Optimising Quantified Expressions in Constraint Models

Ian Gent, Ian Miguel, and Andrea Rendl

University of St Andrews
School of Computer Science, North Haugh, St Andrews, Scotland, UK
ipg,ianm,andrea@cs.st-andrews.ac.uk
http://www.cs.st-andrews.ac.uk

Abstract. One of the key difficulties in Constraint Modeling lies in formulating an effective constraint model of an input problem for input to a constraint solver: many different models exist for a given problem and it is often difficult - even for experts - to determine the model which is solved most effectively by a constraint solver. In recent years, solver-independent modelling languages (MLs) have become increasingly popular among the CP community, such as OPL [10], ESSENCE′ [1] or MiniZinc [5]. These languages are very expressive and allow the user to focus on the problem model rather than on the solver input syntax. An important construct in solver-independent MLs are *quantifiers*, in particular $\forall, \exists, \sum$, which are used to scale constraints and expressions in constraints, similarly to for-loops in program code. However, quantified expressions often contain redundancies, in particular in models formulated by novices. Such redundancies can have a notable impact on the solving performance of the model, in particular since they often increase with problem size. This paper presents new constraint model optimisation techniques concerned with optimising quantified expressions at problem class level. Our experimental results show that quantified expression optimisations can reduce solving time very considerably.

1 Introduction

CP is often inaccessible to novice users, precluding its widespread use. One of the principal reasons that expertise is required is the so-called *modelling bottleneck*: the task of formulating an *effective* constraint model (one that can be solved efficiently) for input to a constraint solver. In order to address this problem, recent research in automated constraint modelling has begun to investigate automated model optimisations, which can compensate for a wide selection of poor modelling choices that novices (and some experts!) often make.

In previous work [6] we have presented several of these optimisation techniques, which can provide impressive speedups. In this paper, we extend this work with *quantified expression optimisations*. Quantified expressions in constraint models perform a similar function to loop constructs in general programming languages. Just as in programming, there are a number of potential pitfalls

in using quantified expressions that can lead an inexperienced user to produce an inefficient model. We will demonstrate that these pitfalls can be avoided automatically through the use of the optimisations described in this paper, which can lead to a substantial increase in model performance over a naive formulation.

2 Background

In recent years, solver-independent constraint modelling languages have become increasingly popular in Constraint Programming. They all provide the same benefits:

- the separation of problem model and parameter (data) specification
- expressive constructs, such as quantified expressions
- advanced data types, such as multi-dimensional arrays
- specific tools, such as syntax highlighting or interactive frontends (languages ESSENCE′ and MiniZinc [5] provide frontends, TAILOR [6] (ESSENCE′) and the MiniZinc-FlatZinc translator, `mzn2fzn` [5], that automatically translate solver-independent models to particular solvers).

Throughout this paper, we use ESSENCE′ as solver-independent modelling language, however, the choice of modelling language is unimportant.

For illustration, we consider the n-queens problem [4]: placing n queens on an $n \times n$ chessboard such that no two queens attack another. A naive constraint model is given in Example 1, which contains n variables (array '`q`'), where each variable represents the row-position of a queen and thus ranges over values $(1..n)$ (line 1). The first constraint (line 5) states that no two queens may be in the same column. The second and third constraints (line 8-10,12-13) disallow two queens positioned on the same NW- and SW-diagonal, respectively.

Example 1. Naive n-Queens problem model formulated in solver-independent constraint modelling language ESSENCE′

```
0    given    n: int(1..)  $ number of queens $
1    find     q: matrix indexed by [int(1..n)] of int(1..n)
2
3    such that
4    $ queens positioned in different columns $
5         alldifferent(q),
6
7    $ no two queens on same NW-SE diagonal $
8         forall i,j:int(1..n) .
9              (i!=j) => (q[i]+i != q[j]+j),
10
11   $ no two queens on same SW-NE diagonal $
12        forall i,j:int(1..n) .
13             (i!=j) => (q[i]-i != q[j]-j)
```

The main difficulty in constraint modelling is to formulate a model of high quality, i.e. a model which will be solved efficiently. For instance, the n-queens model above is inefficient compared to other n-queens constraint models [4],

however, it is an intuitive formulation and hence very likely to come from a novice.

Quantified Expressions (denoted *quantifications*) are a very powerful means in constraint modelling languages to generate sequences of expressions in a compact way. We consider quantifications of the form

$$\varphi \;\; i_1, ..., i_m : int(lb..ub). \quad E(i_1, ..., i_m)$$

where $\varphi \in \{\forall, \exists, \sum\}$ is a *quantifier* that ranges over m *quantifying variables* $I = \{i_1, \ldots, i_m\}$, each defined over the *quantifying domain*, the finite integer range $int(lb..ub)$ where $lb \leq ub$ (denoted D for brevity), and $E(i_1, \ldots, i_m)$ (or E_I) denotes an arbitrary expression that is quantified over $\{i_1, \ldots, i_m\}$. For brevity, we mainly denote quantifications by $\varphi_I D.E_I$.

