EFFECTIVE COMPILATION OF CONSTRAINT MODELS
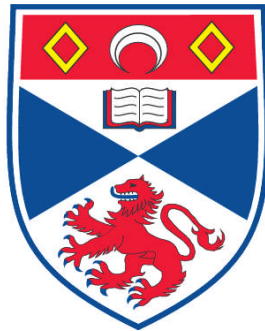
Andrea Rendl

**A Thesis Submitted for the Degree of PhD
at the
University of St. Andrews**
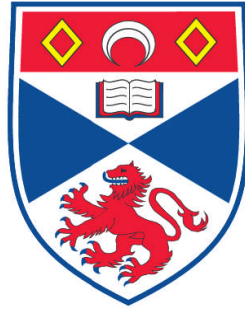
2010

Effective Compilation of Constraint Models

A thesis submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by

Andrea Rendl

School of Computer Science
University of St Andrews

January 2010

ABSTRACT

Constraint Programming is a powerful technique for solving large-scale combinatorial (optimisation) problems. However, it is often inaccessible to users without expert knowledge in the area, precluding the wide-spread use of Constraint Programming techniques. This thesis addresses this issue in three main contributions.

First, we propose a simple 'model-and-solve' approach, consisting of a framework where the user formulates a solver-independent problem model, which is then automatically *tailored* to the input format of a selected constraint solver (a process similar to compiling a high-level modelling language to machine code). The solver is then executed on the input, solver, and solutions (if they exist) are returned to the user. This allows the user to formulate constraint models without requiring any particular background knowledge of the respective solver and its solving technique. Furthermore, since the framework can target several solvers, the user can explore different types of solvers.

Second, we extend the tailoring process with *model optimisations* that can compensate for a wide selection of poor modelling choices that novices (and experts) in Constraint Programming often make and hence result in redundancies. The elimination of these redundancies by the proposed optimisation techniques can result in solving time speedups of over an order of *magnitude*, in both naive and expert models. Furthermore, the optimisations are particularly light-weight, adding negligible overhead to the overall translation process.

The third contribution is the implementation of this framework in the tool TAILOR, that currently translates 2 different solver-independent modelling languages to 3 different solver formats and is freely available online. It performs almost all optimisation techniques that are proposed in this thesis and demonstrates its significance in our empirical analysis.

In summary, this thesis presents a framework that facilitates modelling for both experts and novices: problems can be formulated in a clear, high-level fashion, without requiring any particular background knowledge about constraint solvers and their solving techniques, while (sometimes naturally occurring) redundancies in the model are eliminated for practically no additional cost, improving the respective model in solving performance by up to an order of magnitude.

I, Andrea Rendl, hereby certify that this thesis, which is approximately 78,500 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date _____ *signature of candidate* _____

I was admitted as a research student in September 2006 and as a candidate for the degree of Doctor of Philosophy in September 2006; the higher study for which this is a record was carried out in the University of St Andrews between 2006 and 2009.

date _____ *signature of candidate* _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date _____ *signature of supervisor* _____

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker, that my thesis will be electronically accessible for personal or research use unless exepmt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. We have obtained any third-party copyright permissions that may be required in order t oallow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by the candidate and the supervisor regarding the electronic publication of this thesis:

Access to Printed copy and electronic publication of thesis through the University of St Andrews.

date _____ *signature of candidate* _____

ACKNOWLEDGEMENT

CONTENTS

CHAPTER 1

INTRODUCTION

Constraint Programming is a powerful technique for solving large-scale combinatorial (optimisation) problems. However, it is often inaccessible to users without expert knowledge in the area, precluding the wide-spread use of Constraint Programming techniques. There are two main reasons for this.

1. **Lack of Standards**
   First, unlike similar successful areas, like SAT [15], the Constraint Programming community has not (yet) agreed on a standard format to represent problem instances. This is a major drawback, since every constraint solver takes a different format as input. Furthermore, every constraint solver performs different internal techniques, hence exploiting a solver's strengths requires a thorough study of the solver's internal procedures. This makes it particularly impractical for a novice to explore the strength of Constraint Programming, since he or she has to commit to a single solver.

2. **Generality and the Modelling Bottleneck**
   The second reason stems from the generality of the problems Constraint Programming can tackle: in Constraint Programming, problems are formulated in a rich language, far richer than, for instance, SAT or MIP languages. Typically, many different model formulations exist that represent the same problem and it is vital to choose a high-quality formulation in order to gain a satisfactory solving performance in the solver: a good formulation may be solved in fractions of a second, and poor formulations might not be solvable at all (in a reasonable amount of time). However, it is difficult, often even for experts, to determine the a high-quality CSP formulation. This creates a major bottleneck in Constraint Programming.

This thesis addresses both of these issues in order to render Constraint Programming techniques more accessible to non-experts.

We address the first issue by proposing a simple 'model-and-solve' approach, consisting of a framework where the user formulates a solver-independent problem model, which is

1

then automatically *tailored* to the input format of a selected target solver (a process similar to compiling a high-level modelling language to machine code). The generated solver input is then applied to the solver, from which solutions are retrieved (if they exist) and returned to the user. This allows the user to formulate a constraint model without requiring any particular background knowledge of the respective solver and its solving technique. Furthermore, since the framework can target several solvers, the user can explore different types of solvers.

The second issue is addressed by extending the tailoring process (the translation from solver-independent model to solver input format) with *model optimisations* that can compensate for a wide selection of poor modelling choices that novices (and also experts!) in Constraint Programming often make and hence result in redundancies. Moreover, we will see that for some families of problems, these redundancies naturally occur in constraint models and cannot be easily prevented when modelled as a CSP. Most importantly, in our empirical analysis, we will see that eliminating these redundancies by the proposed optimisation techniques can result in solving time speedups of over an order of *magnitude*, in both naive and expert models. Furthermore, the optimisations are also particularly light-weight, adding negligible overhead to the overall tailoring process.

During the work on this thesis, the framework has been implemented in the tool TAILOR, that currently translates 2 different solver-independent modelling languages to 3 different solver formats and is freely available online. It performs almost all optimisation techniques that are proposed in this thesis and has already contributed to the spread of Constraint Programming techniques in other areas [52].

In summary, this thesis presents a framework that facilitates modelling for both experts and novices: problems can be formulated in a clear, high-level fashion, without requiring any particular background knowledge of the respective constraint solver and its solving techniques, while (sometimes naturally occurring) redundancies in the model are eliminated for practically no additional cost, improving the respective model in solving performance by up to an order of magnitude.

In the following, we want to elaborate on the issues we presented above, by giving a brief introduction to Constraint Programming and discuss the issue of the Modelling Bottleneck in Constraint Programming. Then, we will show how the work presented in this thesis successfully addresses the modelling bottleneck and contributes access to Constraint Programming for non-experts.

1.1   Solving Problems using Constraint Programming

Constraint Programming is a particularly powerful technique that can tackle large-scale combinatorial (optimisation) problems. Solving problems using Constraint Programming proceeds in two steps: *modelling* and *solving*. In the following we will give a brief intro-

duction of how problems are first modelled and then solved in Constraint Programming.

1.1.1 Modelling Constraint Satisfaction Problems (CSPs)

The first step in Constraint Programming, modelling, is concerned with formulating the respective problem as a Constraint Satisfaction Problem (CSP).

Definition 1.1.1. A Constraint Satisfaction Problem (CSP) is a triple (V, D, C) where V is a set of n variables, D a set of n discrete domains and C a finite set of constraints, where each variable $v_i \in V$ is defined over the domain $D_i \in D$, and a constraint $c \in C$ is a relation on a subset of variables of V.

A solution to a CSP is a set of n variable assignments $a_i \in A$ where $a_i \in D_i$ and all constraints in C hold, if every $v_i \in V$ is assigned a_i. As an example, consider the Send-More-Money Problem [23] that is described in Example 1.1.1.

Example 1.1.1. Send-More-Money. A young student in Computer Science asks his parents for some additional money (to buy the latest gadget). The parents decide to grant their son's wish *only* if he can prove to have learnt something during the term. Therefore, they give him the following riddle to solve in order to obtain the money:
Solve the following equation, assigning each letter a distinct number between 0 and 9, where the leftmost letters, S and M must not be zero:

$$
\begin{array}{ccccc}
 & S & E & N & D \\
 & M & O & R & E \\
\hline
M & O & N & E & Y \\
\end{array}
$$

The Send-More-Money Problem can be easily represented as a CSP: first, the set of variables V consists of the 8 letters $V = \{S, E, N, D, M, O, R, Y\}$. Each letter can be assigned a number between 0 and 9, with the exception of the leftmost letters, S and M, that may not be 0. Therefore, we define the set of corresponding domains as $D = \{(1..9), (0..9), (0..9), (0..9), (1..9), (0..9), (0..9), (0..9)\}$. Finally, we define the list of constraints that need to capture (1) the equation and (2) that all variables take different values. We start with the first constraint, that deals with the equation, described as follows:

$$
\begin{array}{rrrrrrrrr}
c_1 \equiv & & 1000\text{*}S & + & 100\text{*}E & + & 10\text{*}N & + & D \\
 & & 1000\text{*}M & + & 100\text{*}O & + & 10\text{*}R & + & E \\
 = & 10000\text{*}M & + & 1000\text{*}O & + & 100\text{*}N & + & 10\text{*}E & + & Y \\
\end{array}
$$

Second, we state that the numbers assigned to each letter have to be different. We can do this either by stating the disequality explicitly, like below.

$$
\begin{array}{rccc}
c_2 \equiv & S & \neq & E \\
 & S & \neq & N \\
 & S & \neq & D \\
 & S & \neq & M \\
 & & \cdots & \\
\end{array}
$$

However, there exists the *global constraint* [84] called *alldifferent* [62, 33] that imposes disequality on all its arguments and is equivalent to the above clique of disequalities:

$c_2 \equiv$ alldifferent(S, E, N, D, M, O, R, Y)

Global constraints [84] represent constraint patterns that often occur in combinatorial problems and for which constraint solvers provide particularly powerful solving techniques.

In summary, we obtain the CSP (V, D, C) consisting of

- variables $V = \{S, E, N, D, M, O, R, Y\}$
- domains $D = \{(1..9), (0..9), (0..9), (0..9), (1..9), (0..9), (0..9), (0..9)\}$
- constraints $C = \{c_1, c_2\}$.

The unique solution to this problem is the assignment $A = \{9, 5, 6, 7, 1, 0, 8, 2\}$, i.e. $9567 + 1085 = 10652$.

1.1.2 Solving Constraint Satisfaction Problems (CSPs)

The solving procedure of a CSP as performed in a typical finite-domain constraint solver, consists of two core parts: propagation [10] and search [81].

Propagation

Propagation is a mean of inference, which prunes values from variable domains. Each constraint is represented by a *propagator* that reduces the domains of the variables in the constraint's scope when possible. In other words, a propagator infers that some value i from a particular variable v is not consistent with the constraint imposed on v. Hence the value i can be removed from the variable's domain.

For example, consider the following, simple CSP:

- variables $V = \{x, y\}$
- domains $D = \{(1..5), (0..4)\}$
- constraints $C = \{x \leq y\}$

where from the constraint $x \leq y$ we can infer two things. First, infer that '5' can be removed from x's domain, since the highest value in the domain of y is 4 and $x \leq y$. Second, we infer that '0' can be removed from y's domain, since '1' is the smallest number in x's domain and $x \leq y$. The $\leq$-propagator applied to x and y prunes their respective domains yielding $x \in (1..4)$ and $y \in (1..4)$.

Search

In many cases, applying propagation is not enough to infer solutions of a CSP: at some point no more values can be removed from the variables' domains. At this point, we have to *search* for solutions.

In principle, search is a structured way of testing various value assignments until a solution is found. More specifically, a *search tree* is constructed, over which a search algorithm iterates in order to detect a solution. The construction ('branching') heuristic and iteration ('search') heuristic concerning the search tree are core features of search.

In many constraint solvers, search is interleaved with propagation, where typically propagation is triggered as soon as search has removed further values of the problem.

In summary, Constraint Programming proceeds in two steps, modelling and solving. This work addresses issues arising during modelling, however, note that modelling and solving are closely linked to another: poor modelling typically has a strong negative effect on the solving performance. In the following section, we want to investigate different established approaches in order to automatically model an efficient constraint model.

1.2   Automated Constraint Modelling

The modelling bottleneck, which stems from the difficulty of selection an appropriate constraint model to represent a given problem, can be addressed by modelling the CSP *automatically*, which is the main concern of *Automated Constraint Modelling*.

At present, there exist several successful automated modelling systems that aim at reducing the modelling bottleneck by generating efficient constraint instances, typically from a rather intuitive (or naive) input. In the following, we present these systems by briefly outlining their approach and objectives.

O'Casey

O'Casey [51] uses case-based reasoning to store, retrieve and reuse constraint programming experience. Case-based reasoning is a means to solve a problem by exploiting experience gained in previous problem-solving episodes [48]. In particular, problems are paired with model instances to form a 'case'. The experience obtained from cases mainly includes propagator selection and search heuristics. Guiding propagator and search heuristic selection by previous experience during tailoring is an interesting possibility for future work.

CONACQ

CONACQ [11] is a SAT-based version space algorithm to acquire constraint networks: given a set of variables (with associated domains), solutions and non-solutions from the user, CONACQ generates a constraint model by applying machine learning. CONACQ has notably evolved over the past years and has been further extended [12] with interesting features: first, CONACQ can assist in the selection of examples for learning, through which a smaller set of examples is needed to generate a good constraint network. Furthermore, CONACQ now learns non-binary constraints and provides interactive assistance to the user. In summary, CONACQ is an established system that is the subject of active research. This application of machine learning to acquiring a good set of valid constraints is a useful idea for automated modelling, however, since (non-)solutions are a prerequisite, it is not applicable to tailoring.

CONJURE

CONJURE [27] is an automated refinement system, that, given an abstract problem specification, returns a set of constraint models derived from the specification. The idea is to allow the user to formulate a problem in an abstract, mathematical way, using abstract constructs such as mappings, sets, functions, etc. This abstract specification is then *refined* to a CSP by non-deterministic refinement rules. Refinement rules include data structure refinement (e.g. refining mappings into arrays), as well as corresponding operator refinement. In its implementation, CONJURE takes specifications formulated in the abstract specification language ESSENCE [26] as input, and returns constraint models in ESSENCE′ (Chapter 2).

Summary

In summary, the presented automated modelling tools aim at generating an efficient constraint instance using different approaches. This thesis is concerned with the *subsequent* step, after a constraint model has been formulated, when it has to be *tailored* to a constraint solver in order to be solved. In the following section, we outline our contributions towards reducing the modelling bottleneck.

1.3   Thesis Statement and Contributions

Modelling in Constraint Programming is a notoriously difficult task, which prevents the widespread use of Constraint Programming techniques, a phenomenom referred to as the *modelling bottleneck*. Automated Constraint Modelling currently addresses the first essential step in modelling, which is to formulate a constraint model from a (sometimes informal) problem specification. However, a further step remains: to represent the model

in solver format. The translation from solver-independent constraint model to solver input is non-trivial and hence desirable to be automated.

This dissertation defends the thesis that this second step in automated modelling, the compilation from solver-independent constraint model to solver input, can be *automated* and extended with light-weight *model optimisations* that can deliver substantial enhancements of the problem model formulation. To defend this claim, we present a set of compilation and optimisation algorithms and thoroughly assess their implementation on a large set of examples. In the following, we explicitly outline our contributions.

1.3.1   Specification of ESSENCE′

The first contribution is the specification of the solver-independent constraint modelling language ESSENCE′. ESSENCE′ is a derivative of the problem specification language ESSENCE, and has not been explictly defined to date. This thesis gives a detailed introduction to ESSENCE′ in Chapter 2, including many examples. In Appendix A we summarise the syntax of ESSENCE′ in a more formal way; its semantics is inherited from that of ESSENCE [26].

1.3.2   Tailoring

The second contribution is the detailed discussion of *tailoring*. Tailoring is the process of compiling a solver-independent (high-level) problem formulation into low-level solver input. The main challenge is to produce a valid solver input from an input file, using as little time as possible.

As we will show in this thesis, the process shares many properties and concerns with the compilation of programs, from which we can gain substantial inspiration on how to perfect the process. In particular, we present the architecture of an effective tailoring engine, consisting of three parts (like a compiler), with two important properties: first, the engine is easily extendable to support other input- or output-languages (Sec. 3.1). Second, the key compilation entities (for preprocessing and flattening) that represent the core of the translation process, are re-usable for every translation and needs not be re-implemented for different target solvers or input languages.

Moreover, we do not limit our discussion of tailoring to problem *instances*, but extend it to tailoring whole problem *classes*. Tailoring problem classes has barely been studied, in Constraint Programming or related areas. We show how to extend the tailoring procedure at instance level to class level, highlight the main challenges in tailoring at class level and demonstrate that class-wise tailoring can be a competitive alternative to instance-wise tailoring.

1.3.3 Model Optimisations

The third contribution is the introduction of automated *model optimisations* that aim at enhancing problem models. These optimisations are easily integrated into tailoring, combined with core tailoring tasks, such as normalisation or flattening, which render the optimisations particularly cheap to perform. In the following we briefly discuss two optimisation techniques that we propose.

Common Subexpression Elimination

The most successful optimisation technique from this thesis is that of common subexpression elimination (CSE), a technique that is widely spread in related areas. In this work we present a new CSE technique, that is integrated into the (necessary) process of flattening, and which hence adds practically no notable overhead to the overall tailoring procedure, even if performed in vain. Furthermore, CSE can result in solving time speedups of up to a factor of 2,000 in some problem instances.

We extend the basic approach of CSE with additional measures that aim at *increasing* the benefits from CSE-flattening. CSE-flattening is limited to detecting subexpressions that are *identical* (i.e. they are equivalent wrt their syntax). Therefore, we augment tailoring with optimisation techniques that *detect* equivalent subexpressions that are not identical and then *reformulate* them into an identical representation. We exploit several different kinds of equivalences, each providing different benefits. The most successful one, the *active negation reformulation* has the best impact on solving performance, in some cases reducing the overall solving time to a third of the solving time used for instances tailored without the reformulation.

Applying CSE is also discussed at class level (Sec. 6.2), where CSE has to be performed in an advanced fashion to detect all equivalences that are detected at instance level. We present three different CSE techniques at class level, each with its own advantages and drawbacks. In summary, CSE is a powerful optimisation technique, at both instance and class level, that yields dramatic speedups, on novice and expert models (Sec. 8.2).

Eliminating Duplicate Constraints

Duplicate constraints often arise in naive models of inexperienced users, by specifying Boolean guards that are too weak (Sec. 4.5.1). We show how to eliminate those duplicates at both instance and class level, applying two different techniques. Our empirical results confirm the benefits of this optimisation technique, as duplicate constraints can double the solving time in medium-sized instances (of the problems we have considered).
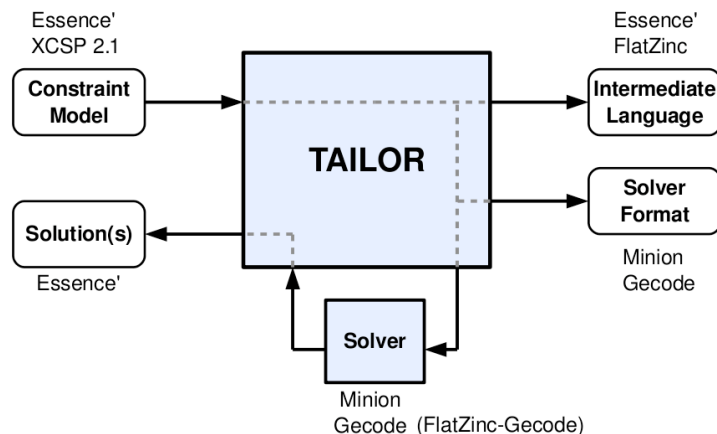
Figure 1.1: **Tailoring Overview**. Given a problem model (paired with a parameter specification), the tailoring engine produces solver input undergoing three major steps: preprocessing, flattening and mapping to solver syntax.

1.3.4 The tool TAILOR

The most practical contribution is the implementation that incorporates almost all procedures that are proposed in this thesis: TAILOR. TAILOR is an interactive modelling assistant that is freely available at TAILOR's website [65]. It provides a graphical user interface in which the user can both model her problem, tailor it to a solver (Minion [32] or Gecode [80]), call the solver externally and retrieve solutions (if they exist).

TAILOR's Capabilities

The flow-graph in Fig. 1.1 illustrates the different operations TAILOR can perform. As input, TAILOR takes constraint models formulated either in modelling language ESSENCE′ or the XML format XCSP 2.1. TAILOR can perform several different translations of this input. First, it can generate intermediate formats, such as flat ESSENCE′ or the FlatZinc format. Second, it can generate solver input format for constraint solvers MINION(text format) and Gecode (C++) (translation to the latter is still restricted). Third, TAILOR can guide the whole solving process for you: first generating solver input, then invoking the solver on the input and finally mapping the solution back to ESSENCE′ or FlatZinc. In this way the user simply has to model the respective problem, click the 'solve' button and does not have to bother about the solver input/settings at all.

A Snapshot of TAILOR

We show a snapshot of TAILOR's graphical user interface (of version 0.3.2) in Fig. 1.2. It consists of the input part on the left hand side and the output part on the right hand side.
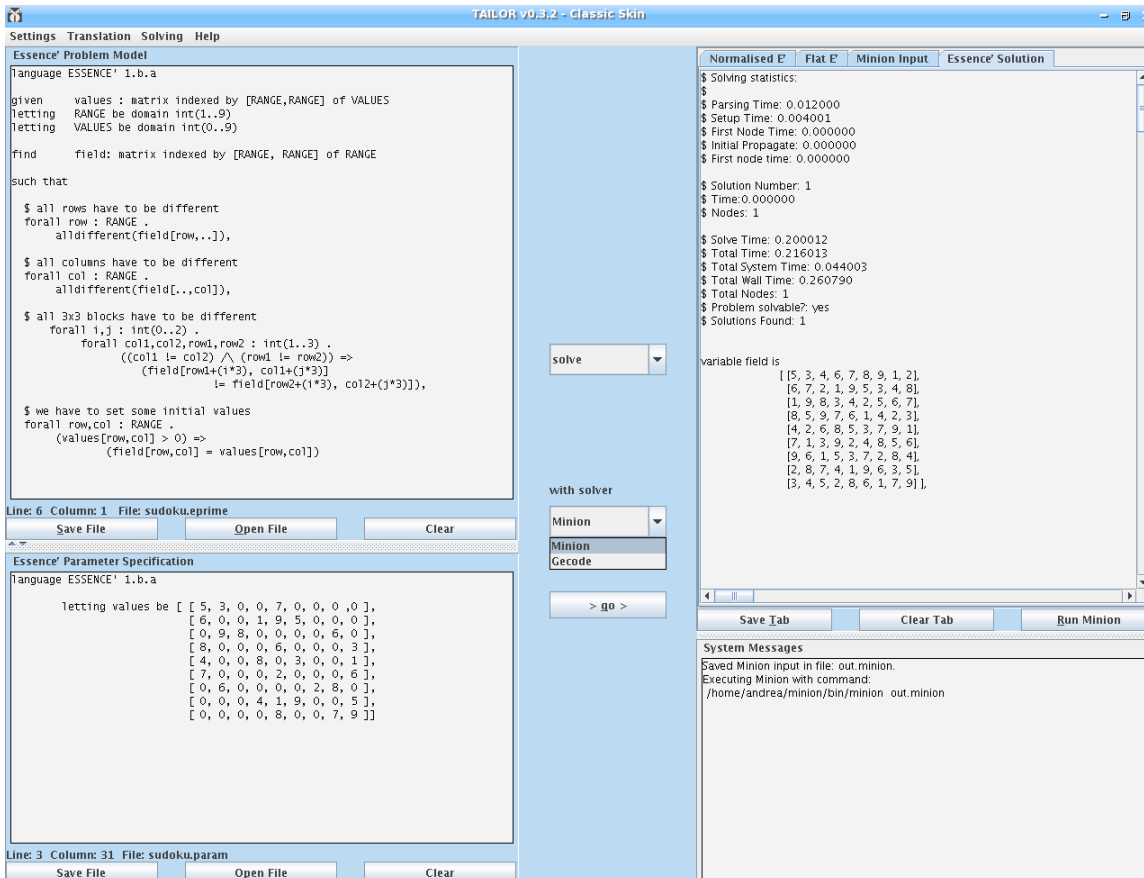
Figure 1.2: **Snapshot of** TAILOR **v0.3.2's graphical user interface (GUI)** solving a Sudoku instance in solver Minion. The GUI is divided into an input part (left) where problem class(top) and parameters(bottom) are specified, and an output part (right) that returns solutions(top) and system messages(bottom). The drop-down menus and buttons in the middle enable the user to select the target solver and select between solving and tailoring

The input part to the left has two fields for modelling: the top field is used to model the problem class; the bottom field to specify parameter values (data). The parameter field can be left empty. There is a series of ESSENCE′ examples that comes with the TAILOR distribution (like the Sudoku problem model in the image) to get familiar with modelling in ESSENCE′.

The output part on the right summarises the results obtained from either tailoring or the target solver: the top right field is a collection of tabs, each showing the translation output at a different stage: normalised ESSENCE′, flattened ESSENCE′, solver input and the ESSENCE′ solution. The content of each tab can be saved using the 'Save Tab' button.

In between the input and output part, there are drop-down menus to select a translation mode: the first option is 'solve' or 'tailor' (in Fig. 1.2 the 'solve' option is selected). If 'solve' is selected, TAILOR will translate the problem from the left hand side into the corresponding solver format, save it in a file, execute the solver on the file (in a separate

process), and return the solver output (in either ESSENCE′ or FlatZinc format) on the right hand side.

In summary, TAILOR is an attractive and useful tool, in particular for novices, to model and efficiently problems using Constraint Programming techniques.

1.3.5 Addressing the Missing Link in Automated Constraint Modelling

To date, automated modelling has mainly focussed on how to *formulate* a given problem as a constraint model, an essential task to reduce the modelling bottleneck. This is, however, not enough. In order for a (generated) model to be *solved*, it has to be formulated in the solver language, a non-trivial step that constitutes another challenge to the modeller, a challenge that has not yet been recognised and investigated for automation. The last and probably most significant contribution of this thesis is the *identification* of this last important step during automated modelling, as well as its *automation*.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2: *The Modelling Language* ESSENCE′ We begin with a detailed specification of the solver-independent modelling language, that is used through-out this thesis: ESSENCE′. ESSENCE′ has not yet been explicitly specified, so this chapter delivers a general contribution in form of a detailed description. The presentation of ESSENCE′ is illustrated with many different examples and demonstrates that ESSENCE′ incorporates all necessary facilities that other constraint modelling languages provide, demonstrating that ESSENCE′ is a suitable choice of modelling language to illustrate the issues discussed in this thesis.

Chapter 3: *Tailoring Problem Instances* The third chapter gives a thorough discussion of the task of *tailoring*, which is the compilation of a solver-independent constraint model to a particular constraint solver. This chapter illustrates how tailoring can be performed in a structured, easily-extendable and efficient fashion, generalising the core tailoring tasks for an arbitrary input (constraint language) and output (solver language), while considering the distinguished features a target solver may provide.

Chapter 4: *Instance Optimisations* This chapter is the largest and most significant part of this work, as it presents instance optimisations that can easily be integrated into tailor-

ing, adding negligible computational effort while resulting in speedups of up to a factor of 3,400. We first explore various optimisation techniques that have been proposed in the context of Constraint Programming and Code Optimisation in Compilers where the latter has inspired many of the optimisation techniques we propose. The optimisations include four main techniques: first, *common subexpression elimination*(CSE) is the most powerful optimisation technique we propose that we show to be applicable to many constraint models. Second, we propose a set of techniques in order to *increase* the number of common subexpressions in order to augment the benefits obtained through CSE. Third, we consider the elimination of redundant constraints, in particular *duplicate constraints* that often occur in constraint models of inexperienced modellers. Fourth, we study *optimisations of quantifications*, in particular the role of Boolean guards and different quantification representations involving loop-invariant expressions.

Chapter 5: *Tailoring Problem Classes*   In this chapter we extend the tailoring process from translating problem instances to whole problem *classes*. We start with presenting two different applications of tailoring problem classes and proceed with a thorough discussion of the extension of the instance tailoring procedure from Chapter 3, where we highlight challenges and current limitations.

Chapter 6: *Class Optimisations*   This chapter discusses automated enhancements of a problem class that can be integrated into tailoring problem classes as presented in the previous chapter. We show how the techniques proposed at instance level are applicable at class level and how they need to be extended and refined in order to provide similar benefits at class level as at instance level.

Chapter 7: *Case Study: Common Subexpressions in CSPs of Planning Problems*   In this chapter we summarise our observations from enhancing a particular kind of CSP: CSPs that represent AI planning problems are particularly amenable to CSE and, the more complex their structure, the more benefits can be gained from CSE. We first present well-established techniques of how to represent an AI Planning problem as a CSP and highlight the sources of common subexpressions in a generic problem formulation. Then we present four case studies, considering four different AI Planning problems of different complexity, and analyse the impact of CSE on each of them.

Chapter 8: *Experiments*   This chapter contains our main empirical analysis where we assess all the proposed optimisation techniques wrt (1) the scope of each enhancement (e.g. reduction of constraints in the model), (2) the impact of the optimisation on the solving performance, and (3) the optimisation's computational effort during tailoring (does performing the optimisation add an significant overhead to the overall translation process).

Chapter 9: *Conclusions*   The final chapter first summarises the work, then gives a thorough conclusion and outlines plans for future work.

CHAPTER 2

THE MODELLING LANGUAGE ESSENCE′

A discussion of translating constraint models to solvers requires a solver-independent modelling language in which to formulate problems. Unfortunately, the Constraint Programming community is still far away from agreeing on a standard constraint format or standard constraint modelling language. However, there exists a small range of solver-independent modelling languages (MiniZinc [58], OPL [83], ESSENCE′ and XCSP 2.1 [68]) from which we had to choose an appropriate candidate at the beginning of this PhD project. We could not chose MiniZinc since it has been developed *after* this project was initiated. Commercial OPL did not apply to our plans of offering free software, and the CSP solver competition format XCSP 2.1 [68] does not permit formulating problem classes, thus our choice fell on ESSENCE′. ESSENCE′ is a derivative of the specification language ESSENCE [26] and has only been specified *implicitly* through ESSENCE. Therefore, this chapter contributes a thorough specification.

We stress that the general choice of modelling language is unimportant, since this work is applicable to any of the modelling languages mentioned above. ESSENCE′ is a deserving candidate for two main reasons: first, ESSENCE′ incorporates all typical features that other constraint modelling languages provide, which we will demonstrate in this chapter. Second, ESSENCE′ is a plain and straight-forward modelling language, whose underlying idea is to allow users to formulate problems in a succinct, mathematical way without requiring any CP background. The syntax is based on typical mathematical notation and the model structure is similar to that of mathematical modelling languages, such as AMPL [24] or ZIMPL [47]. We will use ESSENCE′ as modelling language throughout this thesis.

This chapter is structured as follows: first, Sec. 2.1 introduces ESSENCE′ by formulating a classical combinatorial problem, the Graph Colouring (or Map Colouring) Problem. The section covers basic concepts of ESSENCE′ and gives a brief overview of the available facilities. Section 2.2 provides detailed information about ESSENCE′, in particular its format, features (types, expressions, etc) and its usage. A detailed grammar specification of ESSENCE′ can be found in Appendix A.

2.1 Essence′ by Example

The Graph Colouring (or Map Colouring) Problem (GCP) is a classical combinatorial optimisation problem: given a non-directed graph and a number of colours, assign a colour to each vertex, such that every pair of adjacent vertices has distinct colours and a minimal number of colours is used. Fig. 2.1 illustrates an example of a minimally coloured graph. The GCP can be formally specified (in the $\mathcal{L}$ language from [41]):

parameters	$vertices \in \mathbb{N}_1$	$Vertices$: set(1..$vertices$)
	$colours \in \mathbb{N}_1$	$Colours$: set(1..$colours$)
	$edges \in (Vertices \times Vertices)$	
variables	$colouring$: $Vertices \rightarrow Colours$	
	$usedColours : Colours \rightarrow (0,1)$	
objective	minimise $\sum_{c \in Colours} . usedColours(c)$	
constraints	$\forall (a,b) \in edges . colouring(a) \neq colouring(b)$	
	$\forall v \in Vertices. \forall c \in Colours.$	
	$(colouring(v) = c) \Rightarrow (usedColours(c) = 1)$	

The number of *vertices* and *colours* are given as parameters from which the set of vertices and set of colours, *Vertices* and *Colours*, are defined. The graph *edges* are also parameters, specified by the Cartesian product between the set of vertices (i.e. specified by adjacent vertices). The aim is to find a valid *colouring*, i.e. a mapping of vertices to colours. Additionally, *usedColours* is employed as a helper mapping that records which colours are actually in use: colour c is mapped to '1' if c is used in the colouring and to '0' otherwise. The objective is to use a minimal number of colours, hence the sum of all used colours has to be minimal. Two constraints define the GCP: first, for all vertices a and b that are connected by an edge, the colours assigned to a and b have to be different. Second, colouring vertex v with colour c implies that colour c is used.

Modelling the Problem
The GCP specification can be easily converted into an Essence′ model, which is given in Fig. 2.2. First, parameters are declared: the number of '`vertices`' and '`colours`' are positive integers, and the '`edges`' are represented as adjacency matrix, i.e. a 2-dimensional matrix of zeros and ones, where '`edges[a,b]=1`' if there is an edge between vertices '`a`' and '`b`'. '`VERTICES`' and '`COLOURS`' are labels for the range of vertices and colours, respectively. Labels are practical means to represent constants that often occur in a problem model.
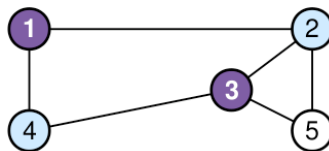
Figure 2.1: Sample solution to the Graph Colouring Problem (GCP)

```
language ESSENCE' 1.b.a
$ ———————————— PARAMETERS and CONSTANTS ————————————
given    vertices    :  int(1..)
given    colours     :  int(1..)

letting VERTICES     be domain int(1..vertices)
letting COLOURS      be domain int(1..colours)

given    edges        :  matrix indexed by [VERTICES,VERTICES] of int(0,1)
$ ———————————— VARIABLES ————————————
find     colouring   :  matrix indexed by [VERTICES] of COLOURS
find     usedColours :  matrix indexed by [COLOURS] of int(0..1)
$ ———————————— OBJECTIVE ————————————
minimising sum c : COLOURS . usedColours[c]
$ ———————————— CONSTRAINTS ————————————
such that
    forall a,b : VERTICES .
        (edges[a,b] = 1) => (colouring[a] != colouring[b]),

    forall v : VERTICES .   forall c : COLOURS .
            (colouring[v] = c) => (usedColours[c] = 1)
```

Figure 2.2: ESSENCE′ **problem model** of the GCP.

Second, the variables 'colouring' and 'usedColours' are defined. In the problem specification, *colouring* and *usedColours* are defined as mappings, however, constraint modelling languages typically do not provide mappings as data structures. Thus every mapping has to be represented by an array ('matrix'). There are several ways of refining a mapping to an array [27]. In this case, every mapping $m : X \rightarrow Y$ is represented by a matrix m' where $m'[x] = y$ for all $x \in X$ and $y \in Y$ where $m(x) \rightarrow y$. As an example, the mapping *colouring: Vertices → Colours* is represented by a 1-dimensional array, 'colouring', of length *vertices*, where every array element 'colouring'[i] stands for the colour assigned to a vertex i. In other words, 'colouring' has an element for every vertex whose domain ranges over all colours, i.e. 'colouring[vertex]=colour'.

Finally, the objective and constraints are specified. Note, that the formulation is very close to the problem specification. The objective states that the sum of all used colours has to be minimal. The first constraint states that for vertices 'a' and 'b' that are connected by an edge, the colours assigned to a and b have to differ. The second constraint states that if colour 'c' is assigned to vertex v then colour 'c' is used, i.e. 'usedColour[c]' is assigned '1'.

Defining Parameter Values

In order to solve an *instance* of the Graph Colouring Problem, parameter values (also known as 'data') have to be specified. Typically, this is done in a separate file, the *parameter specification*. The separation of problem and data facilitates modelling and is standard in many constraint modelling languages. For illustration, Fig. 2.3 shows the ESSENCE′ parameter specification for the graph in Fig. 2.1 that has 5 vertices, 3 colours and 6 edges

defined by an adjacency matrix.

```
language ESSENCE' 1.b.a

letting vertices be 5
letting colours  be 4
letting edges    be [[0,1,0,1,0],
                     [1,0,1,0,1],
                     [0,1,0,1,1],
                     [1,0,1,0,0],
                     [0,1,1,0,0]]
```

Figure 2.3: ESSENCE' **parameter specification** of the GCP in Fig. 2.1.

Presenting Solutions

Constraint solvers differ greatly in how they output solutions, hence it is practical (and standard in most constraint modelling languages) to provide means to describe solutions. For illustration, Fig. 2.4 shows the ESSENCE' solution for the instance obtained by the parameters in Fig. 2.3, which also corresponds to the coloured graph in Fig. 2.1.

```
language ESSENCE' 1.b.a

variable colouring is [1, 2, 1, 2, 3],
variable usedColours is [0, 1, 1, 1]
```

Figure 2.4: ESSENCE' **solution description** of the GCP shown in Fig. 2.1

Quick Summary

ESSENCE' can be used to formulate three specifications: problem models, parameters and solutions. It has a concise syntax which is based on mathematical notations. Furthermore, it embodies all standard features (separation of model and data, quantifications, arrays, etc) that other solver-independent CP modelling languages, such as OPL or MiniZinc, provide.

2.2 Format

The format of ESSENCE' is closely related to that of ESSENCE, which is thoroughly discussed in [26]. In particular, ESSENCE' is a subset of ESSENCE with some additional features that we present in this section. Furthermore, we give an anecdotal overview of the format of ESSENCE', highlighting the most important features, starting with the model structure.

An ESSENCE' model is structured in the following way:

1. Header
2. Declarations (constants, parameters, variables)
3. Objective
4. Constraints

All statements from above are optional, with the exception of the header, which states the ESSENCE′ version number. Comments are line-wise and preceded by the symbol '$'. The following format discussion is focused on the format of *problem models*. Parameter specifications follow the same structure and syntax with the difference, that it consists solely of a header and declaration part. Similarly, a solution specification contains only the header and a set of solution statements, as shown in Fig. 2.1. For a detailed grammar specification, consult Appendix A.

Types

ESSENCE′ is a strongly typed language where every expression has a type. The basic types are **Boolean** and **integer**; advanced types are **domain** and array (**matrix**) types, which are again either Boolean or integer. Note that arrays are denoted '**matrix**' in ESSENCE′'s syntax. All types are further discussed in Sec. 2.2.1.

Identifiers and Scopes

Every identifier that is used in an expression has to be declared before use. ESSENCE′ does not support local variables, with the exception of quantified variables whose scope is limited by the corresponding quantification. The scope of all other identifiers is global. Identifier names have to be unique in their scope.

Categories

ESSENCE′ has four categories of expressions: *constant*, *parameter*, *quantifying variable* and *decision variable*. There is a strict order over the four categories:

constant < parameter < quantifying variable < decision variable

Every expression is assigned a category in a recursive fashion: atoms (terminals) have the category corresponding to their declaration; expressions composed of subexpressions are assigned the highest category of its subexpressions. As an example, $4 + n$ has category *parameter* if n is a parameter; $3 * x$ has category *decision variable* if x is a decision variable.

Categories are necessary to describe the arguments of global constraints. For instance, in the global constraint *atmost*(var,val,occ) (see Tab. 2.4) decision variables are not allowed for argument occ - only constants, parameters and quantifying variables. However, arguments that are typically of category decision variable, like var, can be replaced by any of the other, lower categories. Thus, the ordering incorporates a means to express the 'highest' category that can be used as an argument. In the case of *atmost*, the respective highest feasible category for its arguments would be *atmost*(var:dec-var,val:q-var,occ:q-var).

Notation

Throughout this thesis, Boolean expressions are denoted with uppercase letters from the beginning of the alphabet (A, B, C, etc.); integer expressions are denoted with lower-case

letters from the ending of the alphabet (x, y, z, etc). Arrays are denoted with the prefix 'array$i$', where i gives the dimension of the array. Domains are written in capital letters (e.g. DOM). In the remainder of this chapter, 'Boolean expression' and 'integer expression' refer to expressions that are neither array nor domain expressions; if expressions are of type domain or array, it will be specifically emphasised.

2.2.1 Expressions

Every ESSENCE$'$ expression is either Boolean or integer. Advanced types are domain and array expressions, which are again either Boolean or integer. In this section, we cover *basic* expressions, i.e. expressions that are either atoms or constructed by unary or binary operators. Array expressions are covered in Sec. 2.2.2, followed by domain expressions in Sec. 2.2.3. Complex expressions, such as global constraints are discussed in Sec. 2.2.7 and quantifiers ($\forall$, $\exists$ and $\sum$) in Sec. 2.2.8.

Basic Boolean Expressions are either constants (*true* and *false*), Boolean identifiers or expressions composed by operators that yield Boolean expressions. Table 2.1 summarises all unary and binary Boolean and relational operators that yield Boolean expressions in ESSENCE$'$. These operators are standard in most solver-independent modelling languages. Note that n-ary operators that yield Boolean expressions, such as global constraints and quantifiers, are discussed in Sec. 2.2.7 and Sec. 2.2.8, respectively.

Boolean Operators		
`!A`	negation of A	$\neg A$
`A /\ B`	A and B	$A \wedge B$
`A \/ B`	A or B	$A \vee B$
`A => B`	A implies B	$A \Rightarrow B$
`A <=> B`	A is equivalent to B	$A \Leftrightarrow B$

Relational Operators		
`x = y`	equality	$x = y$
`x <= y`	less or equal	$x \leq y$
`x >= y`	greater or equal than	$x \geq y$
`x != y`	disequality	$x \neq y$
`x < y`	less than	$x < y$
`x > y`	greater than	$x > y$

Table 2.1: **Unary and Binary Operators** yielding **Boolean expressions**. A and B are arbitrary Boolean expressions, x and y are arbitrary integer expressions.

Basic Integer Expressions are either integer constants (e.g. 0, 120, 7), integer identifiers or expressions composed by operators that yield integer expressions. Table 2.2 summarises all unary and binary operators that yield integer expressions. These operators are standard

in most solver-independent modelling languages. Note that the n-ary sum constraint($\sum$) is discussed in Sec. 2.2.6.

-x	negative x	$-x$
|x|	absolute value	$\mid x \mid$
x + y	addition	$x + y$
x - y	subtraction	$x - y$
x * y	multiplication	$x * y$
x / y	integer division	x/y
x mod y	x modulo y	$x \bmod y$
x ^ y	x to the power of y	x^y

Table 2.2: **Unary and Binary Operators** yielding **integer expressions**. x and y are arbitrary integer expressions

Polymorphic Operators. In our implementation, particular operators are polymorphic, i.e. they can be applied to both Boolean and integer expressions. The polymorphic operators are **+, -, *, sum** and the relational operators (**=, <=**, etc). If these operators are applied to Boolean expressions, then *true* is converted to 1 and *false* to 0. This provides two benefits: first, it gives more expressiveness to ESSENCE′ (e.g. one can state that exactly 2 out of 3 statements have to be true, as illustrated in the example below).

```
$ example of polymorphic operations$

 A + B + C = 2        $ exactly 2 statements have to be true   $
 (x=0) * (y=0) = 1    $ expressing (x=0) xor (y=0)$
```

Second, since most solvers allow summation/multiplication of Boolean variables, ESSENCE′ provides direct support of these features. Furthermore, note that many solver-independent modelling languages provide a *bool2Int* function that converts a Boolean expression into an integer expression.

2.2.2 Arrays

Arrays are a means to collect a fixed number of items of the same type into one structure. In general, the array type is defined by the element type and the dimension. For example, a 2-dimensional array containing Boolean elements is of type 'Boolean 2-dimensional array'.

Declaring Arrays
Constants, parameters or decision variables can be declared as an array using the keyword '**matrix**', followed by its dimension ('**indexed by**') and the base domain ('**of**'), i.e. the domain over which the array elements range. An example would be

```
   array1:   matrix indexed by  [int(1..10)] of bool
```

that declares a Boolean array labelled 'array1' containing 10 elements. The *base domain* ('**bool**') denotes the basic type of the array elements. The *index domain* ('**int** (1..10)') states dimension, length and how to dereference the array. In the example above, there is one index domain, '[**int** (1..10)]', hence 'array1' has 10 entries and 'array1[1]' dereferences the first element since the index domain starts with '1'. Arrays can have arbitrary dimensions, for instance, the following statement declares a 2-dimensional integer array with $10 * 6$ elements whose elements range over the integer domain (1..5).

```
array2:   matrix indexed by   [int(1..10), int(0..5)] of int(1..5)
```

Constructing Arrays

Simple expressions of the same type can be combined into an array expression by using the array constructors '[' and ']', e.g. '[x,y,z]'. Multi-dimensional arrays are constructed by nesting '[' and ']', as the example below demonstrates:

```
$ Constructing a 2-dimensional array in a constant definition $
letting myArray be [ [1,2], [3,4], [5,6], [7,8], [9,10], [11,12] ]
```

Note, that in the current implementation, the elements of a constructed array have to be *atomic* expressions, i.e. constants or identifiers. If the corresponding index array of a constant array is not defined, then it is indexed starting from 1. For example, if the index domain of 'myArray' has not been defined, then 'myArray[2,1]' will return '3'.

Dereferencing Arrays

Array elements are accessed by a similar syntax to that used in many programming languages. For instance, given 'myArray' from the example above, 'myArray[4,1]' represents '7', since it is the '1'st element of the '4'th vector (1-dim. array) in the matrix. It is also possible to dereference a set of elements of an array by dereferencing *ranges*, which are similar to domain expressions. Table 2.3 illustrates dereferencing with ranges.

3-Dimensional sample Array	
letting array3 **be**	`[ [[1,2],    [3,4]],`
	`  [[5,6],    [7,8]],`
	`  [[9,10],  [11,12]] ]`
Array Dereference	Referenced Expression
'array3[3,1,..]'	'[9,10]'
'array3[1,..,2]'	'[2,4]'
'array3 [2,..,..]'	'[[5,6], [7,8]]'
'array3[1..2,1,1]'	'[1,5]'
'array3 [2..3,2,..]'	'[[7,8], [11,12]]'

Table 2.3: Sample **Array Dereferencing** in ESSENCE′

2.2.3 Domains

There are two types of domains in ESSENCE′: the Boolean domain (**bool**) and integer domains. There exist both finite and infinite integer domains. Finite Integer domains are restricted by a lower and upper bound lb and ub, where lb and ub are integer expressions and $lb \leq ub$. The elements of a domain can be explicitly given (e.g. '**int** (1,2,3,4) '), specified by the range (e.g. '**int** (1..4) '), or defined by a combination of both (e.g. '**int** (1,2..4) ').

Infinite domains are only allowed when declaring a parameter's domain. They can be open within the lower or upper bound (e.g. '**int** (1..) ' or '**int** (..10) ' or simply specified as the set of integers with '**int**'.

2.2.4 Parameters and Constants

Parameters are problem features that define a particular problem instance. They can represent all kinds of expressions, i.e. basic expressions, arrays or domains. Parameters are declared in the problem class model using the '**given**' statement and then specified in a separate parameter file. After specifying the parameter's name, the parameter's type is declared after a colon symbol, as examples illustrate below:

```
$ parameter declaration examples $
given A : bool
given n : int(1..)
given array_M2 : matrix indexed by [int(1..3),int(1..n)] of int(1..12)
```

Constants are used to either specify parameter values or to label expressions/domains that often occur in a problem model. The '**letting**' statement allows us to assign a name to a constant value. For instance, the statement '**letting** c **be** constant' introduces a new reserved name 'c' that is associated with the constant expression *constant*. Every subsequent occurrence of identifier 'c' in the model is replaced by *constant*. Note that 'c' cannot be used in the model *before* it has been defined. Below are some examples to illustrate constant definitions:

```
$ parameter and constant definition examples $
letting A be false
letting n be 4
letting array2_M be [[1,4,5,10],[11,7,6,2],[8,12,9,3]]
letting length be n*n
letting Dom be domain int(1..n)
```

2.2.5 Decision Variables

Decision variables can be defined over either a finite integer domain or the Boolean domain. As with constants and parameters, finite variable arrays can be defined. A variable is

defined with the statement 'find', followed by a unique name and the underlying domain, which specifies its type (if a variable is defined over a Boolean domain, it is considered a Boolean expression, if defined over an integer domain, it is considered integer).

```
$ variable declaration examples$
find B : bool
find x : int(1..n)
find array1 : matrix indexed by [int(1..n)] of int(1..n)
```

2.2.6 Objective and Constraints

The objective statement is optional; if left out, the problem is a satisfaction problem (i.e. the solver will return the first valid solution). The objective options are 'maximising' and 'minimising', followed by an arbitrary integer expression. Only one objective can be stated. Below are two examples for illustration.

```
$ objective examples $
maximising x*y
minimising sum i : int(1..n) . array1_x[i]
```

Constraints are stated after the 'such that' statement and separated by a comma. Every constraint has to be of type 'Boolean expression'. A list of examples illustrates different kinds of constraints:

```
$ constraint examples $
such that
  x + y*z = 0,
  true,
  A => B,
  alldifferent([x,y,z]),
  n = sum i:int(1..n).  array1_x[i]
```

2.2.7 Global Constraints

Global constraints [67] are particular constraint patterns for which solvers provide strong propagation algorithms. The most popular global constraint is the *alldifferent* constraint [33, 63]. *alldifferent*$(x_1, x_2, \ldots, x_n)$ states that every variable x_i has to take a different value, which corresponds to a clique of disequalities between $x_1, \ldots, x_n$. In many cases, expressing such a clique *explicitly* (i.e. by a set of disequalities) results in a worse solving performance than expressing the clique by *alldifferent* (given the solver provides an *alldifferent* propagator, such as [62]). Therefore, solver-independent modelling languages need to provide global constraints.

So far, 313 global constraints have been defined in the global constraint catalogue [9]. The research on global constraints is very active, thus the number is increasing rather quickly.

There are efforts to decrease the number of global constraints by defining patterns that summarise kinds of global constraints [14]. Typically, solvers only support a small range of global constraints, which differs from solver to solver. In our implementation of ESSENCE′, only a range of the most popular global constraints is supported which is listed in Table 2.4.

Global Constraint	Description	Formally
`alldiff([x,y,z])`	different values are assigned to x,y and z	$x \neq y, x \neq z$ $y \neq z$
`atleast([x,y],[4,5],[1,2])`	4 occurs atleast $1\times$ in [x,y] 5 occurs atleast $2\times$ in [x,y]	
`atmost([x,y],[2,3],[3,1])`	2 occurs atmost $3\times$ in [x,y] 3 occurs atmost $1\times$ in [x,y]	
`element(array1_x,y,z)`	array1_x has value z at position y	$array\_x[y] = z$
`gcc([x,y,z],[6,7],[1,2])`	6 occurs exactly $1\times$ in [x,y,z] 7 occurs exactly $2\times$ in [x,y,z]	
`table([x,y], [[1,2],` `[1,3],` `[4,5]])`	allowed value tuples for [x,y]	$(x = 1) \wedge (y = 2) \vee$ $(x = 1) \wedge (y = 3) \vee$ $(x = 4) \wedge (y = 5)$

Table 2.4: **Global constraints** supported by the current implementation of ESSENCE′

2.2.8 Quantified Expressions $\forall$, $\exists$ and $\sum$

Quantifications are a powerful means to express a (variable) number of constraints in a compact way. In principle, $\forall$ corresponds to n-ary conjunction, $\exists$ to n-ary disjunction and $\sum$ to an n-ary addition. The general syntax is

quantifier quantifying-variables : *domain . expression*

where *quantifier* represents either ' **forall** ', ' **exists** ' or '**sum**', *quantifying-variables* is a list of identifiers that range over *domain* and whose scope is limited to the quantified expression, *expression*. ' **forall** ' and ' **exists** ' yield Boolean expressions; '**sum**' yields integer expressions. *domain* may only contain expressions that consist of constants, parameters or quantifying variables, i.e. may not contain decision variables. ' **forall** ' and ' **exists** ' can only quantify Boolean expressions, while '**sum**' is applicable to both Boolean and integer expressions. Quantifications can be arbitrarily nested, as some examples illustrate:

```
$ quantification examples $
  n != sum i:int(1..n).  x[i]

  forall i:int(1..n).   x[i] != y[i]

  exists k:int(1..m−1).
     x[k] − x[k+1] = k

  forall i:int(1..n) .
     exists j:int(i..n) .
         x[i] = j

  forall i:int(1..n) .   forall j:int(1..m) .
        n*m <= sum k:int(1..l).  x[i,k]*y[j,k]
```

2.3 Summary

In summary, this chapter has introduced the modelling language ESSENCE′ that is used throughout this thesis. So far, ESSENCE′ has only been defined *implicitly* through the abstract problem specification language ESSENCE [26] of which ESSENCE′ is a subset.

This chapter provides the following contributions. First, it gives an anecdotal description of ESSENCE′ of which no concrete definition has been previously published. Second, it demonstrates that ESSENCE′ is an appropriate choice of modelling language throughout this work, by highlighting similarities to other constraint solver-independent modelling languages.

CHAPTER 3

TAILORING PROBLEM INSTANCES

Tailoring is the compilation of solver-independent constraint problem models to solver input. In this chapter, we focus on tailoring problem *instances*, i.e. a constraint model where all parameter values are specified. A constraint instance is typically obtained by pairing a problem *class* with a parameter specification. For example, pairing the n-queens problem class [57] with $n = 8$, yields the 8-queens instance.

Tailoring is a necessary task in order to solve a problem that is formulated in a solver-independent modelling language: solver input is on a low level, consisting of a restricted set of types and granular constraint expressions. Furthermore, every solver has a different input format and supports different constraints and data types. Hence, tailoring is closely related to compiling high-level programming languages to machine code: complex expressions have to be flattened to low-level representation, data types refined to simpler types, and the problem models needs to be checked for syntactic and semantic correctness.

The general aim of tailoring is to produce valid solver input from the given problem instance. Furthermore, it is desirable to design a general, efficient tailoring process that can be easily extended with further input languages and/or target solvers. By applying a similar translation structure as in Compiler Construction [2], we can achieve this goal, as demonstrated in this chapter.

First, we give a general overview of the translation process structure, in particular in comparison to Compiler Construction and introduce TAILOR, a tailoring tool that has been implemented by the author during the work on this thesis and incorporates all translation techniques discussed in this work. Second, the three tailoring components are discussed: the frontend deals with input-related transformations, discussed in Sec. 3.2. The middle-end incorporates the most important transformations, that are parameterised by the target solver's features (Sec. 3.3). Finally, the backend performs solver-specific transformations that cannot be generalised in the middle-end (Sec. 3.4). In Sec. 3.5, tailoring is illustrated on an example, where an instance is tailored to three different target solvers.
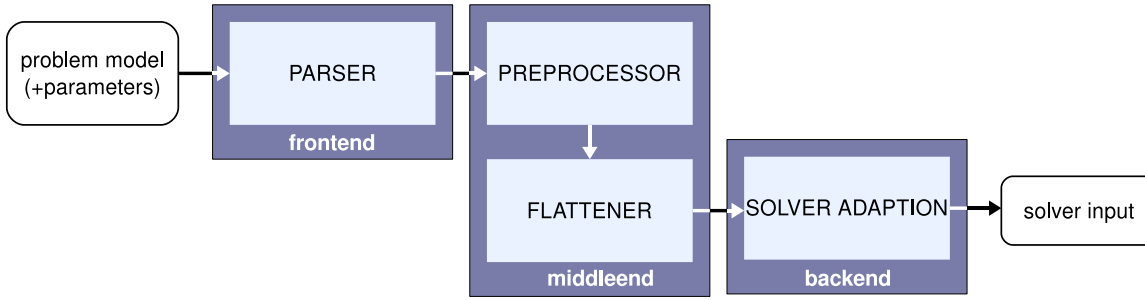
Figure 3.1: **Tailoring Overview**. Given a problem model (paired with a parameter specification), the tailoring engine produces solver input undergoing three major steps: preprocessing, flattening and mapping to solver syntax.

3.1 Tailoring in a Nutshell

A classical compiler consist of three consecutive parts: the compiler *frontend*, *middle-end* and *backend*. The frontend parses the input and generates an intermediate format. The middle-end performs general translations, in particular preprocessing and flattening. The backend applies target-specific operations, generating the final compiler output. This division into frontend, middle-end and backend is very practical, since several frontends and backends can be plugged before/after the middle-end. In other words, the middle-end can be reused for different translations and therefore makes the compiler easily extendable.

We employ a similar structure in tailoring, which is illustrated in Fig. 3.1. The frontend parses the input and performs particular input-language specific preprocessing, such as inserting parameter values or inserting constant labels. Furthermore, it constructs a symbol table that holds information about every declared identifier. This representation is then passed on to the middle-end.

The middle-end contains the two core translation processes: preprocessing and flattening. Though both are generic processes, they include many solver-dependent transformations, so the middle-end requires information about the target solver. How solver information is stored and obtained is explained in Sec. 3.3.3. The middle-end generates a flat, normalised format that is applied to the backend.

The backend performs solver-specific operations that cannot be generalised in the middle-end. These operations include propagator selection, heuristic selection and solver-syntax related issues. Finally, the backend outputs the corresponding solver format.

3.1.1 The tailoring tool TAILOR

During this thesis project, the tailoring tool TAILOR has been implemented by the author. TAILOR follows the tailoring description in this chapter and performs all automated en-
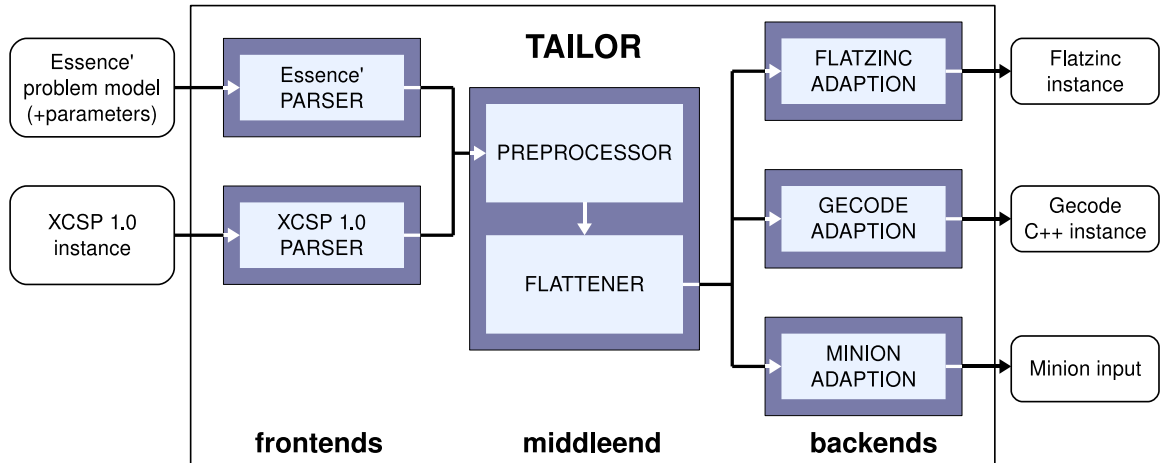
Figure 3.2: **Structure of tailoring tool** TAILOR. TAILOR takes two different inputs: ESSENCE′and XCSP 2.1 format. Three backends allow the translation to 3 different formats: FlatZinc, Gecode C++ and MINION input.

hancement techniques that are discussed in this thesis. TAILOR can process two different input formats, ESSENCE′(Chapter 2) and XML format XCSP 2.1 [68], and generates three different solver input formats: FlatZinc [58], solver Gecode's [80] C++ format and solver MINION's [32] text input format. The structure of TAILOR is shown in Fig. 3.2 and illustrates how several front- and backends can be efficiently plugged into a tailoring system (note, that TAILOR's middle-end (preprocessor, flattener and intermediate format) makes up about 70% of the overall code).

3.2   Frontend

The frontend processes the tailoring input and generates an intermediate format. The intermediate format is the internal expression representation of the translator and is typically a format to which any input can be easily transformed to. For instance, in case of TAILOR, the intermediate format is ESSENCE′. The intermediate format represents the list of constraints and all identifiers that have been declared in the problem model (and parameter file). The following translation steps yield the intermediate format:

1. Lexing and parsing
2. Expression tree construction
3. Symbol table construction
4. Parameter/constant/predicate insertion
5. Computation of subexpression domains

The first step is lexing and parsing the input, which assesses the input for syntactic errors by matching lexems to the specified input language grammar. The matched tokens are then

Production Rules	Semantic Rules
$E \rightarrow (E_1)$	$E.lb = E_1.lb$, $E.ub = E_1.ub$
$E \rightarrow R$	$E.lb = R.lb$, $E.ub = R.ub$
$E \rightarrow A$	$E.lb = A.lb$, $E.ub = A.ub$
$R \rightarrow$ **false**	$R.lb = 0$, $R.ub = 0$
$R \rightarrow$ **true**	$R.lb = 1$, $R.ub = 1$
$R \rightarrow \neg R_1$	$R.lb = 0$, $R.ub = 1$
$R \rightarrow E_1$ **rel** E_2	$R.lb = 0$, $R.ub = 1$
$R \rightarrow R_1$ **boolop** R_2	$R.lb = 0$, $R.ub = 1$
$R \rightarrow Q_{\forall,\exists}$	$R.lb = 0$, $R.ub = 1$
$R \rightarrow G$	$R.lb = 0$, $R.ub = 1$
$A \rightarrow Atom$	$A.lb = Atom.lb$, $A.ub = Atom.ub$
$A \rightarrow -A_1$	$A.lb = \min\{-A_1.lb, -A_1.ub\}$
	$A.ub = \max\{-A_1.lb, -A_1.ub\}$
$A \rightarrow |A_1|$	$A.lb = \min\{|A_1.lb|, |A_1.ub|\}$
	$A.ub = \max\{|A_1.lb|, |A_1.ub|\}$
$A \rightarrow A_1$ **mul** A_2	$A.lb = \min\{(A_1.lb$ **mul** $A_2.lb),$
	$(A_1.lb$ **mul** $A_2.ub),$
	$(A_1.ub$ **mul** $A_2.lb),$
	$(A_1.ub$ **mul** $A_2.ub)\}$
	$A.ub = \max\{(A_1.lb$ **mul** $A_2.lb),$
	$(A_1.lb$ **mul** $A_2.ub),$
	$(A_1.ub$ **mul** $A_2.lb),$
	$(A_1.ub$ **mul** $A_2.ub)\}$
$A \rightarrow Q_{\sum}$	$A.lb = Q_{sum}.lb$, $A.ub = Q_{sum}.ub$
$A \rightarrow$ **min**(A_1, A_2)	$A.lb = \min\{A_1.lb, A_2.lb\},$
	$A.ub = \max\{A_1.ub, A_2.ub\}$
$A \rightarrow$ **max**(A_1, A_2)	$A.lb = \min\{A_1.lb, A_2.lb\},$
	$A.ub = \max\{A_1.ub, A_2.ub\}$
$Q_{\sum} \rightarrow$ **sum** idList : B_D . A	$Q_{\sum}.lb = A.lb *$ idList.*length*
	$Q_{\sum}.ub = (B_D.ub - B_D.lb + 1)$
	$* A.ub *$ idList.*length*
$Atom \rightarrow$ **num**	$Atom.lb =$ **num**, $Atom.ub =$ **num**
$Atom \rightarrow$ **ident**	$Atom.lb =$ symbolTable(**ident**).lb
	$Atom.ub =$ symbolTable(**ident**).ub
$Atom \rightarrow$ **ident** [$\{I$list$\}$']	$Atom.lb =$ symbolTable(**ident**).lb
	$Atom.ub =$ symbolTable(**ident**).ub

Figure 3.3: Syntax-directed definition that defines the computation of the synthesised attribute *domain* for each node in the expression tree. Attributes lb and ub represent lower and upper bound of *domain*. Terminals are boldfaced; keywords are underlined. Single, capital letters represent non-terminals: E (expression), R (Boolean expr.), A (integer expr.), Q (quantification), G (global constraint).

transformed into a syntax tree that represents the parsed expressions.

Second, constraint expressions are extracted from the syntax tree, each represented as a single *expression tree*, where leaves are *atoms* (i.e. constants or identifiers), and nodes represent *operators*, such as $+$, $\wedge$ or *alldifferent*. Every constraint expression is represented by a single tree. Typically, expression trees are represented in a compact way, i.e. all successive n-ary operators of the same type (that are commutative and associative) are merged into nodes of n children. For instance, the subtree $+(a, +(b, c))$ is restructured into the simpler tree $+(a, b, c)$. Note, that throughout this thesis, constraint expressions will be mainly considered as expression trees.

Third, the *symbol table* is generated: from the declaration part of the parsed syntax tree all information about declared identifiers and their type, domain and category is collected in the symbol table. The symbol table acts as main reference for measures like type and category checking during the translation.

The next step is parameter/constant/predicate insertion. Many solver-independent modelling languages allow the user to specify parameters in a separate file or to define labels for particular expressions, such as constant labels or predicates. These are inserted into the problem model to yield an intermediate instance.

Finally, the domain of each subexpression is computed. This is achieved by using a synthesised attribute [2], *domain*, for every tree element e, representing the lower and upper bound of e. *domain* is computed bottom-up from the leaves for every tree. Since atoms must be defined over a finite domain, every leaf's domain is finite, thus we can compute a finite domain for every node in the expression tree. Note, that Boolean nodes are represented by integer values, i.e. *false* is represented by '0' and *true* by '1'. The semantic rules for obtaining the domain of each subexpression is summarised in the syntax-directed definition in Fig. 3.3. Note, that all these steps are standard in program compilation and are further described in [2]. Finally, the frontend returns the intermediate representation which consists of the problem instance and the symbol table, which are input to the next layer, the middle-end.

For illustration, we will consider one of TAILOR's frontends, the XCSP 2.1 frontend. As a sample input, consider the 4-queens instance in XML format XCSP 2.1 in Fig. 3.4(top) which has been taken from the XCSP Benchmark Website [50]. This model defines 4 variables, V0, V1, V2 and V3, ranging over the domain (1..4). Constraints are expressed by referring to a predicate. The frontend parses the XML file and generates the intermediate format in ESSENCE′that is illustrated in Fig. 3.4 (bottom), where all predicates and domains have been inserted into the representation.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<instance> <presentation name="4-Queens" maxConstraintArity="2" format="XCSP_2.0"/>
<domains nbDomains="1"> <domain name="D0" nbValues="4">1..4</domain>
</domains>

<variables nbVariables="4">
  <variable name="V0" domain="D0"/> <variable name="V1" domain="D0"/>
  <variable name="V2" domain="D0"/> <variable name="V3" domain="D0"/>
</variables>

<predicates nbPredicates="1">
  <predicate name="P0"> <parameters>int X0 int X1 int X2 int X3 int X4</parameters>
     <expression> <functional>and(ne(X0,X1),ne(abs(sub(X2,X3)),X4))</functional>
     </expression>
  </predicate>
</predicates>

<constraints nbConstraints="6">
  <constraint name="C0" arity="2" scope="V0_V1" reference="P0">
     <parameters>V0 V1 V0 V1 1</parameters>
  </constraint>
  <constraint name="C1" arity="2" scope="V0_V2" reference="P0">
     <parameters>V0 V2 V0 V2 2</parameters>
  </constraint>
  <constraint name="C2" arity="2" scope="V0_V3" reference="P0">
     <parameters>V0 V3 V0 V3 3</parameters>
  </constraint>
  <constraint name="C3" arity="2" scope="V1_V2" reference="P0">
     <parameters>V1 V2 V1 V2 1</parameters>
  </constraint>
  <constraint name="C4" arity="2" scope="V1_V3" reference="P0">
     <parameters>V1 V3 V1 V3 2</parameters>
  </constraint>
  <constraint name="C5" arity="2" scope="V2_V3" reference="P0">
     <parameters>V2 V3 V2 V3 1</parameters>
  </constraint>
</constraints>
</instance>
```

```
find    V0, V1, V2, V3 : int(1..4)
such that       (1!=(|V0-V1|)) /\ (V0!=V1),    (2!=(|V0-V2|)) /\ (V0!=V2),
                (3!=(|V0-V3|)) /\ (V0!=V3),    (1!=(|V1-V2|)) /\ (V1!=V2),
                (2!=(|V1-V3|)) /\ (V1!=V3),    (1!=(|V2-V3|)) /\ (V2!=V3)
```

Figure 3.4: **4-queens** instance in **XCSP 2.1** format (top) and ESSENCE′**intermediate format** (bottom) generated by the XCSP 2.1 frontend in TAILOR.

3.3 Middle-End

The middle-end contains all transformations that can be generalised for *every* input. The main processes are preprocessing (Sec. 3.3.1) and flattening (Sec. 3.3.2). However, many transformations depend on the target solver. Therefore, both preprocessing and flattening are parameterised by target solver features. How solver features are represented and used to trigger particular transformations is described in Sec. 3.3.3.

3.3.1 Preprocessing

Preprocessing summarises all model transformations that are performed before the problem model is flattened to a low-level representation. There are two main processing steps: model *normalisation* and *type adaptions*. Normalisation simplifies the problem instance and reduces equivalent but syntactically different representations to one unique representation. Type adaptions are transformations of types (e.g. data structures) to conform to the target solver. This includes for instance flattening of multi-dimensional arrays to 1-dimensional arrays. These adaptions typical concern the whole problem instance and are therefore easiest performed during preprocessing, when the instance is still in a compact format: quantifications are not unrolled, so the model contains 'less' expressions.

Normalisation

The grammar of constraint expressions contains many equivalent representations, for instance, $x + y + z$ and $y + x + z$ where the equivalence stems from the commutativity of addition. A normal form without such equivalences provides many benefits. For instance, processing methods, like flattening, need only be implemented for one normalised subterm and not for every other equivalent representation. Further benefits will be explained in Sec. 4.3, where instance optimisations are discussed. The core normalisation steps are evaluation and ordering of expressions, which are described in more detail below.

Expression Ordering   We define an order $\leq_g$ over the expressions in ESSENCE′and transform every expression tree into a minimal form with respect to this order. The general ordering rules are:

- *Numbers* are ordered according to their value
- *Identifiers* are ordered lexicographically
- Expressions separated by a comma are ordered according to the number of expression, i.e.

  $\quad\quad$ *expression*
  $<_g\quad$ *expression, expression*
  $<_g\quad$ *expression, expression, expression*
  $<_g\quad \cdots$

  For instance, $m[c_1, c_2] <_g m[c_1, c_2, c_3]$.
- We impose an order on terms composed by commutative operators $op_c \in \{\ +,\ *,$ $=,\ \neq,\ \Leftrightarrow,\ \vee,\ \wedge\ \}$: a term *expression*1 op_c *expression*2 is ordered, if and only if *expression*1 $\leq_g$ *expression*2.

We summarise the expression order in Fig. 3.5. The topmost expression is weakest; if a non-terminal has several productions, the first production is the weakest. The order of operations is derived from the operator precedence.

Non-terminals (expressions)		Ordering of possible productions
Atom		**<u>false</u>**
	$<_g$	**<u>true</u>**
	$<_g$	**num**
	$<_g$	**ident**
	$<_g$	**ident [** *ArithmExpr* **]**
	$<_g$	**ident [** *ArithmExpr, ArithmExpr* **]**
	$<_g$	**ident [** *ArithmExpr, ArithmExpr, ArithmExpr* **]**
	$<_g$	. . .
Unary Operators		*- ArithmExpr*
	$<_g$	*! RelExpr*
	$<_g$	*| ArithmExpr |*
Binary Operators		*Expression* **=** *Expression*
	$<_g$	*Expression* **!=** *Expression*
	$<_g$	*Expression < Expression*
	$<_g$	*Expression <= Expression*
	$<_g$	*Expression > Expression*
	$<_g$	*Expression >= Expression*
	$<_g$	*Expression* **+** *Expression*
	$<_g$	*Expression* **-** *Expression*
	$<_g$	*Expression* ***** *Expression*
	$<_g$	*ArithmExpr* **/** *ArithmExpr*
	$<_g$	*ArithmExpr* **%** *ArithmExpr*
	$<_g$	*ArithmExpr ^ ArithmExpr*
	$<_g$	*RelExpr <=> RelExpr*
	$<_g$	*RelExpr => RelExpr*
	$<_g$	*RelExpr \\/ RelExpr*
	$<_g$	*RelExpr /\\ RelExpr*
Global Constraints		**<u>alldifferent</u>** (*ArrayExpr*)
	$<_g$	**<u>table</u>** (*ArrayExpr, ConstArray*)
	$<_g$	**<u>element</u>** (*ArrayExpr, Expression, Expression*)
	$<_g$	**<u>atmost</u>** (*ArrayExpr, ArrayExpr, ConstArray*)
	$<_g$	**<u>atleast</u>** (*ArrayExpr, ArrayExpr, ConstArray*)
	$<_g$	**<u>gcc</u>** (*ArrayExpr, ConstArray, ArrayExpr*)
Quantified Expressions		**<u>sum</u>** *Bindingexpression . ArithmExpr*
	$<_g$	**<u>forall</u>** *Bindingexpression . RelExpr*
	$<_g$	**<u>exists</u>** *Bindingexpression . RelExpr*
Binding Expression		**ident** : *SimpleDomain*
	$<_g$	**ident**, **ident** : *SimpleDomain*
	$<_g$	**ident**, **ident**, **ident** : *SimpleDomain*
	$<_g$	. . .

Figure 3.5: **Expression Ordering** in ESSENCE$'$

Expression Evaluation is important to simplify expressions involving constants and is performed only to a certain extent to minimise the computational effort. Note, that ordering expressions simplifies evaluation: constant expressions are listed before decision variables, e.g. $2 + x + 5$ is ordered to $2 + 5 + x$, hence evaluation rules can be applied from left.

We apply constant evaluation (e.g. '$2 + 5$' is evaluated to '7'), simple logical evaluation, as well as several simple algebraic transformations, such as algebraic identity or inverses.

The table below summarises all transformations, excluding constant evaluation, which is straightforward. Note, that even though some expressions, like identity expressions, seem quite uncommon, they occur rather often at instance level, when parameters are replaced by constant values.

Algebraic Identities			Algebraic Inverses			Other Simplifications		
$E + 0$	$\longrightarrow$	E	$E + \text{-}A$	$\longrightarrow$	0	$E \star 0$	$\longrightarrow$	0
$E - 0$	$\longrightarrow$	E	$E - E$	$\longrightarrow$	0	$A \text{ ^ } 0$	$\longrightarrow$	1
$E \star 1$	$\longrightarrow$	E	$A \star (1/A)$	$\longrightarrow$	1	$B \wedge \text{false}$	$\longrightarrow$	false
$A \,/\, 1$	$\longrightarrow$	A	$A \,/\, A$	$\longrightarrow$	1	$B \vee \text{true}$	$\longrightarrow$	true
$A \text{ ^ } 1$	$\longrightarrow$	A				$B \Leftrightarrow \text{true}$	$\longrightarrow$	B
$B \wedge \text{true}$	$\longrightarrow$	B				$B \Leftrightarrow \text{false}$	$\longrightarrow$	$\neg B$
$B \vee \text{false}$	$\longrightarrow$	B		$B \Rightarrow false$		$\longrightarrow$		$\neg B$
						$B \Rightarrow true$	$\longrightarrow$	$true$
						$true \Rightarrow B$	$\longrightarrow$	$true$
						$false \Rightarrow B$	$\longrightarrow$	B

Figure 3.6: Summary of **evaluation transformations** during Normalisation. E represents arbitrary expressions, A integer expressions and B Boolean expressions

Other Normalisations include the transformation to negation normal form, i.e. every expression is transformed such that negation is only applied to atomic expressions. For instance, $\neg(A \vee B)$ is transformed to $\neg A \wedge \neg B$ by applying de Morgan's Law. Furthermore, we unify inequality operators. For instance, expressions of the form $A \geq B$ are transformed to $B \leq A$.

Type Adaptions

Constraint solvers support different kinds of types, in particular different types of arrays or domains. Therefore, the types used in a problem instance must be adapted to the solver's repertory. For instance, some solvers only support 1-dimensional arrays. Hence, if a model is tailored to such a solver, all multi-dimensional arrays have to be flattened to 1-dimensional arrays. Type adaptions typically involve the whole constraint model, therefore it is best performed during preprocessing, when the quantifications are not unrolled and thus the expressions are represented in a compact way (i.e. less expression trees need to be transformed).

Adapting Arrays Arrays are data structures to contain variables of the same type (and typically from the same category) into one structure. Typically, solver-independent modelling languages support multi-dimensional arrays that can be arbitrarily indexed. However, most solvers provide limited support for arrays, which needs to be taken into account during tailoring.

The first limitation concerns *dereferencing of arrays*. In most solvers, arrays are indexed starting from a fixed constant. For instance, in solver MINION, arrays are indexed starting from '0'; in FlatZinc format arrays are indexed from '1'. However, in ESSENCE', the user can specify the value from which an array is indexed. Therefore, during tailoring, the index domain has to be adapted. For instance, if the index domain of array M is defined as $int(1..10)$, but the target solver initialises arrays with '0', the index domain is transformed into $int(0..9)$. Changing the index domain also requires changing the dereferences of array expressions. For instance, the array dereference in the constraint

```
    find M : matrix indexed by [int(1..10)] of int(1..20)

such that
    forall i:int(1..10).
          M[i] = i
```

has to be rewritten to

```
    find M : matrix indexed by [int(0..9)] of int(1..20)

such that
    forall i:int(1..10).
          M[i-1] = i
```

Note that only the dereferencing expression ('M[i−1]') can be adapted and not the quantifying domain ('**int** (1..10) '). Changing the quantifying domain may have serious effects on dereference expressions of other arrays in the quantified expression. As an example, consider a similar constraint below, that involves array M together with another array N whose indices range from $int(0..20)$:

```
    find M : matrix indexed by [int(1..10)] of int(1..20)
    find N : matrix indexed by [int(0..20)] of int(1..10)

such that
    forall i:int(1..10).
          M[i] = N[i]
```

Adapting the quantifying domain, '**int** (1..10) ', instead of the dereferenced expression of M, 'M[i]', yields the following constraint that does not preserve the semantics of the initial constraint:

```
    find M : matrix indexed by [int(1..10)] of int(1..20)
    find N : matrix indexed by [int(0..20)] of int(1..10)

such that
    forall i:int(0..9).
          M[i] = N[i]
```

Note that array index adaptions not only involve transforming array dereferences, but can also involve argument changes in global constraints, such as the element constraint,

element(M,x,y), denoting $M[x] = y$. In such a case, the constraint has to be transformed to *element*(M,$x{-}1$,y).

The second limitation is the *array dimension*, for instances, many solvers only support 1-dimensional arrays. Array dimensions can be adapted in the following way: whenever the solver profile indicates that the target solver provides limited support, e.g. only supports 1-dimensional arrays, then every multi-dimensional array M is flattened in two steps: first, all constraint expressions are searched for references of M, which are typically of the form $M[c_1, ..., c_n]$. Those dereferencing expressions are transformed to dereference a 1-dimensional array:

$$M[c_1, c_2, \ldots, c_{n-1}, c_n] \quad \longrightarrow \quad M[c_n + \sum_i^{n-1..1} c_i * \prod_j^{i..n-1}(ub_j - lb_j + 1)]$$

where lb_i and ub_i are the lower and upper bound of the ith index domain of array M. Consequently, the entry of M in the symbol table is changed to an 1-dimensional array M:

M: matrix indexed by   [int($lb_1..ub_1$), int($lb_2..ub_2$), ..., int($lb_n..ub_n$)] $\longrightarrow$

M: matrix indexed by   [int(lb_{solver} .. $(\prod_i^{1..n} ub_i - lb_i + 1) - (1 - lb_{solver})$)]

Array Adaption: Example   As an example, we consider the array adaption steps when tailoring a simple instance(below) to FlatZinc format. In FlatZinc, index domains start with '1' and only 1-dimensional arrays are supported. Hence, the 3-dimensional array M requires adaption of its index domains, as well as of its dimension.

```
$ unadapted constraint instance with 3-dimensional array $
find M: matrix indexed by [int(1..5),int(0..9),int(2..4)] of int(1..10)
such that
  forall i:int(1..5) . forall j:int(0..9).
      10 <= sum k:int(2..4) .  M[i,j,k]
```

First, index domains are adapted: the second and third index domain, '**int** (0..9 ' and '**int** (2..4) ' are adapted as follows:

```
$ STEP1: adapted array dereferences to starting index '1' $
find M: matrix indexed by [int(1..5),int(1..10),int(1..3)] of int(1..10)
such that
  forall i:int(1..5) . forall j:int(0..9).
      10 <= sum k:int(2..4) .  M[i,j+1,k-1]
```

Second, the 3-dimensional array has to be flattened to a 1-dimensional array, which yields:

```
$ STEP2: flattened multi-dimensional array to 1-dimensional array $
find M : matrix indexed by [1..150] of int(1..10)
such that
  forall i:int(1..5) . forall j:int(0..9).
      10 <= sum k:int(2..4) .  M[i*50 + (j+1)*5 + (k-1)]
```

Adapting Domains The basic domain types are integer and Boolean, however, solvers distinguish between integer domains that represent a range of integers (e.g. int(1..5)) and domains that represent a range with holes (e.g. int(1,3,5)). We will refer to the former as *bound domain* and to the latter as *sparse domain*. Practically every constraint solver supports bound domains, but some have no support for sparse domains. In that case, sparse domains have to be converted into bound domains with additional disequality constraints that explicitly set the holes in the domain.

Algorithm 3.1 FLATTEN_INSTANCE (M_S) flattens 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:    $E_0' \leftarrow$ FLATTEN (E *false*)
5:    $E_S' \leftarrow E_0' \wedge (\bigwedge_i E_i' \in constraintBuffer)$
6:    *flatConstraints*.add(E_S')
7: $M_S'.constraints \leftarrow flatConstraints$
8: $M_S'.vars \leftarrow \{M_S.vars \cup auxVars\}$
9: **return** M_S'

3.3.2 Flattening

In many cases, a target solver does not directly support expressions as they are formulated in a rich, solver-independent modelling language: in solvers, constraints are implemented as *propagators* which are 'granular' constraints that take only variables as arguments i.e. their expression tree depth is 1 (exceptions are solvers such as ECLiPSe Prolog, that flatten their input internally). Thus, many 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*.

Definition 3.3.1. 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 ex-

---
Algorithm 3.2 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:        $e_i' \leftarrow$ 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

---

pression 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). 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. If all constraints in a model M are adapted to solver S in this way, we will refer to it as M_S.

Flattening is a well-understood technique, however, we summarise a standard flattening algorithm, FLATTEN_INSTANCE , in Alg. 3.1 that applies a recursive helper procedure, FLATTEN (Alg. 3.2), 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 7-9 in Alg. 3.2). After flattening every subnode N_i of expression E, the '$Aux_i=N'_i$'-constraints are conjoined with the flat root node (line 5 in Alg. 3.1), yielding the flat representation of E. As an example, consider Figure 3.7 that illustrates how FLATTEN decomposes the expression $a \Rightarrow ((x{<}3){\wedge}(y{>}5){\wedge}(z{=}0))$.

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. 3.2). lb and ub are obtained from the domain attribute of node E that contains the lower and upper bound of the subtree E (see Sec. 3.2 or Fig. 3.3). In this way, we can assign tight bounds to the auxiliary variables.

After generating the flat representation E'_S for every constraint E in instance M_S, FLATTEN_INSTANCE constructs the flat instance M'_S, consisting of the flat constraints and M_s's decision variables combined with the auxiliary variables (line 7) in Alg. 3.1). Clearly, FLATTEN_INSTANCE 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 3.3.1. *If constraint instance M_S contains n constraints that contain m nodes in their expression trees (with $m \geq n$), then* FLATTEN_INSTANCE *will generate flat instance M'_S with m constraints and $m - n$ auxiliary variables.*

Proof. FLATTEN_INSTANCE applies FLATTEN to every constraint E in M_S (line 4 in Alg. 3.1). FLATTEN is recursively invoked on every subnode that is not a leaf (line 4 in Alg. 3.2), and creates an auxiliary variable and constraint for every node, except the root node
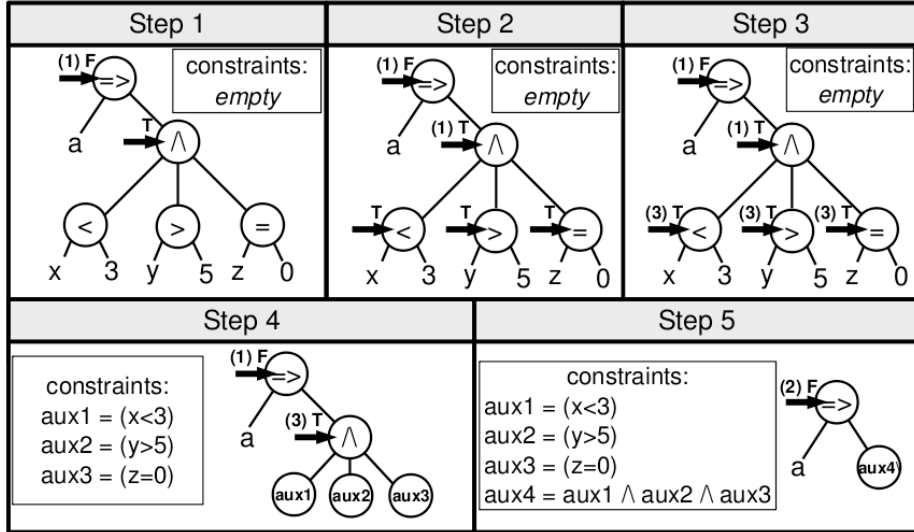
Figure 3.7: **Example: Flattening** $a \rightarrow ((x{<}3){\wedge}(y{>}5){\wedge}(z{=}0))$ using FLATTEN (Alg. 3.2). Arrows pointing at nodes denote that FLATTEN has been invoked on the subtree. **T** and **F** represent the Boolean value for *flatten2Aux*. The numbers represent parts of Algorithm 3.2: **(1)** invoke FLATTEN on non-leaf children (line 4-5), **(2)** return expression (line 10-11), **(3)** flatten to auxiliary variable (line 7-9).

(*flatten2Aux=false* only for the root node, line 4 in Alg. 3.1). 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 11 in Alg. 3.2). Therefore, FLATTEN generates $m - n$ auxiliary variables and m constraints. $\square$ $\square$

As an example, consider the constraint instance consisting of the constraint from Fig. 3.7: $a{\Rightarrow}((x{<}3){\wedge}(y{>}5){\wedge}(z{=}0))$. 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.

Theorem 3.3.1. *The* **time complexity** *of* FLATTEN_INSTANCE *lies in* $O(n)$.

Proof. Let f be the number of atomic operations required to flatten a node of an expression (note that f is a constant that is the same for every subexpression but which might differ between machines). From Lemma 3.3.1 we know that FLATTEN_INSTANCE performs these operations n times, since it is applied to every node in M_S. Therefore, FLATTEN_INSTANCE has a runtime of $f * n$ which lies in $O(n)$, since f is a constant (recall that we are assuming atomic operations in the implementation). $\square$

Theorem 3.3.2. *The* **space complexity** *of* FLATTEN_INSTANCE *lies in* $O(n)$.

Proof. FLATTEN_INSTANCE employs the lists *flatCts*, *auxVars* and *ctBuffer*, as well as the representation of constraint model M_S and M'. All of these data structures require a maximal capacity of n, where n is the number of subexpressions in M_S (Lemma 3.3.1). Therefore the space complexity of FLATTEN_INSTANCE lies in $O(n)$. $\square$

3.3.3 Representing Solver Features

Many constraint solvers exist, but each differs in provided variable types, propagators, data structures and search heuristics.

Flattening constraint expressions to propagators in solver format is a particularly difficult issue, since many expressions are flattened differently for each solver. As an example, consider flattening the nonlinear constraint '$a+b+c \neq e*f$' to three different solvers: ECLiPSe Prolog [87], Gecode [80] and MINION [32]. The appropriate constraint representation for each solver is given in the table below:

Target Solver	ECLiPSe Prolog	Gecode	MINION
Propagators	$a + b + c \neq e * f$	$aux_1 = e * f$	$aux_1 = e * f$
		$a + b + c \neq aux_1$	$a + b + c \leq aux_2$
			$a + b + c \geq aux_2$
			$aux_1 \neq aux_2$

ECLiPSe takes arbitrarily complex expressions (since it performs flattening internally), hence no flattening is required. Gecode provides a linear disequality constraint, allowing variables as arguments only, hence we flatten $e * f$ by introducing auxiliary variable aux_1, and post $a + b + c \neq aux_1$. MINION only supports binary disequality, hence we introduce another auxiliary variable, aux_2, representing $a + b + c$.

Note, that the flat representation of MINION would also be valid for solvers Gecode and ECLiPSe Prolog, but would contain additional variables, aux_1 and aux_2, respectively. Such a representation can result in worse propagation/runtime than a representation that is exactly tailored to the solver's repertory. Therefore, the flattening engine should decompose expressions *only* if the expression is not directly supported by the solver. In order to do this, the flattening engine requires information about the solver's constraint repertory - information a *solver profile* can provide.

Solver Profiles

We propose the notion of a *solver profile*, similar to rule-based systems in retargetable compilers [25], that captures important features of a particular solver. Those features include

- variable information (variable and domain types, available data structures),
- propagator information (constraint type, consistency level, arity, reifiability, etc.),
- provided search heuristics
- and other, solver-specific features

Given a general list of features, every solver profile associates a Boolean value to each feature that indicates if the feature is supported or not. For instance, if solver S provides

one n-ary conjunction propagator that is not reifiable, then the the feature *n-ary conjunction* will be set to true, but feature *reifiable n-ary conjunction* will be set to false. Solver profiles can also include solver-specific features, e.g. variable labelling.

Solver profiles can be used to customise preprocessing and flattening by setting choice-points in the procedure that depend on the information retrieved from the solver profile. For illustration, we will consider in detail how flattening can be customised for the target solver by using the solver profile's propagator and variable information.

Solver Profile-driven Flattening

The flattening engine works recursively, i.e. when given an expression, the flattening procedure is again invoked on the expression's arguments. A solver profile can guide the flattening engine: when given an expression, e.g. an n-ary multiplication, the flattening engine consults the solver profile about the availability of the corresponding propagator. If no applicable propagator exists, flattening proceeds (e.g. the n-ary multiplication is flattened into a binary multiplication). In this manner, an expression is only flattened, if the target solver does not support it.

This approach provides three key benefits: first, it assists in reducing the overhead when flattening expressions, since expressions are only flattened if necessary for the target solver. Second, a general flattening engine can be used for different solvers. Third, the flexibility of the solver profile allows to easily adapt to changes in the target solver (e.g. a new constraint is supported by simply changing the settings in the solver profile). Solver profiles can also assist in other parts of the tailoring process, such as selecting an appropriate propagator or search heuristic (in case favoured propagators or search heuristics are not already defined in the modelling language, as possible in MiniZinc).

A notable limitation of solver profiles is that they cannot provide alternatives if a particular constraint is not supported. For instance, consider the case when the flattening engine encounters the global constraint 'atmost' and learns from the solver profile that it is not supported. In this case, the solver profile currently cannot provide assistance as how to reformulate 'atmost' to be supported by the target solver. This can be resolved by either extending the tailoring engine with additional reasoning or extend the modelling language to support *predicates* with which one can define alternative representations (as in MiniZinc [58]).

3.4 Backend

The backend performs solver-specific transformations that cannot be generalised in the middle-end. These transformations include propagator selection, search heuristic selection and solver-specific issues.

Propagator Selection   Some solvers provide several propagators for the same constraint expression. As an example, solver MINION provides two variants of the 'alldifferent' constraint: a standard, arc consistent version, `alldiff`, and a general arc consistent version, `gacalldiff`. The former version is computationally cheap but will not prune many values from the variables' domains. The latter version can be computationally expensive but can prune more values than the first variant. Note that the modeller cannot specify which variant to choose in his/her model, so the backend has to decide which version to apply. In the case of 'alldifferent' there is a fairly good heuristic: if the number of arguments corresponds to the domain size of the arguments, then `gacalldiff` is preferable, otherwise it is likely to get too expensive, hence `alldiff` should be chosen.

However, typically, the choice is difficult, even for expert modellers, and strongly depends on the problem instance and other factors, such as the underlying variable types. The analysis of when which propagator is most effective can therefore seldom be generalised. Hence, in TAILOR, the standard propagators are chosen, unless a special known case occurs, in which another propagator is known to perform better.

Search Heuristic Selection   Similarly, the best search heuristic for a given instance is very difficult to choose and this choice is not an objective in our work. Therefore, the default search heuristic is chosen in the solver backend.

3.5   A Tailoring Example

The tailoring steps are best illustrated by an example: the Graph Colouring Problem (GCP) from Sec. 2.1. Starting from a problem model and a parameter specification (Fig. 3.8), we consider the steps to generate solver input. In particular, we consider the instance at three different levels: after preprocessing, after flattening, and the final format.

We tailor the instance to three different formats: input for solver MINION [32], FlatZinc [58] format, and C++ for solver Gecode [80], as it is performed by TAILOR. Note, that FlatZinc is a low-level language that 'differs' for every target solver. In this example, we generate FlatZinc that conforms to target solver MINION.

1. Preprocessing

The first major tailoring step is preprocessing. Fig. 3.9 shows the problem instance from Fig. 3.8 as it is preprocessed for solver MINION. Preprocessing performs three main steps: parameter insertion, data structure adaption and normalisation.

The first step, involves inserting parameter values into the problem model and eliminating all constant labels. Therefore, in Fig. 3.9, all constant and parameter values are replaced

by the corresponding integer value/domain. Second, data structures are adapted to the solver, which includes the adaption of array dereferences. For example, the array dereferences in expressions like 'usedColours[c]' are transformed to 'usedColours[c−1]' in Fig. 3.9, since MINION starts indexing arrays from '0' and the index domains in the problem model start with '1'. The third step is normalisation, which includes steps like ordering and evaluation. For illustration, in Fig. 3.9, expressions such as 'edges[a,b]=1' are ordered to '1=edges[a,b]'.

2. Flattening

After preprocessing, the instance is flattened, i.e. quantifications are unrolled and expressions are simplified by introducing auxiliary variables. The flat instance for the GCP when targeting solver MINION, is given in Fig. 3.10.

The objective is flattened to a sum constraint (1); the first constraint is flattened to six disequality constraints (2); and the second constraint is flattened to a list of implication constraints, introducing auxiliary variables (3).

```
given    vertices   : int(1..)
given    colours    : int(1..)
letting VERTICES    be domain int(1..vertices)
letting COLOURS     be domain int(1..colours)
given    edges      : matrix indexed by [VERTICES,VERTICES] of int(0,1)

find     colouring  : matrix indexed by [VERTICES] of COLOURS
find     usedColours : matrix indexed by [COLOURS] of int(0..1)

minimising sum c : COLOURS . usedColours[c]

such that
   forall a,b : VERTICES .
       (edges[a,b] = 1) => (colouring[a] != colouring[b]),

   forall v : VERTICES .  forall c : COLOURS .
           (colouring[v] = c) => (usedColours[c] = 1)
```

```
letting vertices be 5
letting colours  be 3
letting edges    be [[0, 1, 0, 1, 0],
                     [1, 0, 1, 0, 1],
                     [0, 1, 0, 1, 1],
                     [1, 0, 1, 0, 0],
                     [0, 1, 1, 0, 0]]
```

Figure 3.8: ESSENCE′**problem model** and **parameter** specification of Graph Colouring Problem(GCP) instance from Fig. 2.1

```
letting edges be [[0,1,0,1,0],
                  [1,0,1,0,1],
                  [0,1,0,1,1],
                  [1,0,1,0,0],
                  [0,1,1,0,0]]

find    colouring   : matrix indexed by [int(0..4)] of int(1..3)
find    usedColours : matrix indexed by [int(0..2)] of int(0..1)

minimising sum c:int(1..3). usedColours[c-1]

such that
   forall a,b: int(1..5).
       (1 = edges[a,b]) => (colouring[a-1] != colouring[b-1]),

   forall v:int(1..5).   forall c:int(1..3).
       (c = colouring[v-1]) => (1 = usedColours[c-1])
```

Figure 3.9: ESSENCE′GCP problem instance **after preprocessing** for MINION

3. Final Solver Formats

Finally, the flat problem instance is mapped to solver format. We show three different solver formats that were all generated automatically by our tailoring tool TAILOR. Fig. 3.11 shows the MINION input file, Fig. 3.12 the FlatZinc representation and Fig. 3.13 the C++ file for solver Gecode.

MINION **Instance**   The instance for MINION consists of three main parts: the variable declaration, the search specification and the constraint specification, where each part is initialised with double stars following the corresponding keyword, e.g. **VARIABLES****. The comments are (optionally) generated by TAILOR and denote what expression each auxiliary variable represents from the original model. The variables are ordered according to their declaration order in the original model. Auxiliary variables are last in the search order.

FlatZinc Instance   The FlatZinc instance is tailored for solver MINION, i.e. it contains those constraints that are supported by solver MINION. The format is very similar to MINION's input format, also consisting of three main parts: the variable declarations, followed by the constraints and a search declaration at the end. Note, that there are no particular search specifications, such as variable ordering, which is optional in FlatZinc.

Gecode C++ File   The C++ class for library-based solver Gecode is the most complicated format. Gecode is a C++ library, so problems are formulated as C++ classes. Note, that the translation to Gecode is mainly inspired by the examples provided by the Gecode distribution [80]. All comments are automatically added by TAILOR.

```
find colouring: matrix indexed by [int(0..4)] of int(1..3)
find usedColours: matrix indexed by [int(0..2)] of int(0..1)

$ auxiliary variables
find aux0 : int(0..3)    find aux1 : bool    find aux2 : bool    find aux3 : bool
find aux4 : bool         find aux5 : bool    find aux6 : bool     find aux7 : bool
find aux8 : bool         find aux9 : bool    find aux10 : bool    find aux11 : bool
find aux12 : bool        find aux13 : bool   find aux14 : bool    find aux15 : bool
find aux16 : bool        find aux17 : bool   find aux18 : bool


minimising aux0
such that
        $ (1) flat objective
        usedColours[0] + usedColours[1] + usedColours[2]=aux0,

        $ (2) first constraint flattened
        colouring[0]!=colouring[1],      colouring[0]!=colouring[3],
        colouring[1]!=colouring[2],      colouring[1]!=colouring[4],
        colouring[2]!=colouring[3],      colouring[2]!=colouring[4],

        $ (3) second constraint flattened
        aux1 <=> (1=colouring[0]),   aux2 <=> (1=usedColours[0]),
        aux1 => aux2,
        aux3 <=> (2=colouring[0]),   aux4 <=> (1=usedColours[1]),
        aux3 => aux4,
        aux5 <=> (3=colouring[0]),   aux6 <=> (1=usedColours[2]),
        aux5 => aux6,
        aux7 <=> (1=colouring[1]),
        aux7 => aux2,
        aux8 <=> (2=colouring[1]),
        aux8 => aux4,
        aux9 <=> (3=colouring[1]),
        aux9 => aux6,
        aux10 <=> (1=colouring[2]),
        aux10 => aux2,
        aux11 <=> (2=colouring[2]),
        aux11 => aux4,
        aux12 <=> (3=colouring[2]),
        aux12 => aux6,
        aux13 <=> (1=colouring[3]),
        aux13 => aux2,
        aux14 <=> (2=colouring[3]),
        aux14 => aux4,
        aux15 <=> (3=colouring[3]),
        aux15 => aux6,
        aux16 <=> (1=colouring[4]),
        aux16 => aux2,
        aux17 <=> (2=colouring[4]),
        aux17 => aux4,
        aux18 <=> (3=colouring[4]),
        aux18 => aux6
```

Figure 3.10: ESSENCE′GCP problem instance **after flattening**, targeting solver MINION

The C++ class representing the problem inherits from the class 'Example', which is de-
fined in Gecode's library. First, the variables on which search is performed are declared as
protected class members. Then the class constructor is defined: given the search variables,
auxiliary variables are initialised, followed by the set of constraints. Finally, the branching
option for each variable is stated, for which TAILOR picks a standard approach (smallest
domain and smallest value first).

After the class constructor, several additional methods need to be defined. First, method 'constrain' is used to express the objective. Second, 'print' is overwritten to handle how solutions are printed to stdout. Third, the 'copy' method and the constructor for cloning are necessary for Gecode's search procedure. Finally, a main function is defined in order to initiate the solving the problem.

Summary

We have considered tailoring a sample instance of the Graph Colouring problem to three targets: solver MINION, FlatZinc and C++-based Gecode, as it is performed by our implementation, TAILOR. Note, that TAILOR takes about 0.25 seconds to generate each of the respective MINION, Gecode and FlatZinc files (on an rather old machine, an Intel(R) Pentium(R) 4 CPU 3.00GHz with 512 RAM using Java 1.5.0).

Though the formats differ in their syntax, the (low) level of abstraction is the same. Examples for differences are for instance the different starting values for array indexing in MINION and FlatZinc.

The examples demonstrate that a modeller requires a lot of specific knowledge to formulate a problem directly in solver language. For instance, in order to construct the C++ instance for Gecode, it is necessary to be aware of the Gecode API.

Furthermore, the instance we have considered in this example is very small - more practical instances will probably involve graphs that contain far more than 5 vertices and 4 colours as in the example. However, modelling larger instances *by hand* can easily become unfeasible: writing an instance for MINION, FlatZinc or Gecode with 50 vertices by hand is expected to take longer than the 2.5 seconds TAILOR takes (on the same machine as mentioned above).

3.6 Summary

In this chapter we have considered the task of *tailoring*, the compilation of a solver-independent constraint model to low-level solver input. We started by giving a brief overview of the tailoring task by describing its features in a nutshell. Then we proposed a general, efficient and easily-extendable architecture for a tailoring engine that consists of three core parts: frontend, middlelend and backend, which are each thoroughly discussed. During this discussion, we introduced the tool TAILOR that incorporates all tailoring steps described in this chapter. Finally, we illustrated tailoring on an example, where we use TAILOR in order to generate three different solver formats: solver MINION's text format, solver Gecode's C++ format and FlatZinc format.

The contributions of this chapter are clear: first, we specified the main objectives of tailoring and presented a generic and efficient tailoring engine that can easily be extended to

support further input- or output-languages. Second, we showed how core tailoring steps, like preprocessing and flattening, can be generalised for any target solver by the use of *solver profiles*. This allows us to re-use these central parts of the implementation (that, in the case of TAILOR nearly make up 70% of the code) for all target solvers. Third, we demonstrated that tailoring can be efficiently implemented by introducing the tool TAILOR that can currently generate three output formats from two different modelling languages, which is illustrated throughout the chapter, in particular, in the tailoring example.

In conclusion, we have seen that tailoring is not only an important item in the context of automated constraint modelling, but can be realised in a general, efficient and extendable way. In the following chapter, we will consider how we can *augment* particular tailoring processes in order to *enhance* the instance that is tailored.

```
MINION 3
**VARIABLES**
DISCRETE colouring[5] {1..3}
SPARSEBOUND usedColours[3] {0,1}
# auxiliary variables
DISCRETE aux0 {0..3} # usedColours[0] + usedColours[1] + usedColours[2]
BOOL aux1   # 1=colouring[0]
BOOL aux2   # 1=usedColours[0]
BOOL aux3   # 2=colouring[0]
BOOL aux4   # 1=usedColours[1]
BOOL aux5   # 3=colouring[0]
BOOL aux6   # 1=usedColours[2]
BOOL aux7   # 1=colouring[1]
BOOL aux8   # 2=colouring[1]
BOOL aux9   # 3=colouring[1]
BOOL aux10  # 1=colouring[2]
BOOL aux11  # 2=colouring[2]
BOOL aux12  # 3=colouring[2]
BOOL aux13  # 1=colouring[3]
BOOL aux14  # 2=colouring[3]
BOOL aux15  # 3=colouring[3]
BOOL aux16  # 1=colouring[4]
BOOL aux17  # 2=colouring[4]
BOOL aux18  # 3=colouring[4]

**SEARCH**
MINIMISING aux0

PRINT [colouring,usedColours]
VARORDER [colouring,usedColours,aux2,aux4,aux6,aux0,aux1,aux3,aux5,
aux7,aux8,aux9,aux10,aux11,aux12,aux13,aux14,aux15,aux16,aux17,aux18]

**CONSTRAINTS**
diseq(colouring[2], colouring[4])    diseq(colouring[2], colouring[3])
diseq(colouring[1], colouring[4])    diseq(colouring[1], colouring[2])
diseq(colouring[0], colouring[3])    diseq(colouring[0], colouring[1])
sumleq([usedColours[0],usedColours[1],usedColours[2]], aux0)
sumgeq([usedColours[0],usedColours[1],usedColours[2]], aux0)
reify(eq(1, colouring[0]), aux1)
reify(eq(1, usedColours[0]), aux2)
ineq(aux1,aux2,0)
reify(eq(2, colouring[0]), aux3)
reify(eq(1, usedColours[1]), aux4)
reify(eq(3, colouring[0]), aux5)
reify(eq(1, usedColours[2]), aux6)
ineq(aux5,aux6,0)   ineq(aux3,aux4,0)
reify(eq(1, colouring[1]), aux7)
reify(eq(2, colouring[1]), aux8)
reify(eq(3, colouring[1]), aux9)
ineq(aux9,aux6,0)   ineq(aux8,aux4,0)
reify(eq(1, colouring[2]), aux10)
reify(eq(2, colouring[2]), aux11)
reify(eq(3, colouring[2]), aux12)
ineq(aux12,aux6,0) ineq(aux11,aux4,0)
reify(eq(1, colouring[3]), aux13)
reify(eq(2, colouring[3]), aux14)
reify(eq(3, colouring[3]), aux15)
ineq(aux15,aux6,0) ineq(aux14,aux4,0)
reify(eq(1, colouring[4]), aux16)
reify(eq(2, colouring[4]), aux17)
reify(eq(3, colouring[4]), aux18)
ineq(aux18,aux6,0) ineq(aux17,aux4,0)
ineq(aux16,aux2,0) ineq(aux13,aux2,0)
ineq(aux10,aux2,0) ineq(aux7,aux2,0)
**EOF**
```

Figure 3.11: MINION **input** for the Graph Colouring Problem instance from Fig. 2.1

```
array [1..5] of var 1..3 :colouring ::output_array([1..5]);
array [1..3] of var 0..1 :usedColours ::output_array([1..3]);

% auxiliary variables
var 0..3: aux0;  % usedColours[0] + usedColours[1] + usedColours[2]
var bool: aux1;  % 1=colouring[0]
var bool: aux2;  % 1=usedColours[0]
var bool: aux3;  % 2=colouring[0]
var bool: aux4;  % 1=usedColours[1]
var bool: aux5;  % 3=colouring[0]
var bool: aux6;  % 1=usedColours[2]
var bool: aux7;  % 1=colouring[1]
var bool: aux8;  % 2=colouring[1]
var bool: aux9;  % 3=colouring[1]
var bool: aux10; % 1=colouring[2]
var bool: aux11; % 2=colouring[2]
var bool: aux12; % 3=colouring[2]
var bool: aux13; % 1=colouring[3]
var bool: aux14; % 2=colouring[3]
var bool: aux15; % 3=colouring[3]
var bool: aux16; % 1=colouring[4]
var bool: aux17; % 2=colouring[4]
var bool: aux18; % 3=colouring[4]

% constraints
constraint int_lin_eq([1, 1, 1, -1],
  [usedColours[1], usedColours[2], usedColours[3], aux0], 0);
constraint int_ne(colouring[1], colouring[2]);
constraint int_ne(colouring[1], colouring[4]);
constraint int_ne(colouring[2], colouring[3]);
constraint int_ne(colouring[2], colouring[5]);
constraint int_ne(colouring[3], colouring[4]);
constraint int_ne(colouring[3], colouring[5]);
constraint bool_le(aux1, aux2);
constraint bool_le(aux7, aux2);
constraint bool_le(aux10, aux2);
constraint bool_le(aux13, aux2);
constraint bool_le(aux16, aux2);
constraint bool_le(aux17, aux4);
constraint bool_le(aux18, aux6);
constraint int_eq_reif(3, colouring[5], aux18);
constraint int_eq_reif(2, colouring[5], aux17);
constraint int_eq_reif(1, colouring[5], aux16);
constraint bool_le(aux14, aux4);
constraint bool_le(aux15, aux6);
constraint int_eq_reif(3, colouring[4], aux15);
constraint int_eq_reif(2, colouring[4], aux14);
constraint int_eq_reif(1, colouring[4], aux13);
constraint bool_le(aux11, aux4);
constraint bool_le(aux12, aux6);
constraint int_eq_reif(3, colouring[3], aux12);
constraint int_eq_reif(2, colouring[3], aux11);
constraint int_eq_reif(1, colouring[3], aux10);
constraint bool_le(aux8, aux4);
constraint bool_le(aux9, aux6);
constraint int_eq_reif(3, colouring[2], aux9);
constraint int_eq_reif(2, colouring[2], aux8);
constraint int_eq_reif(1, colouring[2], aux7);
constraint bool_le(aux3, aux4);
constraint bool_le(aux5, aux6);
constraint int_eq_reif(1, usedColours[3], aux6);
constraint int_eq_reif(3, colouring[1], aux5);
constraint int_eq_reif(1, usedColours[2], aux4);
constraint int_eq_reif(2, colouring[1], aux3);
constraint int_eq_reif(1, usedColours[1], aux2);
constraint int_eq_reif(1, colouring[1], aux1);

solve minimize aux0;
```

Figure 3.12: Graph Colouring Problem instance from Fig. 2.1 in **FlatZinc**

```cpp
#include "examples/support.hh"
#include "gecode/minimodel.hh"

class GraphColouring1 : public Example {

protected:    // variables:
    IntVarArray colouring;
    IntVarArray usedColours;
    IntVar aux0;

public:       // actual problem
    GraphColouring1(const Options& opt) :            colouring(this, 5, 1, 3),
                                                     usedColours(this, 3, 0, 1),
                                                     aux0(this,0,3)   {
        // defining the ArgsVarArrays that hold the aux variables
          BoolVarArgs _aux_bool_var_buffer(18);
          IntVarArgs __int_var_buffer_part_7331(1);

        // defining auxiliary boolean variables
          BoolVar
          aux1(this, 0,1), aux2(this, 0,1), aux3(this, 0,1), aux4(this,0,1),
          aux5(this, 0,1), aux6(this, 0,1), aux7(this, 0,1), aux8(this,0,1),
          aux9(this, 0,1), aux10(this, 0,1), aux11(this, 0,1), aux12(this, 0,1),
          aux13(this, 0,1), aux14(this, 0,1), aux15(this, 0,1), aux16(this, 0,1),
          aux17(this, 0,1), aux18(this, 0,1);

        // assigning each auxilary variable to the corresponding container/buffer
          _aux_bool_var_buffer[0] = aux1; _aux_bool_var_buffer[1] = aux2;
          _aux_bool_var_buffer[2] = aux3; _aux_bool_var_buffer[3] = aux4;
          _aux_bool_var_buffer[4] = aux5; _aux_bool_var_buffer[5] = aux6;
          _aux_bool_var_buffer[6] = aux7; _aux_bool_var_buffer[7] = aux8;
          _aux_bool_var_buffer[8] = aux9; _aux_bool_var_buffer[9] = aux10;
          _aux_bool_var_buffer[10] = aux11;_aux_bool_var_buffer[11] = aux12;
          _aux_bool_var_buffer[12] = aux13; _aux_bool_var_buffer[13] = aux14;
          _aux_bool_var_buffer[14] = aux15; _aux_bool_var_buffer[15] = aux16;
          _aux_bool_var_buffer[16] = aux17; _aux_bool_var_buffer[17] = aux18;
          __int_var_buffer_part_7331[0] = aux0;

        /* constraints */
          IntVarArgs _lours_1__usedColours_2_(3);
            _lours_1__usedColours_2_[0] = usedColours[0];
            _lours_1__usedColours_2_[1] = usedColours[1];
            _lours_1__usedColours_2_[2] = usedColours[2];
          linear(this, _lours_1__usedColours_2_, IRT_EQ, aux0);
          rel(this,colouring[0], IRT_NQ, colouring[1], opt.icl());
          rel(this,colouring[0], IRT_NQ, colouring[3], opt.icl());
          rel(this,colouring[1], IRT_NQ, colouring[2], opt.icl());
          rel(this,colouring[1], IRT_NQ, colouring[4], opt.icl());
          rel(this,colouring[2], IRT_NQ, colouring[3], opt.icl());
          rel(this,colouring[2], IRT_NQ, colouring[4], opt.icl());
          rel(this,aux1, IRT_LQ, aux2, opt.icl());
          rel(this,aux7, IRT_LQ, aux2, opt.icl());
          rel(this,aux10, IRT_LQ, aux2, opt.icl());
          rel(this,aux13, IRT_LQ, aux2, opt.icl());
          rel(this,aux16, IRT_LQ, aux2, opt.icl());
          rel(this,aux17, IRT_LQ, aux4, opt.icl());
          rel(this,aux18, IRT_LQ, aux6, opt.icl());
          rel(this,colouring[4], IRT_EQ, 3, aux18, opt.icl());
          rel(this,colouring[4], IRT_EQ, 2, aux17, opt.icl());
          rel(this,colouring[4], IRT_EQ, 1, aux16, opt.icl());
          rel(this,aux14, IRT_LQ, aux4, opt.icl());
          rel(this,aux15, IRT_LQ, aux6, opt.icl());
          rel(this,colouring[3], IRT_EQ, 3, aux15, opt.icl());
          rel(this,colouring[3], IRT_EQ, 2, aux14, opt.icl());
          rel(this,colouring[3], IRT_EQ, 1, aux13, opt.icl());
          rel(this,aux11, IRT_LQ, aux4, opt.icl());
          rel(this,aux12, IRT_LQ, aux6, opt.icl());
          rel(this,colouring[2], IRT_EQ, 3, aux12, opt.icl());
          rel(this,colouring[2], IRT_EQ, 2, aux11, opt.icl());
          rel(this,colouring[2], IRT_EQ, 1, aux10, opt.icl());
```

```cpp
        rel(this,aux8, IRT_LQ, aux4, opt.icl());
        rel(this,aux9, IRT_LQ, aux6, opt.icl());
        rel(this,colouring[1], IRT_EQ, 3, aux9, opt.icl());
        rel(this,colouring[1], IRT_EQ, 2, aux8, opt.icl());
        rel(this,colouring[1], IRT_EQ, 1, aux7, opt.icl());
        rel(this,aux3, IRT_LQ, aux4, opt.icl());
        rel(this,aux5, IRT_LQ, aux6, opt.icl());
        rel(this,usedColours[2], IRT_EQ, 1, aux6, opt.icl());
        rel(this,colouring[0], IRT_EQ, 3, aux5, opt.icl());
        rel(this,usedColours[1], IRT_EQ, 1, aux4, opt.icl());
        rel(this,colouring[0], IRT_EQ, 2, aux3, opt.icl());
        rel(this,usedColours[0], IRT_EQ, 1, aux2, opt.icl());
        rel(this,colouring[0], IRT_EQ, 1, aux1, opt.icl());

        branch(this, colouring, INT_VAR_SIZE_MIN, INT_VAL_MIN);
        branch(this, usedColours, INT_VAR_SIZE_MIN, INT_VAL_MIN);
        branch(this, _aux_bool_var_buffer, INT_VAR_SIZE_MIN, INT_VAL_MIN);
        branch(this, __int_var_buffer_part_7331, INT_VAR_SIZE_MIN, INT_VAL_MIN);
    }

    // Method to state the objective
    void
    constrain(Space* s) {
        rel(this, aux0, IRT_LE, static_cast<GraphColouring1*>(s)->aux0.val());
    }

    // method for printing solutions
    virtual void print(std::ostream& os) {
        os << "\n";
        os  << " colouring:" << std::endl;
        for(int i=0; i<5; i++) {
            os  << colouring[i] << ",  ";
            if(i % 10 == 0 && i!=0) os << "\n";
        }
        os << std::endl;

        os  << " usedColours:" << std::endl;
        for(int i=0; i<3; i++) {
            os  << usedColours[i] << ",  ";
            if(i % 10 == 0 && i!=0) os << "\n";
        }
        os << std::endl;
    }

    // copy during cloning
    virtual Space*
    copy(bool share) {
        return new GraphColouring1(share, *this);
    }

    // constructor for cloning
    GraphColouring1(bool share, GraphColouring1& s) : Example(share,s){
        colouring.update(this, share, s.colouring);
        usedColours.update(this, share, s.usedColours);
        aux0.update(this,share,s.aux0);
    }
};


int
main(int argc, char* argv[]) {
    Options opt("GraphColouring1");
    opt.solutions(1);
    opt.parse(argc, argv);
    Example::run<GraphColouring1, BAB, Options>(opt);
    return 0;
}
```

Figure 3.13: **Gecode C++ input** for the GCP instance from Fig. 2.1

CHAPTER 4

INSTANCE OPTIMISATIONS

The main aim of instance tailoring is to produce valid solver input from a given problem instance. However, we want to strengthen this aim with two objectives: first, to generate an *effective* instance with respect to solving time and search space used in the target solver. Second, to tailor and enhance in as *little* computational time (and memory) as possible. In other words, we want to perform *instance optimisations* during tailoring that add little computational overhead. Embedding optimisations into tailoring provides two benefits: first, the combination of optimisations with necessary tailoring tasks, such as flattening, saves computational time and memory (e.g. see Sec. 4.2). Second, since the problem instance is processed at different levels of abstraction (e.g. unflat versus flat representation) enhancement techniques can be applied at the most appropriate abstraction level.

Automated enhancement of constraint instances has been subject to previous work in Constraint Programming (summarised in Sec. 4.1). However, most of these techniques require solutions of an instance to perform enhancements (see Sec. 4.1), hence the problem needs to be tailored and solved before the model is enhanced, which is infeasible during tailoring. Therefore, we do not mainly focus on enhancements proposed in Constraint Programming, but explore approaches from related fields, such as Compiler Construction [2], that deal with processes similar to tailoring.

Optimising instances during tailoring is strongly related to Code Optimisation [3], a well-established and well-researched area of Compiler Construction that is concerned with automatically enhancing program code during compilation. Compiling a high-level programming language (like C++) to machine code is a very similar process to tailoring. Both program compilation and tailoring deal with the conversion of a high-level language to a low-level format consisting of granular expressions that differ slightly between the targets. However, there is one key difference: program compilation processes *instructions* whose variables represent registers at a particular state; tailoring processes *relations* that are imposed on decision variables that range over integer domains. Hence, the former deals with expressions 'in series' and the latter with expressions 'in parallel'. Though this difference renders many code optimisations inapplicable to tailoring, we can draw conclusions and inspiration from them: by assessing promising code optimisation techniques in terms of

their objectives and applicability to tailoring, we exploit parallels between tailoring and program compilation, and propose a set of effective instance optimisations.

In summary, the contributions of this chapter are a set of efficient instance optimisation techniques that were mainly inspired by Code Optimisation. Each technique is easily integrable into tailoring and adds negligible overhead to the tailoring process, even if performed in vain. More importantly, as we demonstrate in our empirical evaluation (Chapter 8), the combination of all optimisation techniques can achieve dramatic solving time speedups, in some cases of a factor of more than 3,000, including reductions in search space. Finally, we stress that *none* of these optimisation techniques are routinely performed by constraint solvers or flattening tools at present (with the exception of basic common subexpression elimination, which has been recently added to the MiniZinc-FlatZinc converter, following our work). Since almost all constraint systems perform some translation of the expressions they allow the user to input, the benefits of instance optimisation during tailoring could be made available in most constraint systems.

This chapter is organised as follows. First, we analyse established optimisation techniques and consider their objectives, aims and applicability during tailoring in Sec. 4.1: we begin with an overview of techniques proposed in the context of Constraint Programming, followed by techniques from Compiler Construction. Then each optimisation technique is presented, starting with common subexpression elimination (CSE) in Sec. 4.2, followed by techniques in order to increase the number of common subexpressions (Sec. 4.3), and the scope of CSE (Sec. 4.4), then techniques of eliminating redundant constraints (Sec. 4.5) and finally Quantification Optimisations (Sec. 4.6). We wrap up with a chapter summary in Sec. 4.7).

4.1 Established Optimisation Techniques

In this section, we summarise optimisation techniques proposed in the context of Constraint Programming and Code Optimisation. Our aim is to use related work as an inspiration for optimisation techniques that are cheap, effective and easily integrable into the tailoring process.

Interestingly, the enhancement techniques from Code Optimisation are far more useful and inspirational than techniques proposed in Constraint Programming. There are several reasons for this: first, many enhancement techniques in CP are stand-alone techniques that are not easily integrable into tailoring. Second, many CP techniques require instance solutions *prior* to performing enhancement, which is infeasible during tailoring. Third, as opposed to optimisation techniques in Constraint Programming, Code Optimisation is a well-established field that has been studied for many decades, thus many more ideas have been presented. In the following, we briefly present each technique and discuss its significance and applicability into the context of performing instance optimisations during tailoring.

4.1.1 Optimisation Techniques in Constraint Programming

There exists a set of automated techniques that aim at generating an effective problem instance in Constraint Programming which we summarise in this section. Note, that we only consider *static* enhancement techniques, i.e. techniques that alter the problem model, as opposed to *dynamic* techniques that are performed during search, when the instance is solved. After giving a brief description of each technique's aims and possible approaches, we discuss the benefits and its applicability during tailoring.

Adding Implied Constraints

A constraint is called *implied* if the solutions of a constraint instance are the same with or without the constraint [77]. We strengthen this definition with the following requisite: adding an implied constraint has to result in additional propagation (i.e. an implied constraint reduces search). In summary, adding an implied constraint to an instance is beneficial. However, inferring an implied constraint and proving that it enhances propagation is costly and difficult. Several approaches have been proposed on how to add implied constraints to instances.

Charnley *et al* [18] automatically infer implied constraints by setting up an architecture consisting of a theorem prover, a constraint solver and an automated theory formation system: given a constraint class and parameter values, some solutions are generated by the solver which are passed to the theory formation system that generates conjectures related to the constraints in the instances. Each conjecture is passed to the theorem prover in an attempt to proof that the conjecture is implied from the constraint instance. If there is a proof, the conjecture is added as additional constraint to the instances which are solved again. If the constraint improves the solving performance, it is kept in the problem model. Note, that this enhancement technique improves both problem instances and classes.

Bessiere *et al* [13] introduce a generic framework to learn implied *global* constraints, in particular the global 'gcc' constraint in their implementation. A brute-force learning algorithm is used to determine the tightest set of parameters for a gcc constraint that is an implied constraint (which is tested in the constraint solver Choco [19]). Specific heuristics that exploit constraint properties reduce the computational effort. The empirical evaluation (that was performed on instances for which implied constraints could be found), demonstrates that the learning effort pays off for the solving time reduction obtained by the implied constraints. However, the study does not consider instances for which implied constraints can *not* be found and therefore does not analyse the potential *penalty* one might have to pay for attempting to enhance an instance.

Both Charnley *et al* [18] and Bessiere *et al* [13] propose interesting optimisation techniques that effectively enhance particular problem instances. However, we have not include them into tailoring for two main reasons. First, both techniques require solutions of instances in order to prove the constraints to be implied. This is not useful in the context of tailoring,

since solutions can only be obtained by first tailoring the instance to a solver, i.e. it would require to tailor twice. Second, both techniques cannot be integrated into any necessary tailoring step, and hence have to be added *on top* of tailoring. However, since these techniques fire only for a limited number of constraints instances, but would be performed for *every* instance that is tailored, this is expected to add overhead to tailoring.

Therefore, we conclude that the techniques from above would be most successfully applied as an *extension* to tailoring, to particular instances that have a high potential of benefit (e.g. learning implied gcc constraints for particular scheduling problems before/after tailoring). Formulating heuristics of when these techniques fire on a particular instance is an interesting item for future work.

Adding Symmetry Breaking Constraints

Many constraint instances contain *symmetries* that are mainly introduced during modelling, when entities that are indistinguishable in the original problem are represented by distinct values or variables. For instance, in the n-queens problem [57], it does not matter which of the n queen figures is placed into the first row of the chessboard, however, many models assign a variable to each queen. This results in *symmetric* solutions, that are permutations of another and all represent the same setting in the actual problem. If a constraint solver searches for all solutions of a problem, then looking for symmetric solutions can add significant overhead. Therefore, one can add *symmetry breaking constraints* that *eliminate* symmetries in the problem and hence reduce the search space. For more details on symmetries and symmetry breaking in Constraint Programming, consult [35].

Symmetry breaking is an important component of efficient modelling, which is subject to active research in Constraint Programming [35]. Unfortunately, including symmetry breaking into tailoring is out of scope of this thesis, but an interesting candidate for future work.

The CGRASS System

The CGRASS system [28] takes a low-level constraint instance as input and returns an automatically enhanced instance, by help of a proof planner. The instance enhancements include ordering and evaluation by normalisation, the removal of particular redundant constraints, domain and bounds propagation, adding inequalities that break symmetries and a simple form of common subexpression elimination. Its structure was inspired by proof planning [17].

CGRASS is a successful tool from which we have drawn inspiration, such as performing extensive normalisation of expressions during preprocessing (Sec. 3.3.1). However, many optimisation techniques in CGRASS have limitations: first, since CGRASS is restricted to flat constraint instances, all optimisation techniques are tailored to simple expressions and

do not exploit high-level constructs such as quantifications. Furthermore, the enhancement techniques are not particularly cheap: the authors give no concrete translation times, but note that the time invested into enhancement often exceeds the solving time reduction gained from the enhancements. In fact, the authors argue that the input format is too low-level to perform enhancements effectively and that transformations on more abstract formulations (involving e.g. quantifications) would be more beneficial. Therefore, we focus on more general enhancement rules to be included during tailoring.

Constraint Representation and Propagator Selection

Harvey *et al* [40] present efficient representations of linear constraints to improve propagation. In particular, they investigate equivalent representations of several linear constraints by comparing them in terms of bounds and domain propagation. This study has been very inspirational and given us important insights on how to map linear constraint expressions to efficient but cheap propagators, which we have applied (to a certain extent) in the implementation of TAILOR.

Schulte and Stuckey [72] investigate when expensive domain propagators can be replaced by cheaper bounds propagators while resulting in the same amount of propagation. These cases can be statically determined, as the authors show. Furthermore, in [73] the authors extend their work to an even more effective dynamic approach. This work shows important results that are easily integrable into tailoring to improve the propagator selection, which we have, to an extend applied. Note however, that these results cannot be directly applied for every solver, since some solvers do not allow to state the propagator type (bounds or domains) explicitly. For instance, in solver Minion [32], the *variable type* determines the type of propagation (bounds or domains), while in solver Gecode [80], the propagation type is explicitly set within the constraint specification. Therefore, replacing an expensive domain propagator with a cheap bounds propagator in Minion cannot be achieved as easily as in Gecode, since it requires a change of the variable type, which can have consequences on the propagation behaviour of other constraints and possibly impair the model.

Automated Modelling Systems

There exist several automated modelling systems that aim at generating efficient constraint instances, typically from a rather intuitive (or naive) input. This is an important step that is typically *independent* of a particular target solver and therefore constitutes the automated modelling step which is performed *before* tailoring.

4.1.2 Optimisation Techniques in Compilers

Code Optimisation [3] is aimed at automatically enhancing program code during the compilation process. It is a well-researched area in Compiler Construction [2] and standard in current compilers. In this section, we discuss the most established optimisation techniques in compilers from which we can draw parallels to tailoring constraint instances. Obviously inapplicable optimisation techniques are excluded from this discussion, e.g. altering statement blocks to improve the execution paths.

Most of the optimisation techniques presented below depend on the *data flow analysis* [4, 20] which is concerned with deriving information about the data flow along particular execution paths. This analysis is not useful for constraint instances, since expressions represent relations and not instructions. However, though the compiler optimisation approaches cannot be directly applied to tailoring, the objectives are very similar and we can draw inspiration on how to proceed. Compiler Optimisation has influenced many related fields of Constraint Programming, such as SAT [55], Proof Theory [60] or Model Checking [49].

4.1.3 Summary

In summary, we have seen a broad selection of enhancement techniques of which some provide inspiration for instance enhancements. In the following sections, we present each optimisation technique, show how to integrate them into tailoring by proposing algorithms which we analyse according to their time complexity. A thorough empirical evaluation of each technique follows in Chapter 8.

Eliminating Redundant Operations

Redundant operations in program code are instructions that, if eliminated, do not change the behaviour of the program. Redundant operations can stem from poor programming (e.g. unnecessary recalculations), but also occur on a very low-level to which the programmer has no access: for instance, accesses to the same data structure often share operations of which the programmer is not (and should not be) aware of. Eliminating redundant operations shortens the program while preserving the program semantics. In tailoring, the elimination of redundant operations corresponds to eliminating *redundant constraints*, which we discuss in Sec. 4.5.

Global Common Subexpression Elimination

In program code, an expression E is called common, if E has been previously computed and the variables of E have not changed since the previous computation. A common subex-

pression is eliminated [20] by reusing the previously computed value (and register) and discarding E.

In constraint instances, variables have no underlying state, hence two expressions E_1 and E_1 that have the same representation are also common. More details on common subexpression elimination are given in Section 4.2.

Copy Propagation

Copy Propagation exploits instructions of the form 'x := y' by reusing variable 'y' for variable 'x', saving instructions and registers.

The corresponding case in tailoring is exploiting linear relations of the form $x = y + c$ where x and y are decision variables and c a constant value, by reusing y for x, which saves constraints and variables. Exploiting explicit linear equalities has been well studied in the context of Constraint Programming [40, 61, 57]. 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 decision variable can be introduced with the intersection of the domains of x and y and replace both throughout.

Dead Code Elimination

Dead code corresponds to statements that compute values that are never used elsewhere in the program. Dead code results either from poor programming or low-level transformations during compilation to which the programmer has no access to. Obviously, elimination of dead code is beneficial since it avoids unnecessary computations.

In constraint instances, one could consider *unconstrained* decision variables as 'dead code'. There are two kinds of decision variables in a constraint instance: 'core' decision variables that are declared in the problem model, and auxiliary variables that have been added to the instance during flattening. If a core decision variable is unconstrained, removing it would mean a severe alteration of the initial model that does not preserve the model's semantics (the user might have a reason to leave a decision variable unconstrained). Auxiliary variables are *always* constrained, since they are introduced during flattening to represent subexpressions (see Sec. 3.3.2). Therefore, we do not detect and eliminate unconstrained decision variables.

Loop Optimisations (Code Motion)

Loops, such as for-loops or while-loops, are important constructs to express a set of related instructions in high-level programming languages. However, redundancies in loops can

have a big effect on the program's quality. Code Motion [5] is a means to reduce the number of instructions in a loop by detecting computations that are always the same, independent on how many times the loop is executed. These *loop-invariant* computations can be moved outside the loop.

In constraint modelling languages, there are similar constructs to loops: quantifications. Quantifications can contain redundancies whose elimination we discuss in Sec. 4.6.

Reducing Induction Variables

An induction variable is defined as a variable whose value increases by a constant value $c \in \mathbb{Z}$ each time it is assigned. Such variables occur in loops and it is often possible to reduce the set of induction variables in a loop to a particular subset.

The equivalent to induction variables in constraint models are quantifying variables. However, since quantifying variables correspond to constant values (and not registers as in program code), there is no benefit in reducing the number of quantifying variables. For instance, the quantifying variables in the example below