As an example, consider the following quantification that has been taken from the n-Queens model from Example 1:

$$\forall i, j : int(1..n). \quad (i \neq j) \Rightarrow q[i] + i \neq q[j] + j$$

This is a universal quantification, where quantifier $\forall$ ranges over two variables i and j, hence the set of quantifying variables is $I{=}\{i, j\}$, which ranges over the quantifying domain $D{=}(1..n)$. The quantified expression is $(i \neq j) \Rightarrow q[i] + i \neq q[j] + j$.

Similarly to for-loops in program code, quantifications can include redundancies, in particular when formulated by novices. Typically, the negative effect of redundancies increases with the size of the quantifying domain and the number of quantifying variables, i.e. with $m(ub{-}lb{+}1)$, which can be vast in large instances. The elimination of redundancies in quantifications is therefore vital. In the following two sections, we will discuss two types of redundancies in quantifications and will show how to automatically eliminate them.

3 Moving Loop-invariant Expressions

In some cases, quantified expressions can be reformulated into equivalent, more efficient representations. Therefore, in this section, we study equivalent representations involving *loop-invariant* expressions. We call A in $\varphi_I.A \oplus E_I$ loop-invariant, if A is not quantified by any $i \in I$ and there exists an operator $\oplus'$ such that

$$\varphi_I.A \oplus E_I \equiv A \oplus' \varphi_I.E_I \tag{1}$$

As an example, consider $\forall_I (x{=}0) \vee (y[i]{=}i)$ that contains the loop-invariant expression $(x{=}0)$ and can be reformulated into $(x{=}0) \vee \forall_I y[i]{=}i$ using the law of distributivity. In this case, the latter representation (*outside*-representation) is typically far more efficient than the former (*inside*-representation).

First, we notice that the inside-representation introduces *common subexpressions* as soon as the corresponding quantification is unrolled. More specifically, every quantification $\varphi \; i_1, ..., i_m{:}int(lb..ub).\; A \oplus E_I$ is unrolled to

$$(A \oplus E_1) \oplus_\varphi (A \oplus E_2) \oplus_\varphi \cdots \oplus_\varphi (A \oplus E_k)$$

where $\oplus_\varphi$ represents the corresponding operation for φ, i.e. $\oplus_\forall = \wedge$, $\oplus_\exists = \vee$ and $\oplus_\sum = +$. The number of unrolled subexpressions, k, depends on whether $A \oplus E_I$ is guarded: if unguarded, $k = m*(ub-lb+1)$, otherwise $1 \leq k \leq m*(ub-lb+1)$. Evidently, the unrolled quantification contains k occurrences of the loop-invariant expression A, i.e. common subexpressions that can be eliminated by CSE [6]. We therefore see that if CSE is applied, the redundancies from loop-invariant expressions *in* quantifications can be easily eliminated, therefore the inside-representation is not necessarily always worse than the outside-representation.

3.1 Example: Peaceful Armies of Queens

For illustration, we want to consider an example, a model of the Peaceful Army of Queens Problem from [8]: place two maximal, equally-sized "armies" of queens (black and white) on an $n \times n$ chessboard such that no two queens of the opposite colour attack another.

Example 2. Peaceful Army of Queens Model from [8]

```
 0  given      n : int
 1  letting    N   be domain int(1..n)
 2
 3  find       board           : matrix indexed by [N,N] of int(0..2)
 4  find       numberOfQueens  : int(1..(n*n)/2)
 5
 6 maximising  numberOfQueens
 7 such that
 8 $ we have the same number of white and black queens
 9   (sum row : N.  sum col : N.
10          board[row,col] = 1) = numberOfQueens,
11   (sum row : N.  sum col : N.
12          board[row,col] = 2) = numberOfQueens,
13
14 $ white at [row,col] => no black on same row/col/diagonal $
15   forall row,col : N.
16      (board[row, col] = 1) =>
17      (forall i : N.
18               ((i!=row) => (board[i,col] < 2))    $ row $
19             /\
20               ((i!=col) => (board[row,i] < 2))    $ column $
21             /\
22               (((row+i <= n) /\ (col+i <= n)) =>
23                      (board[row+i,col+i] < 2))    $ S-W diag $
24             /\ etc,    $ other diagonals $
25
26 $ black at [row,col] => no white on same row/col/diagonal $
27   forall row,col : N .
28      (board[row, col] = 2) =>
29      (forall j : N.
30               ((j!=row) => (board[j,col] != 1))    $ row $
31             /\
32               ((j!=col) => (board[row,j] != 1))    $ column $
33             /\
34               (((row+j <= n) /\ (col+j <= n)) =>
35                      (board[row+j,col+j] != 1))    $ S-W diag $
36             /\ etc     $ other diagonals $
```