```
forall i,j:int(1..10) .
    (j=i+1) => x[i] != x[j]
```

can be reduced to the quantification over one quantifying variable:

```
forall i:int(1..10) .
    x[i] != x[i+1]
```

However, the resulting set of constraints after unrolling the quantification is the same, so the transformation has no effect and is therefore unnecessary.

Reduction in Strength

Reduction in strength is replacing an operation with an equivalent, but less expensive operation. For instance, replacing multiplication '2∗x' with addition 'x+x', or replacing a simple addition 'x+1' with the increment-operation 'x++'.

In the context of constraint instances, we can also perform reduction in strength: given two equivalent constraint representations, we pick the more efficient one wrt solving time and search space. However, the efficiency of a constraint representation is often closely related to the propagators provided by the target solver and typically *not* straightforward. For instance, replacing a multiplication $2∗x$ with $x + x$ will not improve, but possibly even impair propagation in a weighted linear sum constraint.

4.2 Basic Common Subexpression Elimination (CSE)

Common subexpression elimination (CSE) is a well-established optimisation technique originating from code optimisation [20]. In program code, an expression E is called common, if E has been previously computed and all variables in E have not changed since the previous computation. E can be eliminated [20] by reusing the previously computed value (and register) and discarding E.

Unlike in code optimisation, in CP we are dealing with expressions that represent *relations* and not consecutive *instructions*. Therefore, we distinguish two kinds of common subexpressions: first, we call two expression trees E_1 and E_2 *identical*, if they are the same wrt their syntax. Second, we call E_1 and E_2 *equivalent* if they are semantically equivalent under all satisfying assignments. In other words, *identical* subexpressions are common with respect to their syntax, *equivalent* subexpressions are common with respect to their semantics. For example, the two expressions $a * (b + c)$ and $(b + c) * a$ are not identical, but equivalent. However, their subexpressions $(b + c)$ and $(b + c)$ are identical and equivalent. Eliminating all *equivalent* common subexpressions is hard, therefore, we restrict our CSE-algorithm to eliminating all *identical* subexpressions. However, we will see that some families of equivalent common subexpressions can be easily *reduced* to identical subexpressions.

Expressions in CP are similar to expressions in related areas like Proof Theory, SAT or Model Checking, where CSE is a wide-spread technique [60, 55, 49]. In most of those disciplines, CSE is performed by transforming expression trees into acyclic directed graphs (DAG) where common nodes are merged [71]. This approach has also been applied to Numerical CSPs [6]. However, big finite-domain constraint instances can contain 10,000s of complex constraints, resulting in an extremely large DAG. Therefore, we introduce an alternative approach of CSE, that is embedded into the necessary task of flattening.

Embedding CSE into flattening provides three main benefits. First, it is easy to do. Second, extending a necessary task like flattening with some cheap operations that apply CSE, instead of performing an additional CSE-algorithm *on top* of tailoring, saves computational

---

Algorithm 4.1 FLATTEN_INSTANCE_CSE (M_S) flattens constraint instance M_S to M_S' with CSE. Differences/extensions to Algorithm 3.1 are given in *red font*.

Require: M_S: problem instance
1: **global** *flatConstraints, constraintBuffer, auxVars $\leftarrow$ empty lists*
2: global *hash-table $\leftarrow$ empty hash-table*
3: **for all** $E \in M_S.constraints$ **do**
4:     *constraintBuffer $\leftarrow$ 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: $M_S'.constraints \leftarrow flatConstraints$
9: $M_S'.vars \leftarrow \{M_S.vars \cup auxVars\}$
10: **return** M_S'

time and memory. Third, if an instance contains common subexpressions, we also save flattening time, since duplicate subexpressions are never flattened twice.

This section delivers the following contributions. First, we introduce an alternative, lightweight CSE approach that is embedded into flattening. Second, we show that for many instances, the proposed CSE-flattening approach lies in the same complexity class (wrt flattening time and memory) as standard flattening; additionally, we theoretically describe the potential benefit in instance reduction that can result from CSE (Sec. 4.2.1). In the following sections, we present techniques to *increase* the number of identical common subexpressions in an instance in order to further improve the instance (Sec. 4.3). Furthermore, in Sec. 4.4, we discuss the scope of our CSE approach and show which CSs we do not eliminate (and why). Finally, a thorough empirical analysis can be found in Chapter 8.

4.2.1 Extending Flattening with CSE

In the previous chapter we have seen a typical flattening procedure (Sec. 3.3), FLATTEN_INSTANCE , 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:

1. FLATTEN creates a different auxiliary variable for each subtree, while each identical subtree could/should be represented by the same auxiliary variable.
2. Creating redundant auxiliary variables also creates redundant constraints to initialise these variables.
3. 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.

The CSE-flattening Algorithm

Flattening can be easily extended with CSE: first, a hash-table is introduced to map every flattened subexpression to the corresponding auxiliary variable. Second, every time a subexpression E is about to be flattened, the hash-table is consulted: if E was flattened before, then the hash-table returns the corresponding auxiliary variable and E does not have to be flattened. Otherwise, E is flattened to an auxiliary variable aux and an entry $E \longrightarrow aux$ is added to the hash-table. The main advantage of using a hash-table for this process is that all operations involving adding or retrieving data from the hash-table are constant on average [46].

We formalise CSE-based flattening in the algorithm FLATTEN_INSTANCE_CSE (Alg. 4.1), an extension of FLATTEN_INSTANCE (Alg. 3.2). The *hash-table* maps every flattened

---

Algorithm 4.2 FLATTEN_CSE (E, *flatten2Aux*), recursive procedure based on FLATTEN (Alg. 3.2), extended with a *hash-table*, mapping all flattened subexpressions to the corresponding auxiliary variable. 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:       $String_{e_i} \leftarrow toString(e_i)$
  5:       **if** *hash-table.contains*($String_{e_i}$) **then**
  6:         $aux \leftarrow hash\text{-}table.get(String_{e_i})$
  7:       **else**
  8:         $aux \leftarrow$ FLATTEN_CSE (e_i, S, *true*)
  9:         *hash-table.add*($String_{e_i}$, aux)
 10:       $E.replaceChildWith(e_i, aux)$
 11: **if** *flatten2Aux* **then**
 12:   $Aux \leftarrow createNewVariable(E.\text{lb}, E.\text{ub})$; *auxVars.add(Aux)*
 13:   *constraintBuffer*.add('$Aux = E$')
 14:   **return** Aux
 15: **else**
 16:   **return** E

---

subtree to its corresponding auxiliary variable and is used by the new recursive flattening procedure, FLATTEN_CSE (Alg. 4.2), based on FLATTEN , to detect identical subexpressions that have been previously flattened:

FLATTEN_CSE :

1. whenever a non-leaf child e_i of current node E is flattened, we look for an entry of e_i in *hash-table* (line 5). Note that all subexpressions are stored in String format, hence e_i has to be converted into String format (line 4) before the hash-table is consulted.
2. If there is an entry, we re-use the auxiliary variable to which e_i is mapped (line 6), instead of flattening e_i again.
3. Otherwise (i.e. if there is no match in *hash-table*), we flatten e_i to auxiliary variable e_i' (line 8) and add $e_i \rightarrow e_i'$ to *hash-table* (line 9).

Clearly, FLATTEN_INSTANCE_CSE will flatten each *unique* subnode exactly once.

Lemma 4.2.1. *If constraint instance M_S contains m constraints that contain n subexpressions of which n_u are unique (with $n \geq n_u \geq m$), then* FLATTEN_INSTANCE_CSE *will generate a flat instance M_S' with $n_u - m$ auxiliary variables and n_u constraints.*

Proof. FLATTEN_INSTANCE_CSE applies FLATTEN_CSE to every constraint E in M_S (line 5 in Alg. 4.1). If E has a non-leaf child e_i (a subnode), then there are two cases (line 5 in Alg. 4.2): first, if there is no entry of e_i in *hash-table*, e_i is flattened to auxiliary variable aux, and $e_i \rightarrow aux$ is added to *hash-table* (line 9); hence, if e_i appears again in

M_S, there will be an entry in *hash-table*. Second, if there is an entry of e_i in *hash-table*, (e_i must have been flattened before), the corresponding auxiliary variable aux is retrieved from *hash-table* (line 6). 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 3.3.1). Therefore, M_S' contains $n_u - m$ auxiliary variables (one for each unique node, minus the root nodes) and n_u constraints ($n_u - m$ '*Aux=E*'-constraints and m root-node-constraints). $\square$

Comparing CSE-flattening with Standard Flattening

We analyse the differences of applying FLATTEN_INSTANCE and FLATTEN_INSTANCE_CSE on problem instance M_S with m constraints and n subexpressions of which n_u are unique, We denote M' the flat instance generated by FLATTEN_INSTANCE and M'_{CSE} the flat instance generated by FLATTEN_INSTANCE_CSE . The maximal number of subexpressions in any expression in instance M_S is denoted $\hat{k}$, and the longest String representation of any subexpression in M_S is denoted $\hat{s}$. We consider the *worst case* with respect to the translated instance, in which the instance contains no common subexpressions, i.e. the number of unique subexpressions equals the number of subexpressions, $n_u = n$.

We assume that the hash-table is implemented in such a way that the cost of each lookup is independent of the number of elements stored and operations to add and retrieve entries to the hash-table are performed in constant time. Note that this assumption holds for most standard implementations of hash-tables, e.g. in Java 1.5.0 [78] which we used in our implementation.

Theorem 4.2.1. M'_{CSE} *contains* $n-n_u$ **fewer** *constraints and auxiliary variables than* M'.

Proof. M' contains n constraints and $n-m$ auxiliary variables (Lemma 3.3.1), and M'_{CSE} contains n_u constraints and n_u-m auxiliary variables (Lemma 4.2.1). Since $n \geq n_u$, M'_{CSE} contains $n-n_u$ fewer constraints and auxiliary variables than M'. $\square$

Theorem 4.2.2. *The* **space complexity** *of* FLATTEN_INSTANCE_CSE *is* $O(\hat{s}n)$, *where* $\hat{s}$ *is the maximal String length of any subexpression in instance* M_S.

Proof. From Theorem 3.3.2 we know that the space complexity of FLATTEN_INSTANCE lies in $O(n)$. FLATTEN_INSTANCE_CSE uses the same data structures as FLATTEN_INSTANCE , with the addition of the hash-table to store flattened subexpressions. The hash-table 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 n_u unique nodes and the corresponding auxiliary variables, thus the hash-table uses $2 * n_u * \hat{s}$ units of memory, where $\hat{s}$ is the maximal String length of a subexpression in M_S. Therefore, FLATTEN_INSTANCE_CSE uses $2 * \hat{s} * n_u*$ more units of memory than FLATTEN_INSTANCE . Since $n_u \leq n$, the space complexity lies in $O(\hat{s}n$. $\square$

Theorem 4.2.3. *The **time complexity** FLATTEN_INSTANCE_CSE is $O((\hat{k} + \hat{s})n)$, where $\hat{k}$ is the maximal number of subexpressions in any expression in instance M_S and $\hat{s}$ is the maximal String length of a subexpression in instance M_S.*

Proof. From Theorem 3.3.1, we know that FLATTEN_INSTANCE lies in $O(n)$. FLATTEN_INSTANCE_CSE adds instructions to the flattening process of FLATTEN_INSTANCE : First, the subexpression is converted to String format, an operation that is in $O(k)$ where k is the number of subexpressions the to-be-flattened expression E contains. In the worst case, this is performed for all n subexpressions, yielding a runtime of $O(\hat{k} * n)$, where $\hat{k}$ is the maximal number of subexpressions an expression contains. Second, the hash-table check followed by either retrieving an object from the hash-table (if the check is positive) or creating an entry to the hash-table (if the check is negative). Since operations on hash-tables are in $O(1)$ on average they are all together in $O(1)$ since $2 * O(1) = O(1)$. However, the hash-table operations require the String (representing the subtree) to be read, which lies in $O(\hat{s})$, where $\hat{s}$ is the maximal String length of a subexpression in instance M_S. From Lemma 4.2.1 we know that FLATTEN_INSTANCE_CSE flattens all n_u unique nodes/-subexpression exactly once, thus the hash-table operations lie in $O(\hat{s}n_u)$. In summary, the runtime complexity of the CSE-operations (in addition to flattening) is $O(\hat{k}n) + O(\hat{s}n)$, since $n_u=n$ if the instance contains no common subexpressions and f is a constant. In summary, this results in an overall runtime of $O(n) + O(\hat{s}n) + O(\hat{k}n) = O((\hat{k}+\hat{s})n)$. $\qquad\square$

Conclusion Standard flattening and CSE-flattening differ with respect to the factors $\hat{s}$ for space and $\hat{k}$ and $\hat{s}$ for time complexity, where $\hat{s}$ denotes the longest String representation of any subexpression in M_S and $\hat{k}$ denotes the maximal number of subexpressions in any expression in M_S. Note, that $\hat{s}$ and $\hat{k}$ are often independent of n (and can hence be considered constants), since parameters often only scale the number of constraints, but not the width or depth of the expression trees. As an example, consider again the Graph Colouring Problem (Fig. 2.2 in Section 2.1), where the number of constraints increases with the number of vertices and colours, but the width and depth of the corresponding expression trees stays the same. Therefore, for many problem classes, the time complexity of flattening with or without CSE lies in the same complexity class, since $O(\hat{k}n) = O(n)$ if $\hat{k}$ is a constant. Furthermore, in cases where $\hat{k}$ is not a constant, note that $\hat{k}$ is always strictly smaller than n with the exception of the special case, where the instance contains only one constraint, where $\hat{k} = n$.

Common subexpression elimination as described in this section has been implemented in the tool TAILOR(Sec. 3.1.1) and we have studied the effects of CSE on various problem classes in an empirical analysis that is further described in Sec. 8.2. In this empirical study we observe that in practice, the difference between both flattening approaches is negligible, and, if the instance contains common subexpressions, CSE-flattening often clearly outperforms standard flattening in both runtime and memory.

4.3 Increasing the Number of Common Subexpressions

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

The number of unique nodes n_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 n_u if performed for every E_2 in the instance.

As a simple example, consider the special case of constant evaluation, that 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. Evaluation is part of normalisation, which is a main step during tailoring, and further discussed in Sec. 3.3.1, Note, that evaluation is performed in many systems that perform some sort of tailoring, e.g. the MiniZinc to FlatZinc converter [58]. In the following, we will consider a generic approach to increase the number of identical subexpressions.

4.3.1 Overview: Reformulating Equivalent Subexpressions

We investigate a set of *equivalence properties*(or law) p that state when two non-identical expressions E_1 and E_2 are equivalent, denoted by $E_1 \sim_p E_2$. Given p, we can either reformulate E_1 into E_2 or E_2 into E_1. For example, given commutativity as property of conjunction, expressions $A \wedge B$ and $B \wedge A$ are equivalent, which we denote $(A \wedge B) \sim_{comm} (B \wedge A)$. Therefore, we can replace the former with the latter and vice versa.

Thus, we consider two different things: first, an equivalence property or law that states when two expressions are equivalent, e.g. commutativity. Second, we consider reformulations that rewrite one expression into another representation, exploiting the equivalence property. Note, that such reformulations need not be deterministic. As an example, the expression $A \wedge B \wedge C$ can be rewritten into 5 different representations exploiting commutativity, such as $C \wedge B \wedge A$ or $A \wedge C \wedge B$. Furthermore, each such reformulation has an inverse. Note, that a reformulation is beneficial, if the chosen representation is not worse than the representation that is replaced.

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 required to detect equivalences can be arbitrarily large, especially for very powerful reformulations. For instance, detecting a maximal 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.

In summary, we can increase the number of identical subexpressions in an instance by exploiting an equivalence property p between two distinct expressions E_1 and E_2 and rewriting one representation into the other by applying a corresponding reformulation r. Ideally, this approach has the following properties:

- The *detection* of two equivalent expressions is cheap and integrable into tailoring
- The *reformulation* from E_1 to E_2 (or E_2 to E_1) is cheap.
- It is easy to determine which of the two equivalent representations E_1 or E_2 is *preferable* so that the reformulation does not impair the instance.

In the following, we present an algorithm that applies a generic reformulation r (with respect to a generic equivalence property p) to rewrite subexpressions into identical representations. Subsequently, we discuss a set of equivalences property and their applicability for increasing the number of identical subexpressions.

An Algorithm for Reformulation

We consider an algorithm to increase the number of identical subexpressions by reformulating expressions using a reformulation r based on an equivalence property p. The algorithm is embedded into flattening when common subexpressions are detected. Whenever a to-be-flattened expression E has no common subexpression, then the algorithm performs the following steps:

1. Reformulate E to E_r, using reformulation r
2. Generate the String representation $String_{E_r}$ from expression E_r
3. Check the hash-table for an entry of $String_{E_r}$
4. If successful, return the corresponding auxiliary variable, aux_r, after adding another entry into the hash-table: $E \longrightarrow aux_r$
5. Otherwise continue flattening

Note, that step 4 is necessary in case E has common subexpressions in the constraint instance: if another occurrence of E is flattened, the hash-table check will be positive during standard CSE and the reformulation need not be repeated.

More formally, the algorithm is summarised in Alg. 4.3 that illustrates the extensions to the recursive flattening procedure that performs CSE, FLATTEN_CSE in red font: whenever a non-leaf child e_i of E has no CS, e_i is reformulated using reformulation r (line 9), yielding the reformulated expression e_r. Next, e_r is converted to the String $String_R$ (line 10) and the hash-table is checked for an entry of $String_R$ (line 11). If the check is successful, the hash-table returns auxiliary variable aux (line 12), e_i is replaced with it (line 20), and another

entry is added to the hash-table, linking the original expression to aux, i.e. $String_{e_i} \longrightarrow aux$. This assures that if e_i appears again in the instance, the hash-table will have an entry and the whole reformulation process won't be repeated. Otherwise, if the hash-table has no entry of $String_{e_r}$, flattening proceeds as usual.

Algorithm 4.3 Reformulation for CS-Increase. The recursive procedure FLATTEN_REF (E, *flatten2Aux*) is based on the CSE-flattening procedure from Alg. 4.2, (FLATTEN_CSE), and performs a general reformulation r in order to increase the number of identical common subexpressions. Extensions are given in red font.

```
 1: if ¬ (all of E's children are leaves) then
 2:     for all e_i ∈ children(E) do
 3:         if ¬(e_i.isLeaf) then
 4:             String_{e_i} ← toString(e_i)
 5:             if hash-table.contains( String_{e_i}) then
 6:                 aux ← hash-table.get( String_{e_i})
 7:             else
 8:                if r is applicable to e_i then
 9:                    e_r ← r(e_i)
10:                    String_R ←toString(e_r)
11:                    if hash-table.contains(String_R) then
12:                        aux ← hash-table.get(String_R) ;
13:                        hash-table.add(String_{e_i}, aux) ;;
14:                    else
15:                        aux ← FLATTEN_REF (e_i, true);
16:                        hash-table.add(String_{e_i}, aux)
17:                else
18:                    aux ← FLATTEN_REF (e_i, true);
19:                    hash-table.add(String_{e_i}, aux)
20:             E.replaceChildWith(e_i,aux)
21: if flatten2Aux then
22:     Aux ← createNewVariable(E.lb, E.ub); auxVars.add(Aux)
23:     constraintBuffer.add('Aux = E')
24:     return Aux
25: else
26:     return E
```

Generic Time Complexity

Alg. 4.3 is very general and its complexity depends on two factors. First, the *applicability* of reformulation r matters: r is usually applicable only to a certain kind of expressions. For instance, de Morgan's Law can only be applied to particular Boolean expressions that are composed by disjunction and conjunction. Hence the algorithm also depends on the frequency of occurrence of the expression type to which r is applicable. We denote m_r the number of subexpressions in instance n to which r is applicable, where $n \geq n_r \geq 0$. Furthermore, we denote $n_{r,u}$ the number of unique nodes to which r is applicable, i.e. if $n_r - n_{r,u} > 0$ then there exist common subexpressions amongst the nodes to which r is applicable.

Second, it depends on the cost of reformulating expression E to E_r, i.e. the cost of applying r on E, which we denote $cost(r,k)$, where k is the number of nodes in the expression tree.

Since Alg. 4.3 is based on CSE-flattening (Alg. 4.2), we analyse the corresponding extensions to derive the time complexity:

Applicability Check First, the to-be-flattened expression e_i is tested for applicability, whose cost we denote $applic_r(k)$ where k represents the number of nodes in the tested expression. This test is performed on all n' subexpressions that have no previously flattened common subexpression. In the worst case (if the instance contains no common subexpressions) $n' = n$, hence performing the check lies in $O(n)*applic_r(\hat{k})$, where $\hat{k}$ is the maximum number of subexpressions of any expression in the instance (in the worst case, if the constraint instance has only one constraint, $\hat{k} = n$).

Reformulation Second, the reformulation r is applied to those nodes that pass the check, which are all unique nodes to which r is applicable, i.e. $n_{r,u}$ nodes. Note, that the other $n_r - n_{r,u}$ nodes to which r is applicable, are common subexpressions, and hence have a match in the hash-table, to which we add e_i and aux (line 13). We denote the cost of the reformulation $cost_r(k)$, where k is again the number of subexpressions in the reformulated expression. Therefore, the reformulation step lies in $O(n)*cost_r(\hat{k})$. since in the worst case $n_{r,u} = n_r = n$, and where $\hat{k}$ is the maximum number of subexpressions of any expression in the instance (note that if the constraint instance has only one constraint, $\hat{k} = n$).

toString Operation . Third, the reformulated expression e_r is converted to String format $String_R$, an operation that lies in $O(k)$ where k is the number of subexpressions the to-be-flattened expression E contains. This is performed for all $n_{r,u}$ subexpressions, where in the worst case, $n_{r,u} = n$, yielding a runtime of $O(\hat{k}n)$, where $\hat{k}$ is the maximal number of subexpressions an expression contains.

Hash-Table Operations Finally, hash-table operations are performed in order to retrieve a common subexpression. The first hash-table check (line 11) is performed on all $n_{r,u}$ nodes and the following two hash-table operations are performed on all those subexpressions that have a common subexpression. All hash-table operations are constant in average, but require to read the String representation, so we summarise the complexity with $O(\hat{s}n)$, since in the worst case, $n_{r,u} = n$, where $\hat{s}$ denotes the maximal String length of a subexpression in instance n.

In summary, the additional runtime complexity of Alg. 4.3 compared to CSE-flattening (Alg. 4.2) is:

$$O(n) * applic_r(\hat{k}) + O(n) * cost_r(\hat{k}) + O(\hat{k}n) + O(\hat{s}n) \tag{4.1}$$

In the following, we investigate different equivalence properties on both integral and Boolean expressions: associativity, commutativity, negation, distributivity, Horn Clauses and De

Morgan's Law. In each case, we will apply the reformulation in Alg. 4.3 (if necessary) and analyse the corresponding runtime from Eq. 4.1.

4.3.2 Associativity and Commutativity

Associativity and commutativity (AC) are well-known properties that hold for several operators, such as addition or conjunction. For instance, $A \wedge (C \wedge B)$ and $(C \wedge A) \wedge B$ are equivalent, since conjunction is commutative and associative.

Fortunately, reducing equivalent AC expressions can be performed during preprocessing, when expressions are *normalised*, which is fairly cheaper than applying Alg. 4.3 during flattening. Normalisation reduces expressions that are equivalent by AC to an identical representation: by imposing an *order* on the arguments of all AC operators, the reduction is performed even without requiring a detection step. Ordering is a main step during tailoring, and further discussed in Sec. 3.3.1, Note, that ordering is a common procedure in many systems that perform some sort of tailoring, such as the MiniZinc to FlatZinc converter [58].

In summary, the AC property can be exploited during *normalisation*, in particular by ordering, which is preferable to applying Alg. 4.3 for two main reasons: first, it requires no detection step, since simply all AC operators are normalised in the same fashion. Second, the number of subexpressions that are processed during normalisation is *much smaller* than the number of subexpressions that processed during flattening and Alg. 4.3, since all quantifications are unrolled. Note, that this approach satisfies the three properties that we required our reformulation approach to fulfill: both detection and reformulation are cheap and the reformulated representation is never worse than the original representation.

4.3.3 Negation

Negation can be propagated and extracted from particular expressions, which we consider as an equivalence property. For illustration, the two expressions $x \neq y$ and $\neg(x = y)$ are equivalent, which we denote $x \neq y \sim_{neg} \neg(x = y)$.

Note, that these expressions are *not* equivalent if either of them contain an *undefined* expression and the underlying semantics is relational. The relational semantics interprets every expression as a relation where undefined Boolean expressions are interpreted as *false*. For instance, the constraint $\neg(1/y = 1)$ with $y \in \{0, 1\}$ would have the solution $y = 0$ under the relational semantics: $(1/0)$ is interpreted as *false*; (*false* $= 1$) is undefined hence it is also *false*, and $\neg$*false* is *true*. However, in the reformulated case, $(1/y \neq 1)$, the assignment $y = 0$ is not a solution under the relational semantics: $(1/0)$ is undefined, i.e. *false*, and (*false* $\neq 1$) is again undefined and hence *false*. Therefore, this discussion is restricted to cases where all expressions are well-defined (in which case the underlying semantics does not matter wrt equivalence of expressions). For a detailed discussion on different semantics

for constraint languages with respect to undefinedness, see [29].

Similar to AC, normalisation reduces this kind of equivalence: every expression is transformed to negation normal form where negations are moved as far 'inside' the expression as possible. Hence, $\neg(x = y)$ is normalised to $x \neq y$. This is a common procedure, also in other tailoring tools, such as the MiniZinc to FlatZinc converter [58].

Active Negation Reformulation

However, negation can be exploited in an proactive fashion, employing a slight alteration of Alg. 4.3. The idea is to reformulate a subtree to its negated form, check the hash-table for an entry of the negated form, and, if successful, replace the expression with the negated corresponding auxiliary variable.

For illustration, consider the constraint instance in Example 4.3.1:

Example 4.3.1. Example for Active Negation Reformulation

```
given  n     :  int (1..)
find  x,y,z  :  int (0..n)

such that
 (x=0)   =>  (y=z)
 (x!=0)  =>  (y>0)
```

The instance contains two subexpressions $(x = 0)$ and $(x \neq 0)$, where one is the negated version of the other. Hence, the auxiliary variable used to represent the former can be used to represent the other, if it is negated. More specifically, if aux represents $(x = 0)$, then $\neg aux$ can represent $(x \neq 0)$. This is particularly beneficial, if the target solver allows negated variables as arguments of its propagators: then $\neg aux$ could just replace $(x \neq 0)$ which saves one auxiliary variable. Otherwise, the implication (reification) $(x \neq 0) \Leftrightarrow aux^\star$ (that would result from flattening $(x \neq 0)$) can be replaced by the simpler expression $\neg aux \Leftrightarrow aux^\star$.

Fig. 4.1 illustrates the example in more detail: the expression tree in (top, left) shows the original expression tree and (bottom, left) shows how the tree would be flattened using CSE-flattening. The tree in (top, right) shows the reformulated version of the original after applying active negation reformulation. This tree can be flattened in two different ways according to the following cases: first, if the solver supports negated arguments (bottom, middle), and second, otherwise (bottom, 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.
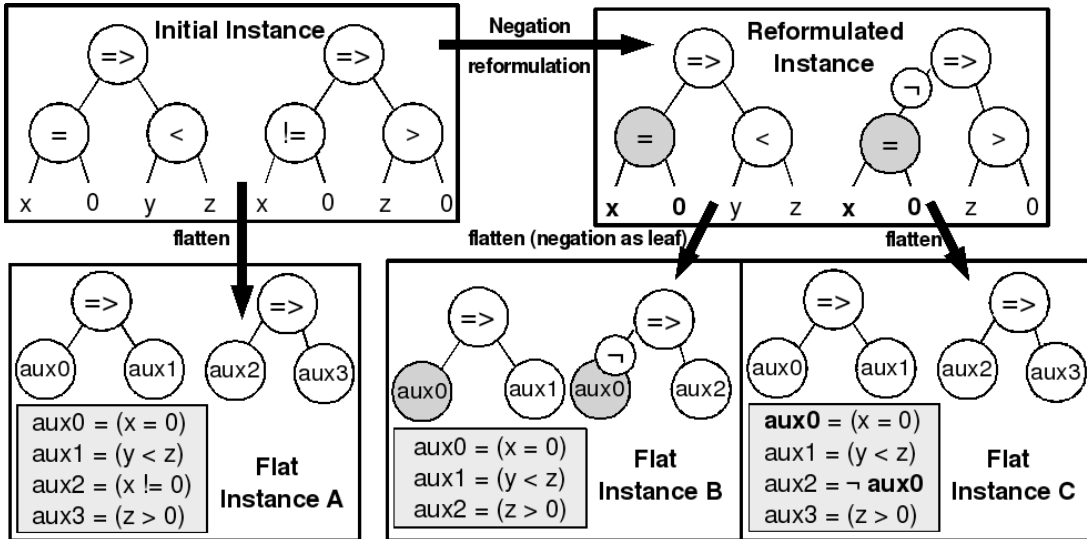
Figure 4.1: **Example for Active Negation Reformulation**: Flattening an instance (top left) with constraints $(x=0) \Rightarrow (y=z)$ and $(x\neq 0) \Rightarrow (y>0)$ in three different ways: if flattened directly (with or without CSE), we get *flat instance A* (bottom, left). If applying neg-reformulation 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*.

Active Negation Reformulation Algorithm

We summarise the procedure (for target solvers that allow negated arguments, e.g. MIN-ION) in Alg. 4.4 where alterations/adaptions to Alg. 4.3 are illustrated in red font: whenever a non-leaf child e_i of E has no CS and is relational, e_i is negated to $\neg e_i$ (line 9), and the hash-table is consulted with the String representation of $\neg e_i$ (line 11). If there is a match that returns auxiliary variable aux (line 12), e_i is replaced by the negated auxiliary variable $\neg aux$ (line 14), after adding another entry of $e_i \longrightarrow \neg aux$ to the hash-table (line 13)

If the target solver does *not* allow negated variables as arguments in constraints, $\neg e_i'$ has to be flattened to an auxiliary variable.

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. There-fore, the negation-reformulation may only be applied for solvers where Boolean negation is cheaper than all relational operators it substitutes.

To determine the runtime complexity, need to define the complexity of the applicability check, $applic_r(\hat{k})$, and the reformulation cost, $cost_r(\hat{k})$. First, the applicability check deter-mines if the expression is relational, a simple check, which is independent of the number of subexpressions k in the corresponding expression. Therefore, this check is constant, i.e. lies in $O(1)$. Second, the reformulation cost: since we limit the reformulation to relational expressions (i.e. we exclude Boolean expressions), the reformulation corresponds to an

Algorithm 4.4 Active Negation Reformulation. Excerpt of the recursive procedure FLAT-TEN_NEG (*E*,*flatten2Aux*) is based on the generic reformulation procedure FLATTEN_REF (Alg. 4.3). Changes are given in red font.

```
 1:  if ¬ (all of E's children are leaves) then
 2:      for all eᵢ ∈ children(E) do
 3:          if ¬(eᵢ.isLeaf) then
 4:              String_{eᵢ} ← toString(eᵢ)
 5:              if hash-table.contains( String_{eᵢ}) then
 6:                  aux ← hash-table.get( String_{eᵢ})
 7:              else
 8:                  if eᵢ is relational then
 9:                      e_r ← ¬(eᵢ)
10:                      String_R ←toString(e_r)
11:                      if hash-table.contains(String_R)  then
12:                          aux ← hash-table.get(String_R) ;
13:                          hash-table.add(String_{eᵢ},¬aux)
14:                          E.replaceChildWith(eᵢ,¬aux)
15:                      else
16:                          aux ← FLATTEN_NEG (eᵢ, true)
17:                          hash-table.add(String_{eᵢ}, aux)
18:                          E.replaceChildWith(eᵢ,aux)
19:                  else
20:                      aux ← FLATTEN_NEG (eᵢ, true)
21:                      hash-table.add(String_{eᵢ}, aux)
22:                      E.replaceChildWith(eᵢ,aux)
```

operator switch of the topmost node in the expression tree, which is again independent of the number of subexpressions k in the respective expression, and thus lies in $O(1)$.

Inserting the respective complexities for $applic_r(\hat{k})$ and $cost_r(\hat{k})$ in Eq. 4.1 (that represents the additional effort to increase the number of identical subexpressions) yields:

$$O(n) * O(1) + O(n) * O(1) + O(\hat{k}n) + O(\hat{s}n) \;\; = \;\; O((\hat{k} + \hat{s})n) \tag{4.2}$$

In summary, the time complexity of the active negation reformulation lies in $O((\hat{k} + \hat{s})n)$.

Summary

In summary, the equivalence stemming from negation can be easily exploited during normalisation. Furthermore, actively searching for negated common subexpressions, as incorporated by the active negation reformulation, is a cheap reformulation approach that satisfies the three desired properties: both detection and reformulation are cheap and the reformulated representation is never worse than the original representation.

The active negation reformulation is implemented in the tool TAILORand experiments on a set of problem classes have shown that it can deliver a solving time speedup of a factor 10, additional to basic common subexpression elimination. We have also observed a slight

tailoring time overhead for particular problem classes when performing the active negation reformulation. However, the overhead arises mainly in cases where instances are enhanced and hence in most cases the slight overhead in tailoring time is compensated by a notable solving time speedup. For more details on the empirical analysis of the active negation reformulating, see Sec. 8.3.2. g

4.3.4 De Morgan's Law

De Morgan's Law is another equivalence property that we can exploit. As an example, consider the expressions $\neg A \vee \neg B$ and $\neg(A \wedge B)$, where $(\neg A \vee \neg B) \sim_{deMor} (\neg(A \wedge B))$ and thus the former can be rewritten into the latter representation.

De Morgan's law is applicable to both conjunctions and disjunctions and is 'moves' a negation either 'inside' or 'outside' the respective expression. Similar to negation, normalisation automatically transforms all expressions of the form $\neg(A \wedge B)$ into $\neg A \vee \neg B$ which corresponds to the negation normal form.

Active De Morgan's Law Reformulation

However, similar to the active negation reformulation, we can apply an active form of De Morgan's reformulation, where normalised expressions of the form $\neg A \vee \neg B$ are reformulated to $\neg(A \wedge B)$ whose subexpression, $(A \wedge B)$ can be checked for a common subexpression in the instance.

For illustration, consider the example below

```
(A /\ B) => C,
(!A \/ !B) <=> D
```

The example contains two subexpressions, $(A \wedge B)$ and $(\neg A \vee \neg B)$, where the former corresponds to the negation of the latter, which becomes evident as soon as $(\neg A \vee \neg B)$ is reformulated to $\neg(A \wedge B)$. After this reformulation, the active negation reformulation (Alg. 4.4) can detect the equivalence and (in case the solver allows negation of arguments), $(A \wedge B)$ and $(\neg A \vee \neg B)$ could be represented by the same auxiliary variable:

```
(A /\ B) <=> aux,
 aux   =>  C,
!aux <=> D
```

More formally, the active De Morgan reformulation performs the following steps whenever a to-be-flattened subexpression e_i has no common subexpression:

1. if De Morgan's law is applicable to e_i, i.e. if e_i is a conjunction or disjunction whose arguments are negatable, goto 2. else continue flattening

Algorithm 4.5 Active De Morgan Reformulation. Excerpt of the recursive procedure FLATTEN_MORGAN (E,*flatten2Aux*) which is based on the generic flatten+reformulation procedure from FLATTEN_REF (Alg. 4.3). Changes are given in red font.

```
 1: if ¬ (all of E's children are leaves) then
 2:     for all eᵢ ∈ children(E) do
 3:         if ¬(eᵢ.isLeaf) then
 4:             String_{eᵢ} ← toString(eᵢ)
 5:             if hash-table.contains( String_{eᵢ}) then
 6:                 aux ← hash-table.get( String_{eᵢ})
 7:             else
 8:                 if eᵢ applicable to De Morgan then
 9:                     eᵣ ← deMorgan(eᵢ).getNegatedArg()
10:                     String_R ←toString(eᵣ)
11:                     if hash-table.contains(String_R)  then
12:                         aux ← hash-table.get(String_R) ;
13:                         hash-table.add(String_{eᵢ},¬aux)
14:                         E.replaceChildWith(eᵢ,¬aux)
15:                     else
16:                         aux ←  FLATTEN_MORGAN (eᵢ, true)
17:                         hash-table.add(String_{eᵢ}, aux)
18:                         E.replaceChildWith(eᵢ,aux)
19:                 else
20:                     aux ←  FLATTEN_MORGAN (eᵢ, true)
21:                     hash-table.add(String_{eᵢ}, aux)
22:                     E.replaceChildWith(eᵢ,aux)
```

2. reformulate e_i using the reformulation exploiting De Morgan's law. This results in a negation of a conjunction or disjunction, respectively (since all expressions have been normalised to negation normal form and hence De Morgan's Law can only be applied in one way). Retrieve the negation argument and store it as e_r.

3. Generate the String representation *String$_R$* of e_r

4. Check the hash-table for an occurrence of *String$_R$*

5. if successful, retrieve the corresponding auxiliary variable aux, otherwise continue flattening.

6. add an entry to the hash-table for $e_i \longrightarrow \neg aux$

7. replace e_i with $\neg aux$.

The Active De Morgan Algorithm

The reformulation algorithm is summarised in Alg. 4.5 where extensions/alterations to Alg. 4.3 are given in red font. Note, that the differences are marginal compared to the negation reformulation.

In order to determine the runtime complexity, we define the complexity of the applicability check, $applic_r(\hat{k})$, and the reformulation cost, $cost_r(\hat{k})$. First, the applicability check determines if the expression is a disjunction or conjunction and of its arguments are negatable.

In the worst case, this check involves all k subexpressions of the corresponding expression and therefore lies in $O(\hat{k})$, where $\hat{k}$ is the maximal number of subexpressions in any expression of the instance.

Second, the reformulation cost: applying De Morgan's law requires to negate all arguments of the corresponding conjunction/disjunction which in the worst case again affects the whole expression tree and therefore lies in $O(\hat{k})$.

Inserting the respective complexities for $applic_r(\hat{k})$ and $cost_r(\hat{k})$ in Eq. 4.1 (that represents the additional effort to increase the number of identical subexpressions) yields:

$$O(n) * O(\hat{k}) + O(n) * O(\hat{k}) + O(\hat{k}n) + O(\hat{s}n) \quad = \quad O((\hat{k} + \hat{s})n) \qquad (4.3)$$

Thus, the time complexity of the active De Morgan reformulation lies in $O((\hat{k} + \hat{s})n)$.

Summary   We can exploit De Morgan's law in order to reduce equivalent expressions that are not identical. The first reformulation, that rewrites a negation of a disjunction/conjunction to a conjunction/disjunction of negated arguments, like $\neg(A \vee B) \longrightarrow \neg A \wedge \neg B$, is performed during normalisation that transforms each Boolean expression into Negation Normal Form. The second reformulation (the inverse of the former), can be used in order to actively search for negated common subexpressions. For instance, in the example above, $\neg A \wedge \neg B$ can be reformulated to $\neg(A \vee B)$ and if $(A \vee B)$ has been previously flattened to aux, then $\neg aux$ can replace $\neg A \wedge \neg B$. The active De Morgan approach is a fairly cheap approach where both detection and reformulation are cheap and the resulting representation is not worse than the original one.

The active De Morgan approach has been implemented in the tool TAILORand empirically evaluated on a set of problems, which is further outlined in Sec. 8.3.4. In this study we have seen that attempting the De Morgan reformulation does not add significant overhead, however, only very few problems apply to the reformulation that, in these few cases, does not deliver very impressive speedups. To conclude, in practice, we have not seen any problems where the De Morgan approach provides particular benefits, however performing it during tailoring is cheap and therefore it is one of the standard optimisation techniques in TAILOR.

4.3.5   Horn Clauses

Another equivalence to consider is that between Horn Clauses and implications. A Horn Clause is a disjunction of literals where at most one literal is positive, i.e. the disjunction can be expressed as an implication. As an example, consider the expression $\neg A \vee B \vee \neg C$ which is a Horn clause and is hence equivalent to the implication $(A \wedge C) \Rightarrow B$, denoted $\neg A \vee B \vee \neg C \sim_{horn} (A \wedge C) \Rightarrow B$. Hence $\neg A \vee B \vee \neg C$ can be reformulated into $(A \wedge C) \Rightarrow B$ and vice versa. There are two different cases to consider: first, reformulating a

disjunction that is a Horn Clause into an implication. Second, reformulating an implication into a disjunction.

Horn Clauses during Normalisation

We include the first reformulation into normalisation, where we reformulate every disjunction into a Horn Clause (if possible), which is then reformulated into an implication. The first step is achieved by reformulating the arguments of disjunctions so that each disjunction contains exactly one positive literal. This is often possible, by 'pushing' negation outside an expression, in particular if arguments are relational expressions. As an example, consider the disjunction $A \vee (x < y)$ where we can rewrite argument $(x < y)$ into $\neg(x \geq y)$, yielding the Horn Clause $A \vee \neg(x \geq y)$.

Note, that disjunctions can have several different representations as Horn Clauses. For instance, the expression $(x = y) \vee (z > y) \vee (z \neq x)$ can be represented by three different Horn Clauses, as illustrated below:

$$
\begin{aligned}
(x = y) \vee (z > y) \vee (z \neq x) \quad &\longrightarrow \quad \neg(x \neq y) \quad \vee \quad \neg(z \leq y) \quad \vee \quad (z \neq x) \\
&\longrightarrow \quad \neg(x \neq y) \quad \vee \quad (z > y) \quad \vee \quad \neg(z = x) \\
&\longrightarrow \quad (x = y) \quad \vee \quad \neg(z \leq y) \quad \vee \quad (z \neq x)
\end{aligned}
$$

In our normalisation procedure, we pick the argument, that is largest with respect to our expression ordering to be the positive literal. Hence, in the example above, we would pick the first representation, since disequality '$\neq$' is the 'largest' operation according to our order. Note, that this (more or less random) choice is currently not beneficial (see our empirical analysis in Sec. 8.3.3) and has to be further enhanced by considering all combinations of arguments in the disjunction, which is an item of future work. For further details on common subexpressions stemming from arguments of disjunctions, see Sec. 4.4.2.

Since all disjunctions that can be reformulated as Horn Clauses are normalised into disjunctions, it is useless to attempt the inverse reformulation (reformulating implications into Horn Clauses) in order to find common subexpressions. However, similar to negation and De Morgan's Law, we can use the reformulation from implication to disjunction in an active manner in order to match common subexpressions, which we denote the *active Horn Clause reformulation*.

Active Horn Clause Reformulation

First, we want to illustrate the active Horn Clause reformulation on a simple example

```
(x < y) \/ A,
(x >= y) => B
```

where '`x`' and '`y`' are integer variables and '`A`' and '`B`' are Booleans. The normalisation step transforms the disjunction into an implication (choosing ('`x < y`') as positive literal, since it is largest according to the ordering), which yields:

```
!A        => (x < y),
(x >= y) =>  B
```

The active Horn clause reformulation aims at detecting a common subexpressions for the lefthand side expression of each implication: when flattening the second implication, the active Horn clause reformulation checks for a common subexpression of the negation of '`(x >= y)`', i.e. $(x < y)$, since $(x >= y) \Rightarrow B$ is equivalent to $(x < y) \vee B$. Since '`(x < y)`' appears in the first constraint, '`(x >= y) => B`' can be reformulated to '`(x < y) / B`', which yields the flat constraints:

```
$ using the active Horn Clause reformulation $
(x < y) <=> aux,
!A     => aux,
aux \/  B
```

Note, that the active negation reformulation would be successful as well in this example, but would result in a negated auxiliary variable, as illustrated below:

```
$ using the active negation reformulation $
(x < y) <=> aux,
!A     => aux,
!aux =>  B
```

Furthermore, there is another option: if the Horn Clause normalisation (that reformulates all Horn clauses into implications) would pick '`A`' as positive literal, then this would yield different normalised constraints:

```
(x >= y) => !A,
(x >= y) =>  B
```

In this case, standard common subexpression elimination will replace the two occurrences of '`(x >= y)`' with the same auxiliary variable:

```
$ standard CSE+ different Horn clause normalisation  $
(x >= y) <=> aux,
aux =>   !A,
aux =>   B
```

Hence, this example demonstrates the effects of the choice of positive literal when reformulating Horn Clauses into implications during normalisation, in particular, that this choice needs to be refined in order to provide an efficient reformulation.

The Active Horn Clause Algorithm

We now consider the algorithm for active Horn Clause reformulation, that attempts to reformulate implications into disjunctions. The algorithm is summarised in Alg. 4.6 which

Algorithm 4.6 Horn Clause Reformulation. Excerpt of the recursive procedure FLAT-TEN_HORN (E,*flatten2Aux*) which is based on the generic flatten+reformulation procedure from FLATTEN_REF (Alg. 4.3). Changes are given in red font.

```
 1:  if ¬ (all of E's children are leaves) then
 2:      for all e_i ∈ children(E) do
 3:          if ¬(e_i.isLeaf) then
 4:              String_{e_i} ← toString(e_i)
 5:              if hash-table.contains( String_{e_i}) then
 6:                  aux ← hash-table.get( String_{e_i})
 7:              else
 8:                  if e_i is an implication then
 9:                      e_r ← ¬((e_i).getLeftArg())
10:                      String_R ←toString(e_r)
11:                      if hash-table.contains(String_R) then
12:                          aux_1 ← hash-table.get(String_R) ;
13:                          e_{right} ← e_i.getRightArg()
14:                          e_i ← aux_1 ∨ e_{right}
15:                      aux ← FLATTEN_HORN (e_i, true)
16:                      hash-table.add(String_{e_i}, aux)
17:                      E.replaceChildWith(e_i,aux)
18:                  else
19:                      aux ← FLATTEN_HORN (e_i, true)
20:                      hash-table.add(String_{e_i}, aux)
21:                      E.replaceChildWith(e_i,aux)
```

is based on the generic reformulation algorithm in Alg. 4.3 - alterations are denoted in red font. The alterations are as follows: if subexpression e_i has no common subexpression, proceed as follows:

1. if e_i is an implication, goto2., otherwise continue flattening
2. negate the left-hand argument of the implication and store it in e_r.
3. transform e_r into its String representation $String_r$
4. check for a common subexpression of e_r, if successful, goto 5., otherwise continue flattening
5. retrieve the corresponding auxiliary variable of e_r, denoted aux_1
6. retrieve the right-hand side argument of the implication, e_{right} and create a new expression e_i, defined as $aux_1 \vee e_{right}$ and continue flattening.

For a runtime analysis, we define the complexity of the applicability check, $applic_r(\hat{k})$, and the reformulation cost, $cost_r(\hat{k})$. First, the applicability check determines if an expression is an implication. This is a simple test that is independent of the number of nodes in the corresponding expression tree and hence in $O(1)$.

Second, the reformulation cost: applying Horn Clause reformulation requires to (1) retrieve the left-hand side argument of the implication, (2) negate the argument and finally, in case the CSE-test is successful, create a new expression by (3) retrieving the right-hand side argument of the implication and (4) constructing a new disjunction. Retrieving the impli-

cation arguments, (1) and (4), lie in $O(1)$, since retrieving a subtree from a binary node is independent of the tree size (since we assume atomic operations). Negating the argument, (2), has a runtime of $O(k)$ where k is the number of subexpressions in the tree (i.e. number of nodes and leaves), and thus lies in $O(\hat{k})$ where $\hat{k}$ is the largest number of subexpressions in any expression of the instance. Finally, (4), constructing a new binary disjunction out of two existing subtrees lies again in $O(1)$. Hence, in summary, the reformulation cost lies in $O(1) + O(1) + O(\hat{k}) + O(1) = O(\hat{k})$.

Inserting the respective complexities for $applic_r(\hat{k})$ and $cost_r(\hat{k})$ in Eq. 4.1 (that represents the additional effort to increase the number of identical subexpressions) yields:

$$O(n) * O(1) + O(n) * O(\hat{k}) + O(\hat{k}n) + O(\hat{s}n) \quad = \quad O((\hat{k} + \hat{s})n) \qquad (4.4)$$

Thus, the time complexity of the active Horn Clause reformulation lies in $O((\hat{k} + \hat{s})n)$.

Summary We can exploit the equivalence between Horn clauses and implications in order to create further identical subexpressions. The equivalence is exploited during normalisation and flattening: first, during normalisation, all disjunctions are reformulated into Horn clauses which are then reformulated into implications. Second, during flattening of an implication, its left argument is negated and checked for a common subexpression. If this check is positive, the resulting auxiliary variable can be combined in a disjunction with the right argument of the implication.

Note, that this approach is *not* complimentary to the active negation reformulation, which also detects the negated common subexpression of the left hand side argument. However, the active Horn Clause reformulation uses the auxiliary variable *without* negating it, as opposed to the active negation reformulation.

The active Horn Clause reformulation is implemented in TAILORand has been tested on a selection of problems (see Sec. 8.3.3). It fires only in one problem class, but the attempt adds no significant overhead. However, we observe a solving time increase of about 40% from the active Horn Clause reformulation, which we presume results from the immatureness of our reformulation from disjunction to Horn Clauses. However, we expect an improvement of solving performance after enhancing the approach in order to create the 'best' set of common subexpressions.

In summary, the active Horn Clause reformulation is a promising reformulation in addition to the active negation reformulation. However, at its current state, it is impractical and there is still room for improvement: there are several choices on how to reformulate a disjunction into a Horn Clause and the 'best' choice is expected to depend on other subexpressions in the problem instance. Investigating this issue is an important item of future work.

4.3.6 Distributivity

The law of distributivity can be exploited to create expressions that are likely to match other subexpressions. For instance, representing $x * y + x * z$ with the equivalent representation $x * (y + z)$ in order to match further occurrences of $x * (y + z)$ (or its subexpression, $(y + z)$).

As noted earlier, a reformulation is beneficial, if the chosen representation is not worse than the original representation. This is, however, not always clear and depends on many different factors. Consider again the example above: in many cases, $x * (y + z)$ is expected to provide better propagation than $x * y + x * z$. However, other subexpressions in the instance can play an important role: what if the instance contains several occurrences of $x * y$ and $x * z$ but no further occurrence of $(y + z)$ - in that case the latter representation might be preferable. Evidently, the preferred representation cannot be determined in a local manner (i.e. without considering the rest of the constraint instance). Therefore, applying the distributivity reformulation requires further investigations that are part of our future work.

4.4 The Scope of CSE

As noted before, our basic technique of CSE (Alg. 4.2) does not eliminate all possible common subexpressions in a constraint instance, but only handles *identical* subexpressions that are equivalent according to their syntax. By performing measures like normalisation or active reformulations, as discussed in the previous subsection, we can still detect a fair amount of *equivalent* subexpressions by reducing them to *identical* subexpressions. However, there are still many kinds of equivalent subexpressions that we *could* detect at instance level, but choose not to. In the following, we discuss two particular families of common subexpressions that we neither detect nor eliminate and explain our choice.

4.4.1 Matching Global Constraints

A powerful family of common subexpressions that we do not generally eliminate at instance level is that of matching global constraints with their equivalent, decomposed representation. For instance, detecting a maximal clique of disequalities to replace with the global *alldifferent* constraint [62, 33]. This procedure is NP complete and therefore not feasible to be integrated into tailoring. However, when tailoring whole problem *classes* instead of single instances, these expensive detections are feasible in a restricted manner, since they can be highly beneficial (as demonstrated, for instance, with the Golomb Ruler Problem [77]). More details on problem class optimisations will follow in Chapter 6.

4.4.2 Common Subexpressions in n-ary Arguments

Another interesting family of common subexpressions is that of common subexpressions in arguments of n-ary operators (note, that we consider binary associative and commutative operators, like $\wedge$ or $+$, as n-ary). Expressions that are constructed from n-ary commutative and associative operators can share common subexpressions, that we denote 'argument-CS' for brevity. For instance, consider Example 4.4.1 where the conjunctions in the two constraints share the arguments A and C, hence they share the subexpression $A \wedge C$.

Example 4.4.1. Common subexpression $A \wedge C$ in the arguments of two conjunctions

```
  A /\ B /\ C  => D,
 (A /\ C) \/ E
```

The argument-CS $A \wedge C$ in Example 4.4.1 can be eliminated by introducing an auxiliary variable 'Aux' to replace $A \wedge C$, as illustrated below:

```
 Aux <=> A /\ C,
 Aux /\ B => D,
 Aux \/ E
```

The Cost of argument-CS Elimination

The CSE-flattening algorithm from Sec. 4.2.1 does not detect and eliminate argument-CSs. The reason for this is that CSE-flattening checks each argument *separately* and does not look for *common subsets* of arguments. The CSE-approach using Directed Acyclic Graphs(DAG) [71, 6] eliminates argument-CSs by disjoining the nodes of the DAG. This approach can be integrated into CSE-flattening by testing every possible argument combination for a common subexpression. However, this (naive) approach would add significant overhead to the algorithm: testing every possible argument subset (whose cardinality is greater than 1) for a common subexpression requires $2^n - (n + 1)$ additional checks for every n-ary commutative associative expression in the constraint instance. Furthermore, the hash-table would need to store all those argument combinations. This can result in a vast overhead, since in practice, n can be very large (e.g. in sums or conjunctions). Another approach would be to find a new data structure that replaces the hash-table and which can perform the matching steps using set operations. Such an approach would be far cheaper in computational cost to detect argument-CS than the naive approach using the hash-table. However, it would require to perform the standard CSE approach using the same datastructure, which would have to perform adding- and retrieving-operations in constant time in order to compete with the hash-table approach. This difficulty is one reason why we have not included the detection and elimination of argument-CSs into CSE-flattening at present.

The Benefits of Eliminating Argument-CSs

From our experimental evaluation, we observe that argument-CS elimination is beneficial in only very restricted cases: manually eliminating argument-CSs has often no effect, neither on solving time, nor search space. However, there are cases where argument-CS elimination is beneficial. For illustration, consider the constraint instance in Example 4.4.2 where the two sums share the common arguments '`x + y + z`'.

Example 4.4.2. Argument CS in two sums

```
  find x,y,z : int(−5..5)
  find t,u,v : int(1)
  find w : int(2)
such that
  x + y + z + t = v,
  x + y + z + u = w
```

In this example, propagation will not fire and thus not detect unsatisfiability. However, if the argument-CS is eliminated using a variable '`aux`', as illustrated in Example 4.4.2 below, then propagation would be triggered in the sums '`aux + t = v`' and '`aux + u = w`', since in the former case '`aux`' is assigned '`1`' and in the latter '`aux`' is assigned '`0`'.

Example 4.4.3. Eliminated argument-CS in two sums

```
  find x,y,z : int(−5..5)
  find t,u,v : int(1)
  find w : int(2)
  find aux : int(−15..15)
 such that
  x + y + z = aux,
  aux + t = v,
  aux + u = w
```

Obviously, eliminating the argument-CS is highly beneficial in this case, since variables '`t`', '`u`', '`u`' and '`w`' have small domains so propagation immediately derives unsatisfiability. However, in our experience, argument-CS elimination is beneficial mainly in constructed cases, like the one above, which we have not (so far) encountered in 'real' constraint instances, which leads us to the analysis of argument-CS in practice.

Argument-CS in Practical Examples

Our study of instance optimisations has been mainly driven by performing manual enhancement on a large selection of different problem classes (drawn from both experts and novices). From this process we have gained a certain extent of experience on what kind of redundancies typically appear in constraint instances. During our investigations, the only

problem class model in which we found argument-CS, was that of 'Plotting' (see Sec. 7.3.4 for a detailed discussion of the problem class). Interestingly, the respective argument-CS are of a particular kind that can be eliminated by slightly extending CSE-flattening.

More specifically, Plotting contains a constraint that we simplify (for clarity) as follows:

```
forall a:int(1..uba) .
    forall b:int(1..ubb).
        (A(a) /\   exists c:int(1..b) .
                        B(a,c))
                =>
                    C(a,b)
```

Expression 'A(a)' denotes a Boolean expression 'A' that is quantified by quantifying variable 'a'. Note that the upper bound of the existential quantification is limited by the quantifying variable 'b'. Unrolling the quantification reveals the argument-CS, therefore, we set 'uba=1' and 'ubb=4' and unroll the quantification:

```
A(1) /\    B(1,1)                                => C(1,1),
A(1) /\   (B(1,1) \/ B(1,2))                     => C(1,2),
A(1) /\   (B(1,1) \/ B(1,2) \/ B(1,3))           => C(1,3),
A(1) /\   (B(1,1) \/ B(1,2) \/ B(1,3) \/ B(1,4)) => C(1,4)
```

Evidently, the disjunctions conjoined with A(1) share arguments. Flattening-CSE cannot detect the argument-CS and would flatten the example to (simplified for clarity):