In our model, we introduce a variable for each cell of the board that can take 3 values: '0' if empty, '1' if occupied by a black queen and '2' if occupied by a white

queen. The array '`board`' contains these variables (line 3). Then we introduce another variable, '`numberOfQueens`' (line 4), that we want to maximise (line 6). The first set of constraints (line 9-12) sets the number of white and black queens. The second set of constraints (line 15-24) states the non-attacking constraints for the black army: for any position (*row*,*col*) on the board, if (*row*,*col*) is occupied by a black queen, then no white queen may be positioned on the same row (line 18), same column (line 20) and the same diagonals (we only show the S-W diagonal in line 22; the other three diagonal constraints are similar). Similarly, the third set of constraints (line 27-36) describes the non-attacking constraints for the white army.

Hence, the non-attacking constraints are of the form

$$\forall col, row. queenAt(row, col) \Rightarrow \forall i. \bigwedge_j positionConstraint_j(col, row, i) \qquad (2)$$

where $queenAt(col, row)$ represents the constraint 'a black/white queen is positioned at (*row*,*col*)' and $positionConstraint(col, row, i)$ corresponds to the six position constraints, i.e. 'no white/black queen may be positioned on the same row/column/4 diagonals'. Expression $queenAt(col, row)$ is loop-invariant of $\forall i. \bigwedge_j positionConstraint_j(col, row, i)$ and can hence be moved inside the quantification, yielding the inside representation:

$$\forall col, row, i. \bigwedge_j queenAt(row, col) \Rightarrow positionConstraint_j(col, row, i) \qquad (3)$$

Equation 2 represents the *outside*-representation while Equation 3 represents the respective *inside*-representation. Hence, the model in Example 2 expresses the non-attacking constraints using the outside-representation. For illustration, we give an excerpt of an equivalent model using the inside-representation in Example 3, showing the non-attacking constraint for the black army (corresponding to the constraint in line(14-24) in Example 2).

Example 3. Excerpt of Peaceful Army of Queens Model, showing the non-attacking constraint for the black army in inside-representation.

```
$ white at [row,col] => no black on same row/col/diagonal $
 forall row,col,i : N.
   (board[row, col]=1) => ((i!=row) => (board[i,col] < 2)) $ row $
 /\
   (board[row, col]=1) => ((i!=col) => (board[row,i] < 2)) $ column $
 /\
   (board[row, col]=1) => (((row+i <= n) /\ (col+i <= n)) =>
                           (board[row+i,col+i] < 2))     $ S-W diag $
 /\ etc,    $ other diagonals $
```

We compare both models (inside- versus outside-representation) in our empirical comparison in Section 5 where we will see that - against common expectations - the inside-representation (Example 3) clearly dominates the outside representation (Example 2).

Status of Expression	Inside-Representation	Outside-Representation
Original	$(\forall_I A \Rightarrow E_I)$	$A \Rightarrow (\forall_I E_I)$
Unrolled	$(A \Rightarrow E_1) \wedge \cdots \wedge (A \Rightarrow E_k)$	$A \Rightarrow (E_1 \wedge \cdots \wedge E_k)$
Flat unnested	$a \Rightarrow e_1, a \Rightarrow e_2, \ldots, a \Rightarrow e_k$ 0 *auxiliary variables* k *constraints*	$aux \Leftrightarrow e_1 \wedge \cdots \wedge e_k$ $a \Rightarrow aux$ 1 *auxiliary variable* 2 *constraints*
Flat nested	$aux_1 \Leftrightarrow (a \Rightarrow e_1)$ ... $aux_k \Leftrightarrow (a \Rightarrow e_k)$ $aux_q \Leftrightarrow (aux_1 \wedge \cdots \wedge aux_k)$ k+1 *auxiliary variables* k+1 *constraints*	$aux_1 \Leftrightarrow e_1 \wedge \cdots \wedge e_k$ $aux_q \Leftrightarrow (a \Rightarrow aux_1)$ 2 *auxiliary variables* 2 *constraints*

Table 1. Comparing the flat representations of the two expressions $\forall_I A \Rightarrow E_I$ (inside) and $A \Rightarrow (\forall_I E_I)$ (outside)

3.2 Inside- versus outside-Representation

We want to study the differences between the *inside*- and the *outside*-representation wrt its representation and efficiency in a constraint solver. Therefore, it is essential to consider each representation at the abstraction level in which it is processed by the solver: its *flat* representation. The flat representation is obtained by *flattening* [3] every constraint to the constraints ('propagators') provided by the target solver, which involves introducing auxiliary variables and additional constraints. For instance, '$A \Rightarrow (x{=}y)$' would be flattened to '$(A \Rightarrow aux) \wedge (aux \Leftrightarrow x{=}y)$', introducing auxiliary variable aux. Flattening is a strictly solver-dependent procedure, therefore we do not conduct a *generic* analysis (covering all possible cases of possible propagators wrt arity, etc), but restrict our analysis to solvers that provide n-ary conjunction, disjunction and summation propagators, which holds for most constraint solvers.

For illustration, Tab.1 depicts how $\forall_I(A{\Rightarrow}E_I)$ and $A{\Rightarrow}(\forall_I E_I)$ are flattened for such solvers: first, the quantification is unrolled, and then each unrolled expression is flattened, for which we distinguish two cases: (1) if to-be-flattened expression E is *not* nested in another constraint, E is flattened to a propagator (constraint), otherwise, (2) E is flattened to an auxiliary variable (denoted aux_q in Tab.1). Then we compare the resulting flat expressions. For instance, the unnested flat inside-representation consists of k implication constraints, introducing no auxiliary variables, while the flat outside-representation consists of 2 constraints (1 reification, 1 implication), introducing 1 auxiliary variable.

We perform this analysis of flat representations for *every* case for which Eq. 1 holds (inside versus outside):

	Inside-Representation	Outside-Representation	
Case (1)	$\forall_I A \wedge E_I$	$A \wedge \forall_I E_I$	
Case (2)	$\forall_I A \vee E_I$	$A \vee (\forall_I E_I)$	
Case (3)	$\forall_I A \Rightarrow E_I$	$A \Rightarrow (\forall_I E_I)$	
Case (4)	$\exists_I A \vee E_I$	$A \vee \exists_I E_I$	
Case (5)	$\exists_I A \wedge E_I$	$A \wedge (\exists_I E_I)$	
Case (6)	$\sum_I A + E_I$	$mA + \sum_I E_I$	where $m = \|I\|$
Case (7)	$\sum_I A * E_I$	$A * (\sum_I E_I)$	

Note, that if loop-invariant A is a constant, then both the *inside*- and *outside*-representations are the same. For instance, in case (3), if A evaluates to *false* then both representations evaluate to *true*. Otherwise, if A evaluates to *true*, both ($\forall_I$ *true*$\Rightarrow E_I$) and (*true*$\Rightarrow (\forall_I E_I)$) evaluate to $\forall_I E_I$. Furthermore, cases (1), (4) and (6) yield identical flat representations (if CSE is applied) and are therefore not included in our comparison.

We summarise our comparison in Table 2: for each of the four cases, we compare the inside and outside representation wrt the number of auxiliary variables (aux) and constraits (cts) in the flat model. This comparison itself is not conclusive, therefore, we tested each representation in artificial problem models (consisting of nothing but the respective constraint) and highlight the representation that was solved in less time in **bold face**.

The overall result is rather surprising: neither representation generally dominates the other. This is particularly interesting, since the outside-representation would be expected to *generally* outperform the inside-representation. However, clearly, the outside-representation is preferable in most cases. Therefore, in our empirical analysis, we investigate the special case where the inside-representation performs better (see Section 5).

Note that it is difficult to make a *generic* statement on which representation is preferable: the representations are not comparable wrt propagation since solvers provide many different propagators. Moreover, we expect that the preferable representation depends on *other* expressions in the problem model: e.g. if the inside-representation shares common subexpressions in the model it might be preferable, if the outside-representation does not.

The observations from this study are integrated into Tailor that can automatically reformulate expressions into the preferable representation. This means

Table 2. Comparing the number of auxiliary variables and constraints in the flat models of inside- and outside representation, in (1) the unnested case and (2) the nested case. The more efficient representation wrt solving time on our test cases is highlighted in **bold face**.

	Case (2)				Case (3)				Case (5)				Case (7)			
	inside		outside		inside		outside		inside		outside		inside		outside	
	aux	cts	aux	cts	aux	cts	aux	cts	aux	cts	aux	cts	aux	cts	aux	cts
(1)	k	k+1	**1**	**2**	**0**	**k**	1	2	k	k+1	**1**	**2**	k	k+1	**1**	**2**
(2)	k+1	k+1	**2**	**2**	k+1	k+1	**2**	**2**	k+1	k+1	**2**	**2**	k+1	k+1	**2**	**2**

that the user can choose a standard representation for each quantifier-operator combination (e.g. outside representation for Case (2), $\forall$ and $\vee$) which allows the user to experiment with different representations. For future work, it would be interesting to explore learning which representation is best for the given model.