```
aux1 /\    aux2                         => aux3,
 (aux2 \/ aux4)                         <=> aux5,
aux1 /\   aux5                          => aux6,
 (aux2 \/ aux4 \/ aux7)                 <=> aux8,
aux1 /\   aux8                          => aux9,
 (aux2 \/ aux4 \/ aux6 \/ aux10)        <=> aux11,
aux1 /\   aux11                         => aux12
```

while elimination of argument-CS would yield:

```
aux1 /\    aux2                         => aux3,
 (aux2 \/ aux4)                         <=> aux5,
aux1 /\   aux5                          => aux6,
 (aux5 \/ aux7)                         <=> aux8,
aux1 /\   aux8                          => aux9,
 (aux8 \/ aux10)                        <=> aux11,
aux1 /\   aux11                         => aux12
```

Note, that argument-CSE does not save auxiliary variables but decreases the arity of the corresponding disjunction.

The kind of argument-CS from the example above has an important feature: the argument-CSs stem from a quantification and are 'extracted' while unrolling it. This is an important

point, since it allows us to make the following assumptions about two associative and commutative subexpressions A and B of the form $E_1 \otimes E_2 \otimes \cdots \otimes E_n$ that share this kind of argument-CS:

First, the arguments of A and B are ordered in the same way, starting with the arguments that are shared. This is a valuable feature, since it facilitates matching two expressions at String level: we can compare if expression A corresponds to the prefix of expression B.

Second, the number of (common) arguments strictly increases in each iteration (typically by 1), since quantifying domains are all *ascending*. A quantifying domains is *ascending* if in every iteration the value assigned to the quantifying variable(s) increases. *Descending* quantifying domains, such as $int(5..1)$ are considered the empty range and therefore never occur in quantifications that are unrolled. If quantifications are unrolled starting from the lower bound of the quantified domain, then we can conclude that if unrolled subexpression A has more arguments than unrolled subexpression B, then the first occurrence of B is *always before* the first occurrence of A. Therefore, the detection of these special kinds of argument-CS (by checking if a subexpression is the prefix of another subexpression) is confluent.

An Algorithm for Detecting Special Argument-CS

In summary, we can extend CSE-flattening to detect argument-CS stemming from quantifications by adding a set of steps to the flattening procedure FLATTEN_CSE (Alg. 4.2) which we informally summarise in the algorithm ARGUMENT_CS as follows:

ARGUMENT_CS $(E, bool)$

1. for each subtree e of E

    (a) if e has a common subexpression in the hash-table, replace e with the respective auxiliary variable aux and stop, otherwise goto (b)

    (b) if e consists of an AC operation with more than c arguments goto (c) , otherwise goto (f).

    (c) create the String representation e' of e, excluding the last i arguments of E, where i is the constant value for which the quantifying variable increases in every iteration (typically $i = 1$).

    (d) check if e' has an entry in the hash-table (i.e. a common subexpression) - if successful, goto (e), otherwise to (f).

    (e) replace the first $n - 1$ arguments of e with the corresponding auxiliary variable aux and goto (f) procedure

    (f) apply ARGUMENT_CS $(e, true)$ which returns auxiliary variable aux and replace e with aux in E

2. if *bool* is true, reify E to auxiliary variable aux and return aux, otherwise return E

This algorithm matches all argument-CSs that stem from quantifications of similar structure as in Plotting, if the quantifying domain increases by i. Since expressions with less arguments always first appear before those with more arguments, all argument-CS of this kind will be detected. From our empirical analysis, $c = 3$ has shown to be the most efficient choice.

The detection algorithm has been implemented in the tool TAILORand its effects have been tested on the Plotting example, further documented in Sec. 8.4. In the Plotting example, the enhancement has shown a notable benefit and reduced the overall solving time by about 50% for small instances and 25% for larger instances. Furthermore, it has not caused any significant overhead wrt tailoring time to perform the optimisation. Therefore, we conclude that there is no overall penalty for removing redundancies stemming from these special argument-CS and therefore include this measure to the set of optimisations that are performed by default during tailoring in TAILOR.

Conflicting Common Subexpressions

Another difficulty can arise with argument-CS: *conflicting* common subexpressions [6]. Two common subexpressions E_1 and E_2 are in conflict if $E_1 \cap E_2 \neq \emptyset$ but $E_1 \nsubseteq E_2$ and $E_2 \nsubseteq E_1$. For instance, consider Example 4.4.4, which is a slight alteration of Example 4.4.2 that contains the conflicting common subexpressions 'x + y + z' and 'x + y + t'.

Hence, the elimination of one subexpression prevents eliminating the other. In the case of conflicting common subexpressions, one has to choose which CS is best to select. In the context of numerical CSPs, Araya *et al* [6] propose a technique to eliminate those CS that have the most occurrences. However, it is not clear if this approach is best for finite integer domain CSPs, since some common subexpressions might dominate others in terms of propagation but might not have the most occurrences. For instance, in Example 4.4.4 from above, subexpression 'x+y' has most occurrences, but eliminating 'x+y' would not improve propagation. Hence, determining which common subexpression to eliminate from a set of conflicting CS is complex and requires a detailed investigation, including propagation behaviour.

Example 4.4.4. Conflicting common subexpressions

```
   find x,y,z,s,r : int(-5..5)
   find t,u,v : int(1)
   find w : int(2)

such that
   x + y + z + t = v,
   x + y + z + u = w,
   x + y + t + s = r
```

Note, that in our restricted approach of argument-CS, conflicting common subexpressions are not an issue, since we still compare subexpressions according to their syntax and do not test several combinations of arguments. Furthermore, conflicting common subexpressions *only* occur in n-ary commutative associative expressions, i.e. in the form of argument-CS, and are hence not an issue during CSE-flattening, as described in the previous subsections.

Summary

In summary, the decision of whether it is worthwhile to eliminate *all* argument-CS is difficult. It is not clear if the computational effort into detecting and eliminating *all* argument-CS would pay off in practice. We propose a detection algorithm for a special kind of argument-CS that occurs in practical examples. This algorithm is implemented in TAILORand from our respective empirical analysis (Sec. 8.4), we see that argument-CS elimination can provide a benefit.

From our experience with enhancing problem classes, it seems that general argument-CS elimination provides notable improvements in only in very restricted cases, that hardly occur in practical examples. Therefore, for now, we choose not integrate a general argument-CS elimination procedure into tailoring. However, there are interesting open questions (such as the question of how to deal with conflicting argument-CS) hence further investigation of argument-CS is part of our future work.

4.5   Eliminating Redundant Constraints

In this work, we call a constraint *redundant*, if the problem solutions are the same with or without the constraint and the constraint has no effect on the solving procedure in terms of propagation. This is opposed to the notion of an *implied* constraint, which we define as a constraint where the problem solutions are the same with or without the constraint, but the constraint adds further propagation. Evidently, *implied* constraints are beneficial, whereas redundant constraints are not. As an example, consider the two constraints $x + y \leq 5$ and $x + y \leq 10$ where the latter constraint is redundant since it does not add any further information than the former. Often, redundant constraints are a result of poor modelling from inexperienced users.

There exist many kinds of redundant constraints and detecting and eliminating *all* redundant constraints in a problem instance requires actually solving the problem and analysing propagation, which is infeasible in the context of tailoring. Therefore, we restrict the elimination of redundant constraints to special cases that often occur in practice: duplicate constraints.

4.5.1 Duplicate Constraints

The most common cases of redundant constraints are duplicate constraints that arise when quantified expressions are poorly guarded. A *guard* B for an expression E is a Boolean expression that has to hold in order to enforce E. As an example consider the expression $B \Rightarrow E$, where E is enforced if B is true. Guards are often used in Boolean quantifications to restrict the number of expressions that the quantification yields. This is often done by enforcing restrictions on quantifying variables. For instance, consider the following naive model of the n-Queens problem [57] in Example 4.5.1, where each variable 'queens[i]' represents the column position of the ith queen on the chessboard. The second and third constraint (restricting the queens positioned in diagonals) use the guard '(i!=j)'.

Example 4.5.1. Naive n-Queens problem model with weak guards

```
    given    n:           int(1..)   $ number of queens $
    find     queens:      matrix indexed by [int(1..n)] of int(1..n)

such that
    $ all queens are positioned on different columns $
    alldifferent(queens),

    $ no two queens positioned on same NW–SE diagonal$
    forall i,j:int(1..n) .
        (i!=j) =>  queens[i]+i != queens[j]+j,

    $ no two queens positioned on same SW–NE diagonal$
    forall i,j:int(1..n) .
        (i!=j) =>  queens[i]-i != queens[j]-j
```

Unrolling the second constraint (concerning the NW-SE diagonals), yields the following set of disequality constraints:

```
    queens[1]+1 != queens[2]+2,        queens[1]+1 != queens[3]+3,
    queens[2]+2 != queens[1]+1,        queens[2]+2 != queens[3]+3,
    queens[3]+3 != queens[2]+2,        queens[3]+3 != queens[1]+1,
```

Note, that the list of constraints contains *duplicates*, since the guard in the quantification, (i!=j), is not as strict as possible. These 'weak' guards are the main reason for duplicate constraints in constraint models. For a more thorough discussion on redundancies stemming from guards, see Sec. 4.6.

Eliminating Duplicate Constraints

Duplicate constraints can be eliminated in two different ways. First, by analysing guards in quantifications and strengthen them if necessary. For instance, the guard in the quantification above can be strengthened to '$i<j$' to avoid duplicates, as illustrated below:

```
    forall i,j:int(1..3) .
        (i<j) =>  queens[i]+i != queens[j]+j
```

Reasoning over guards and the guarded expression will be discussed in the context of enhancing whole problem classes in Sec. 6.1.1.

At instance level, we consider an alternative, simpler approach to eliminate duplicates: testing two constraints for syntactical equivalence. Normalisation (see Sec. 3.3.1) assists us in this matter: since all constraints are *ordered*, duplicates are positioned *consecutively* in the ordered list of constraints. Therefore, duplicates can be detected by simply iterating once over the list of constraints and test consecutive constraints for equivalence and is thus performed in linear time wrt the number of constraints in the problem instance [28].

4.5.2   Benefits of Eliminating Duplicate Constraints

In practice, the benefit of removing duplicate constraints depends on their number: the more duplicate constraints are posted, the more propagation time increases without pruning domains.

We study the effects of duplicate constraints on two problem models containing weak constraints: the n-Queens model from Example 4.5.1, and a naive model of the Golomb Ruler Problem [77] with a weak guards in the first constraint, depicted in Example 4.5.2.

Example 4.5.2. Naive Golomb Ruler Problem model

```
    given n : int(1..)  $ number of ticks $
    find  ruler : matrix indexed by [int(1..n)] of int(0..n^2)

$ find a ruler with minimal length $
minimising ruler[n]
such that
     $ distances between all n ticks are distinct $
    forall i1,i2,i3,i4 : int(1..n).
        ((i1 > i2) /\ (i3 > i4) /\ (i2!=i4)) =>
            (ruler[i1] - ruler[i2] != ruler[i3] - ruler[i4]),

     $ monotonicity $
    forall i : int(1..n-1)  .
        (ruler[i] < ruler[i+1]),
```

The experiments were performed on a Mac Pro 4.2 with 8 GB RAM that contains 8 Quad-Core Intel Xeon 5500 series processors, each 2.26 GHz (note that hyper-threading was turned off), using v3.2.0 , Minion 0.9 and Gecode 3.2.2 with Gecode/FlatZinc 3.2.1 and a timeout of 20 minutes.
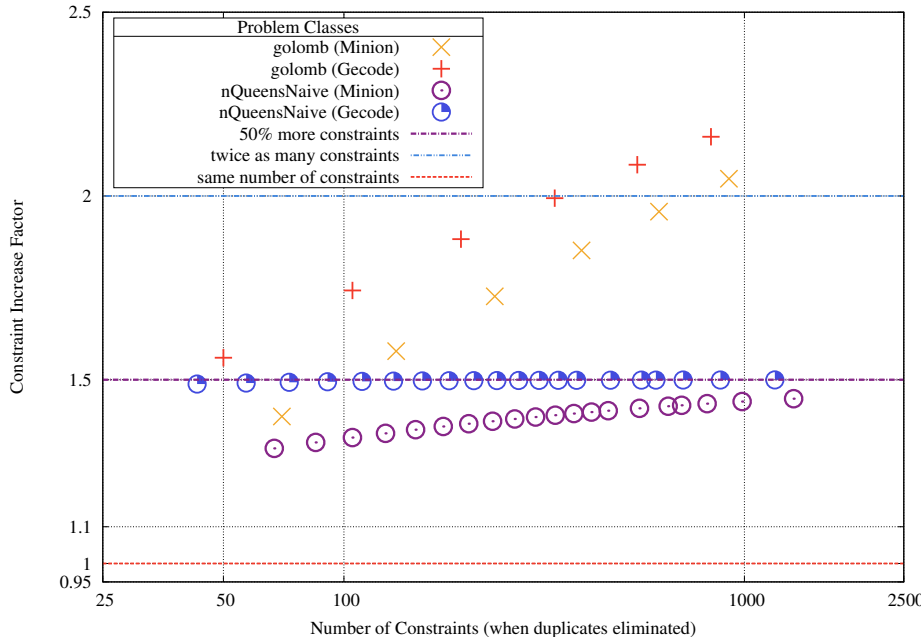
Figure 4.2: **Constraints increase** (with n) tested in **two different solvers** on naive models of the Golomb Ruler Problem (Example 4.5.2) and n-Queens (Example 4.5.1) that both contain **duplicate constraints** from weak guards. The y-axis represents the factor with which the number of constraints increases if the duplicate constraints are not eliminated. For example, the n-Queens instances with duplicates for solver Gecode contain 50% more constraints than those where duplicates are eliminated.

The Growth of Constraints with Increasing n

First, we consider the *growth* of constraints (duplicates) with increasing n in both problems. Fig. 4.2 illustrates the constraint growth for both models targeting two different solvers: Gecode [80] and Minion [32]. The y-axis gives the constraint increase factor between instances with duplicates and instances without. We can see that for the n-Queens problem, the number of duplicates remains fairly the same with increasing n, both for Gecode and Minion. In instances of the Golomb Ruler Problem, however, the number of duplicates stemming from the weak guard *increases* linearly with n. This means that the larger the instances, the larger the number of redundancies.

Effect on Solving Performance

Second, we consider the effects of duplicates on the solving performance. Fig. 4.3 illustrates the solving performance of instances that contain duplicates versus instances where duplicates are eliminated. The y-axis represents the solving time increase factor in the case when instances contain duplicates. For example, all Golomb instances solved in solver Gecode lie above $y = 2$, this means that all Golomb instances with duplicates have been

Figure 4.3: **Solving Performance** of naive models of the Golomb Ruler Problem (Example 4.5.2) and n-Queens (Example 4.5.1) that both contain **duplicate constraints** from weak guards, tested in **two different solvers**. The y-axis represents the factor with which the solving performance increases increases if the duplicate constraints are not eliminated. For example, many n-Queens instances with duplicates for solver Gecode is solved using 30% more time than those where duplicates are eliminated.

solved in more than twice the amount of time used to solve the Golomb instances without duplicates.

First, we observe that most instances with duplicates are solved using *more* time than those without duplicates (since most lie above $y = 1$). The exceptions are several n-Queens instances solved in Minion - in particular those that have been solved withing 0.005 and 0.1 seconds. Since all larger n-Queens instances for Minion are solved in *less* time without duplicates, the differences might stem from external factors, in particular since a 30% solving time difference is very little in the time frame of 0.005 and 0.1 seconds.

Second, we see that the Golomb Ruler instances with duplicates perform worse than the n-Queens instances with duplicates. A possible explanation would be that the Golomb Ruler instances are more severely affected by duplicate constraints, since they increase linearly with n.

Third, there is a notable difference with respect to the two constraint solvers. Duplicate constraints seem to have a far more negative effect in solver Gecode than in solver Minion. The reason for this difference is probably that propagation of constraints in Gecode is performed differently than in Minion.

Figure 4.4: **Constraints increase** (with n) tested with respect to **common subexpression elimination** on naive models of the Golomb Ruler Problem (Example 4.5.2) and n-Queens (Example 4.5.1) that both contain **duplicate constraints** from weak guards. The y-axis represents the factor with which the number of constraints increases if the duplicate constraints are not eliminated. For example, the n-Queens instances with duplicates tailored without CSE contain twice as many constraints than those where duplicates are eliminated (without CSE).

Duplicate Constraints and Common Subexpressions

Note, that for solver Minion, both problems contain common subexpressions which are eliminated. Out of curiosity, we also consider the impact of common subexpressions elimination (CSE) on the effect of duplicate constraints. Therefore, we compare instance size and solving time of instances that were generated without CSE. Note, that this is not an evaluation of CSE (since we do not compare enhanced instances with unenhanced instances).

First, we consider the growth of duplicates with n. In Fig. 4.4 we show the differences in constraint growth with CSE and without CSE: the y-axis shows constraint increase in the case when the instances contain duplicates. As expected, instances generated with CSE contain less constraints in general, but the constraint growth is the same with or without CSE - in Golomb Ruler, the number of duplicates still increases linearly with n.

Second, we consider the solving time of instances that were all tailored *without* CSE in Fig. 4.5. When comparing the solving times of instances generated without CSE (Fig. 4.5) with the solving times of instances generated with CSE (Fig. 4.3) we make the following interesting observation: duplicate constraints have a worse impact on solving performance if common subexpressions are not eliminated.
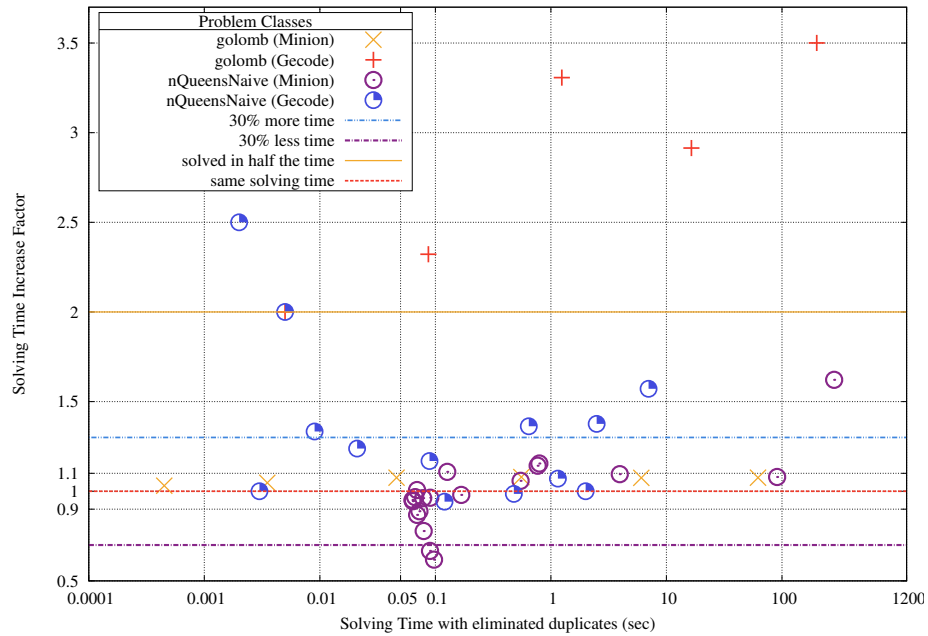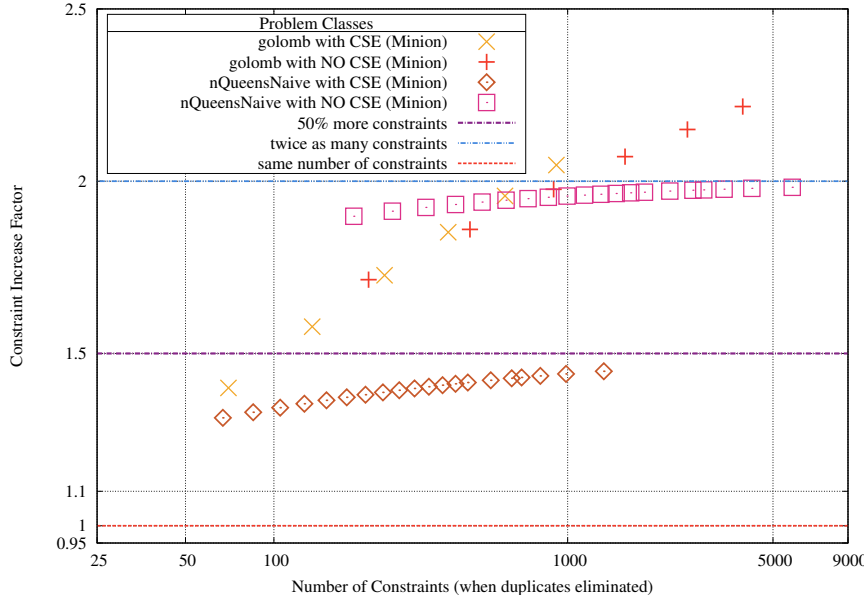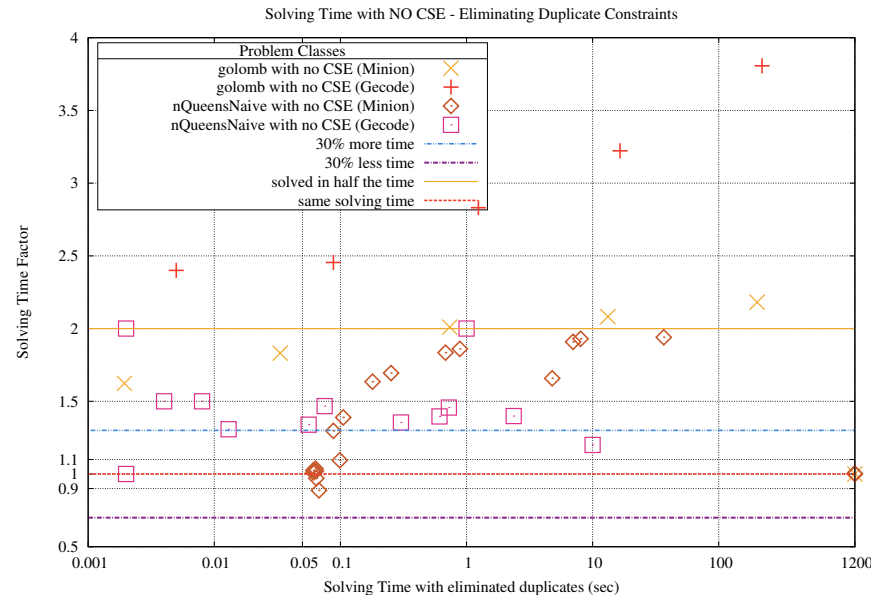
Figure 4.5: **Solving Performance** of naive models of the Golomb Ruler Problem (Example 4.5.2) and n-Queens (Example 4.5.1) that both contain **duplicate constraints** from weak guards, tested in **two different solvers** where **no common subexpressions** where **eliminated** in **all** instances. The y-axis represents the factor with which the solving performance increases increases if the duplicate constraints are not eliminated. For example, many n-Queens instances with duplicates for solver Gecode is solved using 30% more time than those where duplicates are eliminated.

Summary

In summary, we have seen that he practical benefits of eliminating duplicate constraints depend on two things: first, it depends on the respective problem class, where we observed a connection between the growth of duplicates wrt class parameters and the benefits of removing the duplicates. From this we propose the assumption that the benefits of removing duplicate constraints will increase if the duplicates increase with some parameter in the respective problem class. Second, the benefits of eliminating duplicate constraints depends on the target solver and the way the solver calls propagators. To conclude, removing duplicate constraints has shown to be beneficial in many cases, leading to speedups up to a factor of 3.5.

4.6 Quantification Optimisations

Quantifications are a powerful means to formulate a set of related expressions. We consider quantifications of the form

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

where $\varphi \in \{\forall, \exists, \sum\}$ is a quantifier that ranges over the set of quantifying variables $I = \{i_1, \ldots, i_n\}$, each defined over the finite range of integers $int(lb..ub)$ where $lb \leq ub$, and $E(i_1, \ldots, i_n)$ denotes an expression that is quantified over $\{i_1, \ldots, i_n\}$. Example 4.6.1 shows a set of sample quantifications. Note, that if φ represents a universal ($\forall$) or existential ($\exists$) quantifier, E_I has to be a Boolean expression. Apart from this restriction, the quantified expression E_I can be arbitrary (e.g. contain further quantifications, etc). For brevity, we will refer to quantifications as $\varphi_I.E_I$, since the exact quantifying domain and number of quantifying variables is not relevant in this discussion.

Example 4.6.1. Simple quantifications

```
forall i,j:int(1..10) .
    (i<j) => (x[i]+i != x[j]+j),

exists i,j:int(1..5) .
    (i<j) /\ (y[i] = y[j]),

10 = sum k:int(0..3). z[k]
```

Quantifications are a very powerful means to represent a set of constraints in a compact way. However, similarly to for-loops in program code, quantifications can include redundancies, in particular when formulated by novices. Typically, the negative effect of redundancies in quantifications increases with the length of the quantifying domain (combined with the number of quantifying variables), which can be vast in large instances. Therefore it is crucial to detect and eliminate redundancies in quantifications. In this section we investigate how to optimise quantifications in order to address redundancies that might arise from poor modelling. We begin by identifying frequent sources of redundancies in quantifications.

4.6.1 Sources of Redundancies in Quantifications

Weak Guards

A *guard* B for an expression E is a Boolean expression that has to hold in order to enforce E. As an example, consider the expression $B \Rightarrow E$, where E is enforced if B is true. Guards are often used in Boolean quantifications to restrict the number of expressions that the quantification yields. For instance, consider the guard '(i<j)' in the two Boolean quantifications in Example 4.6.1.

As the example illustrates, in universal quantification, guards are typically of the form $\forall_I B_I \Rightarrow E_I$ where every false B_I eliminates the corresponding E_I since *false*$\Rightarrow E$ is evaluated to *true* (which is the identity of conjunction, i.e. of $\forall$). In existential quantifications,

guards are typically of the form $\exists_I B_I \wedge E_I$ where every false B_I eliminates the corresponding E_I, since *false*$\wedge E$ is evaluated to *false* (which is the identity of disjunction, i.e. of $\exists$). If guards are too weak, they usually cause redundancies.

Definition 4.6.1. Weak Guards. A Boolean expression B_I guarding expression E_I in a Boolean quantification $\forall_I B_I \Rightarrow E_I$ or $\exists_I B_I \wedge E_I$ is called *weak guard*, if the conjunction (or disjunction, respectively) of the expressions *allowed* by guard B_I contains symmetric arguments with respect to commutative operators in E_I. In other words, a weak guard B_I allows at least two expressions $E_{I,2}$ and $E_{I,1}$ that are equivalent because of the commutativity of an operator in E_I.

Example 4.6.2. based on Example 4.6.1 with weak guard '(i!=j)'.

```
forall i,j:int(1..10) .
    (i!=j) => (x[i]+i != x[j]+j),

exists i,j:int(1..5) .
    (i!=j) /\ (y[i] = y[j])
```

Since weak guards do not eliminate the symmetry stemming from commutative operators, they yield duplicate expressions in the unrolled quantification. For illustration, consider Example 4.6.2, which is a slight alteration of Example 4.6.1, where the guard '(i<j)' has been replaced by the weak guard '(i!=j)'. Unrolling the quantification in Example 4.6.2, will result in duplicate constraints, like 'x[1]+1!=x[2]+2' and 'x[2]+2!=x[1]+1', since guard '(i!=j)' does not break the symmetry of commutative operator '!=', which guard '(i<j)' does.

In general, two different kinds of duplicates can arise from weak guards. First, if the weak guard B is a *constant* expression (like in Example 4.6.2), then duplicate *constraints* arise when unrolling the quantification. Otherwise, duplicate *subexpressions* arise after unrolling the quantification.

Both cases are covered by previously introduced optimisation techniques. Duplicate constraints are easily eliminated due to constraint ordering (Sec. 4.5), duplicate subexpressions are eliminated by common subexpression elimination (Sec. 4.2). In summary, redundancies from weak guards arise in form of duplicate expressions that are handled by previously discussed instance optimisation techniques.

Structural Reasons

In some cases, expressions involving quantifications can be reformulated into an equivalent representation that is generally more efficient. Our aim is to explore these equivalent representations and determine the most efficient one. More specifically, we consider reformulations in which expressions are 'moved' inside or outside the corresponding quantification. Naturally, only particular expressions can be moved in or outside a quantification

- expressions we denote *loop-invariant* (discussed in Sec. 4.6.2). The number of reformulations involving loop-invariant expressions is small and manageable for a case-wise analysis, which we conduct in Sec. 4.6.3.

4.6.2 Loop-invariant Expressions

Definition 4.6.2. *Loop-invariance*. If expression A in quantification $\varphi_I.A \oplus E_I$ is *not* quantified by any of the quantifying variables in I and there exists an operator $\oplus'$ such that

$$\varphi_I.A \oplus E_I \equiv A \oplus' \varphi_I.E_I \tag{4.5}$$

then we call A *loop-invariant*. In other words, a subexpression A in quantification $\varphi_I.A \oplus E_I$ is loop-invariant, if it can be moved outside the quantification while preserving the semantic of the constraint.

Example 4.6.3 shows a quantification where the guard, '(x=0)', is loop-invariant since $(x=0) \Rightarrow (\forall_i y[i]=i)$ is equivalent to $(\forall_i (x=0) \Rightarrow y[i]=i)$.

Example 4.6.3. A quantification containing the **loop-invariant expression** '(x=0)'

```
forall i:int(1..5) .
    (x=0)  => (y[i] = i)
```

In this section, we want to analyse the effect of loop-invariant expressions in quantifications and address redundancies they introduce. Loop-invariant expressions introduce redundancies in form of *common subexpressions* as soon as the corresponding quantification is unrolled. More specifically, every quantification $\varphi\ i_1, ..., i_n:int(lb..ub).\ A \oplus E_I$ is unrolled to

$$(A \oplus E_1) \oplus_\varphi (A \oplus E_2) \oplus_\varphi \cdots \oplus_\varphi (A \oplus E_{k-1}) \oplus_\varphi (A \oplus E_k)$$

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 if $A \oplus E_I$ is guarded: if unguarded, $k = n*(ub-lb+1)$, otherwise $n \le k \le n*(ub-lb+1)$. Evidently, the unrolled quantification contains k occurrences of the loop-invariant expression A. These multiple occurrences are *common subexpressions*, which, if not eliminated, can cause a significant increase in solving time, and in some cases, a significant increase in search space (see CSE experiments in Sec. 8.2). Common subexpressions can be easily eliminated, as discussed in Sec. 4.2.

We illustrate the effects caused by common subexpressions stemming from loop-invariant expressions on a simple example. Consider the unrolled set of constraints from Example 4.6.3, where the loop-invariant expression '(x=0)' has 5 occurrences:

```
(x=0)  => (y[1]=1),    (x=0)  => (y[2]=2),
(x=0)  => (y[3]=3),    (x=0)  => (y[4]=4),
(x=0)  => (y[5]=5)
```

If common subexpressions are *not* eliminated, the flat instance will contain redundancies:

```
(x=0) <=> aux1,        aux1 => (y[1]=1),
(x=0) <=> aux2,        aux2 => (y[2]=2),
(x=0) <=> aux3,        aux3 => (y[3]=3),
(x=0) <=> aux4,        aux4 => (y[4]=4),
(x=0) <=> aux5,        aux5 => (y[5]=5)
```

However, with common subexpression elimination, 'x=0' is represented by the same auxiliary variable 'aux', yielding:

```
(x=0) <=> aux,
aux => (y[1]=1),       aux => (y[2]=2),
aux => (y[3]=3),       aux => (y[4]=4),
aux => (y[5]=5)
```

In summary, we have seen that loop-invariant expressions in quantifications cause redundancies in form of common subexpressions, which can be effectively eliminated by common subexpression elimination (Sec. 4.2).

4.6.3 Moving Loop-invariant Expressions

In the previous subsection, we have seen that, thanks to common subexpression elimination (Sec. 4.2), loop-invariant expressions in quantifications do not impair the constraint instance per se. Therefore, we are interested if there are cases, where moving a loop-invariant expression *inside* a quantification (and performing CSE) is preferable to moving it outside. Hence, we conduct a thorough comparison between the two equivalent representations: the quantification where the loop-invariant expression is *inside* the quantification ('*inside*-representation'), and where the loop-invariant expression is *outside* the quantification ('*outside*-representation'). More formally, we compare $\varphi_I.A \oplus E_I$ with $A \oplus' \varphi_I.E_I$ and determine which representation is preferable for specific φ, $\oplus$ and $\oplus'$.

In order to conduct a proper analysis, we need to consider the *flat* representation of both expressions. The flat representation is obtained by *flattening* the expression to the propagators provided by the target solver, which typically involves introducing auxiliary variables and additional constraints (see Sec. 3.3.2 for more details). Flattening is a strictly solver-dependent procedure, however, considering every possible combination of propagators in the target solver is out of scope of this analysis. Therefore, we restrict our analysis to solvers that provide n-ary conjunction, n-ary disjunction and n-ary summation propagators, which holds for most constraint solvers. Furthermore, during flattening we need to take into account if the 'to-be-flattened' expression E is nested in another expression: if E is not nested in another constraint, E is flattened to a propagator (constraint), otherwise, if E is nested, e.g. in '$A \Rightarrow E$', E is flattened to an auxiliary variable. If an expression E is flattened to a constraint, it is denoted by $\underline{E}$, if E is flattened to an auxiliary variable, it is denoted in lowercase, i.e. e.

Note, that the reformulation to the respective *inside-* and *outside*-representation is implemented in TAILORand performed during preprocessing as part of normalisation. This means that every quantifier that contains a loop-invariant expression can be reformulated either into the inside or outside representation. An empirical analysis of the differences in loop-invariant representations can be found in the Chapter on Experiments, in Sec. 8.5. Note, however, that only one kind of quantification optimisation fired to the large set of problem classes we consider. The remaining optimisations have each been tested on constructed examples, but require a more thorough investigation in practical problems, which is an item of future work.

In summary, we compare the *inside*-representation with the *outside*-representation at flat level considering both the nested and unnested case, assuming that the target solver provides n-ary conjunction, disjunction and summation propagators. Note, that we assume that all common subexpressions that result from the *inside*-representation are eliminated. We start our investigation with the universal quantifier.

Moving Loop-invariant Expressions in Universal Quantifications

We consider universal quantifications $\forall_I E_I$ that can be combined with a loop-invariant expression A, such that

$$(\forall_I A \oplus E_I) \equiv A \oplus' (\forall_I E_I) \tag{4.6}$$

which holds for operators $\oplus = \oplus' = \wedge$, $\oplus = \oplus' = \vee$ and $\oplus = \oplus' = \Rightarrow$. In the following, we compare both representations for each operator. Note, that it not necessary to consider the case $(\forall_I E_I \oplus A) \equiv (\forall_I E_I) \oplus' A$ since all operators for which this equation holds (i.e. $\wedge$ and $\vee$) are commutative and associative and thus covered by Equation 4.6. In the following we consider all three cases and will see that the *inside*-representation can sometimes be preferable to the *outside*-representation.

Case 1: $(\forall_I A \wedge E_I) \equiv A \wedge (\forall_I E_I)$
Since $\forall$ corresponds to n-ary conjunction, which is commutative and associative, Equation 4.6 holds for $\oplus = \oplus' = \wedge$. The flat representation in the unnested and nested case are given in Tab. 4.1. Evidently, the *inside-* and *outside*-representation yield practically the same flat constraints (the only difference being a permutation of arguments). Hence we conclude that it makes no difference if the loop-invariant expression A is moved in- or outside the universal quantification with $\oplus = \oplus' = \wedge$.

Case 2: $(\forall_I A \vee E_I) \equiv A \vee (\forall_I E_I)$
The second case follows the law of distributivity of conjunction and disjunction, i.e. $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$. Tab. 4.2 shows the flat representations for *inside-* and *outside*-representation. First, we consider the unnested case. Moving the loop-invariant

$\forall$-**CASE 1**	*inside*-representation	*outside*-representation
Original	$(\forall_I A \wedge E_I)$	$A \wedge (\forall_I E_I)$
Unrolled	$(E_1 \wedge E_2 \wedge \cdots \wedge E_k \wedge A)$	$A \wedge (E_1 \wedge E_2 \wedge \cdots \wedge E_k)$
Flat (unnested)	$\underline{E_1}$ $\underline{E_2}$ $\cdots$ $\underline{E_k}$ $\underline{A}$	$\underline{A}$ $\underline{E_1}$ $\underline{E_2}$ $\cdots$ $\underline{E_k}$
Flat (nested)	$aux_q \Leftrightarrow (e_1 \wedge e_2 \wedge \cdots \wedge e_k \wedge a)$	$aux_q \Leftrightarrow (a \wedge e_1 \wedge e_2 \wedge \cdots \wedge e_k)$
Summary		
unnested	$k + 1$ constraints	$k + 1$ constraints
nested	1 constraint, 1 aux. variable	1 constraint, 1 aux. variables

Table 4.1: **Case 1** of comparing *inside*- and *outside*-representation.

expression A *inside* the quantification, yields k binary constraints, while moving A *outside* the quantification yields one k-ary and one binary constraint, introducing one Boolean auxiliary variable. Obviously, the number of constraints of the *inside*-representation increases linearly with k while the number of constraints of the *outside*-representation is constant and only one constraint's arity increases linearly with k. The choice of which representation is preferable is not obvious in the unnested case. In an empirical investigation on some constructed examples we have not observed a difference in the performance (wrt solving time and search space) of both representations.

In the nested case, the *outside*-representation yields a constant number of constraints with respect to k: 2 constraints and 2 auxiliary variables, where only the constraint arity increases with k. On the other hand, with the *inside*-representation, the number of constraints and auxiliary variables increases linearly with k. Therefore, the *outside*-representation is preferable in the nested case for $\oplus = \oplus' = \vee$.

Note, that if the loop-invariant expression A is a constant expression, then both representations are the same: if A evaluates to *true*, then both $(\forall_I \text{ } true \vee E_I)$ and $true \vee (\forall_I E_I)$ evaluate to *true*. Otherwise, if A is false, both $(\forall_I \text{ } false \vee E_I)$ and $false \vee (\forall_I E_I)$ evaluate to $\forall_I E_I$.

$\forall$-**Case 3:** $(\forall_I A \Rightarrow E_I) \equiv A \Rightarrow (\forall_I E_I)$
The third and last case considers $\oplus = \oplus' = \Rightarrow$, following from the fact that $A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C)$. Tab. 4.3 depicts the flat representations for the unnested and nested case. We start with the unnested case.

Note, that in the unnested case, the flat *inside*-representation has an alternative representa-

$\forall$-**CASE 2**	*inside*-representation	*outside*-representation
Original	$(\forall_I A \vee E_I)$	$A \vee (\forall_I E_I)$
Unrolled	$(A \vee E_1) \wedge \cdots \wedge (A \vee E_k)$	$A \vee (E_1 \wedge E_2 \wedge \cdots \wedge E_k)$
Flat (unnested)	$a \vee e_1$ $a \vee e_2$ $\cdots$ $a \vee e_k$	$aux \Leftrightarrow (e_1 \wedge e_2 \wedge \cdots \wedge e_k)$ $a \vee aux$
Flat (nested)	$aux_1 \Leftrightarrow (a \vee e_1)$ $aux_2 \Leftrightarrow (a \vee e_2)$ $\cdots$ $aux_k \Leftrightarrow (a \vee e_k)$ $aux_q \Leftrightarrow (aux_1 \wedge aux_2 \wedge \cdots \wedge aux_k)$	$aux_1 \Leftrightarrow (e_1 \wedge e_2 \wedge \cdots \wedge e_k)$ $aux_q \Leftrightarrow (a \vee aux_1)$
Summary		
unnested	0 aux. variables, k constraints	1 aux. variable, 2 constraints
nested	$k + 1$ aux. variables, $k + 1$ constraints	2 aux. variables, 2 constraints

Table 4.2: $\forall$-**Case 2** of comparing *inside*- and *outside*-representation.

tion, given in brackets '[' and ']', which we discuss separately below. The unnested case is similar to that in Case 2, where the more efficient representation is not straight forward. The *inside*-representation yields k binary constraints; the *outside*-representation yields one k-ary and one binary constraint, introducing one auxiliary variable. In our empirical evaluation the *inside*-representation clearly dominated the *outside*-representation in solving time on the same search space.

In the alternative *inside*-representation, we have $a \Rightarrow \underline{E_i}$ instead of $a \Rightarrow e_i$. Recall the difference between $\underline{E_i}$ and e_i: $\underline{E_i}$ corresponds to the flat *constraint* representing E_i, while e_i corresponds to the *auxiliary variable* representing $\underline{E_i}$. Hence, $\underline{E_i}$ introduces less overhead, since it does *not* introduce an additional auxiliary variable, as e_i does. Therefore, the alternative flat *inside*-representation, $a \Rightarrow \underline{E_i}$, introduces k fewer auxiliary variables than $a \Rightarrow e_i$, but also k fewer auxiliary variables than the corresponding *outside*-representation, which flattens all E_i to e_i. Note, that the alternative representation is only available for constraint solvers that provide a 'reify-imply' propagator of the form *var⇒propagator* where *propagator* is an arbitrary propagator (e.g. the solver MINIONsupports reify-imply). Surprisingly, in our experimental evaluation, the alternative *inside*-representation performs worse than the original *inside*-representation and slightly better than the *outside*-representation. In summary, in the unnested case, our empirical analysis (Sec. 8.5) suggests that the standard *inside*-representation is preferable for $\oplus = \oplus' =\Rightarrow$.

$\forall$-**CASE 3**	*inside*-representation		*outside*-representation
Original	$(\forall_I A \Rightarrow E_I)$		$A \Rightarrow (\forall_I E_I)$
Unrolled	$(A \Rightarrow E_1) \wedge \cdots \wedge (A \Rightarrow E_k)$		$A \Rightarrow (E_1 \wedge E_2 \wedge \cdots \wedge E_k)$
Flat (unnested)	$a \Rightarrow e_1$ $a \Rightarrow e_2$ $\cdots$ $a \Rightarrow e_k$	$[a \Rightarrow \underline{E_1}]$ $[a \Rightarrow \underline{E_2}]$ $[a \Rightarrow \underline{E_k}]$	$aux \Leftrightarrow e_1 \wedge e_2 \wedge \cdots \wedge e_k$ $a \Rightarrow aux$
Flat (nested)	$aux_1 \Leftrightarrow (a \Rightarrow e_1)$ $aux_2 \Leftrightarrow (a \Rightarrow e_2)$ $\cdots$ $aux_k \Leftrightarrow (a \Rightarrow e_k)$ $aux_q \Leftrightarrow (aux_1 \wedge aux_2 \wedge \cdots \wedge aux_k)$	$[aux_1 \Leftrightarrow (a \Rightarrow \underline{E_1})]$ $[aux_2 \Leftrightarrow (a \Rightarrow \underline{E_2})]$ $[aux_k \Leftrightarrow (a \Rightarrow \underline{E_k})]$	$aux_1 \Leftrightarrow e_1 \wedge e_2 \wedge \cdots \wedge e_k$ $aux_q \Leftrightarrow (a \Rightarrow aux1)$
Summary			
unnested	0 aux. variables k constraints [saves up to k flat. variables]		1 aux. variable 2 constraints,
nested	$k+1$ aux. variables $k+1$ constraints, [saves up to k flat. variables]		2 aux. variables 2 constraints,

Table 4.3: $\forall$-**Case 3** of comparing *inside*- and *outside*-representation. The alternative *inside*-representation is given in brackets '[' and ']', representing the special case when the solver provides a '*reify-imply*'-propagator.

In the nested case, the *outside*-representation yields a constant number of constraints with respect to k: 2 constraints and 2 auxiliary variables, where only the constraint arity increases with k. On the other hand, with the *inside*-representation, the number of constraints and auxiliary variables increases linearly with k. Therefore, the *outside*-representation is preferable in the nested case for $\oplus = \oplus' =\Rightarrow$.

Again, if loop-invariant A is a constant, then both the *inside*- and *outside*-representations are the same: if A evaluates to *false* then both representations evaluate to *true*. Otherwise, if A evaluates to *true*, both $(\forall_I \text{ true} \Rightarrow E_I)$ and $(true \Rightarrow (\forall_I E_I))$ evaluate to $\forall_I E_I$.

Loop-invariant Expressions in Existential Quantifications

Now we consider existential quantifications $\exists_I E_I$ that can be combined with a loop-invariant expression A, such that

$$(\exists_I A \oplus E_I) \equiv A \oplus' (\exists_I E_I) \tag{4.7}$$

This equation holds for the operators $\oplus = \oplus' = \vee$ and $\oplus = \oplus' = \wedge$. Note, that it not necessary to consider the case $(\exists_I E_I \oplus A) \equiv (\exists_I E_I) \oplus' A$ since all operators for which this equation holds, $\wedge$ and $\vee$, are commutative and associative, and thus covered by Equation 4.7. In the following two case studies we will see that the *outside*-representation is generally preferable or equivalent to the *inside*-representation.

$\exists$-**CASE 1**	*inside*-representation	*outside*-representation
Original	$(\exists_I A \vee E_I)$	$A \vee (\exists_I E_I)$
Unrolled	$(A \vee E_1 \vee E_2 \vee \cdots \vee E_k)$	$A \vee (E_1 \vee E_2 \vee \cdots \vee E_k)$
Flat (unnested)	$(e_1 \vee e_2 \vee \cdots \vee e_k \vee a)$	$(a \vee e_1 \vee e_2 \vee \cdots \vee e_k)$
Flat (nested)	$aux_q \Leftrightarrow (e_1 \vee e_2 \vee \cdots \vee e_k \vee a)$	$aux_q \Leftrightarrow (a \vee e_1 \vee e_2 \vee \cdots \vee e_k)$
Summary		
unnested	1 constraint	1 constraints
nested	1 aux. variable, 1 constraint	1 aux. variable, 1 constraint

Table 4.4: $\exists$-**Case 1** of comparing *inside*- and *outside*-representation.

Case 1: $(\exists_I A \vee E_I) \equiv A \vee (\exists_I E_I)$
Since $\exists$ corresponds to n-ary disjunction, which is commutative and associative, Equation 4.6 holds for $\oplus = \oplus' = \vee$. The flat representation in the unnested and nested case are given in Tab. 4.4. Evidently, the *inside*- and *outside*-representation yield practically the same flat constraints (the only difference being a permutation of arguments). Hence we conclude that it makes no difference if the loop-invariant expression A is moved in- or outside the existential quantification with $\oplus = \oplus' = \vee$.

Case 2: $(\exists_I A \wedge E_I) \equiv A \wedge (\exists_I E_I)$
The second case follows the law of distributivity of conjunction and disjunction, i.e. $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$. Tab. 4.5 shows the flat representations for *inside*- and *outside*-representation.

First, we consider the unnested case. Moving the loop-invariant expression A *inside* the quantification, yields k reification constraints and 1 disjunction, introducing k auxiliary variables, while moving A *outside* the quantification only yields 2 constraints without introducing any auxiliary variables. Since the *outside*-representation results in far less constraints and auxiliary variables, it is preferable to the *inside*-representation. Furthermore, propagation behaves differently in the two representations: in the *outside*-representation, A can be propagated straight away, however in the *inside*-representation, A is not propagated before the variables in the E_is are pruned such that the disjunction holds.

In the nested case, again the *outside*-representation dominates the *inside*-representation; the former using only 2 reification constraints and 2 auxiliary variables while the latter requires

∃-CASE 2	*inside*-representation	*outside*-representation
Original	$(\exists_I A \wedge E_I)$	$A \wedge (\exists_I E_I)$
Unrolled	$(A \wedge E_1) \vee \cdots \vee (A \wedge E_k)$	$A \wedge (E_1 \vee E_2 \vee \cdots \vee E_k)$
Flat (unnested)	$aux_1 \Leftrightarrow (a \wedge e_1)$ $aux_2 \Leftrightarrow (a \wedge e_2)$ $\cdots$ $aux_k \Leftrightarrow (a \wedge e_k)$ $(aux_1 \vee aux_2 \vee \cdots \vee aux_k)$	$\underline{A}$ $(e_1 \vee e_2 \vee \cdots \vee e_k)$
Flat (nested)	$aux_1 \Leftrightarrow (a \wedge e_1)$ $aux_2 \Leftrightarrow (a \wedge e_2)$ $\cdots$ $aux_k \Leftrightarrow (a \wedge e_k)$ $aux_q \Leftrightarrow (aux_1 \vee aux_2 \vee \cdots \vee aux_k)$	$aux_1 \Leftrightarrow (e_1 \vee e_2 \vee \cdots \vee e_k)$ $aux_q \Leftrightarrow (a \wedge aux_1)$
Summary		
unnested	k aux. variables, $k + 1$ constraints	0 aux. variables, 2 constraints
nested	$k + 1$ aux. variables, $k + 1$ constraints	2 aux. variables, 2 constraints

Table 4.5: ∃-**Case 2** of comparing *inside*- and *outside*-representation.

$k + 1$ reification constraints and $k + 1$ auxiliary variables.

Note, that if the loop-invariant expression A is a constant expression, then both representations are the same: if A evaluates to *true*, then both $(\exists_I \textit{true} \vee E_I)$ and $\textit{true} \vee (\exists_I E_I)$ evaluate to *true*. Otherwise, if A is false, both $(\exists_I \textit{false} \vee E_I)$ and $\textit{false} \vee (\exists_I E_I)$ evaluate to $\exists_I E_I$.

Loop-invariant Expressions in Quantified Sums

Now we consider quantified sums $\sum_I E_I$ that can be combined with a loop-invariant expression A, such that

$$(\sum_I A \oplus E_I) \equiv A \oplus' (\sum_I E_I) \tag{4.8}$$

where $\oplus$ and $\oplus'$ are arithmetic operators. Equation 4.8 holds only for operator $\oplus = \oplus' = *$. Note, that the case $(\sum_I E_I * A) \equiv (\sum_I E_I) * A$ is covered by Equation 4.8, since multiplication is associative and commutative.

Table 4.6 summarises the *inside*- and *outside*-representation in the sum case. Note, that

$\sum$-CASE	*inside*-representation	*outside*-representation
Original	$(\sum_I A * E_I)$	$A * (\sum_I E_I)$
Unrolled	$(A * E_1) + \cdots + (A * E_k)$	$A * (E_1 + E_2 + \cdots + E_k)$
Flat (nested)	$aux_1 = a * e_1$ $aux_2 = a * e_2$ $\ldots$ $aux_k = a * e_k$ $aux_q = aux_1 + aux_2 + \cdots + aux_k$	$aux_1 = e_1 + e_2 + \cdots + e_k$ $aux_q = a * aux_1$
Summary	$k + 1$ aux. variables, $k + 1$ constraints	2 aux. variables, 2 constraints

Table 4.6: $\sum$-**Case 1** of comparing *inside*- and *outside*-representation.

sums, being integer expressions, are *always* nested in another expression, so we only consider the nested flat case. Evidently, the *outside*-representation is preferable to the *inside*-representation since it uses $k-1$ less constraints and auxiliary variables. In our experiments on constructed examples we even observe a drastic difference in propagation. Unfortunately, reasoning about the general propagation behaviour in such a case is very difficult and in some cases impossible due to the vast variety of propagators in constraint solvers.

Summary

In this subsection, we have seen that, against our expectations, there exist cases where it is beneficial to move a loop-invariant expression *inside* a quantification. However, we have also seen cases where moving loop-invariant expressions *outside* the quantification is far more beneficial. Hence, in general, neither representation dominates the other.

Unfortunately, it is not always clear which representation is preferable, in particular when they are not comparable with respect to propagation: solvers provide many different propagators that work differently, so in some cases our analysis remains empirical. Moreover, we expect the preferable representation to depend on *other* expressions in the instance: for example, if the first representation shares common subexpressions in the instance, but the other does not, then the former might provide better propagation if the common subexpressions are eliminated. Investigating the presence of common subexpressions in other constraints, however, is infeasible on instance level: consider an instance with n quantifications involving loop-invariant expressions. To determine the representation with the *highest* number of common subexpressions, 2^n combinations need to be considered and compared, which, at instance level, is too expensive in time and memory.

The observations from this study can be easily integrated into tailoring by reformulating quantifications involving loop-invariant expressions into the preferable representation. This step is best performed during preprocessing, before the quantifications are unrolled.

4.7 Summary

This chapter covered *instance optimisations*, i.e. automated approaches to enhance constraint instances and introduces novel instance optimisation techniques that are cheap and easily integrable into tailoring.

First, we presented established instance optimisation techniques in the field of Constraint Programming and Compiler Construction, some of which have inspired the instance optimisation techniques that we propose during tailoring.

Second, we discussed each technique, starting with the elimination of redundant constraints, followed by the most successful enhancement technique, common subexpression elimination and finally, the enhancement of quantifications.

We stress again that *none* of the techniques proposed in this chapter are routinely performed by constraint solvers or flattening tools at present, which the exception of the MiniZinc to FlatZinc converter that has recently included CSE. Many constraint systems could benefit from these optimisation techniques since almost all constraint tools or solvers perform some translation of their input, in which the proposed techniques could be easily integrated.

CHAPTER 5

TAILORING PROBLEM CLASSES

In this chapter, we discuss how to extend tailoring *instances* to tailoring problem *classes*. Problem classes represent a whole family of instances, where *parameters* scale the features of the problem such that a complete parameter instantiation yields a problem instance. Constraint problems are typically formulated as classes, but solved as instances.

There are two main reasons for flattening problem classes: first, to support solvers that take problem classes as input: solvers such as Gecode [80], ECLiPSe [87] or Choco [19] 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 and enhancement techniques *once* at class level instead of *several times*, i.e. once per instance.

The main contribution of this Chapter stems from the novelty of tailoring problem classes which has barely been investigated. Tailoring tools in constraints, such as the MiniZinc-Flatzinc converter [44], or the internal flattener of ECLiPSe Prolog [87] are limited to tailoring instances. Wuille and Schrijvers [89] have recently presented a translation from constraint instances formulated in the functional programming language Haskell to solver Gecode's C++ representation, which in future will include class translations. However, so far, TAILOR, is the only tool to perform tailoring of constraint problem classes to our knowledge.

In this chapter, we show how to extend the procedure of instance-wise tailoring to class-wise tailoring. First, we motivate the idea of class-wise tailoring by presenting different applications tailoring whole problem classes. Second, we discuss how to represent problem classes by extending the notion of expression trees from Chapter 3 so as to include parametrised expressions (Sec. 5.2). Third, we discuss necessary extensions of the instance tailoring procedure in order to process problem classes in Sec. 5.3. This includes a discussion on the current limitations of class tailoring. Finally, we wrap up and conclude in Sec. 5.5.
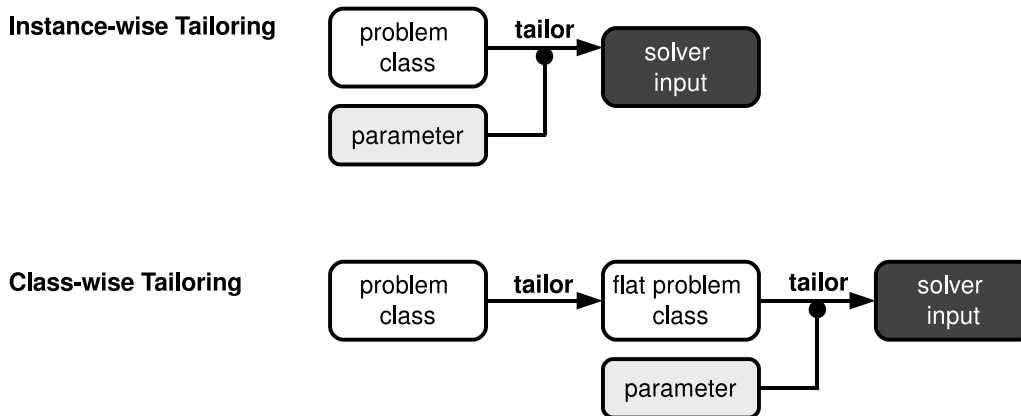
Figure 5.1: **instance-wise** (top) and **class-wise** tailoring (bottom) to solve **instances**

5.1   Applications of Problem Class Tailoring

In this section, we discuss when and how tailoring whole problem classes can be useful and applied in practice, as opposed to the approach of tailoring instances as presented in Chapter 3. More specifically, there are two ways in which tailoring problem classes can be employed.

First, tailoring problem classes can be used to generate input for solvers that allow problem classes as input. Such solvers are typically libraries of programming languages (e.g. solver Gecode [80] is a library of C++, ECLiPSe [87] of Prolog or Choco [19] of Java). As an example, consider tailoring a problem class to a C++ program tailored to solver Gecode, a C++ library. All library-based solvers take constraints in a *flat* format, i.e. constraints are not nested within another. Some library-based solvers, such as Gecode or ECLiPSe, allow nesting of constraint expressions (in case of Gecode, only linear and Boolean constraints). However, though these nested expressions are typically normalised and flattened so as to match n-ary constraints, CSE and other enhancements are not performed on them up to date [74]. Therefore, we conclude that it is more beneficial to flatten and enhance problem classes during tailoring instead of pushing those tasks onto the solver.

Second, tailoring problem classes can also be used when tailoring to a solver that accepts only instances. Typically, such solvers receive input by *instance-wise*-tailoring, as we have seen in Chapter 3: a class is first merged with a parameter specification, yielding an unflattened instance which is then flattened to a flat instance (Fig. 5.1, top). As an alternative approach, we propose *class-wise* tailoring (depicted in Fig. 5.1, bottom) that consists of two tailoring steps: first, the problem class is tailored to a flat class in intermediate format, e.g. an ESSENCE′ problem class is flattened to a flat ESSENCE′ problem class. Then the flat class is tailored together with a parameter specification, yielding a flat instance. Note, that the solver input generated from instance- and class-wise compilation are identical, with certain exceptions (see Sec. 5.3.2). In our empirical analysis (Sec. 5.4), we will see that in particular cases, class-wise flattening is quicker than instance-wise flattening, since a lot of

time-consuming tasks (such as flattening) are only performed *once* at class level and need not be repeated for *every* instance.

The process of tailoring problem classes to solver input consists of the same steps as tailoring instances, with the difference that parameter values are not known (and hence quantifications can often not be unrolled). Automatically generating problem classes has barely been investigated and to the best of our knowledge, TAILOR is the only tool that can perform this step to date.

5.2 Representing Parameterised Expressions

Problem classes and problem instances have very similar structures with the difference that problem classes contain parameters whose values are not specified during tailoring. Parameters 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. These different kinds of parameters need to be integrated into the expression tree representation from the instance level.

Integrating Constant-Scaling and Domain-Scaling Parameters

Constant-scaling and domain-scaling parameters, as illustrated in Example 5.2.1, often appear in constraint problem classes. They affect the computation of each tree node's lower and upper bound (see Sec. 3.3.1 for more details), which has to be extended.

Example 5.2.1. illustrating **constant-** and **domain-scaling** parameters.

```
given n : int (1..)
given m : int (0..)

find x : int (1..n)      $ n as domain−scaling parameter $
find y : int (0..m)      $ m as domain−scaling parameter $

such that      $ m and n as constant−scaling parameters $
  x − m = y + n
```

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 the leaf's lb and ub is defined as $k..k$.

Domain-scaling parameters appear in domain definitions of decision variables. Similarly to constant-scaling parameters, we state that if a domain associated to identifier x is bounded

by the parameters k_1, k_2, 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 $k_1 \leq k_2$.
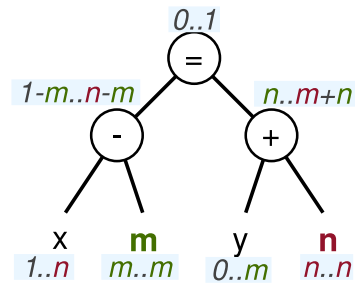

Figure 5.2: **Constant-Scaling and Domain-Scaling Parameters**: illustrating the tree node lower and upper bounds (light-blue labels at each node and leaf) of Example 5.2.1.

In summary, constant- and domain-scaling parameters are easily integrable into the expression tree structure. Furthermore, the extensions do not prevent us from computing finite lower and upper bounds of each expression subtree (using the lb and ub attributes), since every parameter k will be assigned an integer or Boolean value. Fig. 5.2 illustrates constant- and domain-scaling on an example, as well as the computation of the expression nodes' lower and upper bound attributes (represented as light-blue labels), based on the problem class in Example 5.2.1.

Integrating Variable-Scaling Parameters

Variable-scaling parameters arise in the index domain of variable arrays, i.e. they scale the number of elements an array contains, as Example 5.2.2 illustrates. Variable-scaling parameters are easily integrated by extending the symbol table to contain arrays that have a parameterised number of elements.

Example 5.2.2. illustrating **variable-** and **constraint-scaling** parameters.

```
given n : int(1..)