4 Addressing Redundancies from Weak Guards

A *guard* B for an expression E is a Boolean expression that has to hold in order to enforce E, i.e. $B \Rightarrow E$. Guards are often used in constraint modelling languages to restrict the number of expressions that a quantification yields. For instance, consider again the n-Queens model in Example 1, where the diagonal-constraints use the guard '(i!=j)':

```
7     $ no two queens on same NW-SE diagonal $
8         forall i,j:int(1..n) .
9             (i!=j) => (q[i]+i != q[j]+j),
```

If guards are *too weak*, they do not eliminate the symmetry stemming from commutative operators, and hence yield duplicate expressions in the unrolled quantification. For illustration, unrolling the diagonal constraint above yields the following set of constraints:

```
q[1]+1 != q[2]+2,        q[1]+1 != q[3]+3,
q[2]+2 != q[1]+1,        q[2]+2 != q[3]+3,
q[3]+3 != q[2]+2,        q[3]+3 != q[1]+1,    etc
```

Note, that the list of constraints contains *duplicates*, since the guard in the quantification, '(i!=j)', does not break the symmetry of '!=', which a stronger guard, '(i<j)', would.

In general, two different kinds of duplication can arise from weak guards. First, if B is *constant* (like in n-queens), then duplicate *constraints* arise when unrolling the quantification. Otherwise, if B is non-constant, duplicate *subexpressions* arise after unrolling the quantification. Duplicate subexpressions can be easily eliminated by common subexpression elimination [6]. Duplicate constraints are most easily eliminated after unrolling a quantification. If all constraints are *ordered* during compilation, duplicates are positioned *consecutively* in the ordered list of constraints and can be detected by testing consecutive constraints for equivalence (in linear time wrt the number of constraints in the problem instance [2]). However, this approach is applicable only to *instances*, when all parameters are known (for instance, in the diagonal constraints of n-queens, we can only eliminate duplicates after the quantification is unrolled, i.e. if n is known). Therefore, we present a more general elimination technique (that is applicable to a problem *class*) where guards are *automatically strengthened.*

4.1 Strengthening Guards by Unification

Unification is a means to find substitutions that render different logical terms equivalent [7]. Unification is applied through the UNIFY algorithm that, given two logical sentences E_1 and E_2, returns a unifier u (if one exists):

$$\text{UNIFY}(E_1, E_2) = u \text{ where } \text{SUBST}(u, E_1) = \text{SUBST}(u, E_2)$$

where $\text{SUBST}(u, E)$ denotes the result of applying the substitution u to E. As an example, consider the two terms $(x + i)$ and $(x + 3)$ that have the unifier $u = \{3/i\}$, i.e. if i is substituted with (i.e. assigned) 3, then both terms are equivalent. We can exploit unification to eliminate duplicate constraints in the following way: given a quantification

$$\varphi I : D.B_I \ominus_\varphi E_I$$

where $\varphi \in \{\forall, \exists\}$, B_I is a Boolean guard, $\ominus_\forall \; = \!\!\Rightarrow$ and $\ominus_\exists = \wedge$, and E_I is the guarded expression (considered by its expression tree), we define:

STRENGTHEN_GUARD($\varphi I : D.B_I \ominus_\varphi E_I$)

1. If E_I's root node corresponds to a binary commutative operator, goto 2. otherwise stop.
2. Compute the set of unifiers U for the two children of E_I, e_1 and e_2.
3. Search U for unifiers from which we can deduce equivalence of the quantifying variables. For instance, if two unifiers u_1 and u_2 are of the form $u_1 = \{i_k/i_l\}$ and $u_2 = \{i_l/i_k\}$ where $l, k \in \{1..m\}$ and $l \neq k$, then we can deduce that if $i_k = i_l$ then e_1 and e_2 are equivalent. If successful, goto 4, otherwise stop.
4. Add condition C to the guard in order to break the equivalence(s) between all quantifying variables i_l and i_k whose equivalence renders e_1 and e_2 equivalent, denoted $\bigwedge_{k,l}(i_k = i_l)$:
   C: conjunction of lexicographical ordering constraints: for each equivalent pair of quantifying variables, a lexicographical ordering is applied to break the symmetry stemming from the commutative operator. For instance, for the pair $(i_k = i_l)$, we get $(i_k \leq i_l)$ (or $(i_l \leq i_k)$, depending on the order). Note, that the lexicographical ordering constraint has to follow a well-defined order in a consistent fashion.
5. Return $\varphi I : D.(B_I \wedge C) \ominus_\varphi E_I$

Note, that the search for specific kinds of unifiers (step 3) can be implemented rather easily, since we are looking for unifiers that follow a particular pattern:

$u_1 = \{i_1/i_2\}$ $\quad$ $u_2 = \{i_2/i_1\}$
$u_1 = \{i_1/i_2 \wedge i_3/i_4\}$, $u_2 = \{i_1/i_2 \wedge i_4/i_3\}$, $u_3 = \{i_2/i_1 \wedge i_3/i_4\}$, $u_2 = \{i_2/i_1 \wedge i_4/i_3\}$
etc

4.2 Example: Strengthening the Guard in Golomb Ruler

For illustration, we show how to strengthen the guard in an example. We consider a naive model [9] of the Golomb Ruler Problem (find a ruler with n ticks of minimal length, such that all ticks are positioned within different distances from another).