    $ n scaling length of arrays x and y $
find x,y: matrix indexed by [int(1..n)] of bool

such that    $ n scaling number of constraints $
  forall i:int(1..n). x[i] \/ y[i]
```

Integrating Constraint-Scaling Parameters

Constraint-scaling parameters scale the number of constraints, typically by appearing in quantifying domains of quantifications, such as parameter n in the quantification in Example 5.2.2. In a problem *instance*, n would be specified and the quantification unrolled,

generating n disjunctions. However, at class level, such quantifications cannot be unrolled, thus we need to define a notion of parameterised quantified expressions as part of the intermediate expression representation that is used in the tailoring middle-end (Sec. 3.3). Therefore, we extend the corresponding expression tree representation from instance level (Sec. 3.2) to support parameterised quantified expressions, particularly $\forall$, $\exists$ and $\sum$.

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 of N. These attributes are also propagated to 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, Fig. 5.3 illustrates the quantifier attributes (quantifying variables and associated domain) as grey labels in the expression tree of the constraint in Example 5.2.3 below.

Example 5.2.3. Sample Quantification

```
forall i:int(1..n).
    exists j:int(1..m).
        x[i] = y[j]
```

Note, that if the problem class is directly tailored to solver input, then these quantifiers are only allowed if the target solver provides n-ary propagators for conjunction ($\forall$), disjunction ($\exists$) and addition ($\sum$).
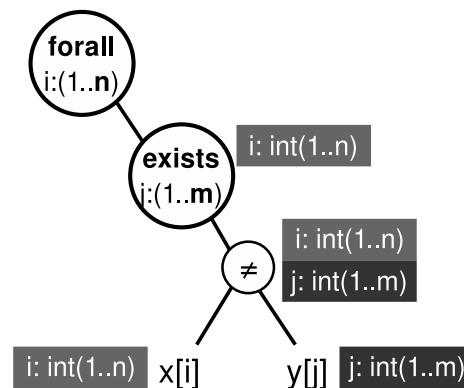
Figure 5.3: illustrating the **Quantifier Attributes** in the expression tree representation of the constraint from Example 5.2.3, where the grey labels represent the quantifier attributes for the corresponding node/leaf.

5.3 Tailoring Problem Classes

In this section, we discuss extensions that are necessary in order to apply the tailoring procedure from Chapter 3 to problem classes. While we have discussed changes in the data structures and internal representation in the previous section, we now discuss extensions to the tailoring *process*.

The tailoring process consists of three stages: (1) the frontend, that processes input to an intermediate format, (2) the middle-end, which performs the core tasks of preprocessing and flattening of the intermediate format, and (3) the backend that deals with solver-related issues that could not be generalised in the middle-end and issues the final output. Each stage requires a (limited) set of adaptions in order to process problem classes.

The frontend requires no notable extensions, (with the exception of some basic, low-level operations to handle the data structure extensions described in the previous section) since its main tasks, parsing, type checking and translation into intermediate format, are the same for classes as for instances since parameters have well-defined types.

Similarly, in the middle-end, all other processes are performed in the same fashion as for instances, with the exception of expression *flattening* that has to be adapted so as to process parameterised expressions. This will be elaborated on in the next subsection. The other core task in the middle-end, preprocessing, is easily extended to deal with problem classes, since the intermediate representation is the same as in instances with the addition of parameterised expressions.

Finally, the backend requires extensions so as to output the problem class. For instance, providing support for 'for'-loops to express a set of universally quantified constraints in the target programming language. These are basic, low-level extensions that are very specific to the target language, and will therefore not be covered.

5.3.1   Flattening Parameterised Subexpressions

The main challenge of flattening parameterised expressions is to handle subexpressions that are quantified by parameters. If an expression is quantified by a parameterised quantifying variable, we need to generate a quantified number of auxiliary variables during flattening. Thus, whenever a parameterised subexpression is flattened, an *array* of auxiliary variables is introduced, whose size is derived by the domain of the corresponding quantifiers. This information is retrieved from the quantification attributes of the quantified subexpression.

Flattening Parameterised Quantifications by Example

To illustrate flattening of a parameterised expression we consider each step in a simple example that contains a parameterised existential quantification:

```
    given  n : int (1..)
    find   x : matrix indexed by [int (1..n)] of int (1..n)
such that
    exists i: int (1..n). x[i]=i
```

The quantified expression, '$x[i]=i$', is a Boolean expression that has the quantifier attributes $i: (1..n)$. Therefore, it is flattened to a Boolean auxiliary array 'auxArray' of length 'n',

which is added to the problem class:

```
find auxArray : matrix indexed by [int(1..n)] of bool
```

The auxiliary array is then linked to the flattened expression 'x[i]=i':

```
forall  i:int(1..n) . auxArray[i] <=> (x[i] = i)
```

Finally, the auxiliary variable array represents the original expression 'x[i]=i':

```
exists  i:int(1..n) . auxArray[i]
```

In summary, flattening yields the flat problem class:

```
given  n : int(1..)
find  x : matrix indexed by[int(1..n)] of int(1..n)
find  auxArray : matrix indexed by [int(1..n)] of bool

such that
    forall  i:int(1..n) . auxArray[i] <=> (x[i] = i),
    exists  i:int(1..n) . auxArray[i]
```

Algorithm 5.1 Excerpt of FLATTEN_CLASS (E,*flatten2Aux*) for flattening problem classes (and instances), based on FLATTEN_CSE (Alg. 4.2) from instance level. 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:             $String_{e_i} \leftarrow toString(e_i)$
5:             **if** $hashMap.contains(String_{e_i})$ **then**
6:                 $aux \leftarrow hashMap.get(String_{e_i})$
7:             **else**
8:                 $aux \leftarrow$ FLATTEN_CLASS (e_i, S, *true*)
9:                 $hashMap.add(String_{e_i}, aux)$
10:            $E.replaceChildWith(e_i,aux)$
11: **if** *flatten2Aux* **then**
12:     **if** E is quantified by parameterised expression **then**
13:         $AuxArray = createNewVarArray(E.lb,E.ub, E.qt.length)$; $vars$.add('*AuxArray*')
14:         $constraints$.add('{'$\forall x \in x$.dom' | $x \in E$.qt.*vars* } . *AuxArray*[E.qt.*index*] = E')
15:         **return** $AuxArray[E.qt.index]$
16:     **else**
17:         $Aux \leftarrow createNewVariable(E.lb, E.ub)$; *auxVars.add(Aux)*; *ctBuffer*.add('*Aux = E*')
18:         **return** Aux

An Algorithm for Flattening Problem Classes

The only difference between instance flattening and class flattening is the generation of auxiliary variables and flat constraints: if a subexpressions is quantified over a parameterised expression, an *array* of auxiliary variables is created, otherwise just a single auxiliary variable. Therefore, the instance algorithm can simply be extended so as to generate auxiliary variables for parameterised quantified expressions.

We summarise the flattening process for problem-class expressions in Alg. 5.1, where extensions to FLATTEN / FLATTEN_CSE are given in red font: It proceeds in three steps: first, an auxiliary variable *array* is generated (line 13). Second, the array is linked to the corresponding flattened expression (line 14). Third, the auxiliary array is returned in such a way that it is referenced correspondingly to the flattened expression (line 15).

Note, that only those nodes are flattened that cannot be evaluated at instance level, i.e. nodes that are of category *decision variable* (see Sec. 2.2). Furthermore, since quantifications cannot be unrolled at class level, flattening typically processes far less nodes than instance flattening. In the following, each flattening step is further explained.

1. Generating the Auxiliary Array   if node E is quantified, it is flattened to an auxiliary variable array (line 13). 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, l, is

$$l = \prod_{1..k}^{i} ub_i - lb_i + 1 \tag{5.1}$$

As an example, in Fig. 5.4, 'aux0' represents an expression that is quantified by $i \in (1..n)$ and $j \in (1..m)$ and hence has length $(n - 1 + 1)*(m - 1 + 1) = n * m$.

2. Linking the Auxiliary Array to the Flat Expression   After creating the auxiliary variable array, the to-be-flattened node E is linked (or 'reified') with *AuxArray* (line 14). The challenge in linking is to dereference the auxiliary array correctly, which needs to take the following into account: the range of each quantifying variable and its associated domain as well as the constant value, from which the target solver starts initialising its arrays. As an example for the latter, in solver MINION, arrays are always dereferenced starting from 0. In FlatZinc, however, arrays are dereferenced starting from 1. We denote this constant value with $\hat{t}$. If E is quantified by k quantifying variables v_i that range over $lb_i..ub_i$, then

the linking constraint defined as:

$$\forall v_1 \in (lb_1..ub_1).\forall v_2 \in (lb_2..ub_2)....\forall v_k \in (lb_k..ub_k).$$

$$E = AuxArray[v_1 + \sum_{2..k}^{i}.(v_i - \hat{t}) * \prod_{1..k-1}^{j} (ub_j - lb_j + 1)] \tag{5.2}$$

As an example, consider the quantified expression from Fig. 5.4:

```
forall i: int(1..n).
    exists j: int(1..m).
        x[i]=y[j]
```

Now we consider flattening this expression to two different targets: first to FlatZinc, where $\hat{t} = 1$, and then to solver MINION, where $\hat{t} = 0$. When flattening to FlatZinc, the tailoring middle-end produces the intermediate representation:

```
forall i: int(1..n).
    forall j: int(1..m).
        (x[i]=y[j]) <=>  auxArray[i+(j−1)*n]
```

When flattening to MINION, the tailoring middle-end produces:

```
forall i: int(1..n).
    forall j: int(1..m).
        (x[i−1]=y[j−1]) <=>  auxArray[(i−1)+(j−1)*(n)]
```

Note, that every 'i' and 'j' has been replaced by 'i−1' and 'j−1' in order to adapt the index to the target solvers array-dereference start value $\hat{t}$, since both index domains initially ranged from '1..n' and '1..m', respectively.

Theorem 5.3.1. *The* **time complexity** *of flattening with* FLATTEN_CLASS *lies in* $O(\hat{k}m)$.

Proof. FLATTEN_CLASS (Alg. 5.1) is an extension of Alg. 4.2 (FLATTEN_CSE) that lies in $O(\hat{k}m)$ where $\hat{k}$ is the maximal number of subexpressions any expression has in the instance/class. FLATTEN_CLASS additionally flattens quantified expressions, which adds the following operations:

1. Checking if an expression is quantified (line 12) requires an atomic check of expression E that is independent of the number of its subexpressions and hence constant. This check is performed for all m_u subexpressions, where in the worst case, when the class has no CS, $m_u = m$. Hence, in summary, the operation lies in $O(m)$.
2. Generating the auxiliary array (line 13): this operation first requires the computation of the length l of the auxiliary array, which depends on v (see Eq. 5.1, the the number of variables that quantify the subexpression, which is constant (the number of quantifying variables cannot be scaled). Second, an array of length l is constructed.

Since we assume atomic operations in the implementation, both operations require constant time, which is performed for m_u subexpressions. In summary, generating an auxiliary variable lies in $O(m)$, since in the worst case, $m_u = m$.

3. Linking the auxiliary array to the flat expression (line 14) involves adding a constraint to the class. This requires to construct a quantification and to compute the index expression from Eq. 5.2 for the auxiliary array where both operations depend on the number of quantifying variables v, which is constant. Again, since we assume atomic operations in the implementation, both operations are dependent only on the number of subexpressions in the to-be-flattened expression, k. The operations are performed for m_u subexpressions, hence, in summary, adding the linking constraint lies in $O(\hat{k}m)$, since in the worst case, $m_u = m$, where $\hat{k}$ is the maximal number of subexpressions of any expression in the class.

In summary, the operations added to flatten classes add $O(m) + O(m) + O(\hat{k}m)$, which, if added to the overall runtime of CSE-flattening at instance level, results in $O(\hat{k}m)$. $\qquad\square$

5.3.2 Redundancies from Flattening Quantified Subexpressions

As mentioned in earlier chapters, quantified expressions often contain guards that enforce particular expressions (see Sec. 4.6). These guards can be either constant (i.e. evaluable at instance level) or entirely composed of decision variables. If a quantified expression is guarded by a constant guard, then class flattening introduces redundancies, which we will illustrate on an example. Consider Example 5.3.1 where the quantified expression 'x[i]*x[j] != y[i]*y[j]' is guarded by the expression '(i<j)'.

Example 5.3.1. illustrating a guarded quantification

```
   given  n,m  :  int(1..)
   find  x,y  :  matrix  indexed  by  [int(1..n)]  of  int(1..m)

 such  that
   forall  i,j  :  int(1..n)  .
      (i<j)   =>   (x[i]*x[j]  !=  y[i]*y[j])
```

Flattening will introduce $2n^2$ auxiliary variables (one array of length n^2 for each multiplication, since both i and j range over $(1..n)$). However, $(i < j)$ will evaluate to *false* in $\frac{n(n+1)}{2}$ cases, hence only $2(n^2 - \frac{n(n+1)}{2})$ auxiliary variables are actually used, the rest are 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. In the following, we discuss different possibilities of addressing these redundancies.
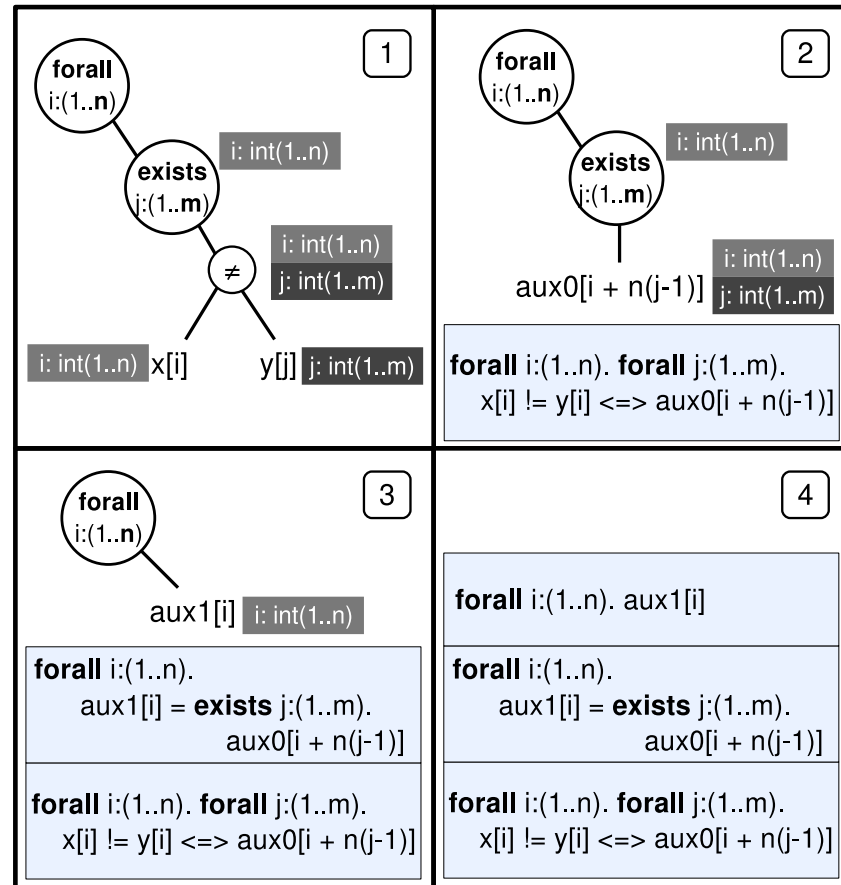
Figure 5.4: **Class Flattening Example**: illustrating the four steps in which the constraint in Example 5.2.3, represented by its expression tree structure, is flattened using Alg. 5.1. Grey labels represent the quantifier attributes of each node/leaf and the light-blue boxes contain the constraints resulting from flattening.

Creating a Minimal Auxiliary Array

Ideally, flattening of guarded expressions at class level would use an auxiliary variable array of *minimal* length, i.e an auxiliary array that contains the *exact* number of elements that are *not* excluded by the guard. For instance, in the Example 5.3.1 from above, we would introduce two auxiliary variable arrays, each containing $2(n^2 - \frac{n(n+1)}{2})$ elements. Creating an auxiliary array of minimal length requires to first determine the exact number of expressions that are excluded by the guard, and second, to find a mapping that allows to properly link the auxiliary array with the flattened expression. These two tasks however, pose major difficulties.

First, determining the exact number of expressions that are excluded by the guard can be extremely difficult, since guard and quantification can be arbitrarily complex. Consider, for instance, quantifications where the guard contains several different quantifiers that can range over domains that are again scaled by quantifiers, as illustrated in Example 5.3.2. In

this example it is difficult to determine the exact number of excluded expressions by the guard, particularly automatically.

Example 5.3.2. illustrating a complex guarded quantification

```
forall i : int(1..n) .
    forall j: int(i..m) .
        (i%n !=0) =>  exists k:int(0..j) .
                          ((i+k < j) /\ (k!=i)) /\
                              (x[i]*y[j] = z[k])
```

Second, finding a mapping that allows us to properly link (reify) the auxiliary variable with the quantified, flattened expression is difficult, even in simple cases like Example 5.3.1: we know the length of the minimal auxiliary array, which is $(n^2 - \frac{n(n+1)}{2})$, hence we create two auxiliary variable arrays of that length:

```
find auxArray0:matrix indexed by [int(1..n*n-(n*(n+1))/2)] of int(1..m*m)
find auxArray1:matrix indexed by [int(1..n*n-(n*(n+1))/2)] of int(1..m*m)
```

The next step is linking each auxiliary variable to the flat expression it represents, i.e. linking 'auxArray0' to 'x[i]*x[j]' and 'auxArray1' to 'y[i]*y[j]', respectively. Using the class-flattening approach described in the previous section, the constraint would be linked in the following way:

```
forall i,j:int(1..n).
  (i<j) =>
      x[i]*x[j] = auxArray0[i+(j-1)*n]
```

However, this mapping works only if the array is is *not* of minimal length, since (i, j)-assignments that are excluded by the guard are included in the mapping. What we want is to map the *first* feasible (i, j)-assignment to the *first* element of 'auxArray0', the *second* feasible (i, j)-assignment to the *second* element of 'auxArray0', and so on. Such a mapping could be formulated in closed form (implicitly) or explicitly. To our knowledge, no implicit formulation exists. An *explicit* mapping is an explicit definition of which (i, j)-assignment is mapped to which auxiliary array index. For illustration, consider again Example 5.3.1 with guard '$i<j$' where both 'i' and 'j' range over $(1..n)$ where the explicit mapping can be summarised as follows:

(i,j)-assignment	$\longrightarrow$	index in 'auxArray0'
(1,1)	$\times$	
(1,2)	$\longrightarrow$	1
(1,3)	$\longrightarrow$	2
...	$\longrightarrow$	...
(1,n)	$\longrightarrow$	$n-1$
(2,1)	$\times$	
(2,2)	$\times$	
(2,3)	$\longrightarrow$	n
(2,4)	$\longrightarrow$	$n+1$
...	$\longrightarrow$	...
(2,n)	$\longrightarrow$	$2n-3$
(3,1)	$\times$	
(3,2)	$\times$	
(3,3)	$\times$	
(3,4)	$\longrightarrow$	$2n-2$
...	$\longrightarrow$	...
(3,n)	$\longrightarrow$	$3n-5$
($n-1$,1)	$\times$	
($n-1$,2)	$\times$	
...	$\times$	
($n-1$,$n-1$)	$\times$	
($n-1$,n)	$\longrightarrow$	$n^2 - \frac{n(n+1)}{2}$
(n,1)	$\times$	
...	$\times$	
(n,n-1)	$\times$	
(n,n)	$\times$	

However, since the exact value of n is unknown at class level and constraint modelling languages do not provide appropriate comprehensions, it is not possible to formulate an explicit mapping.

Using Local Auxiliary Variables

An alternative to minimal arrays is introducing *local* auxiliary variables for every case where the Boolean guard holds. Local variables have a limited scope, i.e. they are only available in the scope of the corresponding quantification. Local variables are common in programming languages but not very common in constraint languages. For illustration, consider again Example 5.3.1 where the two subexpressions 'x[i]*x[j]' and 'y[i]*y[j]' can be flattened to local variables 'aux0' and 'aux1', as demonstrated in the following excerpt of a C++ program tailored to solver Gecode:

```cpp
// using local auxiliary variables 'aux0' and 'aux1'
for(int i=1; i<=n; i++) {
    for(int j=1; j<=n; j++) {
        if(i<j) {
            IntVar aux0(*this,1,m*m);
            IntVar aux1(*this,1,m*m);

            mult(this, x[i-1], x[j-1], aux0, opt.icl());
```

```
            mult(this, y[i−1], y[j−1], aux1, opt.icl());

            rel(this, aux0, IRT_NQ, aux1, opt.icl());
        }
    }
}
```

Evidently, the local variables will only be created if the guard, 'i<j' is true. This is an easy way to resolve the redundancy problem, however, local variables are not standard in constraint modelling languages, nor in constraint solvers. Furthermore, since local variables have a very restricted scope, they cannot be reused in the case of common subexpressions. For instance, if subexpression 'x[i]*x[j]' re-occurs in another constraint in the problem class, it cannot be replaced with the same local variable 'aux0'. Common subexpression elimination (Sec. 4.2) can have a huge impact on model performance (see experiments in Chapter 8) and hence should not be prevented.

The Redundancy in Practice

So far, there is no obvious approach to efficiently tackle redundancies stemming from guards without extending the constraint modelling language and solver format (e.g. to allow comprehensions or local variables) or preventing common subexpression elimination (local variables). Therefore, we have investigated the actual impact of unconstrained auxiliary variables on practical examples. In this empirical analysis (Chapter 8) we observe 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). This suggests that the redundancy is negligible as long as auxiliary variables are not searched upon. However, further investigation of this redundancy is an important part of future work.

5.4 Instance-wise versus Class-wise Tailoring

Typical instance 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 (see Fig. 5.1 (top) and Chapter 3 for more details). In this chapter (Sec. 5.1) we propose *class-wise* compilation, an alternative approach, where first the problem class is tailored to a flat class, which is then again tailored together with a parameter specification, yielding a flat instance (Fig. 5.1 (bottom)). Note, that the flat instances from instance- and class-wise compilation are identical, with certain exceptions (see Sec. 5.3.2). In the following 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 tailors the class by merging C with a parameter specification and then flattening it k times. Class-

wise compilation tailors the class C once and then tailors the instance obtained from pairing the flat class with a parameter specification, k times. We call n'_i the number of nodes of flat instance i, n the number of nodes in unflattened class C, and n' the number of nodes in flat class C, with $n \leq n'$ and $n \leq n'_i$.

Proposition 5.4.1. We propose that the preferred compilation process for class C depends on the kind of parameters that scale C and the number of instances k:

1. First, if C contains *no constraint-scaling* parameters, every flat instance will have the same number of nodes as the flat class, i.e. $n'=n'_1=...=n'_k$. Thus, class-wise compilation will perform the main tailoring work *once*, when flattening the class, while instance-wise tailoring will repeat this work for *every instance*.

2. Furthermore, if the flat problem class contains more nodes than the unflat problem class (i.e. $n' > n$), then tailoring the class must have applied some effort that can be saved when tailoring the flat class to an instance. Therefore, we expect class-wise compilation to be quicker in this case.

3. Finally, in cases where $n' = n$, i.e. tailoring the class to a flat class has not changed the number of nodes (and therefore has not performed any work that can be saved when tailoring instances), instance-wise tailoring is expected to perform better than class-wise tailoring. However, the number of tailored instances, k, also plays an important role: the larger k (i.e. the more instances are flattened), the closer the performance of class-wise and instance-wise tailoring will become wrt tailoring time.

We test this proposition in an empirical analysis in the following subsection, where we consider problem classes with different parameter characteristics.

5.4.1 Empirical Analysis

In our empirical analysis we study the differences in instance-wise and class-wise tailoring on three problem classes that are scaled by parameters in a different way:

1. **English Peg Solitaire** (state-centric model, taken from [43]) has no constraint-scaling parameters, i.e. the number of nodes in the flat class is the same as the number of nodes in every flat instance, hence $n'=n'_1=...=n'_k$.

2. **Balanced Incomplete Block Design (BIBD)** (problem 28 from CSPlib [36]) where $n<n'$, i.e. the number of nodes in the unflat class is smaller than that of the flattened class.

3. **The Langford Number Problem** (problem 24 from CSPlib) where $n=n'$, i.e. the number of nodes in the unflat and flattened class are the same.

Both BIBD and Langford contain constraint-scaling parameters. Tab. 5.1 gives exact figures on the differences of n, n' and n_i in each problem class with respect to the parameters. Note that the instances resulting from both compilation processes are identical (with the difference that the class-wise compiled instances contain arrays of auxiliary variables instead of single variables).

Problem Class	Nodes in **unflat class** n	Nodes in **flat class** n'	Nodes in **flat instance** i n'_i
Peg Solitaire State	30	5,425	5,425
BIBD	17	20	$2(b+v-1)+$ $(b+1)(v^2+v)$
Langford	5	5	$k*(l-1)+1$

Table 5.1: **Node Increase by Parameter Values** m: #nodes in unflat class, m': #nodes in flat class, m'_i: #nodes in flat instance i

We apply both instance- and class-wise compilation on 4 instances of each problem class and compare the overall compilation time (Tab. 5.2). For tailoring, we use TAILORv0.3.2 on Java REv1.6.0 and tailor all instances/classes on the same machine, a MacBook Pro 1,1 with Intel Dual Core (1.83 GHz) and 512MB RAM. Tab. 5.2 gives an overview of our results: for each problem class, it shows the tailoring times for class-wise and instance-wise compilation, showing the times for tailoring the class (first column), followed by the tailoring times for each instance. In the following, we discuss the results for each class in more detail and draw conclusions wrt Proposition 5.4.1.

Peg Solitaire (no constraint-scaling parameters) The results for Peg Solitaire confirm (1) from Proposition 5.4.1: Peg Solitaire contains no constraint-scaling parameters (the number of nodes of the flat class is the same as the number of nodes in every instance, see node increase in Tab. 5.1), therefore, when class-wise tailoring first generates a flat class, it has already performed most of the flattening. This is evident when comparing the figures for class-wise and instance-wise compilation: tailoring the class takes about the same time as tailoring the instance in instance-wise tailoring, since all the flattening is repeated for every instance. These results lead to the conclusion that for problem classes like Peg Solitaire, which contain *no* constraint-scaling parameters, class-wise compilation is preferable to instance-wise compilation wrt tailoring time.

BIBD ($n' > n$) BIBD confirms (2) from Proposition 5.4.1: if class tailoring yields a flat class with more nodes than the initial class ($n' > n$), then it has saved tailoring effort that can be saved when tailoring the instances. This can be observed when comparing the tailoring times of the different instances: the larger the instance, the higher the benefit in

Problem	Tailoring Time for (sec)					
	Class	Instance1	Instance 2	Instance 3	Instance 4	Total
Peg Solitaire						
class	17.96	6.245	6.235	6.237	6.240	**42.917**
instance	-	17.247	17.227	17.230	17.345	59.049
BIBD						
class	0.168	0.335	4.679	9.653	16.875	**31.710**
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
instance	-	0.196	0.197	0.194	0.197	**0.784**

Table 5.2: **Class-wise vs. Instance-wise Compilation Time** with k=4 instances. The first column (class) shows the time used for tailoring the class; the following columns show the time for tailoring each of the four instances. The last column gives the total time, where the faster compilation method is highlighted in bold face.

class-wise tailoring (for the smallest instance, Instance1, the tailoring times are the same, while for the largest instance, tailoring the flat class takes half the time than tailoring the unflat class).

Langford ($n' = n$) In the Langford Number Problem, first tailoring the unflat class to a flat class saves no tailoring effort, since the nodes are the same in the unflat and flat class ($n' = n$). Therefore, the results for Langford confirms (3) from Proposition 5.4.1, stating that instance-wise compilation is preferable in such cases: instance-wise compilation is quicker by the time that class-wise compilation invests into first flattening the class. Furthermore, note, that with increasing k (i.e. the more instances we tailor) the smaller the difference between class-wise and instance-wise tailoring becomes.

Summary

In summary, our empirical analysis has provided some confirmation of our proposition as to when class-wise compilation is preferable to instance-wise compilation and vice versa. However, this study was performed only on a small selection of problems and a sound evidence of the correctness of our proposition requires a more substantial empirical analysis, involving more problem classes, which is another item of our future work. Note that it is difficult to perform a theoretical analysis of this matter, since the tailoring process (of both instances and classes) depends on many factors, which includes the parameter properties of a problem class, so it is difficult to make a general, clear statement.

In summary, our observations are valuable and suggest that class-wise compilation is an interesting and competitive alternative to instance-wise compilation from which we expect promising results in future work.

5.5   Summary

In this chapter we proposed a novel tailoring approach: tailoring problems as classes as an alternative to tailoring instance-wise. Tailoring classes can be used in two different contexts: first, to target library-based solvers, where problems are formulated as programs and parameters can be specified during runtime. In this way, the tailored 'class program' can be reused for each instance.

Second, tailoring classes can also be used to tailor problems to instances by *class*-wise tailoring. Class-wise tailoring takes a problem class and tailors it to a flat format, corresponding to the target solver. Then the flat problem class is used to tailor instance-wise. We have shown that if the problem class contains particular kinds of parameters, then class-wise tailoring is faster than the standard approach of instance-wise tailoring.

Tailoring classes however still has limitations. In particular, the redundancies introduced by constant Boolean guards, in form of unconstrained auxiliary variables. So far, we do not know how to completely overcome this redundancy, however, our experiments have showed that the negative impact is marginal.

In summary, tailoring classes is an interesting and competitive alternative approach to tailoring instances that we expect to be a promising candidate in future work.

CHAPTER 6

CLASS OPTIMISATIONS

In this chapter, we discuss how to automatically enhance problem classes during class-wise tailoring. More specifically, we consider the optimisation techniques from instance level (Chapter 4) and discuss them at class level.

Performing enhancements at class-level is particularly practical, where an improvement affects all instances that are drawn from the class (as opposed to instance-wise enhancement in Chapter 4 where the enhancement of a particular instance has to be repeated for every other instance). This is an important issue, especially since many enhancements are not dependent on one particular instance, but can be performed *in general*, for every instance, by applying them to the problem class. For example, if the instance i_1, drawn from problem class p, contains common subexpressions, then another instance i_2 drawn from p will most likely also contain common subexpressions. Hence, eliminating the common subexpressions in the class p will positively affect both i_1 and i_2 (and all other instances drawn from p). Since class-level enhancements positively affect the whole set of instances drawn from the class, we can invest far more effort and time into enhancement than at instance level.

The main contributions of this chapter are the following. First, we discuss the elimination of redundant constraints at class level. More specifically, we propose an algorithm to eliminate duplicate constraints that arise from weak guards by exploiting *unification*. This is a novel approach in order to strengthen weak guards that arise in problem models of inexperienced modellers. Second, we consider the elimination of common subexpressions at class level where we highlight the main challenges in detecting the same set of common subexpressions that we can detect at instance level. We propose three different CSE approaches, each comprising particular benefits and drawbacks, each tackling (some of) the challenges in different ways. In particular, we discuss the challenge of detecting *shifted common subexpressions* that are particularly hard to detect at class level if the detection is supposed to be performed using little computational effort. However, the third approach we propose can address this challenge and therefore represents the most promising candidate for class-level CSE.

In the general area of constraints, little research has been done on enhancing whole prob-

lems classes. Charnley *et al* [18] automatically infer implied constraints (see Sec. 4.1.1 for more details), which unfortunately is only applicable to simple problem classes. The aim of enhancing a whole problem class is challenging and this Chapter contains a lot of unanswered questions that we want to investigate in future work.

The chapter is structured similarly to Chapter 4 that deals with instance enhancements: First, we discuss the elimination of redundant constraints, in particular duplicate constraints (Sec. 6.1), Second, we discuss the issue of common subexpression elimination at class level in Sec. 6.2. Note, that the quantifier optimisations, as discussed at instance level (Sec. 4.6), are directly applicable to class level (since they performed on quantifications when they are not yet unrolled) and hence require no further discussion at class level. A quick summary wraps up and concludes in Sec. 6.3.

6.1 Eliminating Redundant Constraints

A constraint is called *redundant*, if the set of solutions is unchanged by its addition to or removal from a model and the constraint has no effect on the solving procedure in terms of propagation (as opposed to *implied* constraints with which propagation improves). At instance level, we have seen that constraint instances can contain redundant constraints, in particular *duplicate* constraints, stemming from weak constant Boolean guards (Sec. 4.5.1). In the following we discuss how to tackle duplicate constraints at class level.

6.1.1 Eliminating Duplicate Constraints by Unification

The technique of removing duplicate constraints at instance level is directly applicable to class level. However, at class level, most quantifications are not unrollable and the Boolean guards are not evaluable, since they contain either parameter expressions or quantifying variables that are not specified at class level. As an example, consider the naive n-Queens problem class [57] model in Example 6.1.1 below:

Example 6.1.1. Naive n-Queens Problem Class

```
   given      n:   int(1..)
   find queens:   matrix indexed by [int(1..n)] of int(1..n)

such that
   $ (1) no two queens in the same row $
   alldifferent(queens),

   $ (2) no two queens on the same NW–SE diagonal $
   forall  i,j : int(1..n).
      (i != j) => (queens[i] + i != queens[j]+ j),

   $ (3) no two queens on the same NE–SW diagonal $
```

```
  forall  i,j : int(1..n).
      (i != j) => (queens[i] − i != queens[j]− j)
```

Consider the second constraint that restricts the NW-SE diagonal to contain only one queen. At class level, n is not specified, so the loop cannot be unrolled. However, when unrolled, the resulting set of constraints will contain duplicate constraints of the form

```
  queen[1]+1 != queen[2]+2,
  queen[2]+2 != queen[1]+1,
  ...
```

At instance level, ordering the unrolled constraints facilitated the detection of duplicates (which are placed next to another), so they can easily be eliminated (Sec. 4.5.1). However, at class level, duplicate constraints can only be eliminated by *analysing* and *strengthening* the guard. For illustration, the redundancy in Example 6.1.1 can be prevented by strengthening the guard, '(i!=j)' to '(i<j)'. In the following we discuss one possible approach to strengthen guards in quantifications, which exploits *unification*.

Unification is a means to find substitutions that make different logical expressions equivalent [69]. Unification is applied through the UNIFY algorithm that, given two logical sentenses 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 expressions $(x + i)$ and $(x + 3)$ that have the unifier $u = \{3/i\}$, i.e. if i is substitued with (i.e. assigned) 3, then both expressions are equivalent. Unification is a powerful concept that is applied in logic programming languages, like Prolog [75].

Exploiting Unification

We can exploit unification to eliminate duplicate constraints in the following way: we start with a quantification of the form

$$\forall I : D.B_I \Rightarrow E_I$$

where $I = \{i_1, ..., i_m\}$ is a non-empty set of quantifiers, D denotes a finite integral quantifying domain, B_I is a Boolean guard and E_I is the guarded expression, where both B_I and E_I are quantified over (a subset of) I. Note, that expression E_I is represented as expression tree. This quantification is input to STRENGTHEN_GUARD, which is informally outlined below:

STRENGTHEN_GUARD($\forall I : D.B_I \Rightarrow E_I$)

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 two conditions to the guard in order to break the equivalence in case $i_k = i_l$: first, the restriction $\neg(i_k = i_l)$ to break the equivalence; second, a lexicographical ordering constraint to break the symmetry stemming from the commutative operator: $(i_k \leq i_l)$, (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.

Simple Example: Naive n-Queens For illustration, consider again the excerpt from the n-queens problem class above, where the guarded expression E_I corresponds to

$$E_{i,j} \equiv (queens[i] + i \neq queens[j] + j)$$

whose root node is '$\neq$', a binary commutative operator (step 1). Hence, we compute the unifiers for the two subtrees $(queens[i] + i)$ and $(queens[j] + j)$ (step 2), which are $u_1 = \{i/j\}$ and $u_2 = \{j/i\}$ since $(queens[i]+i)$ is equivalent to $(queens[j]+j)$ if i is substitued by j or vice versa. Therefore, we deduce that $(i \neq j)$ and $(i \leq j)$ (step 3) and add $(i \leq j)$ to the Boolean guard (step 4), yielding:

```
forall i,j:int(1..n).
    ((i!=j) /\ (i<=j))  =>  (queen[i]+1 != queen[j]+j)
```

Advanced Example: Golomb Ruler For an advanced example, we consider an excerpt of a naive Golomb Ruler model [77] that expresses the distinct distances between all ticks:

```
forall i1,i2,i3,i4 : int(1..ticks).

    ((i1 > i2) /\ (i3 > i4) /\ (i2!=i4)) =>

            (ruler[i1] - ruler[i2] != ruler[i3] - ruler[i4])
```

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]),
 ...
```

Hence, we can apply unification in order to determine a stronger guard that prevents duplicate constraints: First, we compute the set of unifiers for $(ruler[i_1] - ruler[i_2])$ and $(ruler[i_3] - ruler[i_4])$ since '$\neq$' is a commutative operator. There are four unifiers:

$$
\begin{aligned}
u_1 &= \{i_1/i_3 \wedge i_2/i_4\} \\
u_2 &= \{i_3/i_1 \wedge i_4/i_2\} \\
u_3 &= \{i_3/i_1 \wedge i_2/i_4\} \\
u_4 &= \{i_1/i_3 \wedge i_4/i_2\}
\end{aligned}
$$

From these unifiers we can deduce that $(ruler[i_1] - ruler[i_2])$ is equivalent to $(ruler[i_3] - ruler[i_4])$ if $(i_1 = i_3) \wedge (i_2 = i_4)$. Therefore, we need to add two restrictions to the guard:

1. The negation of $(i_1 = i_3) \wedge (i_2 = i_4)$, i.e. $(i_1 \neq i_3) \vee (i_2 \neq i_4)$
2. Lexicographic constraints over the equivalent quantifying variables in order to break the symmetry of '$\neq$', i.e. $(i_1 \leq i_3)$ and $(i_2 \leq i_4)$.

In summary, we get the enhanced quantification:

```
forall i1,i2,i3,i4 : int(1..ticks).

    ((i1 > i2) /\ (i3 > i4) /\ (i2!=i4) /\
        ((i1!=i3) \/ (i2!=i4))  /\ (i1 <= i3) /\ (i2 <= i4)) =>

        (ruler[i1] - ruler[i2] != ruler[i3] - ruler[i4])
```

Note that this approach can be extended to all quantifications that contain a quantified commutative expression (and hence can be used to *generate* guards, if necessary). An empirical investigation of eliminating duplicate constraints in n-Queens and the Golomb Ruler problem is given in our experimental chapter, in Sec. 4.5.1, where we observe that eliminating duplicates can reduce solving time by half.

In summary, we have seen that unification can be exploited so as to strengthen guards in quantifications in order to prevent duplicate constraints in the instance after unrolling the respective quantification. A more thorough and more formal investigation of this approach as well as an implementation are important items of future work.

6.2   Common Subexpression Elimination (CSE)

Common subexpression elimination (CSE) at class level is very similar to CSE at instance level. For illustration, consider the quantification from the excerpt from a naive n-queens model [57] in Example 6.1.1, stating that no two queens may be positioned in the same NW-SE diagonal:

```
forall  i,j: int(1..n).
    (i < j) => (queens[i] + i != queens[j]+ j)
```

The subexpressions 'queens[i]+ i' and 'queens[j]+ j' are equivalent, and should hence be eliminated by using the same auxiliary variable array during flattening, as illustrated below:

```
forall i:int(1..n).
    auxArray[i] = queens[i] + i,

forall i,j:int(1..n).
    (i < j) =>  auxArray[i] != auxArray[j]
```

CSE from instance level (Sec. 4.2) will *not* detect this equivalence, since it only matches expressions that are identical, which 'queens[i]+ i' and 'queens[j]+ j' are not. Therefore, we need to extend the detection mechanism from instance level so as to detect equivalences between quantified subexpressions at class level.

The detection at class level has to be done with care, since equivalence of two subexpressions depends on the domains of the quantifying variables. Consider, for example, the two subexpressions 'x[i]*y[i]' and 'x[i]*y[i]' that occur in different constraints. They are identical, but since i might range over different domains in the constraints, 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. Therefore, we need to adapt the hashmap checks from instance level so as to match the right cases with respect to the quantifying domain. This can be achieved in different ways. In the following, we explore three different possible approaches, each with particular advantages and drawbacks.

6.2.1 Approach 1: Quantification Normalisation

The first approach exploits normalisation of quantifications in order to detect common subexpressions. First, quantifiers are normalised (prior to flattening) by renaming quantifiers and creating a *hash-table* of all those quantifiers that range over the *same* quantifying domain. Second, this hash-table is employed during flattening where to-be-flattened subexpressions are reformulated according to common quantifiers.

Quantification Normalisation

The normalisation of quantifiers has two aims: first, renaming quantifiers that range over the same domain so as to render them identical. For instance, consider the two quantifications

```
forall i:int(1..n).   x[i] != i,
```

```
    forall j:int(1..n).    x[j] != y[j]
```

where both 'i' and 'j' range over '(1..n)', and normalisation renames 'j' into 'i', yielding two identical subexpressions 'x[i]'.

```
    forall i:int(1..n).    x[i] != i,

    forall i:int(1..n).    x[i] != y[i]
```

The second aim is to rename all quantifying variables that occur elsewhere in the problem class, ranging over a different domain. As an example, in

```
    forall s:int(0..n).        x[s] != s,

    forall s:int(1..n−1).
        forall j:int(0..m).   x[s] < y[j]+1
```

quantifier 's' ranges over different domains and is hence the latter occurrence is renamed:

```
    forall s:int(0..n).        x[s] != s,

    forall s1:int(1..n−1).
        forall j:int(0..m).    x[s1] < y[j]+1
```

The third aim is to store all quantifying variables that range over the same domain, but which cannot be renamed. We denote such quantifying variables *common* quantifying variables. For illustration, consider the excerpt of the n-queens problem class (Example 6.1.1) again, where the quantifying variables 'i' and 'j' are common since they range over the same domain, '**int** (1.. n)', but which cannot be renamed:

```
    forall  i,j : int(1..n).
        (i < j) => (queens[i] + i != queens[j]+ j)
```

During normalisation, a hash-table is created that stores all common quantifying variables by mapping the quantifying domain to a list of the respective common quantifying variables. In he example above, the hash-table have an entry '**int** (1.. n)' $\longrightarrow$ '[i,j]'.

An Algorithm for Normalisation   Quantifier Normalisation can be achieved by iterating once over all constraints/subexpressions in the constraint model while collecting/matching quantifiers and their domains. The algorithm NORMALISE_Q(L) takes a list of constraints L where each quantification q in every constraint c is normalised.

NORMALISE_Q(L)

1. create empty hash-table H

2. create an empty list Q

3. for each constraint c in list L
   for each quantification q in constraint c
   for each quantifying variable i in quantification q

   (a) if i's domain D has an entry in the hashmap, obtain the m common quantifiers $j_1, \ldots, j_m$ and iterate over them:

      i. if i and j_k are not in the same scope (with $1 \leq k \leq m$) rename i to j_k.

      ii. otherwise, if all m common quantifiers are in the same scope (i.e. i cannot be renamed) add i to the list of quantifiers, Q, and add an entry to hashtable H of the form $D \longrightarrow [j_1, \ldots j_m, i]$.

   (b) otherwise

      i. if the list of quantifiers, Q, contains a quantifier of the same name as i, then iteratively rename i to i_t (where t is an integer, starting from 0) until i_t has no other entry in Q. Then add i_t to Q and add the mapping $D \longrightarrow [i_t]$ to hash-table H.

      ii. otherwise add i to the list of quantifiers Q and add a mapping from i's domain D to i of the form $D \longrightarrow [i]$ into hash-table H.

Eliminating Common Subexpressions Using Common Quantifying Variables

After normalising quantifiers as described above, the following statements hold for every normalised problem class:

- For every quantifier i in the problem class, there exists no other quantifier named i that ranges over a different domain
- If two quantifiers i and j (where $i \neq j$) range over the same domain D, then there is an entry in the quantifier list of the form $D \longrightarrow [i, j]$
- There exist no two quantifying variables i and j where i and j are in different scopes and range over the same domain.

Therefore, when flattening a quantified subexpression E that is quantified by variables $i_1, \ldots, i_n$, we know that any equivalent expression *outside* E's scope (equivalent wrt the quantifying variables) is identical and can hence be matched using the hash-table approach from instance level. Furthermore, we know that any equivalent expression *inside* E's scope (equivalent wrt the quantifying variables) can be detected by renaming E's quantifying variables to the names of their common quantifying variables that are obtained from hash-table H. Hence, we extend the recursive algorithm FLATTEN_CLASS (Alg. 5.1) to algorithm FLATTEN_CLASS_CSE1 (E,*bool*) that performs the following additional operations:

FLATTEN_CLASS_CSE1 (E,*bool*)
for each subtree e of E

1. if e has a common subexpression in the hash-table, replace e with the respective auxiliary variable aux, otherwise goto 2

2. if e has no common subexpression in the hash-table, and e is quantified by quantifying variables $i_1, \ldots, i_n$ goto 3, otherwise goto 4.

3. for each quantifying variable i_k that quantifies e (where $1 \leq k \leq n$):

   (a) retrieve the list of common quantifiers L of i_k from the common quantifiers hash-table H.

   (b) if L is empty, stop. Otherwise iterate over the list of common quantifiers $j_1, \ldots, j_m$:

      i. replace every occurrence of i with j_l ($1 \leq l \leq m$) in e, yielding e' and check for a common subexpression of e'.

      ii. if successful, retrieve the corresponding auxiliary variable and use it for e and return.

4. flatten e to an auxiliary variable aux and replace e with aux and return.

Summary

In summary, the first approach performs two essential steps: first, quantifiers are normalised so that every quantifying variable is unique and common quantifying variables (i.e. those that range over the same domain) are stored in a hash-table. The second step is the exploitation of the hash-table when flattening a subexpression e: Whenever a subexpression e is flattened, the list of common quantifiers is used in order to reformulate e with a common quantifier in order to match identical subexpressions. This approach is easy to implement but can be very impractical if the number of equivalent quantifiers is very high: performing normalisation and checking all possible combinations of common quantifiers in a subexpression can be quite expensive. Therefore, we want to investigate alternative approaches that reduce the detection effort.

6.2.2   Approach 2: Label and Domain Representation

The second approach is to represent quantifying variables by their underlying domain together with a label, which does not require any quantifier normalisation or common quantifier hashmaps.

Domain Representation   Since the hashmap of subexpressions contains expressions in String format, one possibility is to replace every quantified variable identifier with its respective domain in the String representation of every expression (following a specific syntax to avoid conflicts with index domains). Then, in a hash-table check, the actual *domains*

of the quantifying variables are compared, and not the corresponding identifiers. For instance, consider the quantification

```
forall i:int(1..n).
    forall j:int(0..m).
          x[i] mod n != y[j] mod m
```

When flattening '`x[i] mod n`', its String representation in the hashmap is '`x[{1..n}] mod n`'. Similarly, '`y[j] mod m`' is stored as '`y[{0..m}] mod m`'.

Note, however, that this introduces another ambiguity: the ambiguity of dimensions. For illustration, consider representing the two nodes '`x[i]*y[i]`' and '`x[i]*y[j]`', where the first is quantified over one variable and the second over two variables. If '`i`' and '`j`' are quantified over the same domain '`1..n`', then both would be represented by the same expression '`x[{1..n}]*y[{1..n}]`', but they are not equivalent. The difference between both expressions can be detected by comparing their dimensions, however, we would require two entries in the hashmap that have an identical key:

```
  x[i]*y[i] ——> aux1[i]
  x[i]*y[j] ——> aux2[i+j*n]
```

It is not possible (and not useful) to have identical keys in a hashmap - one will overwrite the other: if a map $expr \rightarrow aux_1$ is added to the hashmap where there is already another map $expr \rightarrow value_2$, then adding the first map will delete the second. In summary, it is not possible to just represent quantifying variables simply by their range.

Introducing Labels for Quantifying Variables   However, it is possible to represent the quantifying variables with their range and an additional *label* that represents the quantifier. The labels are used to distinguish different quantifiers that range over the same quantifying domain and are given according to the lexicographical order of the quantifying variables. As an example, consider the following quantification:

```
    forall i,j: int(1..n).
      forall k:int(0..m).
        x[i] + y[k] != z[i,j]
```

where the subexpressions of '`x[i] + y[k] != z[i,j]`' would be represented by

$$
\begin{array}{rcl}
\text{'}\texttt{x[i]}\text{'} & \longrightarrow & \texttt{x}[\{1..n\}\texttt{:}\mathbf{1}] \\
\text{'}\texttt{y[k]}\text{'} & \longrightarrow & \texttt{y}[\{0..m\}\texttt{:}\mathbf{1}] \\
\text{'}\texttt{z[i,j]}\text{'} & \longrightarrow & \texttt{z}[\{1..n\}\texttt{:}\mathbf{1},\{1..n\}\texttt{:}\mathbf{2}]
\end{array}
$$

where the boldfaced numbers represent the labels of the quantifiers. In this way, the actual identifier representing a domain does no longer matter. However, this representation does not consider all possibilities, since the labels might be enumerated differently, but still represent an equivalent expression. For instance, consider:

$$\begin{aligned}
\text{`z[i,j]'} &\longrightarrow \text{z}[\{1..n\}{:}\mathbf{1},\{1..n\}{:}\mathbf{2}] \\
\text{`z[b,a]'} &\longrightarrow \text{z}[\{1..n\}{:}\mathbf{2},\{1..n\}{:}\mathbf{1}]
\end{aligned}$$

Since the quantifying variables are labelled lexicographically, 'i' is labelled **1** and quantifier 'j' labelled **2**, however, 'b' is labelled **2** and quantifier 'a' labelled **1**, therefore 'z[i,j]' and 'z[b,a]' are not matched even though they are equivalent. Hence, it is necessary to compare all *permutations* of labels for common subexpressions during flattening, if the first match is not successful.

CSE using Label and Domain Representation

Representing a quantified subexpression by its domain and adding a label for each quantifying variable according to the lexicographical order of the quantifiers is easily done and requires no formal algorithm. However, we informally describe the CSE approach using the label and domain representation with algorithm FLATTEN_CLASS_CSE2 that is based on FLATTEN_CLASS (Alg. 5.1) and recursively flattens a subexpression E:

FLATTEN_CLASS_CSE2 (E,*bool*)

1. for each subtree e of E

   (a) if e has a common subexpression in the hash-table, replace e with the respective auxiliary variable aux, otherwise goto (b)

   (b) if e is quantified such that the set of labels L has more than one element, goto (c), otherwise goto (d).

   (c) for each permutation p of the set of labels:
       i. rename all labels according to permutation p, yielding expression e'
       ii. check for a common subexpression of e' in the hash-table.
       iii. if successful, retrieve the corresponding auxiliary variable and use it for e and stop.

   (d) call FLATTEN_CLASS_CSE2 ($e, true$) which returns auxiliary variable aux

   (e) replace e with aux and stop.

2. if *bool* is true, reify E to auxiliary variable aux and return aux, otherwise return E

The recursive algorithm FLATTEN_CLASS_CSE2 performs permutations of labels in order to match common subexpressions that are not identical because quantifiers have been labelled differently. Note that though this approach avoids normalisation, it can add significant overhead through the label permutations.

136

Summary

In summary, the second approach replaces quantifying variables by their respective domain in their String representation, together with a label to avoid ambiguities. This String representation renders quantification normalisation unnecessary, but still requires reformulations during flattening in order to detect all equivalent cases wrt the quantifying variables. Furthermore, both Approach 1 and Approach 2 cannot detect a certain kind of common subexpressions that instance-level CSE does detect: shifted common subexpressions.

6.2.3 Shifted Common Subexpressions

Approach 1 and Approach 2 do not detect all common subexpressions that instance-wise flattening detects: they cannot detect *shifted* common subexpressions. As an example, consider the following constraint

```
forall i: int(1..n−1) .
    x[i] mod n + x[i+1] mod n = 1
```

that contains the shifted common subexpressions 'x[i] mod n' and 'x[i+1] mod n' which are identical in different 'iterations' of the quantifications (if $n > 2$). Note, that at instance level, we detect these common subexpressions when the quantification is unrolled:

```
    x[1] mod 10 + x[2] mod 10 = 1,
    x[2] mod 10 + x[3] mod 10 = 1,
    x[3] mod 10 + x[4] mod 10 = 1,
    ...
```

and are then matched accordingly:

```
    aux1 <=> x[1] mod 10,      aux2 <=> x[2] mod 10,
    aux3 <=> x[3] mod 10,      aux4 <=> x[4] mod 10,
    ...
    aux1 + aux2 = 1,
    aux2 + aux3 = 1,
    aux3 + aux4 = 1,
    ...
```

At class level, the shifted common subexpressions can be eliminated by using the same auxiliary array for both subexpressions, since they share subexpressions for $n > 2$:

```
forall i:int(1..n−1).
      auxArray[i] + auxArray[i+1] = 1,

forall i:int(1..n).
      auxArray[i] = x[i] mod n,
```

Shifted Common Subexpressions in Approach 1 and 2

Using either Approach 1 or Approach 2, the shifted common subexpressions are not detected in the example above and the quantification is flattened using two different auxiliary arrays for 'x[i] **mod** n' and 'x[i+1] **mod** n':

```
forall i:int (1..n−1).
      auxArray0[i] + auxArray1[i+1] = 1,

forall i:int (1..n−1).
      auxArray0[i] = x[i] mod n,

forall i:int (1..n−1).
      auxArray1[i] = x[i+1] mod n
```

The equivalence of both subexpressions is not detected since the expressions are compared in String format, in which they differ: in Approach 1, 'x[i] **mod** n' is compared with 'x[i+1] **mod** n'. In Approach 2 'x[$\{1..n-1\}$:**1**] **mod** n' is compared with 'x[$\{1..n-1\}$:**1** + 1] **mod** n'. In order to detect shifted common subexpressions, it is necessary to either alter the String representation or reason over it.

Detecting Shifted Common Subexpressions

One approach to detecting this equivalence consists of two steps. First, we neend to consider the quantifying domains with respect to the array dereference. This requires us to include arithmetic operators that are applied to the quantifying variable(s) in an array dereference into the quantifying domain. For instance, including the constant factor '+1' into the quantifying domain of 'i' would result in the following:

$$
\begin{array}{lcl}
\text{'x[i] } \textbf{mod } \text{n'} & \longrightarrow & \text{x[}\{1..n-1\}\text{:}\textbf{1}\text{] } \textbf{mod } n \\
\text{'x[i+1] } \textbf{mod } \text{n'} & \longrightarrow & \text{x[}\{2..n\}\text{:}\textbf{1}\text{] } \textbf{mod } n
\end{array}
$$

Using this representation, it is easier to detect the shifted common subexpressions by comparing the two quantifying domains, since $\{1..n-1\} \cap \{2..n\} \neq \emptyset$ forall $n > 2$. Hence, the second step performs advanced reasoning over expressions that have the same tree structure but differ in terms of their quantifying domains: if their quantifying domains have a non-empty disjoint set, then both subexpressions can be represented with the same auxiliary variable array. The length of the auxiliary variable array has to be adapted to both subexpressions, where the number of elements is defined as the cardinality of the *union* of both domains. For instance, in the example above, 'auxArray' has n elements since the domain union is $\{1..n-1\} \cup \{2..n\} = \{1..n\}$ whose cardinality is $|\{1..n\}| = n$.

Obviously, the String hash-table cannot perform this sort of reasoning, so a new datastructure is necessary that can perform reasoning and comparison of domains in an efficient

manner. A thorough investigation of possible candidates for such a datastructure was unfortunately out of scope of this work and is an item of future work. However, we propose another, alternative approach on how to represent expressions as Strings that covers the case of shifted common subexpressions by approximating array dereferences.

6.2.4   Approach 3: Approximating Array Dereferences

The main detection difficulty of shifted common subexpressions lies within the comparison of expressions where quantifying variables dereference arrays. The third approach is based on a simple observation about quantified array dereferences: if an array is dereferenced by a quantifying variable, it usually involves (part of) the same domain: the index domain. Therefore, we perform a generalisation by representing each quantifying variable that dereferences a dimension in an array with a simple blank '_', as illustrated below

$$
\begin{array}{rcl}
\text{`x[i]'} & \longrightarrow & \text{x[\_]} \\
\text{`x[k]+z[i,2]'} & \longrightarrow & \text{x[\_] + y[\_,2]} \\
\text{`z[i,j]'} & \longrightarrow & \text{z[\_,\_]}
\end{array}
$$

In this way, we match expressions of the same dimension, i.e. we do not match 'z[j,i]' with 'z[j,2]', but would match it with 'z[a,b]' (if 'a,b,i,j' are all quantifying variables). Evidently, this also renders subexpressions common that are not equivalent. For instance, the two occurrences of 'x[i] **mod** n' are not common in the example below

```
   given   n : int (1..)
   find    x : matrix indexed by [int(1..n)] of int(1..n)

such that

   forall i: int(1..n/2−1) .
       x[i] mod n <= n/2,

   forall i: int(n/2..n) .
       x[i] mod n >= n/2
```

since in the first quantification, 'i' ranges over a different domain than in the second quantification. However, we *always* know, that both ranges of 'i' are a subdomain of the index domain of the respective variable, 'x', which is '**int** (1.. n)'[1]. If the two subdomains overlap, then the two expressions share subexpressions. Therefore, it is feasible to flatten both subexpression, using the same auxiliary variable array that is defined over the whole index domain:

```
   given   n : int (1..)
   find    x : matrix indexed by [int(1..n)] of int(1..n)
   find   aux: matrix indexed by [int(1..n)] of int(0..n−1)
```

---

[1]We assume that the problem model is formulated correctly and no index is out of bounds

```
such that
   forall i: int(1..n).
       aux[i] = x[i] mod n,

   forall i: int(1..n/2-1) .
       aux[i] <= n/2,

   forall i: int(n/2..n) .
       aux[i] >= n/2
```

CSE with Approximated Array Dereferences

CSE using String representations of expressions that approximate the array dereferences is very similar to instance CSE, with the slight differences that auxiliary arrays can be iteratively extended.

For illustration consider flattening the example above: first, the subexpression 'x[i] mod n' is flattened to the auxiliary array 'aux' that has length $\frac{n}{2} - 1$:

```
    find aux: matrix indexed by [int(1..n/2-1)] of int(0..n/2-2)
    forall i: int(1..n/2-1).
        aux[i] = x[i] mod n,
```

The expression 'x[i] mod n' is added to the hash-table: where its String format 'x[_] mod n' is linked to 'aux'. Next, when flattening the second quantification, the other occcurence of 'x[i+1] mod n' is flattened: when consulting the hash-table with its String format, 'x[_] mod n', the hash-table returns a match, which is 'aux'.

In the next step, 'aux' is extended so as to be used for the other subexpression: first, the union of both quantifying domains, D, is computed by $D = \{1..n/2 - 1\} \cup \{n/2..n\} = \{1..n\}$, which defines the new length of 'aux'. Similarly, the new domain of 'aux' is computed: quantifying the subexpression 'x[i] mod n' over D and computing the corresponding lower- and upper-bound attributes, which yields '**int** (0.. n−1)':

```
    find aux: matrix indexed by [int(1..n)] of int(0..n-1)
    forall i: int(1..n).
        aux[i] = x[i] mod n,
```

Obviously, if two matched subexpressions are quantified over the domains D_1 and D_2 respectively, then both subexpressions share exactly $|D_1 \cap D_2|$ common subexpressions. Note, that this iterative detection technique is confluent due to the commutativity of $\cup$: it does not matter if we first construct the auxiliary array from D_2 or D_1 and then merge it, since $D_1 \cup D_2 \equiv D_2 \cup D_1$.

However, if the union of the two domains D_1 and D_2 has a hole, then the gap has to be

filled, which introduces unconstrained auxiliary variables. This is the main drawback of this approach.

Summary

In summary, Approach 3 is a light-weight iterative CSE approach that also detects shifted common subexpressions. Under the assumption that no array is indexed out of bounds, it generalises array dereferences, knowing that all references from quantifying variables have to be a subset of the underlying index domain. Therefore, subexpressions that are not necessary equivalent are matched by this approximation, and in the worst case, Approach 3 can add redundancies in form of unconstrained auxiliary variables. This case however, rarely occurs in practical examples.

6.2.5   Summary: CSE at Class Level

In this section we have seen that the basic approach of CSE at instance level is directly applicable at class level. However, there are many kinds of common subexpressions that the instance-level approach does not detect at class level, hence, it has to be extended so as to match these cases.

In this section we have presented three approaches to extend the basic CSE approach from instance level. The first approach applies special quantification normalisations and maintains a hash-table of common quantifiers that is exploited during flattening, where quantifying variables are renamed in order to match equivalent subexpressions. This approach can be quite expensive, in particular when the number of quantifying variables in the same scope is high.

The second approach uses an alternative String representation for quantified variables in the CS hash-table: quantifying variables are represented with their underlying domain, together with labels (introduced lexicographically) to avoid normalisation. In this approach, the actual domains of quantifying variables are compared and not the respective identifier. While this approach is more time-efficient than Approach 1, it still requires reformulation during flattening in order to detect common subexpressions that stem from a permutation of labels.

Furthermore, Approach 1 and Approach 2 cannot detect shifted common subexpressions, which are expressions that are equivalent in different 'iterations' of a quantification. This equivalence can only be detected by performing additional reasoning on the quantifying domain. Eliminating shifted CS can be achieved by either (1) additional reasoning over the domains which requires either a data structure other than a simple String-based hash-table, or (2) applying another, alternative approach, Approach 3, that approximates dereferences of arrays.

Approach 3 is a simple approach, where array dereferences involving quantifying variables are replaced by a blank, assuming that the array dereference will lie in the range of the index domain of the respective array variable. Following this assumption, all array variables that are dereferenced are considered equivalent. This assumption allows us to detect shifted common subexpressions, however, in the worst case, can result in unconstrained auxiliary variables. Note, that the worst case hardly appears in practice and therefore Approach 3 is an interesting alternative to Approach 1 and Approach 2.

To conclude, this section has highlighted and studied the difficulties that arise when attempting to eliminate all CS at class level that we eliminate at instance level. Three possible approaches are outlined, each of which comprises advantages and drawbacks. A complexity study and an empirical analysis of all three approaches is an item of future work.

6.3   Summary

We have seen that all enhancement techniques from instance level are relevant at class level. Moreover, we have seen that lifting instance enhancement techniques to class level creates many challenges that need to be met.

The first challenge we investigated is that of strengthening weak Boolean guards in quantifications to avoid duplicate constraints. Weak guards often arise in models form inexperienced users and in our empirical evaluation, we have observed that duplicates can increase the solving performance by 100% in some examples. At instance level, duplicates can be simply eliminated by help of normalisation which places identical constraints next to another. However, at class level quantifications cannot be unrolled, hence it is necessary to *reason* over the guard and the guarded expression. Thus, we proposed an approach that exploits *unification* in order to strengthen a guard and which is applicable to any kind of quantified, commutative expression. This approach can easily derive all conditions that need to be included into the guard so as to prevent duplicate constraints.

The second challenge we addressed is that of common subexpression elimination at class level. Though CSE as proposed at instance level is directly applicable at class level, there are many kinds of common subexpression it cannot detect that are detected at instance level. The reason for this is that at class level we deal with *quantified* expressions that cannot be unrolled. Therefore, it is necessary to reason over quantifying variables in order to detect equivalence of two expressions.

In particular, we proposed three approaches on how to perform CSE at class level, each with its own advantages and drawbacks. The first approach consists of quantifier normalisation prior to flattening and requires additional reformulations during flattening, which involves renaming quantifying variables. The second approach utilises a special String representation for quantifying variables, where each is represented by its underlying domain, together with a label to avoid ambiguities. This approach also requires additional reformu-

lation (concerning permutations of labels) in order to match all common subexpressions we detect at instance level.

However, Approach 1 and Approach 2 do not detect a special kind of common subexpressions: shifted common subexpressions, which are expressions that are common in different 'iterations' of the quantification. Detecting this kind of common subexpressions was a major challenge that we identified during class-level CSE. We addressed this challenge by proposing Approach 3, that performs an approximation of array dereferences from the observation that all expressions dereferencing an array are a subdomain of the underlying index domain (assuming that no array is dereferenced out of bounds). Therefore, array dereferences involving quantifying variables can simply be represented by a blank in the String representation which considerably facilitates detection on one hand, but, in the worst case, can result in redundancies in form of unconstrained auxiliary variables on the other hand.

In conclusion, we have successfully lifted all instance-optimisations to class level and met many challenges that the class level imposes. However, the general scope of class optimisations goes much further than the techniques we have discussed in this chapter. For instance, representing expressions by their global constraint representation (e.g. representing a clique of disequalities by *alldifferent*) would be a practical and effective class optimisation. Unfortunately, a more thorough investigation of class optimisation techniques was out of scope of this thesis, hence further class optimisation techniques will be investigated in future work.

CHAPTER 7

CASE STUDY: COMMON SUBEXPRESSIONS IN CSPS OF PLANNING PROBLEMS

In this chapter, we consider a particular family of CSPs: those that represent AI planning problems. Constraint Programming is an attractive approach for solving AI planning problems, however, generating effective constraint models of complex planning problems is challenging, and CSPs resulting from standard approaches often require further enhancement to perform well. One reason for this are *redundancies* in form of common subexpressions that, if not eliminated (Sec. 4.2), can have a negative effect on propagation and solving time.

AI planning is an active, long-established research area, with a wide applicability to such diverse tasks as automating data-processing procedures, game-playing, and large-scale logistics problems. The classical AI Planning problem is to find a sequence of actions (a plan) to transform an initial world state into a goal world state. We consider solving AI Planning problems by using Constraint Programming: first we formulate the planning problem as a CSP, then we solve the CSP using a constraint solver and then map the solution back to the planning problem.

In this context, we identify general causes of common subexpressions from three modelling techniques often used to encode planning problems into constraints. Furthermore, we present four case studies of constraint models of AI planning problems. In each, we describe the constraint model, highlight the sources of common subexpressions, and present an empirical analysis of the effect of eliminating common subexpressions.

7.1 Modelling Planning Problems as CSPs

Constraint modelling and solving of planning problems has been studied in the context of many systems, such as CPlan [82], the planning & scheduling framework in [1], the temporal POCL planner CPT [86] or the distributed multi-agent system in [70]. Hence, there exist many different approaches on how to model a planning problem as a CSP. Barták *et al* [8] describe three different constraint models for planning which are derived from different successful modelling approaches. They all share the same basic set of constraint variables:

- $v \times (n+1)$ **state variables** V_i^s, representing the state of the world at step s, where v is the number of properties of a state, n the length of the plan, i ranges over the state properties (i.e. from 0 to v) and s ranges from 0 to n.

- n **action variables** A^s, representing the action chosen at step s, where s ranges from 0 to $n-1$

State and action variables are connected by logical constraints that summarise the chosen action's *preconditions* and *effects* on the state variables, as well as *frame axioms* (i.e. constraints that enforce that certain state properties stay the same during a state transition). Each of the three models differ in how preconditions, effects and frame axioms are represented.

Throughout, we consider the number of steps of the plan to be a parameter to the constraint model. Hence, to find an optimal solution to a given planning problem, one would iteratively increase the value of the steps parameter of the corresponding constraint model until a solution is found.

1. Basic Model
The basic model describes preconditions and effects by the two constraints

$$(A^s = act) \Rightarrow \quad \text{Pre}(act)^s \quad \forall act \in \text{Dom}(A^s) \tag{7.1}$$

$$(A^s = act) \Rightarrow \quad \text{Eff}(act)^{s+1} \quad \forall act \in \text{Dom}(A^s) \tag{7.2}$$

stating that if action *act* is chosen at step s, then precondition $\text{Pre}(act)^s$ and effect $\text{Eff}(eff)^{s+1}$ have to hold. $\text{Pre}(act)^s$ and $\text{Eff}(act)^{s+1}$ are a conjunction of conditions where appropriate state variables are set to *act*'s preconditions at step s, and effects in step $s+1$, respectively. The frame axiom

$$A^s \in \text{NonAffAct}(V_i) \Rightarrow (V_i^s = V_i^{s+1}), \forall i \in (0, v\text{-}1) \tag{7.3}$$

states that if action A^s has no effect on state property i then V_i^s and V_i^{s+1} are equal.

2. Supporting-actions Model

The second model represents supporting actions [22] by adding variables S_i^s that indicate the action responsible for the value of state property i. Preconditions are formulated as in Eq. 7.1 but effects (Eq. 7.4) and frame axioms (Eq. 7.5) are stated as follows (and val is a state value):

$$(S_i^s = act) \Rightarrow (V_i^s = val) \quad \forall act \in \mathrm{Dom}(S_i^s) \tag{7.4}$$

$$(S_i^s = \text{no-op}) \Rightarrow (V_i^s = V_i^{s+1}) \tag{7.5}$$

3. Successor-state Model

In the third model, effects and frame axioms are merged into so-called successor-state constraints [54]. Successor state constraints state that a state variable has value *val* at step s only if either an action has changed it or it was the same in the previous step, formally

$$V_i^s = val \leftrightarrow$$
$$(A^{s-1} \in C(i, val)) \vee (V_i^{s-1} = val \wedge A^{s-1} \in N(i)) \tag{7.6}$$

where $C(i, val)$ is the set of actions that effect $V_i^s = val$ and $N(i)$ is the set of actions that do not affect V_i.

7.2 Sources of Common Subexpressions

In this section we present the general sources of common subexpressions in constraint models of planning problems, obtained by state-of-the-art modelling techniques as described in the previous section.

Common subexpressions in CSPs of planning problems originate in constraints corresponding to effects, preconditions and frame axioms. Our hypothesis is that a model contains common subexpressions if two or more actions share conditions in preconditions, effects or frame axioms.

Common Subexpressions in the Basic Model

In the basic model, preconditions and effects are expressed by Eq. 7.1 and Eq. 7.2, respectively. Assume two actions act_1 and act_2 share conditions in their preconditions, e.g. $\mathrm{Pre}(act_1)^s = a \wedge b \wedge c$ and $\mathrm{Pre}(act_2)^s = b \wedge d$ share condition b, where a, b, c, d are arbitrary conditions. Then the precondition constraint in Eq. 7.1 will share arguments, hence contain

common subexpression b, which we denote (*case A*):

$$(A^s = act_1) \Rightarrow (a \wedge b \wedge c)^s$$
$$(A^s = act_2) \Rightarrow (b \wedge d)^s$$

The same holds if act_1 and act_2 share a condition in effects $\text{Eff}(act_1)$ and $\text{Eff}(act_2)$, when representing effects by Eq. 7.2. Hence if two actions share a condition in in their preconditions/effects, then the corresponding precondition/effect constraints (Eq. 7.1 and Eq. 7.2) will always contain a common subexpression.

The frame axioms in the basic model are given by Eq. 7.3. If two actions act_1 and act_2 have the same frame axiom, then both frame axiom constraints will share the common subexpression $(V_i^s = V_i^{s+1})$, denoted *(case B)*:

$$(A^s = act_1) \Rightarrow (V_i^s = V_i^{s+1})$$
$$(A^s = act_2) \Rightarrow (V_i^s = V_i^{s+1})$$

Therefore, if two actions share frame axioms then this will always result in a common subexpression in the frame axioms according to Eq. 7.3.

Common Subexpressions in the Supporting-actions Model

Preconditions in the supporting-actions model are formulated as in the basic model (see (*case A*)). The effect formulation uses supporting actions in Eq. 7.4. Hence if two actions act_1 and act_2 share conditions in their effect, the effect constraints from Eq. 7.4 will be (*case C*)

$$(S_i^s = act_1) \Rightarrow (V_i^s = val)$$
$$(S_i^s = act_2) \Rightarrow (V_i^s = val)$$

where the common subexpression $V_i^s = val$ is the effect shared by act_1 and act_2. Note, that shared frame axioms will not result in common subexpressions, in contrast to the basic model. The reason for this is that the supporting action S_i^s in the frame axiom constraint (Eq. 7.4) does not consider the actual action that leaves the state variables unchanged (i.e. effectively, the supporting action representation 'eliminates' this kind of common subexpressions by introducing 'no-op').

Common Subexpressions in the Successor-state Model

While preconditions are formulated using Eq. 7.1 as in the basic model (see (*case A*)), effects and frame axioms are merged into one constraint, stated in Eq. 7.6. If the condition of an action, act_1, effects two state properties V_i^s and V_j^s, then subexpression $A^s = act_1$ is

occurs twice in the successor-state constraint, as illustrated in (*case D*):

$$V_i^s = val_1 \leftrightarrow$$
$$(A^{s-1} = act_1 \lor A^{s-1} = act_2) \lor (V_i^{s-1} = val_1 \land A^{s-1} \in N(i))$$
$$V_j^s = val_2 \leftrightarrow$$
$$(A^{s-1} = act_1 \lor A^{s-1} = act_3) \lor (V_j^{s-1} = val_2 \land A^{s-1} \in N(i))$$

Eliminating a common subexpression, i.e. representing each occurrence by the same auxiliary variable during flattening, saves (at least) one variable and one constraint per occurrence, which reduces the resulting constraint instance. The consequence of this reduction is a significant speed-up in solving time. Note that the process of common subexpression elimination does not add any significant computational effort (see Section 4.2 and Sec. 6.2).

In the following sections, we discuss four planning problems where, when modelled as CSPs, common subexpressions arise. All CSPs are modelled by hand, following one of the three approaches discussed in the background section. In the first two examples, the degree of overlap among preconditions, effects and frame conditions is small, and there are correspondingly few common subexpressions. In the later examples the opposite is true. The case studies depict the scalability of common subexpressions elimination on planning CSPs: the more complex the nature of the planning problem, the more common subexpressions arise and the more we can enhance it. Hence this enhancement particularly addresses complex planning problems.

7.3   Case Studies

In the following, we present four case studies, in the order of their complexity, i.e. the first, Sokoban, is represented by a fairly simple model, while the last, Plotting, has the most complex model representation.

7.3.1   Sokoban

The well known Sokoban puzzle/game (see Fig. 7.1) is played in a 2D virtual warehouse. The problem is to find a sequence of horizontal and vertical moves for the sokoban such that every crate is in a goal cell. The sokoban can push a single crate one cell in the direction that he moves. Neither the sokoban nor the crates can move onto a wall cell, and no two objects (sokoban or crate) can occupy the same cell.

Sokoban Constraint Model

The action variable *move*^{s} represents the direction that the sokoban moves at step s. The domain of *move*^{s} contains four elements, representing the four directions in which the sokoban can move. The state is represented by variables

- *avatar*^{s}: the position of the avatar ('sokoban') at step s

- *cratePosn*$_i^s$: the position of crate i at step s

Both state variables range over the cells of the warehouse, that are enumerated row-wise from left to right. Note, that another possibility would be to have a pair of variables representing the coordinate position of the sokoban/crate, but our formulation allows us to model moving the sokoban and pushing the crates very simply, as described below.

Each problem instance is instantiated by a set of parameters that scale the warehouse. The parameters are the width w of the grid, the total number n of cells, the initial positions of the sokoban (*pInit*) and *crates*, the *goal* positions for the crates, and the positions of the *walls*.

Our constraint model is shown in Fig. 7.2. The first constraints, (1) and (2), initialise the state variables. Constraints (3) and (4) prevent either the sokoban or the crates from ever entering a wall cell. Constraint (5) prevents any two crates from being co-located (that the sokoban can never occupy the same cell as a crate is implied by the push constraints below). The goal is captured by constraint (6).

The effect of a move of the sokoban is a change of its (and possibly the crate's) position. Given the row-wise enumeration of the cells, a move left (right) decreases (increases) the cell index by 1, while a move up (down) decreases (increases) the cell index by w. Hence, movement can be modelled as a simple summation (7). Note that this is a simplification of an otherwise more expensive effect constraint. Furthermore, pushing crates can be modelled in the same way, by adding the precondition that the sokoban occupies the same cell
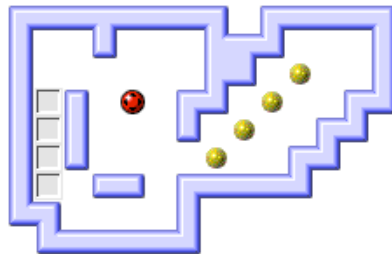

Figure 7.1: A Sokoban instance. The single circle represents the sokoban and the four circles symbolise the crates. Shaded squares are the goal positions of the crates. From http://users.bentonrea.com/~sasquatch/sokoban/

```
given    w, n, pInit, stps          : int (1..)
given    noWalls, noGoals, noCrates : int (1..)
letting WALLS              be domain int (0..noWalls−1)
letting GOALS              be domain int (0..noGoals−1)
letting CRATES             be domain int (0..noCrates−1)
letting POSITIONS          be domain int (0..n−1)
letting MOVES              be domain int(−w,−1,1,w)
letting STEPS              be domain int (0..stps−1)
given walls        : matrix indexed by [WALLS] of POSITIONS
given goals        : matrix indexed by [GOALS] of int (0..n−1)
given crates       : matrix indexed by [CRATES] of int (0..n−1)


find avatar        : matrix indexed by [STEPS] of int (0..n−1)
find move          : matrix indexed by [int (0..stps−2)] of MOVES
find cratePosn     : matrix indexed by [STEPS,CRATES] of int (0..n−1)
find crateMoved    : matrix indexed by [STEPS] of bool


such that
avatar[0] = pInit, $ (1) initialization of the avatar's position

$ (2) initialise crates' positions
forall crate:int (0..noCrates−1).
        cratePosn[0,crate] = crates[crate],

$ (3) avatar must not move on wall
forall s:STEPS .  forall wall : WALLS.
        avatar[s] != walls[wall]

$ (4) crates must not move on wall
forall s:STEPS .  forall c:CRATES .  forall wall : WALLS .
        crates[s,i] != walls[wall],

$ (5) crates have to be at different positions
forall s:STEPS .
        alldifferent(cratePosn[s,..]),

$ (6) in the last step, every crate has to be at a goal position
forall c:CRATES.  exists goal:GOALS.
        cratePosn[stps−1,c] = goals[goal],

$ (7) moving the avatar
forall s:int (0..stps−2).
        avatar[s+1] = avatar[s] + move[s],

$ (8) moving crates
forall s : int (0..stps−2) .  forall c : CRATES .
        (avatar[s+1] = cratePosn[s,c])  => (cratePosn[s+1,c] = (cratePosn[s,c] + move[s])),

$ (9) push crate or leave it
forall s : int (0..stps−2) .  forall c : CRATES .
        (avatar[s+1] = cratePosn[s,c]) \/ (cratePosn[s+1,c] = cratePosn[s,c]),

$ (10) crate has not been moved at the start
crateMoved[0] = 0,

$ (11) crateMoved is true if avatar has moved a crate
forall s:int (1..stps−1).      crateMoved[s] <=>
     exists c:CRATES . avatar[s] = cratePosn[s−1,c],

$ (12) only move in cycles if a crate has been moved
forall s1:int (0..stps−2).    forall s2:int (s1+1..stps−1).
     (avatar[s1] = avatar[s2])  =>  ((sum s3:int (s1..s2). crateMoved[s3]) > 0)
```

Figure 7.2: ESSENCE′ problem class of **Sokoban**

at step $s + 1$ as a crate at step s (8). Following the successor-state formulation (Equation 6), we ensure that crates are either pushed or stay in the same cell (9).

Finally, we exploit that, if no crate is pushed, it is pointless for the sokoban to re-visit the same cell. We introduce a Boolean variable per time step (*crateMoved*), which is true if some crate is pushed (10,11). Then, we allow the sokoban to revisit the same cell only if some crate moved in the interim (12).

Common Subexpressions in Sokoban

The simple effect constraints and frame axioms lead to a small number of common subexpressions in Sokoban. The only source of common subexpressions is the precondition of the sokoban pushing a crate c at step s, stated by `sokPosn[s+1] = cratePosns[s,c]` This condition occurs in constraint (8) that describes the effect of pushing a crate, in the successor-state constraint (9), and in the implied constraint (11) that restricts the sokoban to go in circles only when moving a crate. In summary, we have $(s - 1){*}c$ common subexpressions where s is the length of the plan and c the number of crates.

7.3.2 Settlers

The Settlers problem, introduced in the third International Planning Competition [53], is loosely based on the German board game 'Die Siedler von Catan' (by Franckh-Kosmos Verlags-GmbH & Co.). Each instance has a goal of constructing various buildings across a set of cities. Different cities have access to different raw-materials hence some goods have to be transported between cities in order to construct the required buildings. There are three ways of transporting goods: by cart, train and ship. There are different costs ('labour') associated with creating and operating these forms of transport.

Settlers Constraint Model

In Settlers, actions are the production and transport of goods. The action variables are

- *production*$^s_{i,g}$: units of good g produced in city i at step s

- *export*$^s_{i,g}$: the units of good g exported from city i at step s

- *import*$^s_{i,g}$: the units of good g imported to city i at step s.

Every action variable ranges over the quantity that has been produced/transported. For reasons of brevity, we focus on the main state variables:

```
$ (1) before producing ore, a mine must have been built $
forall city:CITIES . forall s:STEPS .
      production[s, city, ORE] > 0) =>
            buildingBuiltInCity[city,MINE] < s),

$ (2) before exporting ore, a mine must have been built $
forall city:CITIES.  forall s:STEPS.
    (export[s, city, ORE] > 0)   =>
            (buildingBuiltInCity[city,MINE] < s-1),

$ (3) we require goods for constructing buildings and houses $
forall city:CITIES.  forall good:GOODS.  forall s:STEPS.
    buildingRequirement[s,city,good] =
        sum building:BUILDINGS.
            (buildingBuiltInCity[city,building] = s)*
                    requirementTable[building,good]
            + houses[city]*houseRequirements[good]*(s=horizon),

$ (4) the total labour consists of building and production labour $
forall s:STEPS.  forall city:CITIES.
    totalLabour[s,city] =
        sum building:BUILDINGS.
            (buildingBuiltInCity[city, building] = s)*labour[building]
        + sum good:GOODS.
            production[s,city,good]*labour[good]
```

Figure 7.3: Selection of constraints of the Settlers Model in ESSENCE'

- *buildingBuiltInCity$_{i,b}^s$*: true if building b exists in city i at step s

- *buildingRequirement$_{c,g}^s$*: the units of good g required for construction work in city c at time step s.

- *totalLabour$_c^s$*: labour at step s in city c

Fig. 7.3 shows a selection of constraints of the constraint model of Settlers [39]. The maximum number of steps, the amount of cities and building requirements are given as parameters.

The production of a particular material requires the prior construction of the appropriate production site. For instance, to produce ore at time s, a mine must have been built before s. We express these precondition constraints following the standard approach from Eq. 7.1; an example is given in constraint (1). Since export succeeds production, we add similar precondition constraints for exporting a particular good (for an example see constraint (2)). For every city and step, we constrain *buildingRequirement* to equal the sum of goods needed to build construction sites and houses (constraint (3)). We measure the quality of a plan for Settlers by the amount of labour required. The total labour is restricted by constraint (4).

Common Subexpressions in Settlers

There are two main sources of common subexpressions. The first source results from a shared precondition of production and export of a particular good at step s: the appropriate production facility must have been built before s. Fig. 7.3 shows the precondition constraint for production(1) and export(2) concerning ore production. Both constraints share subexpressions of the form

'`buildingBuiltInCity[city,MINE]`< s'

stating the precondition that a mine in *city* was built before step s. This kind of common subexpression occurs for every city, step and production facility (e.g. mine), thus $c*s*5$ times where c is the number of cities, s the number of steps and 5 the number of production facilities.

The second source of common subexpressions arises because the construction of a building b in city c effects both the requirement of goods in c and the amount of labour for building b. Constraint (3) in Fig. 7.3 describes the good requirements, constraint (4) the amount of labour in each city. Both constraints share the subexpression

'`buildingBuiltInCity[city,building]`=s'

stating that b was built in c at step s. In each problem instance, there are $5*c*s$ common subexpressions of this form, where c is the number of cities, s the number of steps and 5 the number of production facilities.

7.3.3 English Peg Solitaire

Peg Solitaire (see CSPLib 37 [36]) is played on a board with a number of holes. In the English version of the game, the board is in the shape of a cross with 33 holes. Pegs are arranged on the board so that at least one hole remains. Moves are draughts/checkers-like and are horizontal or vertical. There are several variations of Peg Solitaire. We focus on the classic *reversal* game in which an initial state with just one peg missing is transformed into a state with a single peg remaining in the same location as the initial hole (Fig. 7.4 shows a sample instance).

Constraint Models of Peg Solitaire

The action variables *moves*^{s} represent the particular move (transition from one cell to another) chosen at step s. *moves* ranges over the 76 possible moves. The state variables

```
      0 1 2 3 4 5 6          0 1 2 3 4 5 6
  0       ● ● ●          0       ○ ○ ○
  1       ● ● ●          1       ○ ○ ○
  2   ● ● ● ● ● ● ●      2   ○ ○ ○ ○ ○ ○ ○
  3   ● ● ● ○ ● ● ●      3   ○ ○ ○ ● ○ ○ ○
  4   ● ● ● ● ● ● ●      4   ○ ○ ○ ○ ○ ○ ○
  5       ● ● ●          5       ○ ○ ○
  6       ● ● ●          6       ○ ○ ○
```
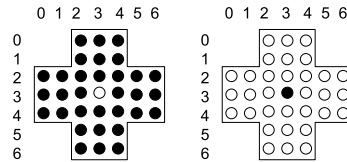
Figure 7.4: Peg Solitaire start (left) and goal (right) board states: black dots mark pegs and white dots mark empty cells.

$board_i^s$ represent the state of cell i at step s as a Boolean value: if true, the cell is occupied by a peg, if false it is empty. It takes 31 steps to remove 31 pegs.

Based upon these variables, we consider two model variants. The first is *action*-centric (Fig. 7.5): for a given move it describes the cells that change and those that stay the same (1), which corresponds to the basic model representation from the Background section. The second is *state*-centric (Fig. 7.6): for each cell, it describes the moves that cause it to change state and those that leave it unaffected (2), which corresponds to the successor-state model. Both models share initial/goal constraints and the implied constraint.

Common Subexpressions in Peg Solitaire

The representation of a state in Peg Solitaire is more complex than for Sokoban, consisting as it does of 33 Boolean variables per step. Since each move affects three cells on the board (and leaves 30 unchanged), there is considerable overlap: The removal of a peg from a particular cell can result from up to 8 different moves, i.e. it is a shared effect. Likewise, the reverse action, placing a peg into a hole, is shared among up to 4 different moves. The biggest overlap occurs in the frame axioms, as a particular cell is left unchanged by up to 72 different actions. It is from this overlap that the common subexpressions in the frame/effect/precondition constraints stem.

Action-centric Model   In the *action*-centric model, we detect common subexpressions that result from effect constraints, as illustrated in (*case A*) in Section 3. For instance, moves '36' and '37' both remove a peg from the centre hole(17), expressed in the legal transition constraint (4) in Figure 7.5), so both actions share effects:

$$(moves^s = 36) \Rightarrow (board_{17}^s > board_{17}^{s+1} \wedge ...)$$
$$(moves^s = 37) \Rightarrow (board_{17}^s > board_{17}^{s+1} \wedge ...)$$

Specifically, a standard instance of the action-centric model has 3,999 common subexpressions, which when eliminated, save 75,857 auxiliary variables (i.e. reducing the number of auxiliary variables from 80,104 to 5,425).

```
letting moveNb :           $ (field1 , field2 ) ——> move
          matrix indexed by [int(1..33),int(1..33)] of int(0..76) be  [...]
letting fieldNb   :        $ (move,{1,2,3}) ——> field
          matrix indexed by [int(1..76),int(1..3)] of int(1..33) be [...]

letting nbSteps            be 31
letting fields             be 33
given   startField         : int(1..fields)
letting STEPS              be domain int(0..nbSteps)
letting FIELDS             be domain int(1..fields)

find moves:          matrix indexed by [int(0..nbSteps-1)] of int(1..76)
find board:          matrix indexed by [STEPS, FIELDS] of bool

such that

$ (1) initial state: all fields are occupied but the one in the centre
forall i : FIELDS .
    (i != startField) => (board[0,i] = true),
board[0,startField] = false ,

$ (2) goal state: only the initial field is occupied
forall i : FIELDS .
    (i != startField) => (board[nbSteps,i] = false),
board[nbSteps,startField] = true ,

$ (3) in every step the number of pegs decreases by 1
forall step : STEPS .
   noFields-step-1 = (sum i : FIELDS . board[step,i]),

$ (4) legal transitions
forall step : int(0..nbSteps-1) .
      forall f1,f2 : FIELDS .

          $ if there exists a legal move from f1 to f2
          ((moveNb[f1,f2] != 0)
           /\ (f1 != f2)) =>

              $ and we select that move, the following holds..
          (   (moves[step] = moveNb[f1,f2]) <=>

                  $ effect and precondition
                ( (board[step, f1] > board[step+1,f1]) /\
                  (board[step,fieldNb[moveNb[f1,f2],2]] >
                   board[step+1, fieldNb[moveNb[f1,f2],2]])
                  /\
                  (board[step, f2] < board[step+1, f2]) /\

                  $ frame axiom
                   forall field : FIELDS .
                      ( (field != f1) /\
                        (field != fieldNb[moveNb[f1,f2],2]) /\
                        (field != f2)
                      ) =>
                        (board[step,field] = board[step+1,field])
                  )
              )
```

Figure 7.5: **Peg Solitaire Action Model** in ESSENCE′

State-centric Model   In the *state*-centric model, common subexpressions arise in the successor-state constraints, illustrated as (*case D*) in Section 3. As an example, consider the constraints (5) and (6) in Fig. 7.6, describing the possible actions when a peg is inserted(5)

```
letting moveNb :          $ (field1 , field2) ──> move
          matrix indexed by [int(1..33),int(1..33)] of int(0..76) be  [...]
letting fieldNb  :        $ (move,{1,2,3}) ──> field
          matrix indexed by [int(1..76),int(1..3)] of int(1..33) be [...]
$ same constants and parameters as in Action Model

find moves:          matrix indexed by [int(0..nbSteps−1)] of int(1..76)
find board:          matrix indexed by [STEPS, FIELDS] of bool

such that
$ same constraints (1)−(3) as in Action Model

$ (4) Frame Axioms:
forall step : int(0..noSteps−1) .
     forall f : FIELDS .
      $ if field f stays the same
     (bState[step, f] = bState[step+1,f]) <=>
          $ then no move has been selected that includes f
        (forall f1 : FIELDS .
           ( (moves[step] != moveNb[f,f1]) /\
             (moves[step] != moveNb[f1,f]) /\

             forall f2 : FIELDS .
               ((( moveNb[f1,f2] != 0) =>
                   ( (f = fieldNb[moveNb[f1,f2],2]) =>
                       (moves[step] != moveNb[f1,f2]) ))
                /\
                (( moveNb[f2,f1] != 0) =>
                   ( (f = fieldNb[moveNb[f2,f1],2]) =>
                       (moves[step] != moveNb[f2,f1]) ))
               )
          )
        ),

$ (5) Moving a peg to field f
forall step : int(0..noSteps−1) .
   forall f : FIELDS .      $ 0 −> 1
      (bState[step,f] < bState[step+1,f]) <=>

          exists f1 : FIELDS .
          (moveNb[f1,f] != 0) /\
          (moves[step] = moveNb[f1,f]),

$ (6) Removing a peg from field f
forall step : int(0..noSteps−1) .
   forall f : FIELDS .      $ 1 −> 0
      (bState[step,f] > bState[step+1,f]) <=>

         (exists f1 : FIELDS .
            $ middle peg
            (exists f2 : FIELDS .
                ( moveNb[f1,f2] != 0) /\
                (moves[step] = moveNb[f1,f2]) /\
                (f = fieldNb[moveNb[f1,f2],2])
            )
             \/
            $ start peg
            ( ((moveNb[f,f1] != 0) /\
              (moves[step] = moveNb[f,f1]))
            )
         )
```

Figure 7.6: **Peg Solitaire State Model** in ESSENCE′

or removed(6): for move '36' (moving the centre peg(17) to north) there are 2 occurrences of $moves^s = 36$:

$$(board_{17}^s > board_{17}^{s+1}) \Leftrightarrow (moves^s = 36 \vee ...)$$
$$(board_{17}^s < board_{17}^{s+1}) \Leftrightarrow (moves^s = 36 \vee ...)$$

A typical instance of the state-centric model contains 5,890 common subexpressions, which when eliminated, save 30,039 auxiliary variables (reducing the number of auxiliary variables from 38,750 to 8,711).

7.3.4 Plotting

Plotting is a puzzle game made by Taito in 1989, see Fig. 7.7. It is played on a 14x14 grid, where the perimeter is composed of solid wall cells. The sub-grid on the bottom-right of the play area contains an arrangement of blocks of one of four types (for simplicity we exclude a fifth, wildcard, type from the grid). The player avatar can move up and down the first column. The avatar carries a single block of one of the types. It can throw this block horizontally along the row it occupies. At the start of the game, the avatar carries a wildcard. The effects of throwing a block against a wall are:

- If it hits a wall as it is travelling right, it falls vertically downwards. Additional walls are arranged to facilitate hitting the blocks from above, as shown in the figure. This arrangement varies with instances of the puzzle — in harder instances wall cells are placed so as to prevent throwing blocks along some rows and columns.
- If it falls onto a wall, it rebounds into the avatar's hand.

A thrown wildcard transforms into the same type as the first block it hits. For the other block types:

- If the first block a thrown block hits is of a different type from itself it rebounds into the avatar's hand.
- If a block A hits a block B of the same type, B is consumed and A continues to travel in the same direction. All blocks above B fall one grid cell each.
- If a thrown block A, having already consumed a block of the same type, hits a block B of a different type, A replaces B, and B rebounds into the avatar's hand.

If, after making a throw, the block that rebounds into the avatar's hand is such that there is now no possible throw that can further reduce the blocks, the player loses a life and the block in the avatar's hand is transformed into a wildcard block. The game is over if the player has no lives remaining. The aim of the game is to reduce the initial configuration of blocks so that at most some specified number remain.

Figure 7.7: Screenshot of Taito's Plotting game.

Plotting Constraint Model

Plotting can be seen as a planning problem. Our model captures an attempt to find a mistake-free solution to an instance, so moves leading to a loss of life will not be allowed. The constraint model is very large, so we restrict our discussion to the main features of the model.

A single action is possible at each step. We abstract away the use of the wall cells, and assume simply that the avatar throws a block either along one of the r rows or down one of the k columns. This is modelled using the pair of action variables:

- $trow_i^s$: the row along which a block is thrown (0..r)

- $tcol_j^s$ the column along which a block is thrown (0..k)

The 0 value is used to record that no block was thrown along a row (or column) at this time step. A simple constraint ensures that exactly one of this pair of variables takes the value one at each time step.

There are several variables representing the state:

- $hand^s$: the block in the avatar's hand at step s; ranges over the different block types (represented by integers)

- $grid_{i,j}^s$: the state of cell at row i and column j; ranges over all block types, including 0 (empty)

An instance is obtained by instantiating the following parameters: the number of steps in the plan, the width k and height r of the grid of blocks; the initial contents of the grid; the number of steps allowed s; the goal number of blocks remaining; and the number of block types (a generalisation of the original problem).

The initial wildcard in the avatar's hand is modelled simply by leaving *hand* at step 0 unconstrained. We constrain each move to be useful (remove at least one block) by insisting that the sum of each $grid^s$ is less than that of $grid^{s-1}$.

Effects and frame axioms are modelled according to the basic model in Section 2. The main effects are: grid cells becoming empty or changing block type and changing the block type in the hand. Frame axioms are: grid cells and the hand remaining unchanged. Due to the extent of the constraint model we do not go into further detail[1].

Common Subexpressions in Plotting

Plotting is the most complex of our case studies and has the most common subexpressions, arising from precondition effect and frame axiom constraints corresponding to (*case A,B*) in Section 3. Important overlaps are:

- *Cell status*: many common subexpressions stem from the shared condition that a cell is empty at step s. It is an effect of hitting blocks, a precondition for hitting consecutive blocks, and also contained in the frame axiom that an empty cell will always stay empty. Similarly, the opposite condition, that a cell contains a block at step s is shared among effects and preconditions.
- *Throwing blocks*: the precondition that a block is thrown from a (particular) row or column is shared among several actions, such as aiming for a particular wall or block type.
- *Frame axioms*: many cells are unaffected by different shots, another source of common subexpressions.
- *Comparing block types*: the precondition that the block in the avatar's hand is the same as a particular block in the grid applies to different actions on the grid. The opposite precondition, that the types differ, is also shared by different actions.
- *Shared conjunctions of conditions*: there are several conjunctions of the above mentioned conditions that form another big group of common subexpressions. For instance, the conjunction of "*cell (1,4) is not empty*" and "*cell(4,1) has the same block type as the hand*" is shared among the action "shoot from row 4 at *cell(4,1)*" and the action "shoot from column 1 at *cell(4,1)*".

7.4 Experimental Results

In this section we illustrate the potential benefits we can achieve when applying common subexpression elimination to our four case studies. We formulated each model in the solver-independent modelling language ESSENCE$'$ and used Tailor v0.2 to flatten each instance for

---

[1]The Plotting constraint model is available at: http://www.cs.st-and.ac.uk/∼andrea/tailor

input to the constraint solver Minion v0.7.0 [32]. Tailor provides optional common subexpression elimination, hence for every problem instance, we generate one file with common subexpression elimination and one without. This translation process takes the same amount of time in both cases since common subexpression elimination is a particularily cheap enhancement technique. We solve both instances with the same branching heuristic(action variables before state variables) and same search heuristic on the same machine (dual-Xeon 5430, 2.66GHz, 8GB RAM, Linux 2.6.18-92.1.13.el5).
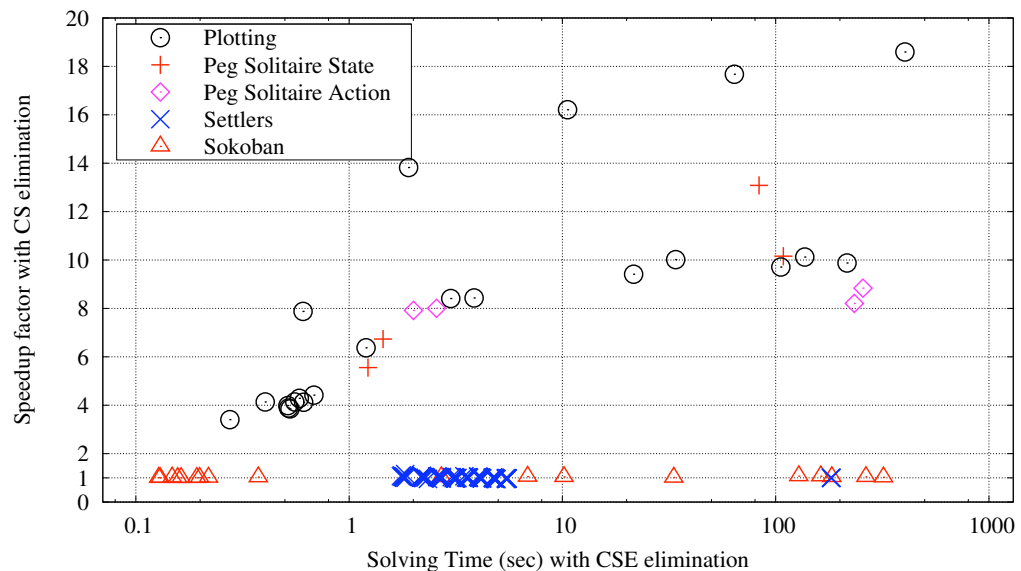
Figure 7.8: **Solving time speed up**. The (logarithmic) x-axis represents the solving time *with* elimination. The y-axis gives the factor to multiply this by to obtain the solving time *without* elimination. As an example, typical Peg Solitaire Action instances are solved $8\times$ faster with common subexpression elimination. Points above $y = 1$ represent instances which is solved faster with elimination than without. Flattening time is excluded but is usually similar with or without elimination.

First, we compare runtimes (summarised in Fig. 7.8). We see significant run time improvements in the Plotting problem and both models of Peg Solitaire. Each of these families can give a $10\times$ or better speedup. The speedups generally improve with problem difficulty. Benefits are slight on the Sokoban instances, ranging from no improvement to only a 7% speedup, although the speedup did improve slightly as problems got harder. For Settlers we saw mixed results: while we did get up to a 15% speedup, a few instances ran up to 6% slower when elimination was used. This slight slowdown may be due to fluctuations in performance from run to run, or detailed features of Minion performing differently on the different instances. Speedups are mostly due to reduction in work for the same node-count. Most instances took the same number of search nodes with and without elimination. The exceptions were the state model of Peg Solitaire, and Plotting. For those Solitaire instances, nodes searched was reduced by about 2.5 times, so even here we see the runtime reduction was greater than the nodecount. A small number of Plotting instances showed a tiny reduction in search.
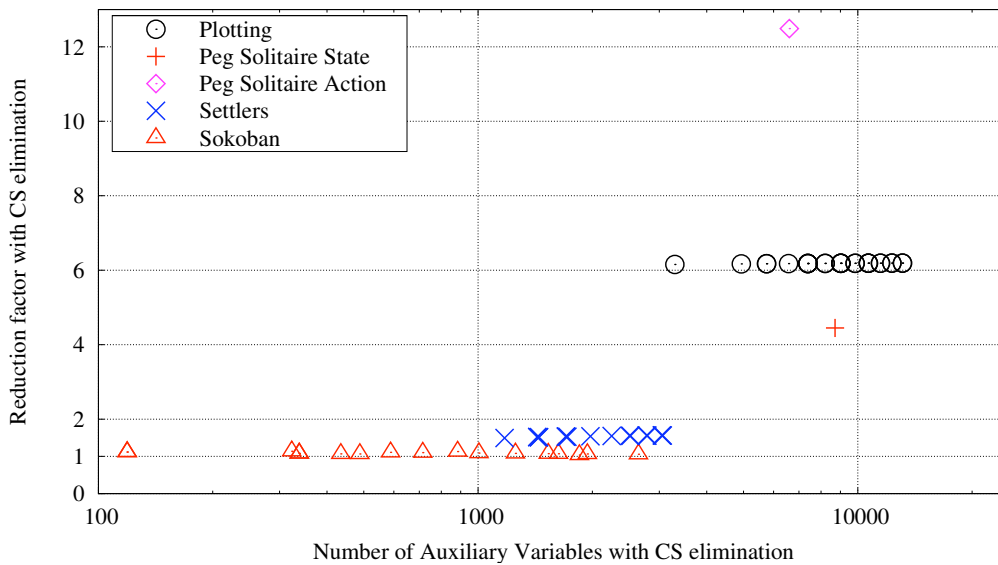
Figure 7.9: **Reduction of Auxiliary Variables**. The x-axis represents the number of auxiliary variables introduced *with* common subexpression elimination, and the y-axis the factor reduction over not using elimination. As an example, Plotting instances with common subexpression elimination have only $\frac{1}{6}$ of the number of auxiliary variables than Plotting instances without. All points are above $y = 1$, so we always use less variables when using elimination. Peg Solitaire instances only differ in the starting hole, so they all have the same number of auxiliary variables.

Second, we compare the size of problem instances with and without common subexpression elimination. Results in Fig. 7.9 show that we always use fewer auxiliary variables this way. The smallest factor is 1.03, i.e. a 3% improvement, and the largest represents a factor of $12.5\times$ fewer auxiliary variables. It is particularly interesting that the reduction in each family is very consistent across problem size. We obtained similar results (not illustrated) when we looked at the number of constraints in each instance. Again results were consistent in each family, the maximum factor being $9.4\times$. We observe that there is, as we expected, a strong correlation between families for which we obtain large reductions in the size of problems, and for which we obtain good runtime improvements.

7.5 Summary

We discussed the general sources of common subexpressions in constraint models of AI planning problems. For illustration, we considered four case studies of different complexity, in which we formulated constraint models using standard techniques and highlighted the sources of common subexpressions. Our empirical analysis demonstrated the potential benefits of eliminating common subexpressions, a computationally cheap procedure. For the problems that exhibited the greatest overlap, Peg Solitaire and Plotting, the reduction

in solving time when common subexpressions were eliminated was dramatic and reached up to a $18\times$ speedup.

[8] suggest the use of table constraints to express preconditions/effects/frame axioms. While this would eliminate many common subexpressions, it is not always feasible. Consider the Plotting problem. The number of state variables involved in, for example, action preconditions would mean that the table constraints required would have a very high arity and would therefore be very cumbersome to specify and to propagate.

162

CHAPTER 8

EXPERIMENTS

In this chapter we summarise our empirical analysis concerning the model optimisation techniques proposed in this thesis. We empirically evaluate each optimisation technique on a large set of examples and give an interpretation of the results. The combination of all optimisation techniques is summarised in the last section, Sec. 8.6 which outlines the practical aspects of the contributions of this thesis, Since all experiments share the same basic setup, we begin with specifying the exact experimental setup.

8.1 Experimental Setup

The basic experimental setup is the same for all our experiments (if it differs it will be explicitly mentioned). The experiments have been conducted on the same machine, a Mac Pro 4.2 with 8 GB RAM that contains 2 Quad-Core Intel Xeon 5500 series processors, each 2.26 GHz (note that hyper-threading was turned off).

In all our experiments we used the tailoring tool TAILOR for generating solver input to either (or both) MINION [32] or (and) Gecode [80]. In the following, we specify the settings for tailoring, solving and give a detailed overview of the set of problem classes that we have used in our experiments. Note that all problem instances and tools that we have used in our experiments are freely available and referenced accordingly.

8.1.1 Tailoring Setup

All problem instances are tailored using TAILORv0.3.2 with Java 1.5.0 on a Mac Pro 4.2 with 8 GB RAM on a 2.26 GHz processor (as mentioned above), where the Java virtual machine had a memory upper bound of 512MB. All problems are tailored on the same machine using the same general settings.

8.1.2   Solving Setup

Problems are solved on either with solver MINION [32] or solver Gecode [80]. Note, that since the translation to the former is more advanced in TAILOR, the majority of experiments is conducted on MINION. We apply the default search heuristics in both solvers and use the same general timeout. Note, that these experiments are *not* intended to compare solvers, but to assess the impact of optimisation techniques during tailoring for different solvers.

Solving Setup for MINION

We use MINION 0.9 with a timeout of 20min (1200sec). We use the default search heuristics (ascending, i.e. smallest value first) and search on decision variables before searching auxiliary variables. Decision variables are searched in order of their declaration in the problem file (which is the same as for solving in Gecode). Note, that those auxiliary variables that represent common subexpressions are at the top of the auxiliary variable search order. When solving satisfaction problems, we search for the first solution, in optimisation problems we (naturally) search for the best solution.

Solving Setup for Gecode

We use Gecode's frontend tool 'fz', Gecode-FlatZinc interpreter [58](freely available at [79]), which is a FlatZinc [44] interpreter for Gecode 3.2.2 that creates a Gecode Space object representing the problem model, executes the object and returns a solution. This means, that the overall solving process with Gecode is setup as follows: first we generate FlatZinc output using TAILOR, and then we solve the instance with Gecode using the Gecode-FlatZinc interpreter that returns the solution(s). Note, that the Gecode-FlatZinc interpreter works as a simple interpreter without performing any instance manipulations. The benefits of using Gecode-FlatZinc interpreter is that in most cases, it is much faster than the alternative approach of first generating a Gecode C++ class with TAILOR, then compiling the class and finally executing it.

We use Gecode/FlatZinc 3.2.1 that operates on Gecode 3.2.2 with a timeout of 20 minutes (1200sec). We use the default search settings (smallest domain first), where decision variables are branched before auxiliary variables, in the order they were defined in the model.

8.1.3   Problem Models

We use a wide selection of problem classes that we divide into three groups that reflect the scope of possible enhancement and which we describe below. Note, that each problem was assigned to a group *after* performing the experiments. All problems are either formulated in ESSENCE′ (Chapter 2) or in XCSP 2.1 Format [68]. All ESSENCE′ models follow either a

model formulation from the literature (which is appropriately cited) or follows descriptions (or other models in other modelling languages) from CSPlib [36]. Note, that all ESSENCE′ problem models are available online at TAILOR's homepage [65]. All XCSP 2.1 instances are taken from the XCSP solver competition benchmark website [50].

In the following, we present each group in more detail, and briefly describe each problem class. Note, that the abbreviation in parenthesis next to each problem class represents the abbreviation used in the experiment graphs. The overall selection of problems contains problems of different flavour: academic combinatorial problems (e.g. n-queens), practical problems (e.g. crosswords), optimisation problems (e.g. Golomb Ruler), planning problems (e.g. English Peg Solitaire) and game puzzles (e.g. Plotting).

Problem Group A

All problems in Group A are problems that have no scope for enhancement (wrt our optimisation techniques). They are problems of different size: for instance, the Langford Number Problem has a very concise problem formulation where typical instances have less than 100 constraints, whereas in the Balanced Incomplete Block Design Problem (BIBD), typical instances contain several 1000s of constraints. The main reason for including this kind of problem classes into our experiments is to determine if our optimisation techniques have a negative impact on the tailoring time when performed *in vain*.

1. **Balanced Incomplete Block Design**, (bibd), is CSPlib Problem 28, concerned with assigning v objects to b blocks such that such that each block contains k different objects, each object occurs in exactly r different blocks and every 2 distinct objects occur in exactly λ blocks.

2. **Langford Number Problem** (langfordN), CSPlib Problem 24 that is to Arrange n sets of positive integers $1..k$ to a sequence, such that, following the first occurrence of an integer i, each subsequent occurrence of i, appears i+1 indices later than the last. For example, for n=2 and k=4, a solution would be 41312432.

3. **Langford Number Problem 2** (langford2), CSPlib Problem 24, where $n = 2$ which results in a simplified version of the general model in 2.

4. n-**Queens Problem (IP formulation)** from [57], which is to place n queens on an $n \times n$ chessboard without attacking each other.

5. **Quasigroup Existence Problem 3, idempotent** (quasigroup3I), is CSPlib Problem 3: an m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where each element occurs exactly once in each row and column and certain multiplication axioms hold (in this case, we want axiom 3 to hold) and enforce idempotency.

6. **Quasigroup Existence Problem 5, idempotent** (quasigroup5I), is CSPlib Problem 3: an m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where

each element occurs exactly once in each row and column and certain multiplication axioms hold (in this case, we want axiom 5 to hold) and enforce idempotency.

7. **Quasigroup Existence Problem 5, non-idempotent** (quasigroup5NI), is CSPlib Problem 3: an m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where each element occurs exactly once in each row and column and certain multiplication axioms hold (in this case, we want axiom 5 to hold) and enforce non-idempotency.

Problem Group B

Problem Group B contains problems with small to medium scope of enhancement (wrt our enhancement techniques). This group contains mainly XCSP 2.1 instances (whose descriptions stem mainly from the corresponding website) and some ESSENCE$'$ instances. Some of them are optimisation problems, such as the Peaceful Army of Queens Problem.

1. **Chessboard Colouring** (cc), XCSP 2.1 benchmark: the task of colouring all squares of a chessboard composed of r rows and c columns. There are exactly n available colours and the four corners of any rectangle extracted from the chessboard must not be assigned the same colour.

2. **Crossword Words Puzzle** (cwM1), XCSP 2.1 benchmark: given a grid and a dictionary, the problem is to fill the grid with the words contained in the dictionary (using the dictionary found under /usr/dict/words under Linux) and the puzzles mentioned in [37].

3. **Crossword Herald puzzle** (cwHerald), XCSP 2.1 benchmark given a grid and a dictionary, the problem is to fill the grid with the words contained in the UK cryptic solvers dictionary and grids taken from the Herald Tribune (Spring, 1999).

4. **Knight's Tour** (knightsXCSP), XCSP 2.1 benchmark: a knight is placed on a chessboard and, moving according to the rules of chess, must visit each square exactly once.

5. **Peaceful Army of Queens, model2** (paq2), from [76]: place two equally sized 'armies' of black and white queens on an $n \times n$ chessboard such that no pair of differently-coloured queens exists that attacks another. The objective is to maximise the size of the armies.

6. **Quasigroup Existence Problem 3, non-idempotent** (quasigroup3NI), is CSPlib Problem 3: an m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where each element occurs exactly once in each row and column and certain multiplication axioms hold (in this case, we want axiom 3 to hold) and enforce non-idempotency.

7. **Quasigroup Existence Problem 4, idempotent** (quasigroup4I), is CSPlib Problem 3: an m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where each element occurs exactly once in each row and column and certain multiplication axioms hold (in this case, we want axiom 4 to hold) and enforce idempotency.

8. **Quasigroup Existence Problem 4, non-idempotent** (quasigroup4NI), is CSPlib Problem 3: an m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where each element occurs exactly once in each row and column and certain multiplication axioms hold (in this case, we want axiom 4 to hold) and enforce non-idempotency.

9. **Queen Attacking Problem** (queenAttacking), problem 29 in CSPlib, XCSP 2.1 benchmark: the task of putting a queen and the n^2 numbers $1,...,n^2$, on an $n \times n$ chessboard so that no two numbers are on the same cell, any number $i+1$ is reachable by a knight move from the cell containing i and the number of cells containing a prime number that are not attacked by the queen is 0.

10. **Queens-Knights Problem** (queensKnightsMul), XCSP 2.1 benchmark: concerned with putting q queens and k knights on an $n \times n$ chessboard such that no two queens can attack each other and all knights form a cycle (when considering knight moves). In this problem version squares on the chessboard may not be shared by queens and knights.

11. **Sokoban** (sokoban1 and sokoban2) from [30]: a Japanese warehouse game where an avatar (the 'Sokoban') has to move a set of crates to designated positions in the warehouse using a specified number of steps (see Sec. 7.3.1 for more details)

Problem Group C

Problem group C contains all problems that have a substantial scope for enhancement. They contain the most complex problems and largest problem models, include optimisation problems, planning problems and general puzzles.

1. **English Peg Solitaire, action-centric and state-centric Model** (pegAction) and (pegState) from [43]: played on a board with holes and pegs where in the initial state all holes are filed with pegs except one and following checkers-like moves such that in the final state all pegs are removed except one that is placed in the hole that was empty in the initial state. See Sec. 7.3.3 for more details.

2. **Golomb Ruler Problem** (golomb), naive model from [77], problem 6 from CSPlib: find a ruler of minimal length such that the distances between all ticks on the ruler are different.

3. **Knight's Tour** (knights), following the C++ model from Gecode's webpage [80] that cites Gert Smolka

4. n-**Queens Model** (nQueensNaive), naive model from [57]

5. **Peaceful Army of Queens, Model1 and Model 3** (paq1), both models from [76]: place two equally sized 'armies' of black and white queens on an $n \times n$ chessboard such that no pair of differently-coloured queens exists that attacks another. The objective is to maximise the size of the armies.

6. **Plotting** (plotting), from [30]: Plotting is a puzzle game played on a 14x14 grid, where an avatar has to reduce a grid of blocks of different types by throwing blocks at the grid in a particular number of steps. For further details see Sec. 7.3.4.

8.2 Basic Common Subexpression Elimination

We start with the empirical evaluation of basic common subexpression elimination (CSE) from Sec. 4.2. Note, that for brevity, we only include our results for targeting solver MIN-ION in this study. However, we have observed similar results for solver Gecode, which is demonstrated in Sec. 8.6, where we conduct a thorough study of the combination of all enhancement techniques that we proposed throughout this thesis, targeting both MINION and Gecode.

8.2.1 Auxiliary Variable Reduction (Eliminated CSs)

We begin with studying the number of common subexpressions that we can eliminate during tailoring. Note, that the reduction of auxiliary variables directly reflects the number of eliminated common subexpressions: for each common subexpression we eliminate, we save one auxiliary variable (Sec. 4.2). Therefore, we study the number of eliminated common subexpressions by considering the reduction of auxiliary variables in each instance. CSE fires only in group B and C and therefore we exclude group A from this discussion, since the instances obtained with and without CSE are identical.

Fig. 8.1 shows the reduction of auxiliary variables of group B(top) and group C(bottom). In both graphs, the x-axis represents the number of auxiliary variables in each instance when CSE was applied. The y-axis denotes the auxiliary variable reduction factor, i.e. it denotes with which factor the auxiliary variables are reduced This means that the further a point is positioned below $y = 1$, the more CSs (and hence auxiliary variables) could be eliminated. For example, points along $y = 0.5$ represent instances where CSE could reduce the number of auxiliary variables to 50% (i.e. the instance contains half as many auxiliary variables than without CSE).
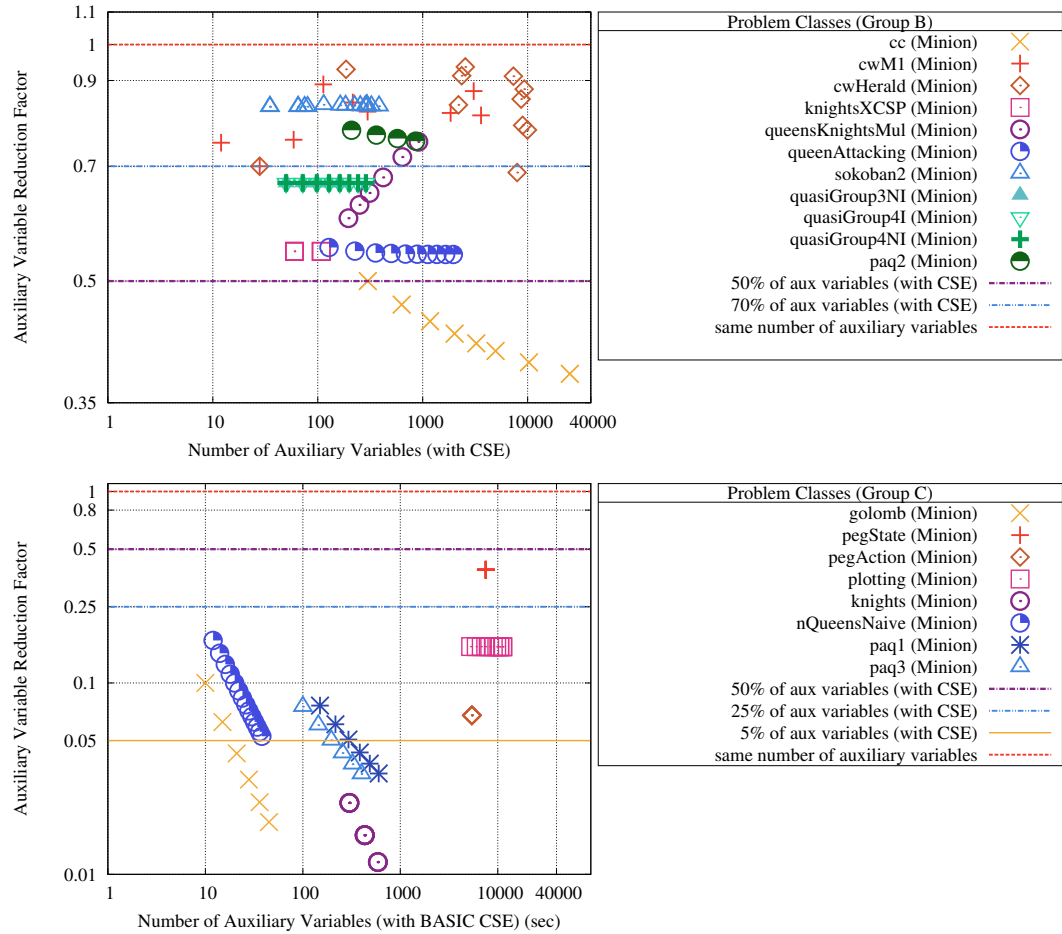
Figure 8.1: **Reduction of Auxiliary Variables** by tailoring using **basic CSE** on problem group B (top) and group C(bottom), which reflects the **number of eliminated common subexpressions** (every eliminated CS saves one variable). The y-axis represents the factor with which the number of auxiliary variables is reduced due to CSE. For instance, points at $y = 0.5$ represent instances where the number of auxiliary variables was reduced to 50% compared to no applying CSE, i.e. CSE has reduced the auxiliary variables by half.

Auxiliary Variable Reduction in Problem Group B

For group B we see that every instance could be reduced with CSE (since all points are below $y = 1$). The reduction is not dramatic, but still considerable, ranging from 10%-20% for crosswords and sokoban, up to over 50% for queenAttacking and Chessboard Colouring. It is also nice to observe *how* the percentage of eliminated common subexpressions changes with instance size (i.e. changes with respect to the parameters scaling the instance): in some problems, such as the quasiGroups or Sokoban, the percentage of eliminated CS remains the same, irrespective of the instance size. In other problems, such as the peaceful Armies of Queens or Chessboard Colouring, the percentage of eliminated CS *increases* with instance size. Furthermore, in Knights-Queens we even observe a *reduction* of eliminated CSs when the instance increases.

Auxiliary Variable Reduction in Problem Group C

The results for group C are even more impressive than those obtained for group B: we observe dramatic reductions by CSE, reducing the number of auxiliary variables down to 5% of the instance obtained without CSE - in the case of the Knight's Tour even down to almost 1%. For illustration, in a medium-sized instance of Peaceful Army of Queens (Model3), CSE reduced the number of auxiliary variables from 5,952 to 256. Note, that all instances of Peg Solitaire (both state- and action-centric Models) are equally sized and therefore represented by only one point. Similar to group B, we can see that in many problem classes, the number of eliminated common subexpressions *increases* with the size of the respective parameters (with the exception of Plotting and Peg Solitaire).

Auxiliary Variable Reduction: Summary

We have considered the reduction of auxiliary variables due to CSE in two problem groups: group B contains problems with fair to medium scope for enhancement and group C contains problems with large scope for enhancement. In both cases, we have seen impressive reductions (in group B up to 50% and in group C up to 1% of the number of auxiliary variables of the unenhanced instance).

In many problem classes, we observed that the number of eliminated auxiliary variables (i.e. the number of eliminated common subexpressions) increases with the parameters that scale the problem. This is an important observation that leads to the conclusion that the benefits of CSE are even greater for large instances of these problem classes.

8.2.2 Reduction in Constraints

Second, we consider the constraint reduction that comes along with CSE. In particular, the constraint reduction stems from the elimination of those constraints that reify auxiliary variables that could be eliminated due to CSE. Since CSE fires only on group B and C, we exclude group A from this discussion.

Fig. 8.2 shows the constraint reduction for problems in group B (top) and group C (bottom). The graphs are set up in the same fashion as the graphs considering the reduction of auxiliary variables: the x-axis represents the number of constraints with CSE and the y-axis shows the reduction factor, so that the further a point is positioned below $y = 1$, the more constraints could be eliminated. We can immediately see that the constraint reduction is not as dramatic as the auxiliary variable reduction. In the following, we discuss the respective reduction in the context of each problem group.
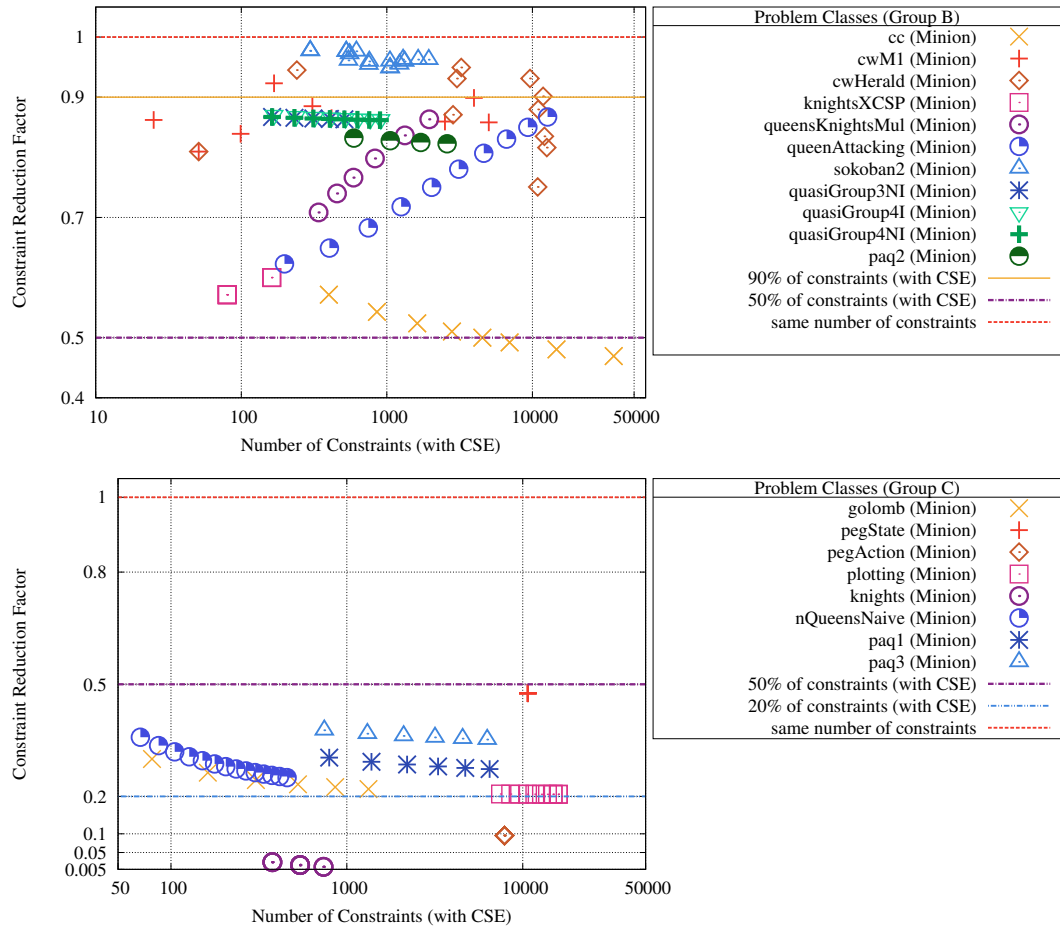
Figure 8.2: **Reduction of Constraints** by tailoring using **basic CSE** on problem group B (top) and group C(bottom). The y-axis represents the factor with which the number of constraints is reduced due to CSE. For instance, points at $y = 0.5$ represent instances whose number of constraints was reduced to 50% of the number of constraints without CSE, i.e. CSE has reduced the constraints by half.

Constraint Reduction in Problem Group B

As we can see in the top graph of Fig. 8.2, the constraint reduction in group B ranges between 5% (Sokoban, Herald Crosswords) and 50% (Chessboard Colouring). Most interestingly, we can see how the reduction behaves with increasing parameters: in some problem classes, like Queen-Attacking and Queens-Knights, the constraint reduction get smaller as the instance size increases. This happens if the set of constraints that increases with the problem parameters is not affected by CSE.

Constraint Reduction in Problem Group C

The constraint reduction in group C is more dramatic than in group B but not as substantial as the reduction of auxiliary variables, ranging from 50% (Peg Solitaire State) to 30-20% for most problems, down to 5% for the Knight's Tour problem. Similar to the reduction of auxiliary variables, the constraint reduction increases for most problems with the parameter size, with the exception of Peg Solitaire and Plotting. As expected, this is the exact same behaviour as for auxiliary variables.

Constraint Reduction: Summary

In summary, we have seen that the constraint reduction is similar to the auxiliary variable reduction but less dramatic, yielding a reduction to 95-45% of constraints in instances of group B and a reduction to 50-5% of constraints in instances of group C. Moreover, the constraint reduction scales with the problem size, similar to the reduction of auxiliary variables. In conclusion, the constraint reduction due to CSE is impressive and often scales with the problem parameters.