Example 4. Naive Golomb Ruler Model from [9].

```
0    given    n      : int    $ number of ticks $
1    letting TICKS   be domain int(1..n)
2    find     ruler : matrix indexed by [TICKS] of int(0..n^2)
3
4    minimising ruler[n]
5    such that
6
7       forall i,j : TICKS.    $ monotonicity $
8             ((i< j) /\ (j <= n)) => (ruler[i] < ruler[j]),
9
10      forall i1,i2,i3,i4 : TICKS.     $ distinction $
11            ((i1>i2) /\ (i3>i4) /\ (i2!=i4)) =>
12                  (ruler[i1]-ruler[i2] != ruler[i3]-ruler[i4])
```

Consider the second constraint (line 10-12) that imposes distinction on the distance between every two ticks. The Boolean guard is weak, since it will result in duplicate constraints after unrolling the quantification:

```
    (ruler[1] - ruler[2] != ruler[1] - ruler[3]),
    (ruler[1] - ruler[3] != ruler[1] - ruler[2]),    etc
```

Hence, we apply STRENGTHEN_GUARD to the quantification: first, since '!=' is commutative, we compute the set of unifiers for the two subtrees $(ruler[i_1] - ruler[i_2])$ and $(ruler[i_3] - ruler[i_4])$. There are four unifiers:

$$u_1 = \{i_1/i_3 \wedge i_2/i_4\} \quad u_2 = \{i_3/i_1 \wedge i_4/i_2\}$$
$$u_3 = \{i_3/i_1 \wedge i_2/i_4\} \quad u_4 = \{i_1/i_3 \wedge i_4/i_2\}$$

From these unifiers we can deduce that $(ruler[i_1] - ruler[i_2])$ is equivalent to $(ruler[i_3] - ruler[i_4])$ if $(i_1 = i_3) \wedge (i_2 = i_4)$ and the following condition is added to the guard:

- C: $i_1, i_2 \leq_{lex} i_3, i_4$, hence $(i_1 \leq i_3) \wedge (i_1 < i_3 \vee i_2 \leq i_4)$.

In summary, STRENGTHEN_GUARD returns the following quantification with a strong guard:

```
  forall i1,i2,i3,i4 : TICKS.
     ( (i1>i2) /\ (i3>i4) /\ (i2!=i4) /\
       (i1<=i3) /\ ((i1<i3) \/(i2<=i4))       $ condition C $
     ) =>       (ruler[i1]-ruler[i2] != ruler[i3]-ruler[i4])
```

The conditions in the new guard need not be simplified (e.g. $(i \neq j) \wedge (i \leq j)$ to $(i < j)$) since they are typically quickly evaluated in the translation frontend (e.g. TAILOR).

We have not yet implemented STRENGTHEN_GUARD and hence cannot give an analysis concerning its runtime. However, we have analysed the effects of duplicate constraints in constraint models (in order to see if duplicate constraints actually affect the solving performance). In our empirical analysis (Section 5) we will see that duplicate constraints stemming from weak guards (in n-Queens and Golomb Ruler) can significantly slow down the solving process by factor 2-3.

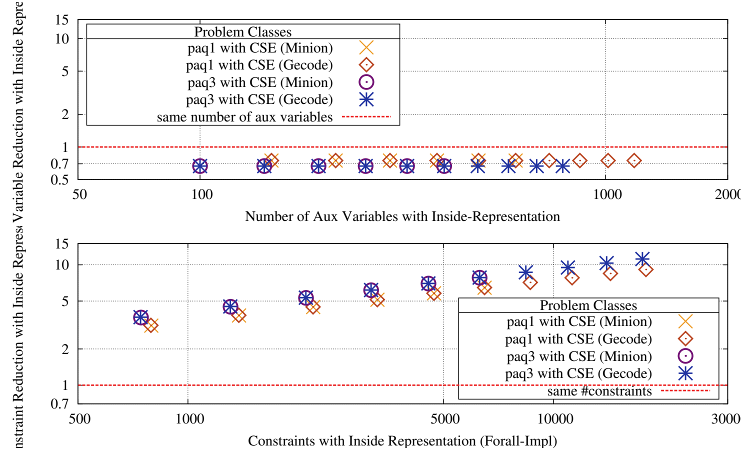

Fig. 1. Moving loop-invariant Expressions: comparing **Instance Size** of different Peaceful Army of Queens instances using inside- and outside-representation. **Auxiliary Variables**(top) and **Constraints**(bottom) for MINION and Gecode. y-axis: increase factor wrt *inside*-representation; e.g. points *above* y=1 depict cases where, using the inside-representation, in aux variables(top)/constraints(bottom) were *increased*.

5 Experimental Results

All our experiments were performed on a Mac Pro 4.2 with 8 GB RAM that contains 2 Quad-Core Intel Xeon 5500 series processors, each 2.26 GHz (hyper-threading off), using TAILORv3.2.0 , Minion 0.9 and Gecode 3.2.2 with Gecode/FlatZinc 3.2.1 and a timeout of 20 minutes. We use default search heuristics in both solvers, first searching on the main decision variables (in their order of definition in the model), followed by auxiliary variables.

5.1 Moving Loop-invariant Expressions

As discussed in Section 3, in some cases, loop-invariant expressions can be moved inside and outside a quantified expression, yielding either the *inside*- or *outside*-representation. Typically, moving loop-invariant expressions outside a quantification is expected to yield the more efficient representation. However, in this study, we investigate a special case, where the inside-representation dominates the outside-representation. More specifically, we compare inside- and *outside*-representation of the case

$$\forall_I A \Rightarrow E_I \equiv A \Rightarrow \forall_I E_I$$

in two different models of the Armies of Queens Problem [8] (one of which is given in Example 2) on solvers Gecode and MINION.

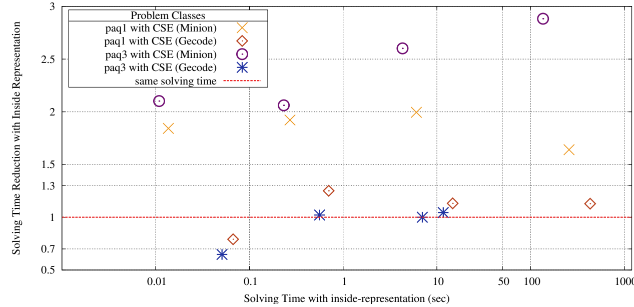

Fig. 2. Moving loop-invariant Expressions: comparing **Solving Time** of different Peaceful Army of Queens instances using inside- and outside-representation. x-axis: solving time with inside-representation; y-axis: increase factor wrt *inside*-representation; e.g. points *above* y=1 depict instances that were solved quicker using the inside-representation. Points below y=1 depict the opposite.

The differences in instances size is depicted in Fig. 1: (top) shows the difference in auxiliary variables and (bottom) shows the difference in constraints: the x-axis shows the number of auxiliary variables/constraints using the inside-representation and the y-axis shows the increase factor in comparison to the outside-representation. This means that all points above $y = 1$ depict instances that contain more auxiliary variables/constraints using the inside-representation. As expected, we observe that for both solvers, the model using the inside-representation contains fewer auxiliary variables but far more constraints than the model using the outside representation.

Figure 2 summarises the comparison in solving time: The x-axis shows the solving time (in sec) for the inside-representation; the y-axis gives the speedup factor using the inside-representation: points above $y = 1$ denote instances that were solved more quickly using the inside-representation, points below depict the opposite.

First, most instances, with the exception of two small instances in Gecode, have been solved quicker using the inside representation. Second, the inside-representation provides a stronger benefit in solver Minion than in Gecode: in Minion the speedup factors goes up to 3, while in Gecode solving time is reduced by moderate 30%. This difference probably stems from the different fashion and cost of posting propagators in each solver, since the inside-representation contains many propagators (constraints). Note, that the reformulation of loop-invariant expressions takes little time and has hence no effect on the overall compilation time. In summary, the results demonstrate that the inside-representation is - in this case - preferable to the outside-representation. We can see that, unlike our expectations, keeping the loop-invariant expression *inside* the universal quantification is actually beneficial.