8.2.3 Tailoring Time with CSE

After considering the instance reductions, we consider the impact of CSE on the overall tailoring time on each of our three problem groups. Fig. 8.3 depicts the difference in tailoring time when tailoring to solver Minion in three graphs, where each point denotes an instance. In each graph the x-axis represents the tailoring time *with* CSE and the y-axis represents the factor of tailoring time increase and decrease, respectively, when applying CSE. This means that all points above $y = 1$ represent instances where tailoring time was *reduced* when applying CSE and points below $y = 1$ show cases where the tailoring time was *increased*.

Tailoring Time in Problem Group A Problem Group A contains problems where CSE does not fire since there is no scope for further enhancement. The topmost graph in Fig. 8.3 depicts the tailoring time difference when applying CSE. We observe that most instances are evenly positioned between the range of $y = 0.9$ to $y = 1.1$ and $x = 0.1$ to $x = 0.5$, i.e. the the difference in tailoring time is about $\mp 10\%$, which, in a tailoring time of 0.1-0.5 seconds is marginal and can be expected between two separate runs on the Java virtual machine. From this we conclude that CSE does not negatively affect tailoring time when applied to instances in vain.

Tailoring Time in Problem Group B Group B includes problems that have little to medium scope for enhancement. We see results that are very similar to those obtained
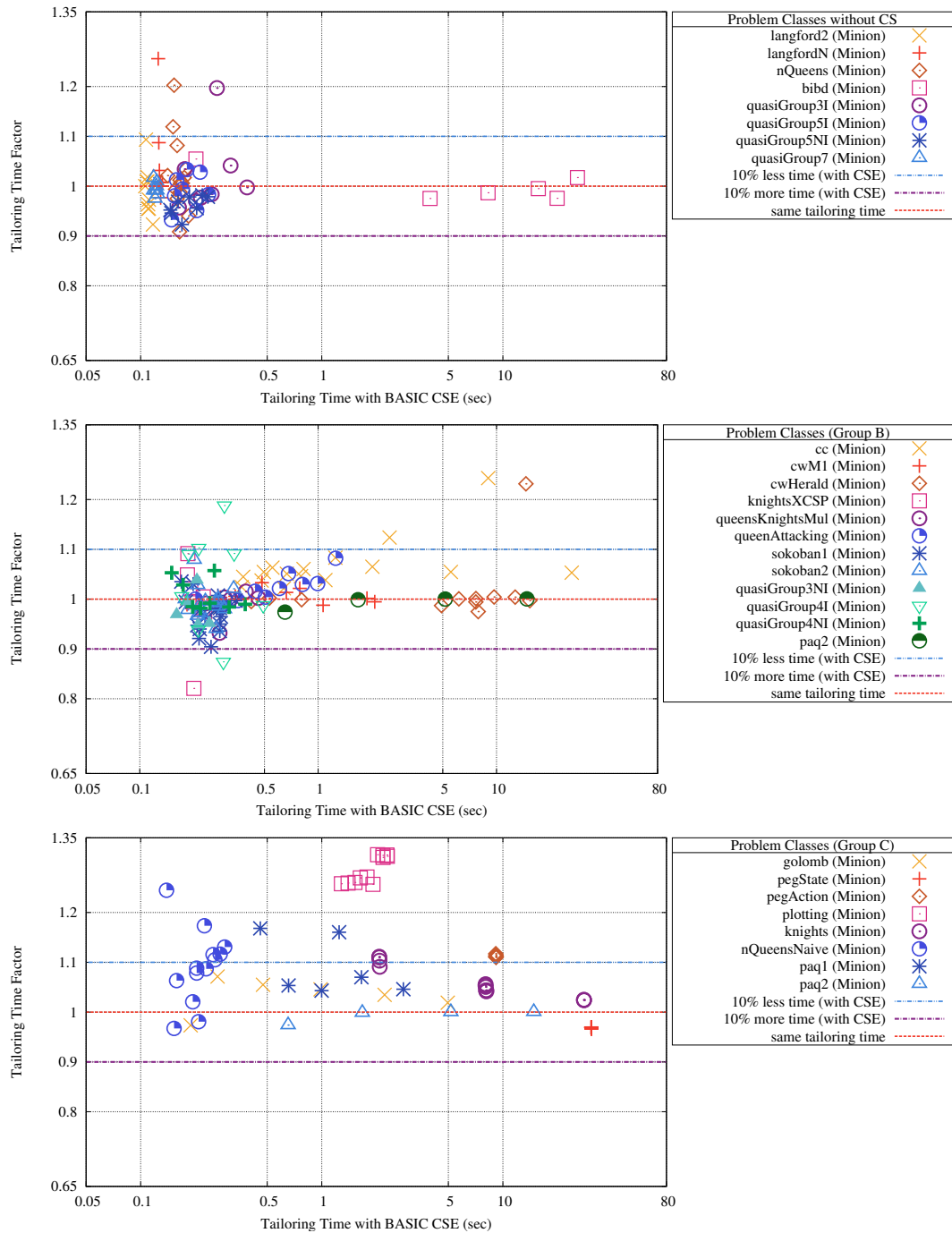
Figure 8.3: **Tailoring Time Comparison** between tailoring **no enhancement** and tailoring with **basic common subexpression elimination** on problem groups A(top), B(middle) and C(bottom). The y-axis represents the factor with which the tailoring time differs between both tailoring options: points above $y = 1$ depict cases where tailoring time was *decreased* when applying basic CSE, points below depict cases where tailoring time was *increased*.

for group A: first, instances that are tailored in 0.5-0.1 seconds are tailored within a time

difference of $\mp 10\%$, which is marginal and can be expected in two separate runs. Moreover, we can see that instances that are tailored in more than 0.1 seconds, the difference becomes smaller with a tendency of a slight *reduction* in tailoring time (more instances are above $y = 1$ than below). Second, we observe that instances of classes where the *most* common subexpressions are eliminated (cc and queenAttacking), the tailoring time is always reduced (though just slightly).

Tailoring Time in Problem Group C   The results for group C, the group containing those problem classes with the largest scope of enhancement, are most impressive: first, we notice that in most cases, tailoring time is reduced. In the case of Plotting even by over 30%. This demonstrates the tailoring time is actually *reduced* if CSE is particularly applicable (note that this makes sense, since the elimination of several subexpressions requires less overall flattening).

Tailoring Time: Summary

In summary, we have seen that there is no significant penalty for applying CSE during flattening. More specifically, we observe that applying CSE in vain (i.e. on problem classes without scope for enhancement) the difference in tailoring time varies within $\mp 10\%$, a figure that further *increases* with the length of tailoring time. This difference is marginal and can be expected within separate runs. Furthermore, in cases where CSE can improve an instance, the tailoring time is reduced proportionally to the number of subexpressions that are eliminated. Therefore, we conclude that the integration of CSE does not impair tailoring time, but can actually enhance it in cases where it fires.

8.2.4   Impact on Solving Performance

Finally, we study the most important feature: the impact CSE can have on solving performance. We start with the reduction in solving time, followed by the reduction in search nodes. Since only group B and C are affected by CSE, we exclude group A in this discussion.

Solving Time Speedup

Fig. 8.4 shows the solving time speedup in solver Minion for problems in group B(top) and C(top). The results are most pronounced - for group C, where we observe speedups of a factor greater than 2000.

Both graphs are setup in the same way: the x-axis denotes the solving time (in seconds) for instances tailored with CSE (we used a timeout of 20 minutes). The y-axis gives the
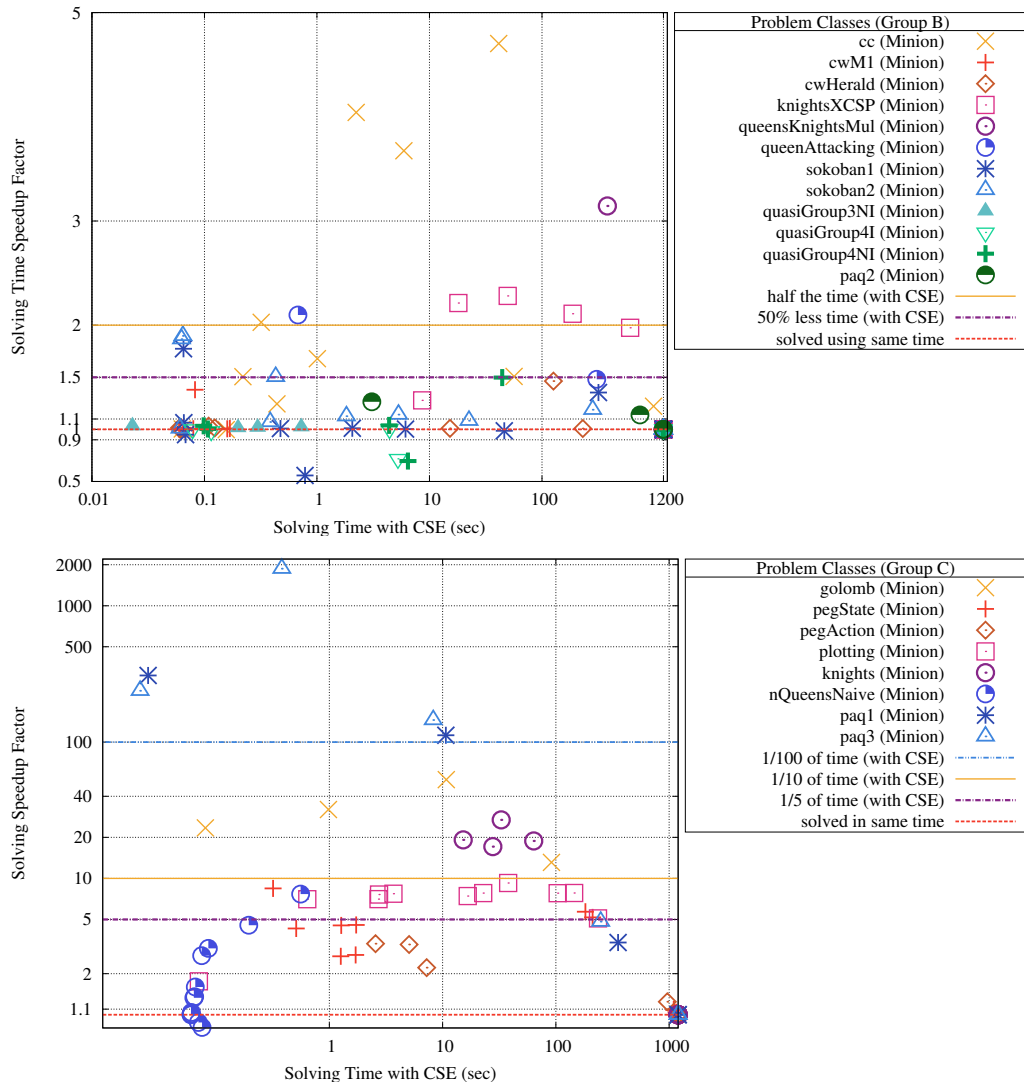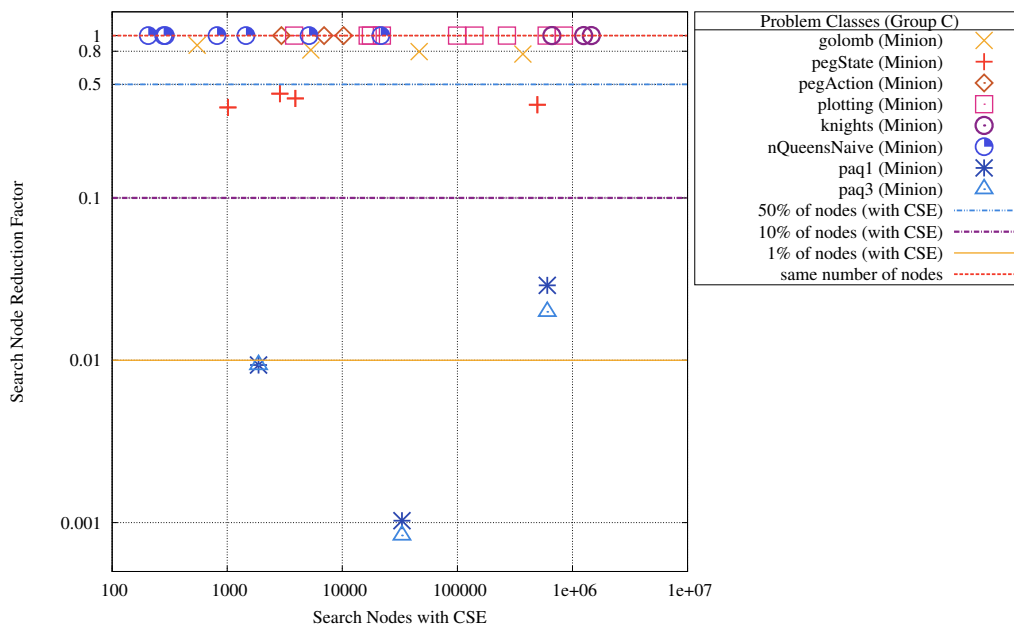
Figure 8.4: **Solving Time Speedup** in solver Minion using the **basic CSE** during tailoring, compared to no enhancement for problem Group B (top) and problem Group C (bottom). The x-axis shows the solving time (in seconds) in case instances were tailored with CSE. The y-axis represents the solving time speedup factor, for example, points at $y = 5$ have been solved 5 times *faster* with CSE then without.

speedup factor from applying CSE. As an example, instances at $y = 5$ have been solved 5 times faster with CSE than without.

Solving Time Speedup in group B   The top graph in Fig. 8.4 depicts the speedup obtained within group B, where we observe a considerable speedup of up to a factor of almost 5. Note, that some enhanced instances are solved using *more* time. Since these are single instances (of which all other instances from the same class are solved using *less* time) we do not expect that the enhancements generally impair instances.

Solving Time Speedup in group C The bottom graph of Fig. 8.4 shows the solving time speedup for group C, that are most impressive: most instances are solved far quicker with CSE than without, mainly by a factor between 2 and 100. Furthermore, in the peaceful Armies of Queens, we observe speedups up to a factor of 2000.

Reduction in Search Space

In some problem classes of group C we observe considerable search space reductions due to CSE. Fig. 8.5 illustrates the reduction of search space for Group C: instances of the Golomb Ruler, Peg Solitaire State, Peaceful Army of Queens Model1 and Model3 are solved using less search space. Most impressively, the search space for Peaceful Armies of Queens could be reduced to 1%-0.1% of the search space used without enhancement. This is probably the main reason of the speed up factor of 2000 that we gain (see Fig. 8.4).

Figure 8.5: **Reduction in Search Space** in solver Minion using **basic CSE** during tailoring, compared to no enhancement for problem Group C. The x-axis denotes the numbers of nodes searched for a solution with CSE. The y-axis represents the factor with which the number of searched nodes is reduced due to CSE. For instance, points at $y = 0.5$ represent instances that were solved using only 50% of the search space of those tailored without CSE, i.e. CSE has reduced the search space by half.

8.3 Active Reformulations to Increase the Number of CS

In this section we investigate the impact of applying a particular kind of instance optimisation: active reformulations that can *increase* the number of common subexpressions (CS) in a constraint instance, as described in Sec. 4.3. More specifically, we first consider the active negation reformulation (Sec. 4.3.3), followed by the De Morgan Reformulation (Sec. 4.3.4) and finally, the Horn Clause Reformulation (Sec. 4.3.5).

8.3.1 Overview

This study investigates two things: first, the impact of the respective reformulation on tailoring time - does the attempt of applying the optimisation slow down the tailoring process? Second, we want to investigate the benefits of the reformulation - can the instance be further reduced by additional CSE and does the reformulation enhance the solving performance? In summary, we will analyse (1) the reduction of auxiliary variables (which reflects the impact of CSE) (2) tailoring time, and (3) solving time. Note that we will not consider the respective search space, since none of the techniques has resulted in a search space reduction.

8.3.2 Active Negation Reformulation

First, we investigate active Negation Reformulation that is discussed in Sec. 4.3.3. We start by investigating the difference in instance size when applying the reformulation.

Auxiliary Variable Reduction with Active Negation Reformulation

We first consider the impact of the active negation reformulation on the instance sizes of problems where the reformulation fired. In particular, it fired on English Peg Solitaire State, Peaceful Armies of Queens model 3 and Plotting. Fig. 8.6 depicts the reduction of auxiliary variables due to the reformulation (which corresponds to the increase of identical common subexpressions). The x-axis represents the number of auxiliary variables in the respective instance *with* active negation reformulation and the y-axis the auxiliary variable reduction factor wrt the reformulation. Hence, all instances below $y = 1$ contain *fewer* auxiliary variables due to the active negation reformulation.

In Plotting, the active negation reformulation reduces the overall number of auxiliary variables by ca.8%, for instance, from 11,714 to 11,109 in a medium-sized instance. This means that about 7% of all subexpressions that required to be flattened could be reduced to an identical representation due to the active negation reformulation. In the Peaceful Armies of Queens model 3, the reduction is even greater, where the negation reformulation saves
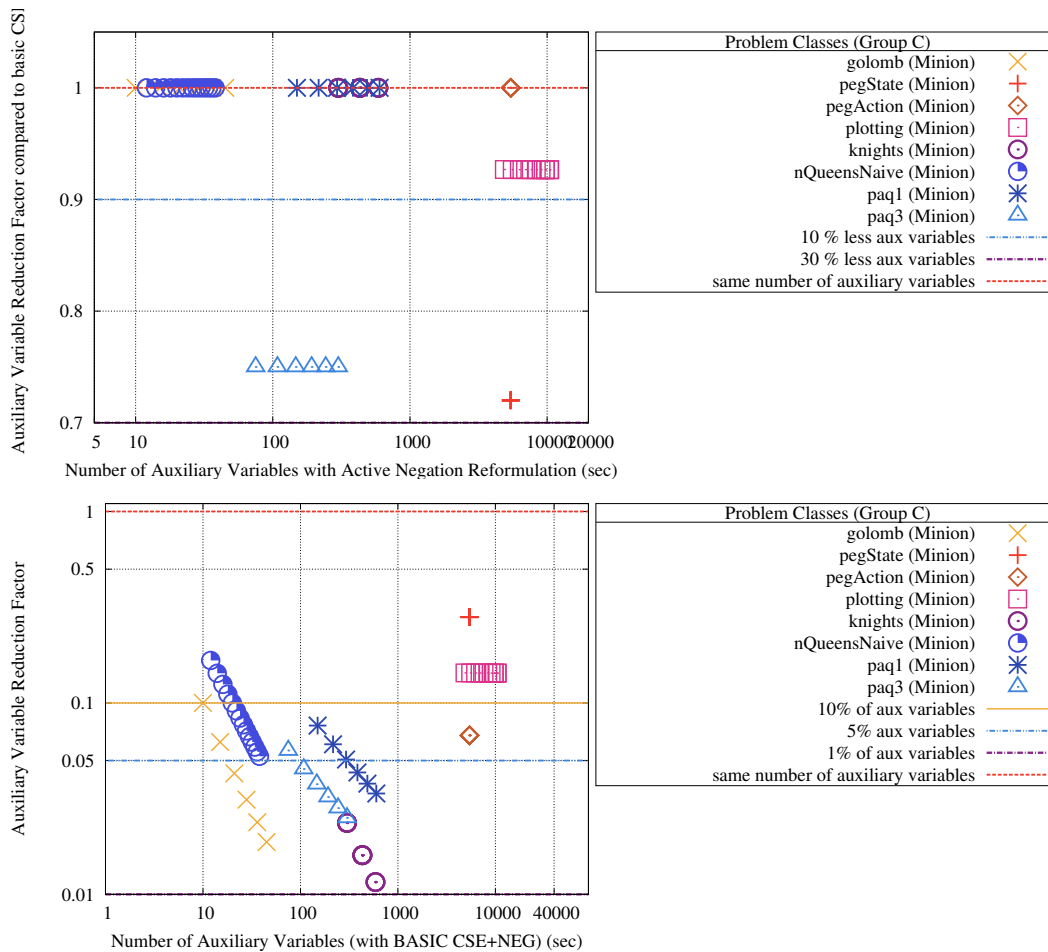
Figure 8.6: **Reduction of Auxiliary Variables** by tailoring using the **active negation reformulation** compared to the reduction achieved by basic CSE (top) and compared to no enhancement (bottom). The y-axis represents the factor with which the tailoring time differs between both tailoring options: points below $y = 1$ depict cases where the number of auxiliary variables was *reduced* when applying the active negation reformulation, the further down, the larger the reduction.

around 25% of auxiliary variables. For instance, in a medium-sized example, basic CSE reduces the number of auxiliary variables from 8610 to 324 auxiliary variables, which the active negation reformulation can further reduce to 243 auxiliary variables. In the state-centric model of English Peg Solitaire, we observe the greatest reduction of about 28%, where in a typical instance, first, basic CSE yields a reduction from 19,313 auxiliary variables to 7,533, which the active negation reformulation further reduces to 5,425 auxiliary variables.

In summary, we observe a considerable reduction of auxiliary variables in three problem classes, whose effect on solving time we investigate in the following.
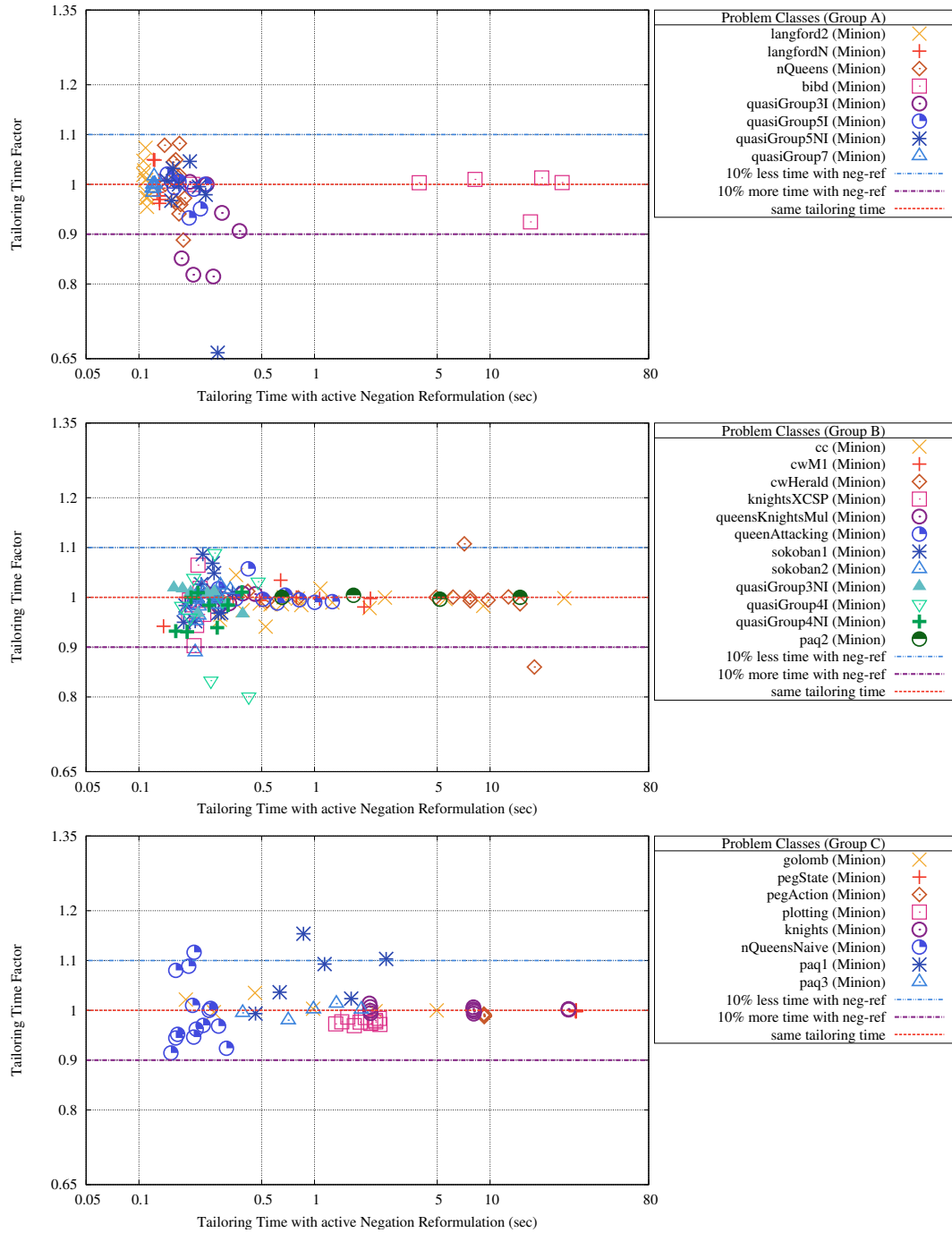
Figure 8.7: **Tailoring Time Comparison** between tailoring **with basic CSE** and tailoring with basic CSE and the **active negation reformulation** on problem groups A(top), B(middle) and C(bottom). The y-axis represents the factor with which the tailoring time differs between both tailoring options: points above $y = 1$ depict cases where tailoring time was *decreased* when applying the active negation reformulation, points below depict cases where tailoring time was *increased*.

Tailoring Time with Active Negation Reformulation

Fig. 8.7 depicts the increase and decrease in tailoring time when applying the negation reformulation *in addition* to the basic CSE procedure. It shows three graphs, one for each problem group. In each graph, the x-axis represents the tailoring time *with* the active negation reformulation and the y-axis depicts the time factor with which tailoring time has been reduced due to the reformulation: all points *above* $y = 1$ represent instances that have been tailored in *less time* with the active negation reformulation and all points *below* $y = 1$ represent instances that took longer to tailor with the negation reformulation. The factor illustrates the percentage of the increase or reduction, respectively. For instance, instances that are positioned at 1.2 have been tailored using 20% less time with the negation reformulation than those tailored only with basic CSE.

Problem Group A The top graph shows tailoring time for Problem Group A, that contains problems that can not be further enhanced. The instances are spread equally below and above $y = 1$ and most are positioned between 1.1 and 0.9, i.e. the tailoring time differs mostly around $\mp 10\%$ which is a small difference (in particular since the main overall tailoring time lies in between 0.1-0.5 seconds) that is to be expected between separate runs on the Java virtual machine. Therefore, we conclude that adding the negation reformulation to the tailoring process has no significant impact on the overall tailoring time of problems without scope for enhancement.

Problem Group B The middle graph in Fig. 8.7 shows the tailoring time difference for problems that have a fair to medium scope for enhancement. Note, that none of these problems actually profit from the negation reformulation. Similar to Problem Group B, there is no significant overhead from adding the negation reformulation: instances are positioned in a balanced way below and above $y = 1$ with the majority within the $\mp 10\%$ range of 1.1 to 0.9. Hence, we conclude that adding the negation reformulation to tailoring does not affect the overall tailoring time of problems in class B.

Problem Group C Problem group C consists of those problems that have the largest scope for enhancement and the respective tailoring time is given in the bottom graph of Fig. 8.7. Most interestingly, the tailoring time does not change notably in those cases where the active negation reformulation applies (Peaceful Army of Queens 3, Peg Solitaire State and Plotting) and, similar to groups A and B, the overall tailoring time is quite the same with or without the negation reformulation.
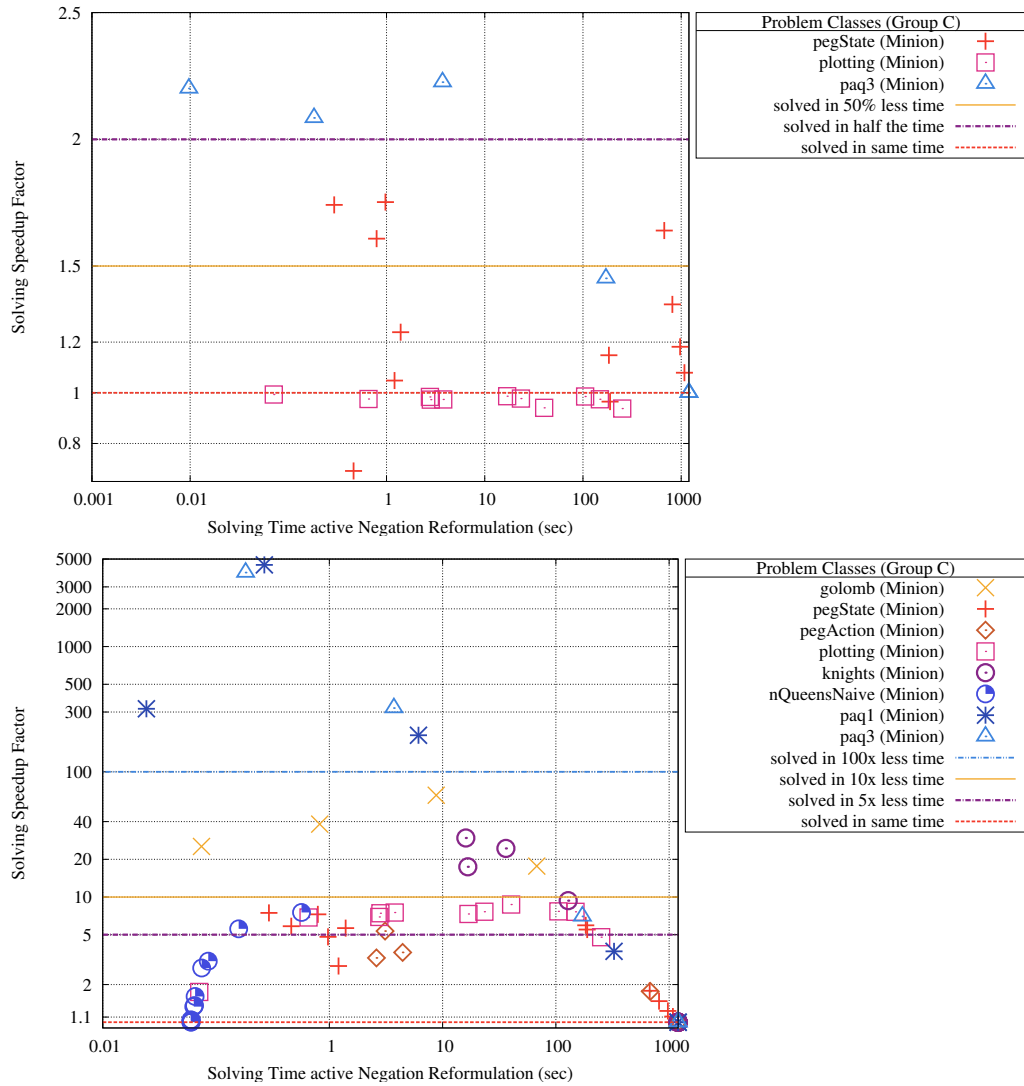
Figure 8.8: **Solving Time Speedup** in solver Minion using the **active negation reformulation** during tailoring, compared to the reduction achieved by basic CSE (top) and compared to no enhancement (bottom). The y-axis represents the solving time speedup factor, for example, points at $y = 3$ have been solved 3 times *faster* with CSE then without.

Solving Time with Active Negation Reformulation

Finally, we consider the impact on solving time in the three problem classes where the active negation reformulation managed to achieve an enhancement. Fig. 8.8 shows the impact of the active negation reformulation in solver Minion. The top graph depicts the speedup obtained in comparison to basic CSE (i.e. how much the active negation reformulation can improve the speedup *on top* of basic CSE). Here we observe that the active negation reformulation can achieve a speedup of 2.2, i.e. instances can be solved in less than half the time than those where only basic CSE is performed. The speedup is also proportional to the number of reduced auxiliary variables - in Plotting, where we have only a reduction

of 7%, the speedup is marginal, while in the peaceful armies of queens, where the active negation reformulation gains a reduction of almost 30%, we observe the largest speedup.

The bottom graph shows the overall enhancement, comparing basic CSE combined with the active negation reformulation to tailoring without any optimisations. The benefits we observe are very significant, reaching speedup factors of almost 3,500. These figures demonstrate that the combination of basic CSE and the negation reformulation can be extremely beneficial.

Summary: Active Negation Reformulation

In summary, our experiments have given evidence that the active negation reformulation does not add any notable overhead to the tailoring process (independent of whether if the respective reformulation is applicable to the instance) and can achieve impressive speedups in problem classes where the reformulation can reduce the number of auxiliary variables. Therefore, we conclude that the active negation reformulation is an effective and important instance optimisation technique that should be integrated into every tailoring tool.

8.3.3 Active Horn Clause Reformulation

The second active reformulation we consider is the active Horn Clause Reformulation (Sec. 4.3.5). Similarly as with the active negation reformulation, we start with investigating the difference in tailoring time when applying the reformulation. Note, that we do not consider the reduction of auxiliary variables since the Horn Clause reformulation does not reduce the number of auxiliary variables.

Tailoring Time with Active Horn Clause Reformulation

Fig. 8.9 depicts the increase and decrease in tailoring time when applying the active Horn Clause reformulation *in addition* to the basic CSE procedure. It shows three graphs, one for each problem group. In each graph, the x-axis represents the tailoring time *with* the active Horn Clause reformulation and the y-axis depicts the factor with which tailoring time has been reduced due to the reformulation: all points *above* $y = 1$ represent instances that have been tailored in *less time* with the active Horn Clause reformulation and all points *below* $y = 1$ represent instances that took longer to tailor with the Horn Clause reformulation. The factor illustrates the percentage of the increase or reduction, respectively. For instance, instances that are positioned at 1.1 have been tailored using 10% less time with the Horn Clause reformulation than those tailored only with basic CSE.

In summary, the results are similar to that of the active negation reformulation: the tailoring times differ mainly within the $\mp 10\%$ margin, a difference that can be expected between
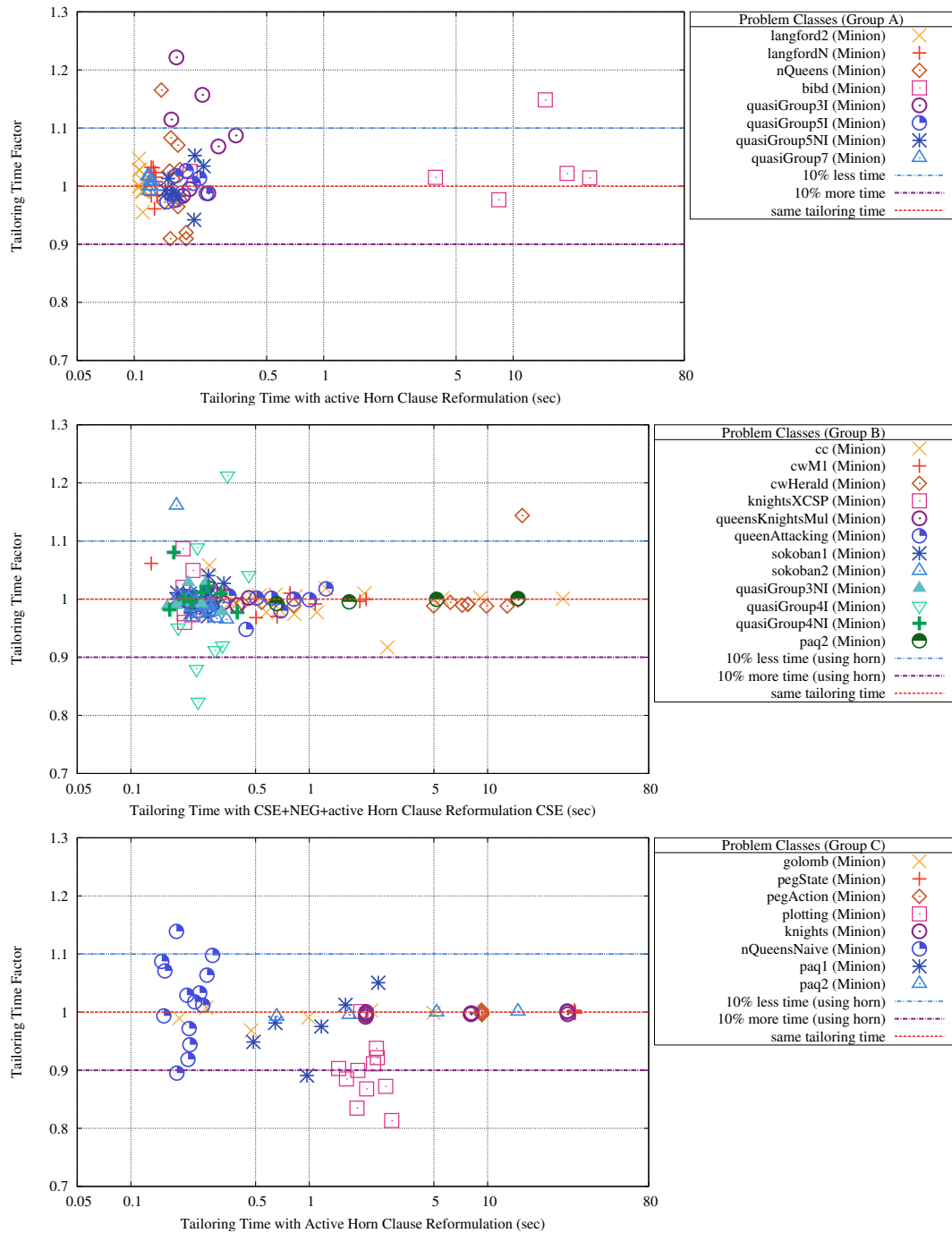
Figure 8.9: **Tailoring Time Comparison** between tailoring **with basic CSE** and tailoring with basic CSE and the **active Horn Clause reformulation** on problem groups A(top), B(middle) and C(bottom). The y-axis represents the factor with which the tailoring time differs between both tailoring options: points above $y = 1$ depict cases where tailoring time was *decreased* when applying the active Horn Clause reformulation, points below depict cases where tailoring time was *increased*.

different runs on the Java virtual machine.
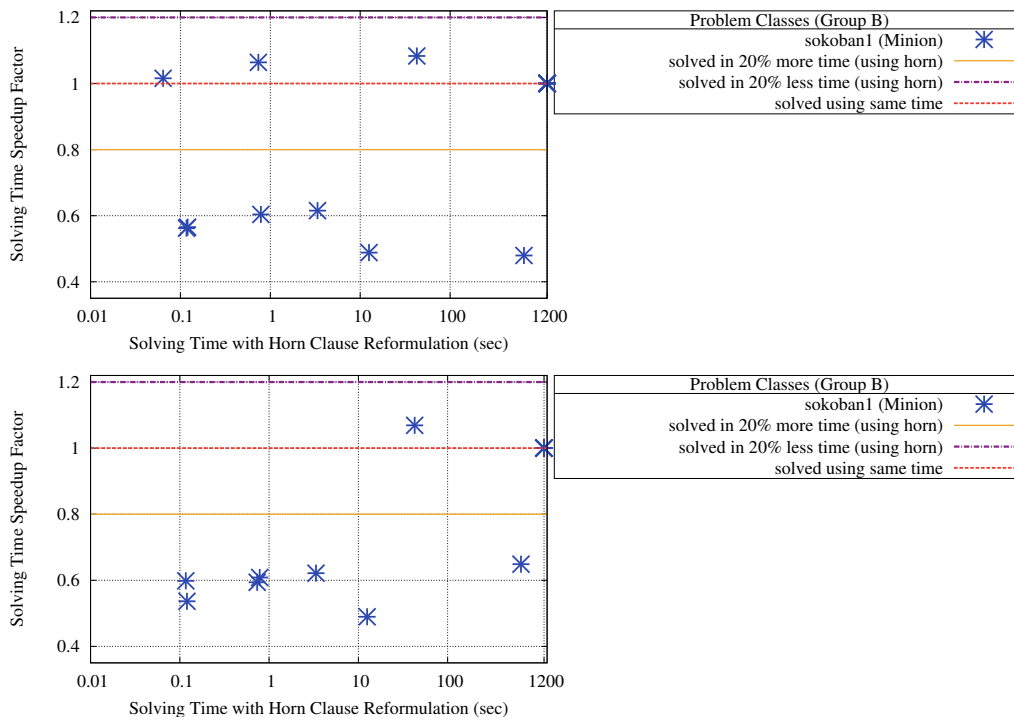
Figure 8.10: **Solving Time Speedup** in solver Minion using the **active Horn Clause reformulation** during tailoring, compared to the reduction achieved by basic CSE (top) and compared to no enhancement (bottom). The y-axis represents the solving time speedup factor: points at $y = 1$ depict cases where the number of auxiliary variables is the same, points below depict cases where the number was reduced. Note, that only Sokoban1 was affected by the reformulation.

Solving Time Speedup with the Horn Clause Reformulation

The Horn Clause reformulation fired in only *one* of all our problem classes: the first model of the Sokoban problem. The results are very unexpected: the solving performance suffers (around 40%) from both basic CSE and the active Horn Clause reformulation. So far, we have no obvious explanation for this. One reason could be that implications are a preferable representation in solver Minion compared to the disjunctive representation. Another reason be that our algorithm still requires enhancements, in particular on how to generate the most effective Horn Clauses from a given disjunction. Therefore, in our implementation, the active Horn Clause reformulation is not applied as default optimisation technique during tailoring at present.

Summary: Horn Clause Reformulation

In summary, we have seen that performing the Horn Clause reformulation adds no significant overhead to the tailoring process. However, the reformulation fired only in one problem class where it resulted in an *increase* of solving time, which was quite unexpected. The observations are inconclusive and require a further investigation into the matter of representing expressions in form of disjunctions and implications, which is an item of future work.

8.3.4 Active De Morgan Reformulation

The active De Morgan Reformulation (Sec. 4.3.4) is another active reformulation that we propose in order to increase the number of identical common subexpressions and hence reduce the number of auxiliary variables in an instance. Note, that the De Morgan reformulation fired in none of our problem classes and hence we limit our investigation on the impact on tailoring time when applying the reformulation.

Tailoring Time with Active De Morgan Reformulation

Fig. 8.7 depicts the increase and decrease in tailoring time when applying the negation reformulation *in addition* to the basic CSE procedure. It shows three graphs, one for each problem group. In each graph, the x-axis represents the tailoring time *with* the active negation reformulation and the y-axis depicts the time factor with which tailoring time has been reduced due to the reformulation: all points *above* $y = 1$ represent instances that have been tailored in *less time* with the active negation reformulation and all points *below* $y = 1$ represent instances that took longer to tailor with the negation reformulation. The factor illustrates the percentage of the increase or reduction, respectively.

The results are very similar to that of the other active reformulations: the tailoring process does not suffer from adding the reformulation, independent on the kind of problem.

8.4 Eliminating Argument Common Subexpressions

Argument CS are subexpressions that are shared among the arguments of n-ary commutative and associative expressions. For instance, the following two sums share the the argument-CS 'y+z'.

```
x + y + z = 10,
y + z + t > 5
```
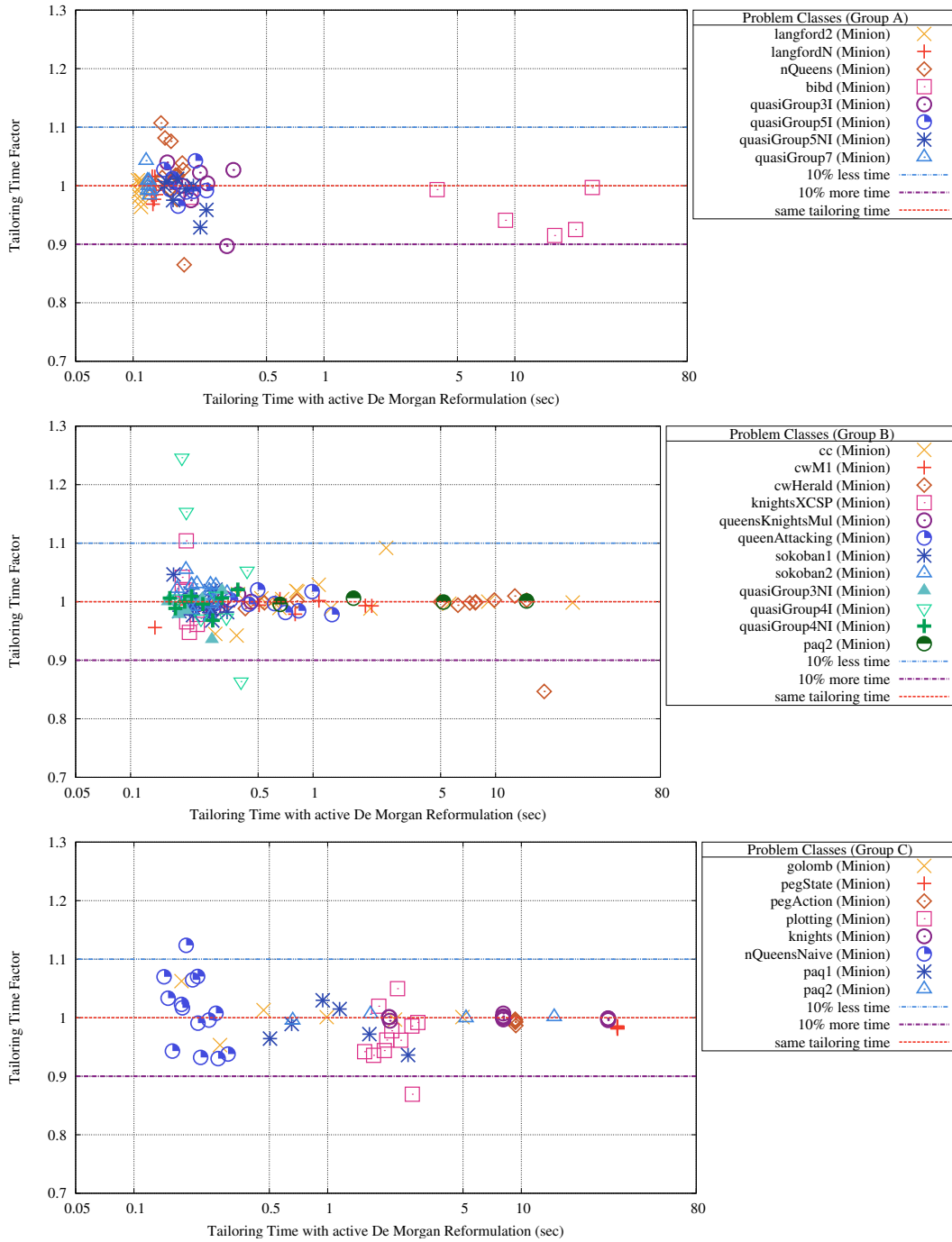
Figure 8.11: **Tailoring Time Comparison** between tailoring **with basic CSE** and tailoring with basic CSE and the **active De Morgan reformulation** on problem groups A(top), B(middle) and C(bottom). The y-axis represents the factor with which the tailoring time differs between both tailoring options: points above $y = 1$ depict cases where tailoring time was *decreased* when applying the active De Morgan reformulation, points below depict cases where tailoring time was *increased*.

Plotting Instance	Eliminated Common Subexpressions (technique)		
	(basic CSE)	(negation CSE)	(**argument CSE**)
(1,14)	58,662	10,220	420
(1,15)	62,854	10,950	450
(2,10)	41,894	7,300	300
(2,11)	46,086	8,030	330
(2,12)	50,278	8,760	360
(3,7)	29,318	5,110	210
(3,8)	33,510	5,840	240
(3,9)	37,702	6,570	270
(4,13)	54,470	9,490	390
(4,14)	58,662	10,220	420
(4,15)	62,854	10,950	450

Table 8.1: The **Number of Eliminated Common Subexpressions** in **Plotting**, tailored to solver Minion. Note, that the number of eliminated argument-CS makes up less than 1% of the number of eliminated basic common subexpressions.

We proposed an algorithm that can eliminate a particular kind of argument-CS that also occur in practical examples (see Sec. 4.4.2 for more details), whose benefits we want to investigate in this section.

Argument-CS occur in only one of the problem classes we consider: in Plotting (Sec. 7.3.4). Hence, we investigate their elimination in the following setup: we compare instance size, solving performance and tailoring time of two different setups: applying all optimisation techniques versus applying all optimisation techniques excluding argument-CS elimination.

Eliminated argument-CS

First, we investigate number of eliminated argument-CS. Recall, that eliminating argument-CS does not reduce the number of auxiliary variables, but reduces the arity of n-ary constraints to binary constraints. Hence, we summarise the number of eliminated common subexpressions in 11 instances of Plotting in Table 8.1. Common subexpressions in Plotting are eliminated by three different techniques: (1) basic CSE (Sec. 4.2), (2) the active Negation Reformulation (Sec. 4.3.3) and (3) argument CSE. Tab. 8.1 lists the number of common subexpressions that can be eliminated using each technique. Evidently, basic CSE is the most powerful technique, followed by the active negation reformulation and finally, argument CSE. Since the common subexpressions eliminated by argument-CSE only make out less than 1% of those eliminated by basic CSE and 10% of those eliminated by the active negation reformulation, we do not expect outstanding solving performance improvements.
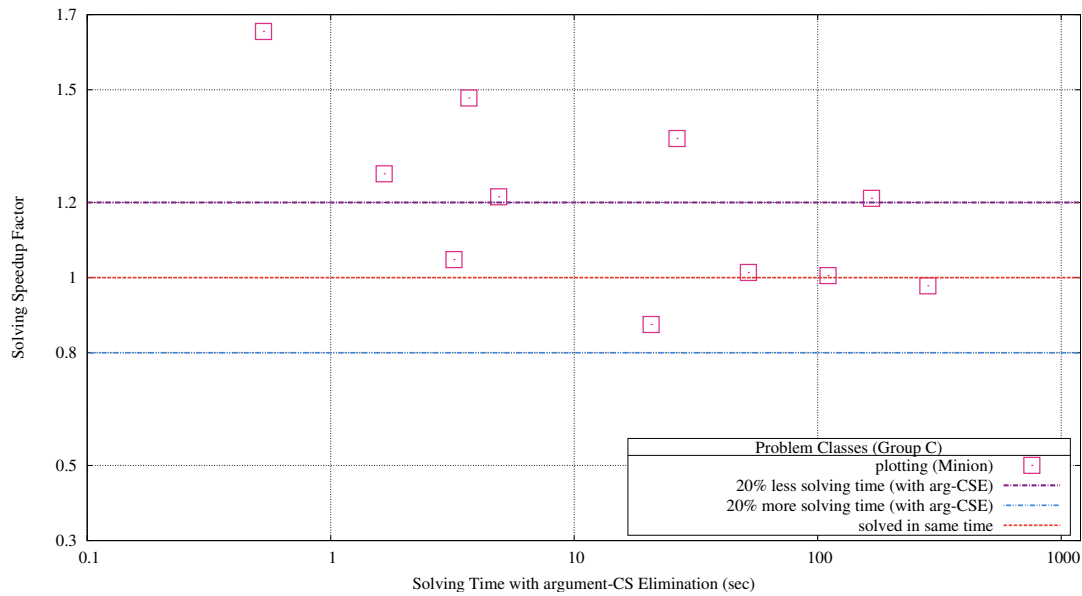
Figure 8.12: **Solving Time Speedup** in Plotting due to **Argument-CS Elimination** in solver Minion.

Impact of Argument-CSE on Solving Performance

Now we consider the impact of argument CSE on the overall solving performance of the Plotting instances. We only consider solving time, since the search space stays the same in all considered Plotting instances.

Fig. 8.12 illustrates the difference in solving time with and without argument-CSE. The x-axis shows the solving time (in seconds) *with* argument CSE, and the y-axis depicts the factor with which solving time differs between instances tailored with and without argument CSE. More specifically, all points above $y = 1$ depict instances that are solved in *less* time using argument CSE, points below denote the opposite case.

First, we observe that most instances could be solved quicker with argument CSE, with the exception of two instances. However, the improvement is moderate, ranging from a speedup of about 50% for smaller instances and about 25% for larger ones. However, since the number of argument CS is not particularly high, this is a considerable positive result and shows the potential of improvement through argument CSE.

Impact of Argument-CS on Tailoring Time

Finally, we investigate the impact of argument CSE on tailoring time. Fig. 8.13 depicts the differences between tailoring with and without argument CSE: as illustrated, the differences lie within a $\mp 5\%$ margin, and are evenly distributed, which leads to the conclusion that performing argument CSE has a negligible impact on the tailoring performance and does
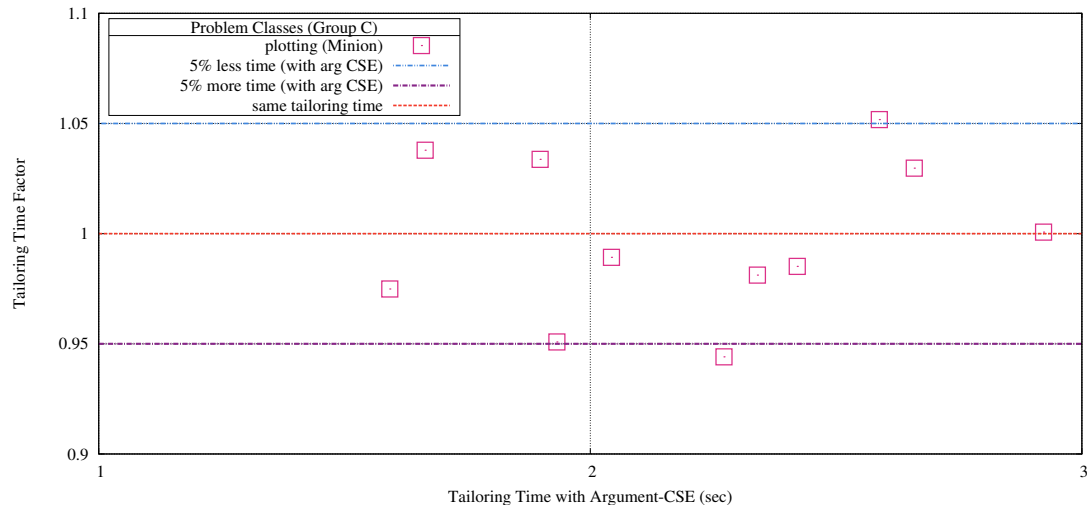
Figure 8.13: **Tailoring Time** differences between tailoring with and without **Argument-CS Elimination**, considering the Plotting problem class targeting solver Minion.

not generally increase tailoring time.

Argument CSE: Summary

We have investigated the effects of argument CSE on the Plotting problem class, the only class that contained argument CS in the set of examples we consider in our empirical analysis. In this analysis we have observed a fair number of eliminated common subexpressions from applying argument CSE (corresponding to about 1% of the number of common subexpressions that basic CSE detects and eliminates). However, the elimination of this small number of argument-CS has resulted in a considerable speedup: we observe solving time reductions up to 70% from argument CSE. Furthermore, we observe that the tailoring time does not suffer from the addition of argument CSE to tailoring which leads to the conclusion that the integration of argument CSE does not impair the tailoring performance and can be applied as general optimisation technique for every tailoring step.

8.5 Loop Optimisations: Inside vs. Outside Representation

In this section we assess the differences between quantifications where loop-invariant expressions are *inside* and those where loop-invariant expressions are *outside* the quantification (see Sec. 4.6.3 for more details). Note, that only one loop optimisation ($\forall\text{-}\Rightarrow$) was applicable to the vast set of problem classes that we consider in our empirical study. Therefore, we limit this analysis to this case, since the remaining loop optimisations could only
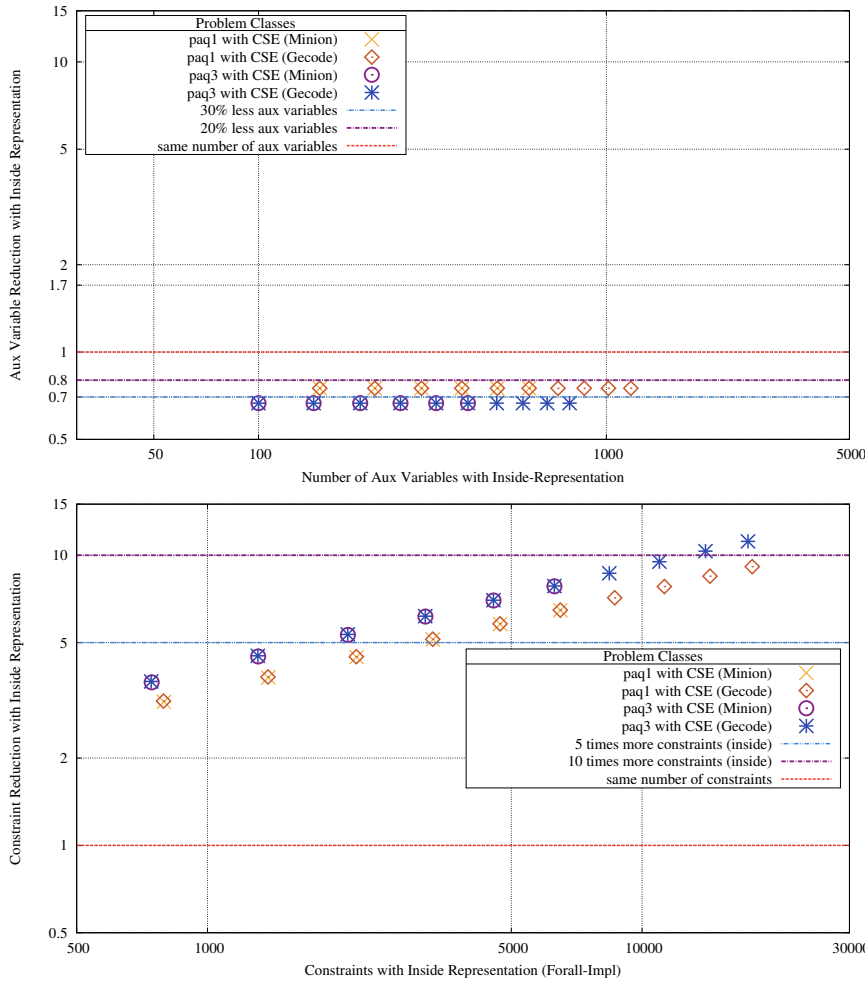
Figure 8.14: **Instance Comparison** between the **inside and outside representation** of the ($\forall$-$\Rightarrow$) case. The top graph gives a comparison of auxiliary variables, the bottom graph shows the difference in constraints, for both solvers MINION and Gecode. The x-axis denotes the number of auxiliary variables/constraints in the *inside*-representation and the y-axis represents the factor of reduction/increase compared to the *outside*-representation. The graphs illustrate how the number of auxiliary variables is *reduced* while the number of constraints *increases* in the *inside*-representation.

be tested on *constructed* examples, which would not provide any meaningful insights for practical examples. We analyse Case 3 from Sec. 4.6.3, where we consider a quantification of the form

$$\forall_I A \Rightarrow E_I$$

where A is loop-invariant and can hence be moved *outside* the quantification, yielding the equivalent representation

$$A \Rightarrow \forall_I E_I$$

In this study, we compare both representations in a practical example: in two different models of the Armies of Queens Problem [76]. We assess the reformulation in two different solvers: Gecode and Minion, for which we get very similar results. We start with considering the difference in instance size between the *inside-* and *outside-*representation.

Difference in Instance Size Fig. 8.14 depicts the differences in instance size when applying the instance and outside representation. First, note, that using the inside-representation, we get a *reduction* in auxiliary variables, but an *increase* of constraints, compared to the outside representation. Furthermore, the reduction/increase is exactly the same for both constraint solvers.
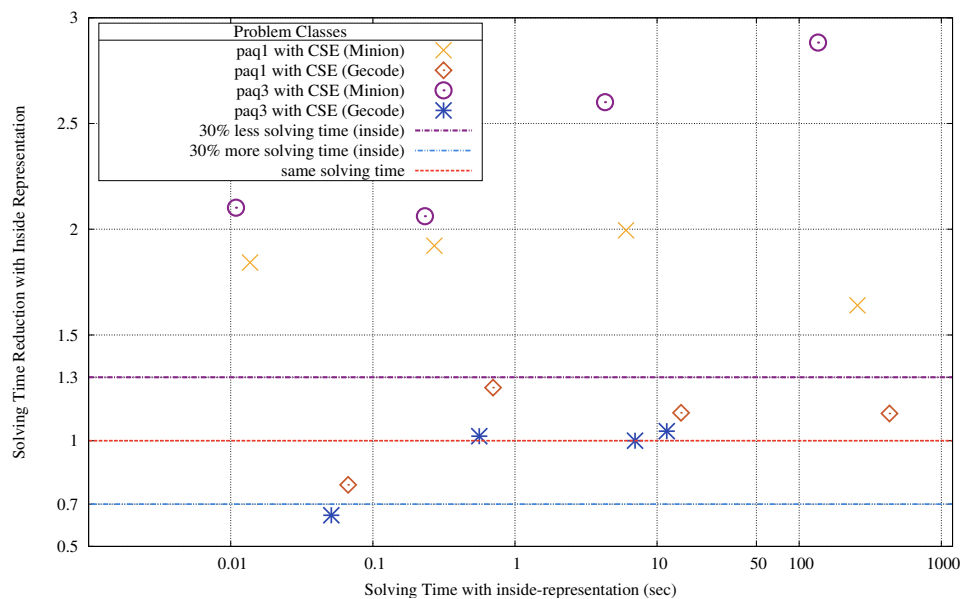

Figure 8.15: **Solving Time Comparison** between **inside and outside-representation** in for solvers MINION and Gecode on the Peaceful Armies of Queens Models (1+3). The y-axis represents the solving time speedup factor: points at $y = 1$ depict cases where the number of auxiliary variables is the same, points below depict cases where the number was reduced. Note, that only Sokoban1 was affected by the reformulation.

Solving Performance We illustrate the solving performance in Fig. 8.15, summarising results for both solver Gecode and MINION, which are very similar: The x-axis shows the solving time for the inside-representation (in seconds) and the y-axis gives the factor over which the solving time for inside-representation differs from the solving time for the outside-representation. As an example, points at $y = 2$ denote those cases where the inside-representation instance was solved in half the time of the outside-representation variant. Therefore, all points above $y = 1$ denote cases, where the inside-representation has been more successful than the outside representation, by means of solving time.

First, we note that most instances, with the exception of two small instances in Gecode,

have been solved quicker using the inside representation. Second, we observe that the inside-representation provides a stronger benefit in solver MINION than in Gecode: for MINION we observe speedup factors up to almost 3, while in Gecode we only observe a moderate improvement, of about 30%. This difference might stem from the vast amount of constraints that the inside representation contains, compared to the outside representation, since Gecode and MINION post propagators (i.e. constraints) in a different fashion.

However, in summary, we can see that, unlike our expectations, keeping the loop-invariant expression *inside* the universal quantification is actually beneficial.

8.6 The Power of Instance Optimisations

In this section we present the power of the combination of all optimisation techniques that we have seen in this chapter (excluding the active Horn Clause reformulation), as they are implemented in TAILOR. This means we will tailor instances *with and without* optimisations, and investigate the differences in

1. Instance Size

2. Solving Performance (solving time and search space)

3. Tailoring Time

in *two different* constraint solvers: Minion and Gecode. Note, that we only consider a *subset* of each problem group for solver Gecode, since the translation is still restricted on one hand, and due to some current restrictions in Gecode on the other.