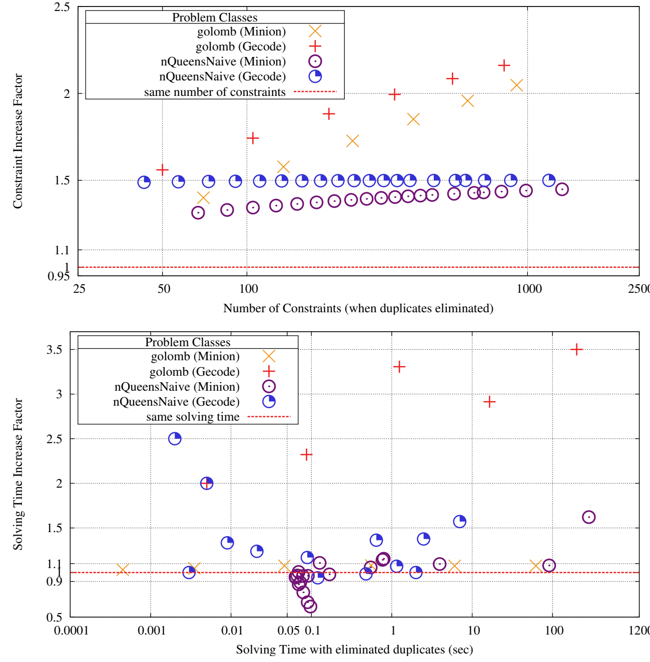

Fig. 3. Eliminating Duplicate Constraints in naive Golomb Ruler and n-Queens models. **Constraints Increase** with n (top) in instances for Gecode and Minion. **Solving Time Increase** (bottom) in solvers Gecode and Minion when duplicate constraints are not eliminated. For both graphs: y-axis: constraint/solving time increase factor if duplicate constraints are not eliminated.

5.2 Duplicate Constraints in Constraint Models

We study the effects of duplicate constraints on the naive n-Queens (Example 1) and Golomb Ruler Problem model (Example 4) as discussed in Section 4, where we compare models with weak guards to models with strong guards since Tailor does not yet perform general duplicate elimination, an item of future work.

First, we consider the *growth* of constraints (duplicates) with increasing n in both problems for Minion and Gecode, illustrated in Fig. 3(top). The y-axis gives the constraint increase factor when duplicates are not eliminated. The number of duplicates remains fairly the same in n-Queens, while it linearly increases with n in Golomb Ruler (for both solvers).

Second, we consider the effects of duplicates on the solving time in Fig. 3(bottom). The y-axis represents the solving time increase factor when instances contain duplicates. For example, all Golomb instances solved in Gecode lie above y=2, i.e. instances with duplicates are solved in more than twice the time used to solve those without duplicates.

We make three main observations. First, most instances with duplicates are solved using *more* time than those without duplicates (most lie above y=1). Exceptions are some n-Queens instances solved in Minion within 0.005 and 0.1 seconds, hence the differences might stem from external factors, since a 30% solving time difference is very small in this time frame. Second, we observe that Golomb Ruler instances with duplicates perform worse than n-Queens instances with duplicates, probably since the number of duplicates linearly increases with n in Golomb Ruler. Third, duplicate constraints seem to have a far more negative effect in Gecode than in Minion, probably stemming from different approaches (and hence different costs) of posting propagators (constraints) in the solvers.

6 Conclusions

This paper has presented model optimisations concerned with improving *quantified expressions*. Quantification optimisations are applicable to both problem instances and classes. As demonstrated on a set of examples, quantification optimisations can compensate for poor modelling choices, resulting in a reduction of solving time to a third. For future work, we plan to extend our investigations of quantification optimisations and other optimisation techniques to provide further enhancement.

References

1. Frisch, A., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008), http://dx.doi.org/10.1007/s10601-008-9047-y
2. Frisch, A.M., Miguel, I., Walsh, T.: Cgrass: A system for transforming constraint satisfaction problems. In: International Workshop on Constraint Solving and Constraint Logic Programming. pp. 15–30 (2002)
3. Gent, I.P., Miguel, I., Rendl, A.: Tailoring solver-independent constraint models: A case study with essence' and minion. In: SARA 07: Syposium on Abstraction, Reformulation and Approximation. pp. 184–199 (2007)
4. Nadel, B.: Representation selection for constraint satisfaction: A case study using n-queens. IEEE Expert 5, issue 3, 16–23 (1990)
5. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: CP 2007. pp. 529–543 (2007)
6. Rendl, A., Miguel, I., Gent, I.P.: Automatically enhancing constraint instances during tailoring. In: SARA 09: Syposium on Abstraction, Reformulation and Approximation. pp. 120–127 (2009)
7. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach (Second Edition), chap. Inference in First-Order Logic, pp. 272–319. Prentice Hall (2003)
8. Smith, B.M., Petrie, K.E., Gent, I.P.: Models and symmetry breaking for 'peaceable armies of queens'. In: CPAIOR-04. pp. 271–286 (2004)
9. Smith, B.M., Stergiou, K., Walsh, T.: Modelling the golomb ruler problem. In: Workshop on Non-binary Constraints (1999)
10. Van Hentenryck, P.: The OPL optimization programming language. MIT Press, Cambridge, MA, USA (1999)