All our problem models come either from the literature or the XCSP solver competition benchmark or follow a standard model from CSPlib, hence this study is based on authentic problem models. We placed our problems into three problem groups: group A, B and C, where group A contains problems with *no* scope for enhancement, group B a small to medium scope of enhancement, while the problems in group C have a large scope of enhancement. Note, that the enhancements only fire in problems in group B and C, hence we will consider group A only in the analysis of tailoring time (since the generated instances in group A are identical in both tailoring approaches). For more details on the general experimental setup, see Sec. 8.1.

The results in this section highlight the practical aspects of the contributions of this thesis.

8.6.1 Instance Reductions

The instance reductions reflect how much a problem class could be enhanced. First, we investigate the reduction in auxiliary variables (that also reflects the number of eliminated
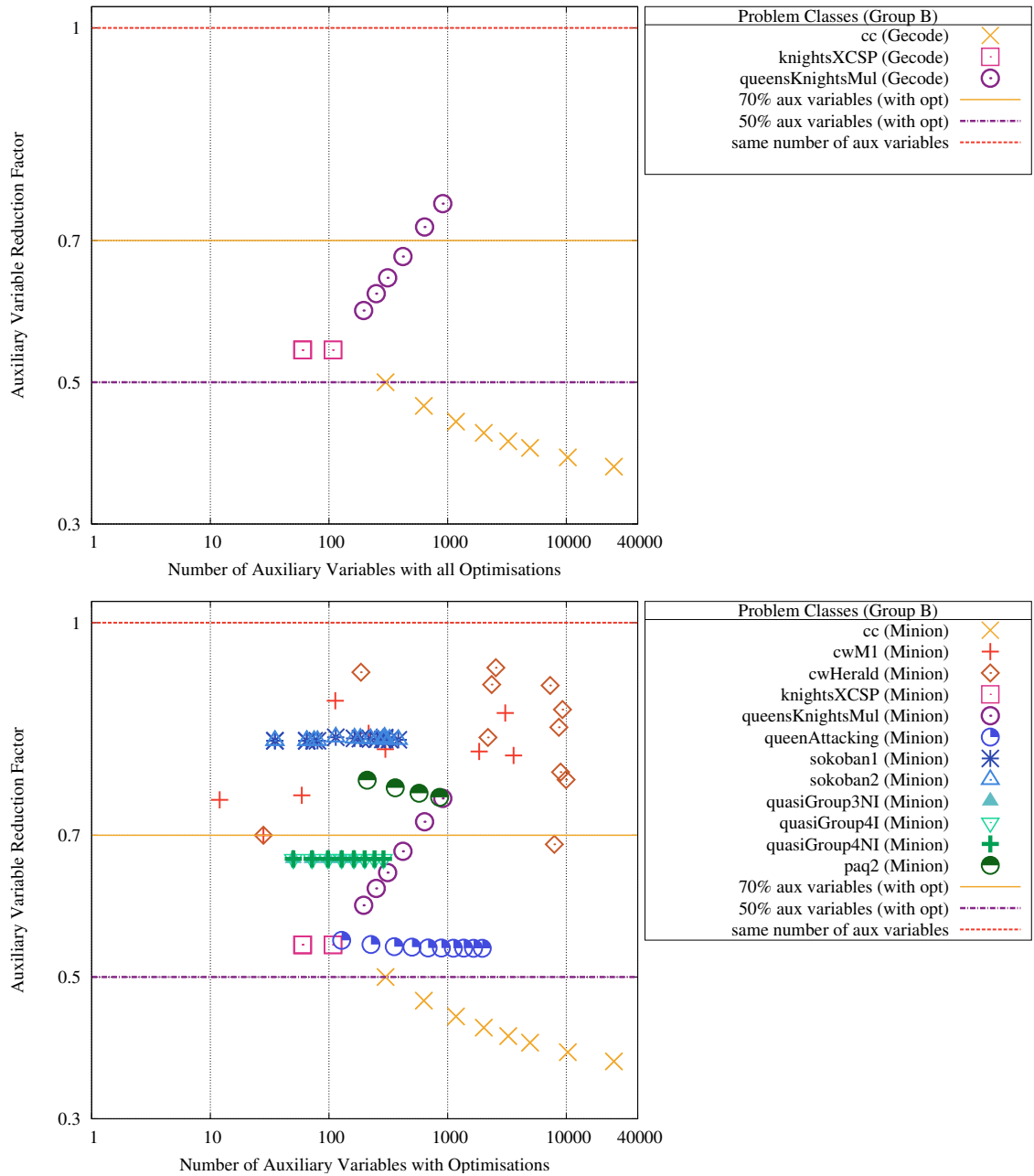
Figure 8.16: **Reduction of Auxiliary Variables in group B** by tailoring using **all optimisation techniques** for solver **Gecode** (top) and solver MINION (bottom). Note that the reduction of auxiliary variables reflects the **number of eliminated common subexpressions** (every eliminated CS saves one variable). The x-axis depicts the number of auxiliary variables with optimisations; the y-axis represents the factor with which the number of auxiliary variables is reduced due to the optimisations. For instance, points at $y = 0.5$ represent instances where the number of auxiliary variables was reduced to 50% compared to not applying optimisations.

common subexpressions). Second, we consider the reduction in constraints.

Reduction in Auxiliary Variables

A reduction in auxiliary variables is typically desirable and reflects the number of eliminated common subexpressions, since for every eliminated common subexpression, we save one auxiliary variable. Fig. 8.16 shows the auxiliary reduction for group B, while Fig. 8.17 shows the reduction for group C. In both figures, the top graph shows results for solver Gecode, while the bottom graph shows results for solver MINION.

First, we notice that the reduction is very similar in both solvers, which demonstrates that our optimisation techniques are generally beneficial and do not compensate for a specific feature in just one solver.

Second, we notice the magnitude of the reduction: for problems in group B, we see reductions down to less than half the number of auxiliary variables. In group C, enhanced instances of some problem classes only contain about 2% of the equivalent unenhanced instance.

In summary, we observe a vast reduction in auxiliary variables due to our optimisation techniques, that are similar in solver MINION and Gecode.

Reduction of Constraints

Second, we consider the reduction in constraints: results for group B are illustrated in Fig. 8.18, those for group C are illustrated in Fig. 8.19.

Similar to the reduction in auxiliary variables, we observe a similarities in the results of solver Gecode and MINION.

Both figures illustrate, how the constraint reduction is related to the parameters that scale the problem: in many classes, the reduction increases with the parameters (e.g. Chessboard Colouring); in other classes, the constraint reduction is independent of the parameters, like in the Peg Solitaire models. Note, that in some problem classes, like Queen-Knights, the constraint reduction linearly *decreases* with the instance size. This is easily explained: this kind of problem classes contain parameters that scale constraints that are not affected by CSE. Hence the larger those parameters get, the smaller is the number of constraints saved by CSE in proportion to the number of constraints *added* by increasing the parameter.

In summary, we see impressive constraint reductions that are similar in both constraint solvers, where the enhanced instances contain about 95%-45% of constraints in group B and 50-1% of constraints in group C, when compared to the respective unenhanced instance.
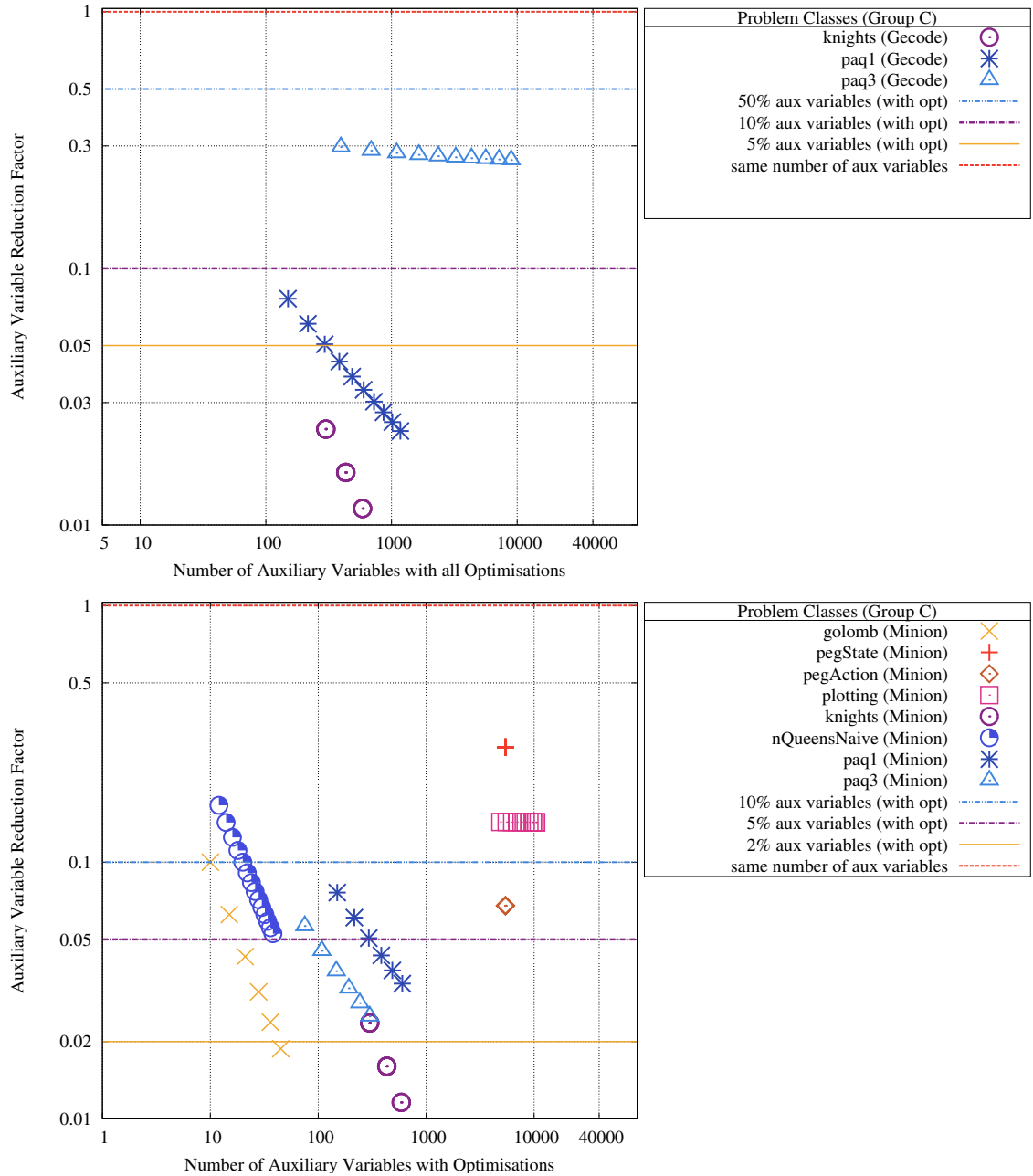
Figure 8.17: **Reduction of Auxiliary Variables in group C** by tailoring using **all optimisation techniques** for solver **Gecode** (top) and solver MINION (bottom). Note that the reduction of auxiliary variables reflects the **number of eliminated common subexpressions** (every eliminated CS saves one variable). The x-axis depicts the number of auxiliary variables with optimisations; the y-axis represents the factor with which the number of auxiliary variables is reduced due to the optimisations. For instance, points at $y = 0.5$ represent instances where the number of auxiliary variables was reduced to 50% compared to not applying optimisations.
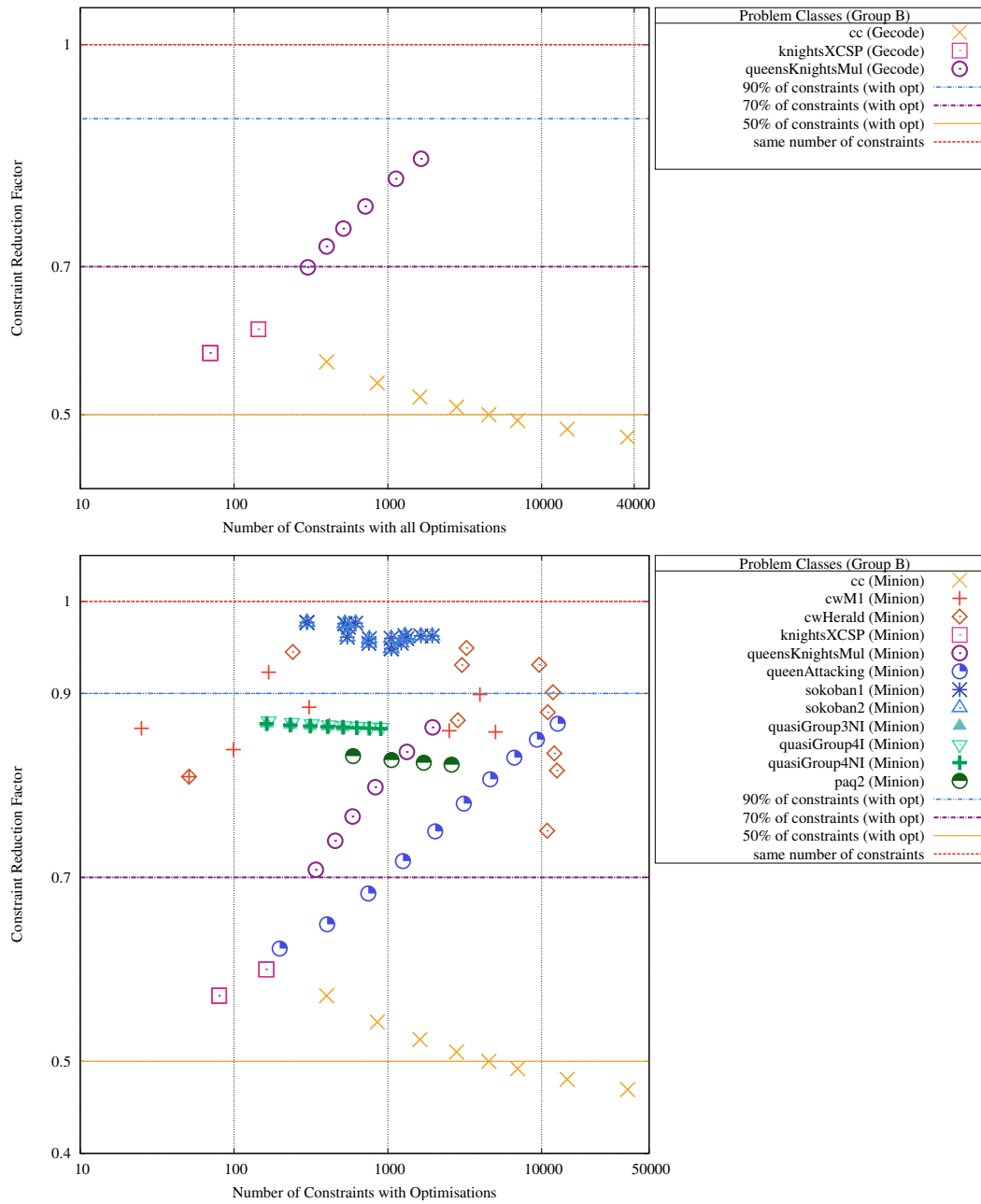
Figure 8.18: **Reduction of Constraints in group B** by tailoring using **all optimisation techniques** for solver **Gecode** (top) and solver MINION (bottom). The x-axis depicts the number of constraints with optimisations; the y-axis represents the factor with which the number of constraints is reduced due to the optimisations. For instance, points at $y = 0.5$ represent instances where the number of constraints was reduced to 50% (i.e. by half) compared to not applying optimisations.

8.6.2 Tailoring Time

Tailoring time is an important feature that assesses the efficiency of applying our techniques. We compare the tailoring times on all three problem groups.
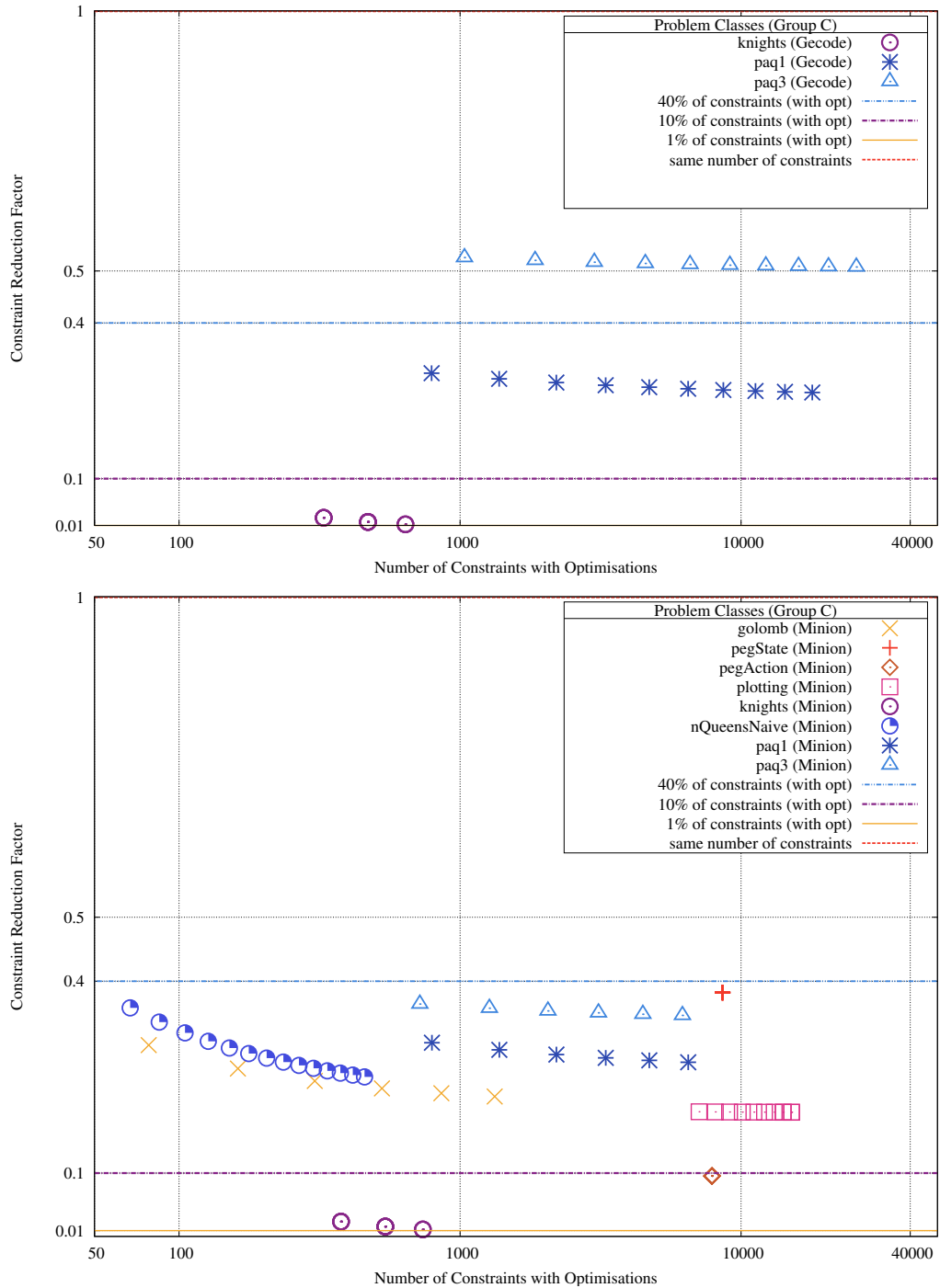
Figure 8.19: **Reduction of Constraints in group C** by tailoring using **all optimisation techniques** for solver **Gecode** (top) and solver MINION (bottom). The x-axis depicts the number of constraints with optimisations; the y-axis represents the factor with which the number of constraints is reduced due to the optimisations. For instance, points at $y = 0.5$ represent instances where the number of constraints was reduced to 50% (i.e. by half) compared to not applying optimisations.
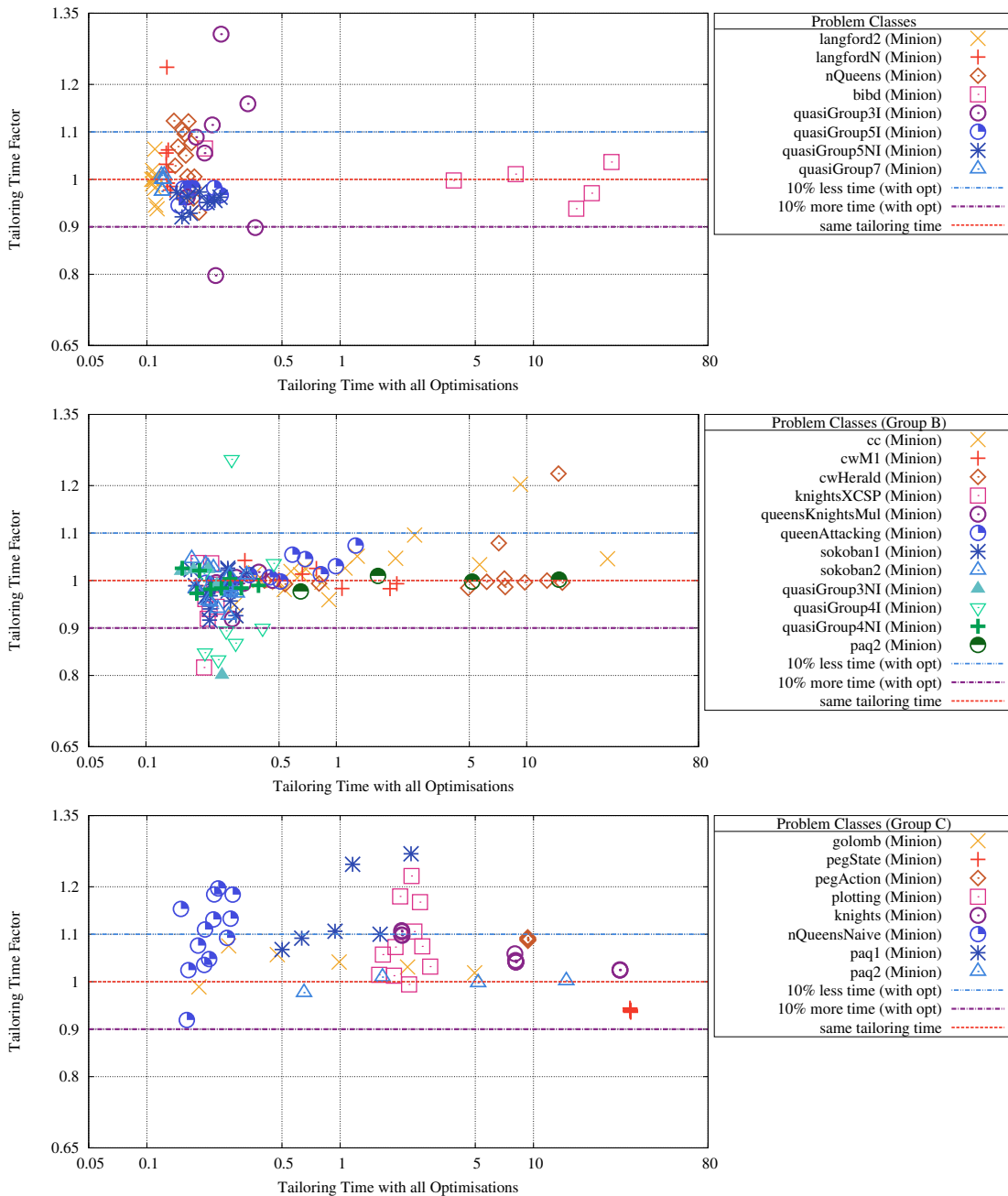
Figure 8.20: **Tailoring Time Comparison** between tailoring to **solver** MINION with and without **using all optimisation techniques** on problem groups A(top), B(middle) and C(bottom). The x-axis represents the tailoring time used for tailoring *with* optimisations. The y-axis represents the factor with which the tailoring time differs between both tailoring options: points above $y = 1$ depict cases where tailoring time was *reduced* when tailoring with optimisations and points below depict the opposite case.

Fig. 8.20 illustrates the tailoring times for solver MINION, Fig. 8.21 illustrates the tailoring times for solver Gecode. Both figures show that tailoring time is not strongly affected
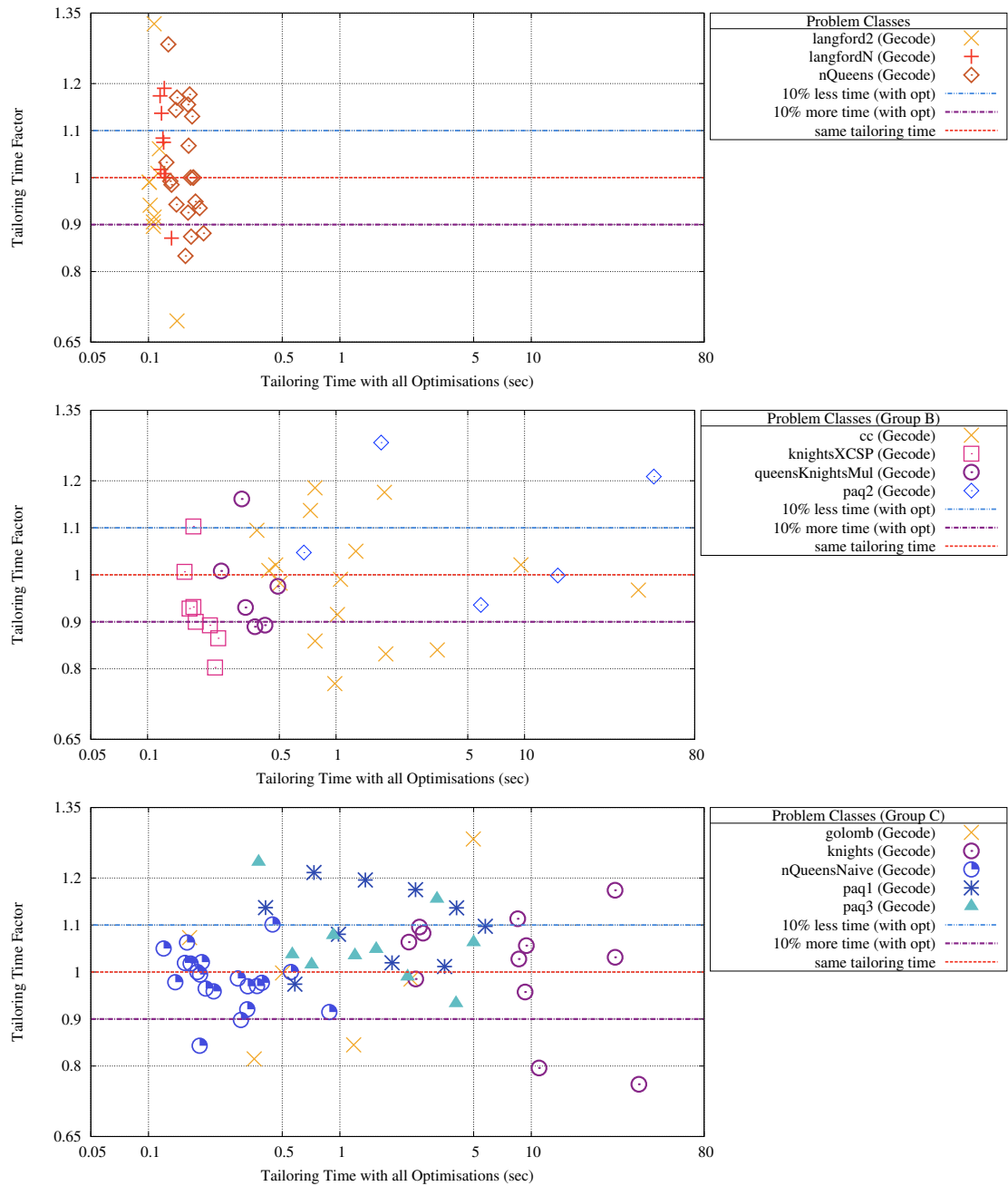
Figure 8.21: **Tailoring Time Comparison** between tailoring to **solver Gecode** with and without **using all optimisation techniques** on problem groups A(top), B(middle) and C(bottom). The x-axis represents the tailoring time used for tailoring *with* optimisations. The y-axis represents the factor with which the tailoring time differs between both tailoring options: points above $y = 1$ depict cases where tailoring time was *reduced* when tailoring with optimisations and points below depict the opposite case.

by integrating all the optimisation techniques: the instances are distributed evenly below and above $y = 1$, most of them in a margin of $\mp 10\%$ for MINION and 20% for Gecode.

Furthermore, for group C, to which most enhancements apply, the tailoring time is actually *reduced* for some problem classes.

In summary, the tailoring time analysis shows that the difference in tailoring with or without optimisations is marginal, and mainly lies within a margin of around $\mp 10\%$, which is fairly small, in particular considering the brevity of the tailoring times. Therefore, we conclude that there is no general penalty for integrating all instance optimisations into tailoring.

8.6.3 Impact on Solving Performance

Finally, we investigate the most important feature: the impact of our instance optimisations on solving performance. We start with the difference in solving time and then consider the search space reduction.

Solving Time

The speedups in solving time are most impressive, for both Gecode and MINION. We illustrate the speedups in Fig. 8.22 for group B, and in Fig. 8.23 for group C. Since MINION benefits slightly more from the enhancements (due to particular limitations of its constraints repertory), the results for MINION are better than those for Gecode: In group B we observe speedups up to 2.7 for Gecode and 3.5 for MINION, while for group C we see speedups ranging up to 1,300 for Gecode and 3,400 for MINION. Note, that these speedups are mainly gained 'for free', since tailoring time hardly suffers from applying optimisation.

We give an overview of the observed speedup factors in Tab. 8.2 for Problem Group B and in Tab. 8.3 for Problem Group C. Each table illustrates the smallest, average and largest speedup factor observed for the respective problem class. Note that instances with timeouts are not included in these speedups (i.e. if an enhanced instance was solved without timeout, but the unenhanced instances has timed out, we do not add this difference in solving time, since it is not useful). As the tables demonstrate, there can be a big difference in speedups between instances of the same class.

Reduction of Search Space

The instance optimisations even resulted in search space reductions for some problem classes (only in group C), for both MINION and Gecode, which are depicted in Fig. 8.24. The most impressive search space reduction is obtained in the Peaceful Armies of Queens Problem, where both in Gecode and MINION some enhanced instances requires only 1% of the search space unenhanced instances require. This demonstrates the potential benefit our instance optimisations can have on the search space.

Problem Group B	Speedup Factor		
	Smallest	Average	Largest
Chessboard Colouring	1.08	1.64	3.24
Crosswords Words	1.00	1.07	1.53
Crosswords Herald	1.00	1.06	1.46
Knight's Tour XCSP	1.31	1.86	2.51
Queens Knights Mul	1.70	1.70	1.70
Queen Attacking	1.35	1.64	1.82
Quasigroup 3 non-idem	1.00	1.02	1.06
Quasigroup 4 idem	1.02	1.18	1.52
Quasigroup 4 non-idem	0.96	1.02	1.06
Peaceful Army of Queens	1.16	1.235	1.31

Table 8.2: **Speedup Overview for Problem Group B**, showing smallest, average and largest speedup observed in the set of instances drawn from the respective class. Timed out instances are excluded from this discussion.

Problem Group C	Speedup Factor		
	Smallest	Average	Largest
Golomb Ruler naive	1.96	22.02	57.82
Peg Solitaire State	2.12	4.76	8.71
Peg Solitaire Action	4.67	6.21	7.27
Plotting	4.83	6.58	7.34
Knights	16.47	20.37	26.32
n-Queens naive	1.15	2.89	8.12
Peaceful Army Queens 1	108.84	1,213.40	3,418.61
Peaceful Army Queens 3	90.86	707.43	1,852.23

Table 8.3: **Speedup Overview for Problem Group C**, showing smallest, average and largest speedup observed in the set of instances drawn from the respective class. Timed out instances are excluded from this discussion.
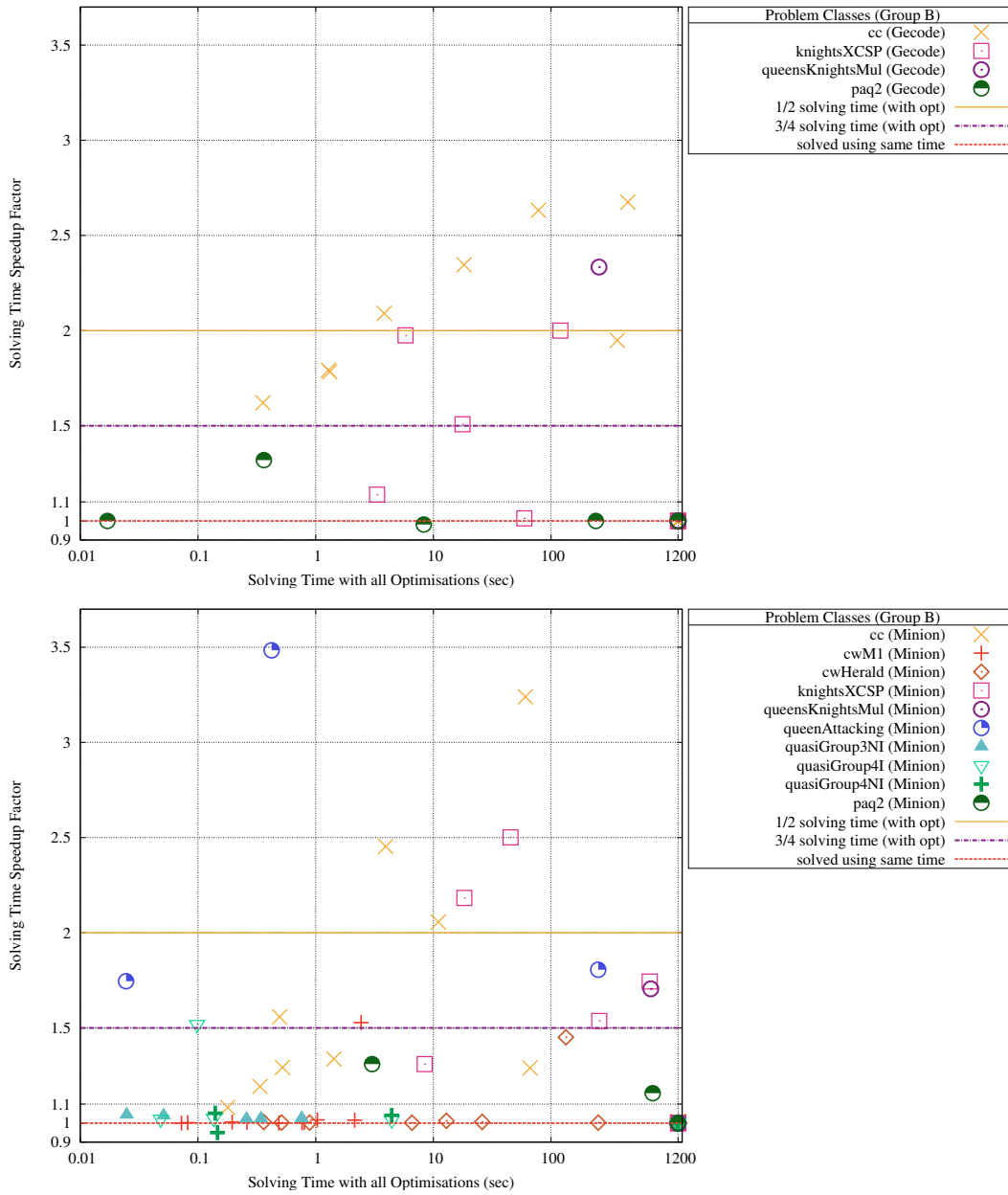
Figure 8.22: **Solving Time Speedup** by tailoring using **all optimisation techniques** for solver **Gecode** (top) and solver MINION (bottom). The x-axis depicts the solving time (in seconds) with optimisations; the y-axis shows the solving time speed up factor of instances tailored using all optimisations. For instance, points at $y = 5$ represent instances that were solved 5 times faster with optimisations than without.

Figure 8.23: **Solving Time Speedup** by tailoring using **all optimisation techniques** for solver **Gecode** (top) and solver MINION (bottom). The x-axis depicts the solving time (in seconds) with optimisations; the y-axis shows the solving time speed up factor of instances tailored using all optimisations. For instance, points at $y = 5$ represent instances that were solved 5 times faster with optimisations than without.
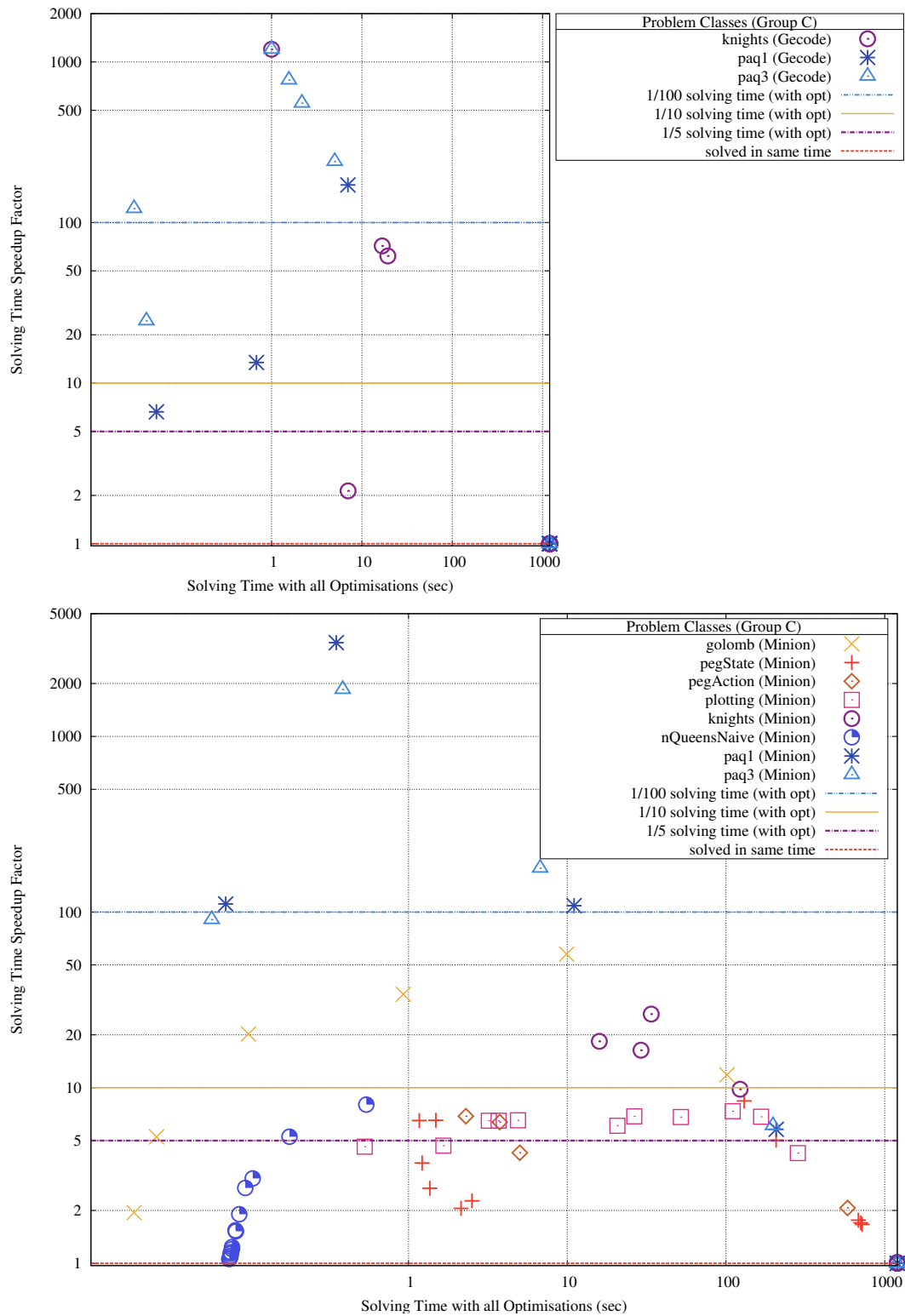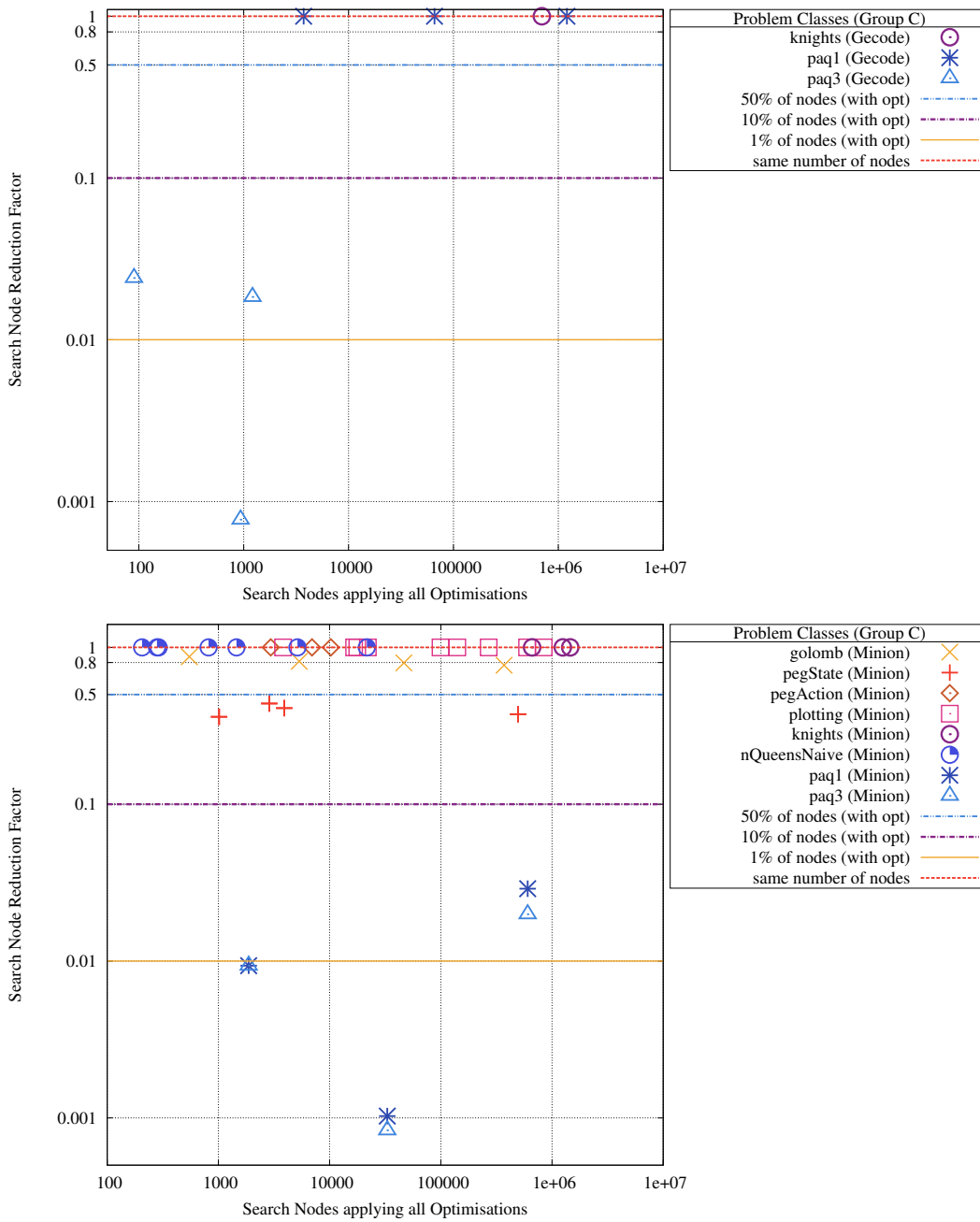
Figure 8.24: **Search Space Reduction** by tailoring using **all optimisation techniques** on group C, in solver **Gecode** (top) and solver MINION (bottom). The x-axis depicts the search nodes with optimisations; the y-axis shows the search node reduction factor of instances tailored using all optimisations. For instance, points at $y = 0.01$ represent instances that were solved using only 1% of the search space of the search space used to solve the same instance without optimisation

CHAPTER 9

CONCLUSIONS

This dissertation has defended the thesis that the compilation from solver-independent constraint model to solver input can be *automated* and extended with light-weight *model optimisations* by presenting an automated modelling framework that automatically tailors problems formulated in a solver-independent modelling language to solver input while performing model enhancements that can result in substantial solving time speedups. This framework clearly facilitates the access to Constraint Programming techniques for novices to the area, since the user requires no particular knowledge of the target solver and poor modelling choices from inexperience can be compensated for by the automated optimisations.

The central contributions are threefold: first, we thoroughly discussed the process of *tailoring*, the compilation of solver-independent constraint models to low-level solver formats. We presented a generic and easily-extendable approach that can process both problem instances and problem classes. The second contribution was the integration of light-weight *optimisation techniques* into the process of tailoring. These techniques add negligible overhead to tailoring time and can produce instances that are solved in fractions of the time that unenhanced instances take. The third contribution was the implementation of all these techniques in the tool TAILOR, that is freely available online. Note, that TAILOR has already contributed in rendering constraint programming techniques more accessible to non-experts, for instance, by providing assistance in scheduling machines that are used in cancer research [52]:

> *"Tailor's implementation of the expressive Essence' constraint modelling language allowed the use of the powerful Minion solver by non experts"*

(taken from the abstract of [52])

The final chapter is organised as follows. First, in Sec. 9.1, we summarise the main features of this work and describe the central contributions in more detail. Then, we outline ideas for future work with respect to this thesis in Sec.9.2.

9.1 Summary

This thesis has presented a framework in which solver-independent constraint models are automatically tailored to solver input and additionally enhanced, which in practice, can result in spectacular solving time speedups. In the following, we summarise the core contributions and features of this thesis.

9.1.1 Specification of ESSENCE′

First, we gave in Chapter 2 a detailed overview of the solver-independent constraint modelling language ESSENCE′. ESSENCE′ is a derivative of the problem specification language ESSENCE, and has not been explictly defined to date. Appendix A gives a formal syntax specification in BNF notation.

9.1.2 Tailoring Constraint Models

Second, we presented the overall structure of a generic tailoring engine: by following an approach similar to Compiler Construction, a generic architecture is introduced that allows us to easily extend the tailoring engine with further input- and output-languages, as well as reusing core parts of the translation (preprocessing and flattening) for any target solver. The generalisation of the core is achieved by the use of solver profiles that summarise a target solver's features. The tailoring engine TAILOR incorporates this general architecture and demonstrates the efficiency of its structure, supporting 2 modelling languages as input and generating 3 solver formats as output.

Furthermore, we consider tailoring of both problem instances and whole problem classes. The latter is a completely novel approach that, to the best of our knowledge, has not yet been investigated. Tailoring classes has two main applications: first, to support solvers that are libraries of programming languages (like C++, Java or Prolog) in which problems can be formulated as classes and parameters specified at runtime, after compilation. The second application is that of *class-wise* tailoring, which denotes an alternative approach for targeting solvers to the standard tailoring method of *instance-wise* tailoring. Class-wise tailoring first generates a flat problem class, which is then used to generate instances from. This can be a faster tailoring approach than instance-wise tailoring, since particular expensive operations, such as flattening, are only performed *once* for the problem class and need not be repeated for every instance. However, tailoring classes is a more advanced translation process than tailoring instances, and still has limitations, like the introduction of unconstrained auxiliary variables that are generated for particular kinds of guards. Fortunately, in our empirical analysis, we have observed that this redundancy does not heavily affect the tailoring process, as long as the auxiliary variables are not included in search. In summary, we showed that tailoring classes is a powerful technique which we expect to be

more significant in the future.

9.1.3 Optimisation during Tailoring

Third, we presented a set of cheap but effective instance optimisation techniques that were mainly inspired by code optimisations in Compiler Construction. We started with a discussion on related work, in particular, enhancement techniques that stem from the areas of Constraint Programming and Compiler Construction, emphasising connections to the techniques we propose.

Removing Duplicate Constraints

The first optimisation technique is that of removing duplicate constraints that often occur in models of inexperienced users as a result of weak Boolean guards. This technique was discussed both at instance and class level. At instance level, the elimination is simple, since preprocessing assists in detecting duplicates: ordering of constraints places duplicates next to another. At class level, duplicate constraints can only be avoided by strengthening the Boolean guard, hence reasoning over the quantified expression is necessary. Therefore, we propose an algorithm for strengthening guards, exploiting unification, which assists us in determining the set of conditions that need to be added to the guard in order to prevent duplicates. In our empirical evaluation, we examined the consequences of duplicate constraints in two naive constraint models and observed that duplicates can cause a doubling of solving time, if not eliminated. In summary, removing duplicate constraints in models of novices is an important optimisation technique that can prevent a considerable increase in runtime.

Common Subexpression Elimination

The second optimisation technique is the most powerful technique of our instance optimisations: common subexpression elimination (CSE). CSE is an already wide-spread technique in related areas, such as SAT or Proof Theory, so its integration into the context of Constraint Programming was an obvious step (a step we were surprised to find that no-one else has taken before us). CSE-techniques in SAT or Proof Theory are performed using Directed Acyclic Graphs (DAGs) through which common subexpressions can be easily detected. However, since our aim is to include instance optimisations during tailoring, we proposed a novel approach: the elimination of common subexpression during flattening. We show that CSE-flattening is a light-weight approach that for many instances even lies in the same complexity class as standard flattening. Furthermore, in our experimental results, we observe that tailoring time actually *decreases* with the number of subexpressions that are eliminated, since fewer expressions need to be flattened. However, the most impressive results are the benefits we obtain from CSE during solving: in our empirical analysis, we

observe dramatic solving time speedups (up to a factor of 2,000) and in some cases even a vast reduction of search space (down to 1% of the search space unenhanced instances require).

We considered CSE at both instance and class level: though CSE from instance level is directly applicable at class level, it does not detect all common subexpressions that are detected at instance level. Therefore, we propose three different CSE-approaches at class level that each provide particular benefits but also drawbacks. The first two approaches cannot detect shifted common subexpressions, which are a particular family of subexpressions that are equivalent in different 'iterations' of a quantification. The detection of shifted common subexpressions requires to either reason over the quantifying domains that dereference arrays, or to approximate the respective quantifying domain, a concept that is embodied in the third approach. The third CSE approach detects the most common subexpressions at class level, but, in the worst case, can introduce redundancies (that however, hardly occur in practice). In summary, we have proposed a set of CSE approaches for class level that aim at detecting the same set of common subexpressions as we detect at instance level.

In conclusion, CSE is a powerful technique that can easily be embedded into flattening for practically no overhead. It is particularly successful, even on expert models, and can provide speedups of an order of a magnitude that sometimes come along with dramatic search space reductions.

Increasing the Number of Identical Subexpressions

In order to extend the benefits from common subexpressions by CSE-flattening, which detects only *identical* subexpressions (i.e. syntactically equivalent expressions), we extend CSE by adding measures that reformulate equivalent, but not identical expressions into an identical format. The detection of equivalence between two subexpressions can be arbitrarily complex, hence we restrict our investigations to simple cases of equivalence, that, nevertheless, often occur in practice.

Active Negation Reformulation   The first reformulation is the most successful reformulation: the active negation reformulation, which is concerned with detecting subexpressions where one subexpression is the negation of the other. In that case, one subexpression can be replaced with the negated auxiliary variable that represents the other (and vice versa). The detection of this equivalence can be easily embedded into tailoring, and also adds no overhead to the basic CSE procedure, since it lies in the same complexity class. This is also confirmed by our experimental results, that illustrate that the reformulation does not cause an overhead in tailoring time, but results in a considerable reduction of solving time, in some cases, to more than half the solving time without the reformulation. In summary, the active negation reformulation is a successful and cheap technique that should be part of a standard tailoring system.

Active Horn Clause Reformulation   The second reformulation exploits the equivalence between implications and Horn Clauses. The important observation in this context is that a disjunction of relational constraints can typically always be represented as a Horn Clause, since we can 'move' the negation 'outside' the relational expression by flipping the relational operator, e.g. reformulating $(x < y)$ into $\neg(x \geq y)$ for all but one disjoint argument, leaving one argument positive. The choice of positive literal is the main difficulty in this approach, since an optimal choice has to take other subexpressions of the instance into account. For now, we apply a simple heuristic that chooses the positive literal according to its expression order. This heuristic, however, has not brought the expected benefits: in our empirical analysis we observe an impairment of solving performance by about 40% (i.e. it took 40% more time to solve the instances with the Horn Clause reformulation). The reason for this might be the consequence of some internal feature of MINION, of which we are not aware of, or, more likely, stem from the immatureness of our heuristic. Improving our heuristic on how to generate the most effective Horn Clause from a disjunction, is an important item of future work.

Active De Morgan Reformulation   The third reformulation is the active De Morgan Reformulation which uses the equivalence between expressions through De Morgan's Law. In particular, since expressions are normalised to Negation Normal Form (where negations are propagated to the leaves of expressions trees) during preprocessing, De Morgan's Law can only be applied one-way, which can yield further common subexpressions if one the arguments is negated somewhere else in the model. In other words, the active De Morgan reformulation can be considered an extension of the active negation reformulation. Unfortunately, the reformulation did not fire in any of our examples in our empirical analysis, however, since attempting it added no overhead to tailoring time, we have chosen to include it as standard optimisation technique during tailoring.

Undetected Common Subexpressions

A discussion on which equivalent subexpressions we can detect cannot lack the discussion on which equivalent subexpressions we do *not* detect. Therefore, this discussion has been included, evolving around the scope of CSE.

First, we mention the detection of equivalences between a global constraint and its decomposed representation, which would be a powerful equivalence to exploit. However, detecting these kind of equivalences is typically very expensive (e.g. detecting a maximal clique of disequalities to match with *alldifferent* is NP-complete) and therefore not useful at instance level. However, at class level, a higher detection effort would be worthwhile, since every instance drawn from the enhanced class would benefit from the enhancement. Unfortunately, a thorough investigation of detecting this type of equivalence was out of scope of this thesis and is expected to be a fruitful item of future work.

Second, there is a family of common subexpressions that we only detect in limited cases:

argument common subexpressions (argument-CS). Argument-CS are arguments that are shared among n-ary associative and commutative expressions, e.g. $b + c + d$ and $a + b + c$ share the argument-CS $b+c$. Detecting this kind of common subexpression does not provide the same improvement as CS detected by basic CSE, since their elimination does not save an auxiliary variable, but only reduces the arity of the respective constraint. There exist examples where the general elimination of argument-CS is beneficial and can even improve search, however, in practice, we have only encountered one particular kind of argument-CS that occurs in special kinds of quantified expressions. We proposed an algorithm in order to detect these argument-CS and our empirical analysis shows that their elimination can yield a notable solving time reduction of about 50% for small instances and 25% for larger instances. These were significant results, since the number of argument-CS was very small (compared to the number of common subexpressions eliminated by basic CSE or the active negation reformulation). However, the question of whether it is worthwhile to detect all argument-CS *in general* is difficult and requires further investigation.

Quantification Optimisations

The third optimisation technique is concerned with expression representations involving quantifications, which are a powerful tool when modelling in a solver-independent modelling language, such as Essence$'$.

First, we investigate the issue of weak Boolean guards, and show that the first two optimisation techniques, CSE and elimination of duplicates, already take care of the redundancies stemming from weak guards.

Second, we consider *loop-invariant* expressions, which are expressions in quantifications, that are independent of the quantifying variable and can hence be moved *outside* the quantification. The choice of moving a loop-invariant expression inside or outside a quantification is the main objective of our discussion on quantification optimisations.

Initially, we expected that moving a loop-invariant expression outside the respective quantification would generally be the better choice. However, we have encountered cases, where the inside representation outperforms the outside representation, in particular, expressions involving universal quantification and implications. We have tested this particular case on the Peaceful Armies of Queens Problem and have seen that the inside representation clearly dominates the outside representation in solving performance. These results are particularly interesting, since they refute a common belief and demonstrate how little we yet know about the right representation of complex, quantified constraint expressions.

9.1.4 The tool TAILOR

Almost all tailoring features in this thesis are implemented in the tool TAILOR, which is a major contribution, in particular since it is freely available on the web. TAILOR and the

optimisation techniques it contains have been thoroughly assessed in our empirical analysis, which has shown very satisfactory results, tailoring most problem instances within fractions of a second and providing speedups factors of up to 3,400. Furthermore, the ease of use of TAILOR is another important feature of our implementation: by providing an inter-active modelling environment, where problems can be modelled in a clean way, using powerful constructs like quantifications, and the problem solution is just a button click away. We hope that the proposed techniques will spread among other tailoring tools and help make Constraint Programming techniques more accessible from outside out community.

9.1.5 The Missing Link in Automated Constraint Modelling

The last and probably most significant contribution of this thesis was the identification and automation of the missing link automated modelling, i.e. the automated translation from solver-independent constraint model to solver input. To date, automated constraint modelling has mainly focussed on how to *formulate* a given problem as a constraint model, an essential task to reduce the modelling bottleneck. This is, however, not enough. In order for a (generated) model to be *solved*, it has to be formulated in the solver language, a non-trivial step that constitutes another challenge to the modeller, a challenge that has not yet been recognised and investigated for automation.

9.2 Future Work

In this section we briefly outline our plans for future work with respect to the work presented in this thesis.

9.2.1 Addressing Redundancies when Tailoring Classes

As we have noted in Sec. 5.3.2 our approach of tailoring classes can introduce redundancies in form of unconstrained auxiliary variables, if the respective quantified subexpression is guarded by a Boolean guard that evaluates to *true* in particular cases. For instance, in the example below,

```
   given n,m : int(1..)
   find x,y : matrix indexed by [int(1..n)] of int(1..m)

 such that
   forall i,j : int(1..n) .
      (i<j)  =>  (x[i]*x[j] != y[i]*y[j])
```

flattening will introduce $2n^2$ auxiliary variables (one array of length n^2 for each multiplication, since both i and j range over $(1..n)$). However, since $(i < j)$ will evaluate to

false in $\frac{n(n+1)}{2}$ cases, only $2(n^2 - \frac{n(n+1)}{2})$ auxiliary variables are actually used, the rest are unconstrained.

This is a considerable limitation that needs to be addressed. We have discussed different possible approaches as how to treat this issue:

1. Introducing new data structures that allow 'holes'

2. Extending the modelling language to support local auxiliary variables and investigate if preventing CSE (since local auxiliary variables have a limited scope) affects the solving performance of the class

3. Extending the modelling language with high-level comprehensions to allow us to explicitly state a mapping between the quantifying variable's assignments and the indices of the corresponding auxiliary array.

Exploring these three possibilities as well as considering other approaches will be an important item of our future work.

9.2.2 Extending the set of Model Optimisations

In this work, we have considered a number of useful optimisation techniques during tailoring, however, there is much scope for extending the optimisations that are performed to date.

Integrating further Optimisations into Tailoring

There exist many more possibilities of optimisations that could be made available during tailoring. First, we could explore further equivalence relations/properties (like Distributivity) that we could exploit in order to formulate further *active* reformulation techniques.

Second, it would be very valuable to detect decomposed representations of global constraints and replace them with their respective global constraint representation. This would be very beneficial for two main reasons: first, global constraints typically provide a better solving performance than their decomposed representation. Second, inexperienced modellers are often not aware of the existence of particular global constraints and simply make no use of them. Therefore, automatically replacing (or proposing a replacement to the user in an inter-active way) would facilitate modelling, in particular for novice users. Evidently, such a detection would be far more expensive than the techniques that we have proposed in this work, so some might have to be integrated in a *restricted* fashion.

Restricted Optimisations during Tailoring

Many optimisation techniques fire only for a small selection of problems. For instance, the algorithm to learn a global gcc constraint [13] is a specific optimisation that is particularly valuable for special problem models. However, since these techniques can be very costly, it is preferable to apply them only in a restricted fashion during tailoring. In other words, some optimisation techniques should only be applied if there is a high probability that the technique fires on the respective instance/class. For example, this could be achieved by introducing heuristic approaches that interpret the model structure (e.g. constraint graph, expression tree structures, etc) and give an estimate of how applicable a given optimisation technique is on the problem model. This estimate can then guide the choice of which optimisation technique to apply to a constraint model.

Exploring Non-deterministic Reformulations for Optimisations

In some cases, there are several different possibilities of how to enhance a problem formulation. We have seen this in two cases during this work:

Active Horn Clause Reformulation In the active Horn Clause reformulation (Sec. 4.3.5), the issue arises that there are several different ways to represent a disjunction as a Horn Clause. More specifically, an n-ary disjunction of relational constraints has $n + 1$ different Horn Clause representations (since any of the n arguments can be chosen as the positive literal, in addition to the Horn Clause with no positive literal). The choice of which argument to select as positive literal can be vital (as our empirical results have shown), and depends on other subexpressions in the constraints instance/class. Therefore, it would be useful to explore possibilities on how to perform this choice in an efficient way.

Conflicting Common Subexpressions A similar issue arises with conflicting common subexpressions in argument common subexpressions (argument-CS) (Sec. 4.4.2). Common subexpressions are conflicting [6] if there are different possibilities of eliminating them. For instance, in the example below, if we eliminate the common subexpression 'x+y', we cannot eliminate the common subexpressions 'y+z' or 'x+y+z'.

```
  find x,y,z,s,r : int(-5..5)
  find t,u,v : int(1)
  find w : int(2)

such that
  x + y + z + t = v,
  x + y + z + u = w,
  x + y + t + s = r
```

One possibility would be to eliminate the common subexpression with the most occurrences (which is 'x+y' in the example above), however, this does not necessarily

provide the greatest benefits (if we would eliminate 'x + y + z', then propagation would immediately detect unsatisfiability). Investigating heuristics on how to deal with conflicting common subexpressions would be a valuable contribution.

Optimising the Choice of Search Strategies

A notable limitation of this work is that the selected search strategy is not adapted to the specific problem that is tailored, but the default heuristic is selected. Note, that this is not a limitation for the expert modeller using a tailoring tool, since she can (and probably *wants* to) try out different search heuristics by editing the solver input manually. However, this can pose a problem for inexperienced users that are not aware of the effects of different search strategies.

Finding a general approach of automatically determining an efficient search strategy for a given constraint model is probably as difficult as finding a general strategy on determining the best constraint model formulation for a given combinatorial problem. However, we hope in future we will be able to perform reasoning of some form in order to determine a good heuristic as to what heuristics to choose.

BIBLIOGRAPHY

[1] M. Arangu A. Garrido, E. Onaindia. Using constraint programming to model complex plans in an integrated approach for planning and scheduling. In *UK Planning and Scheduling SIG Workshop (PLANSIG)*, pages 137–144, 2006.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 9, pages 583–705. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[5] Frances E. Allen. A basis for program optimization. In *IFIP Congress (1)*, pages 385–390, 1971.

[6] Ignacio Araya, Bertrand Neveu, and Gilles Trombettoni. Exploiting common subexpressions in numerical csps. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, pages 342–357, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Fahiem Bacchus and Toby Walsh, editors. *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*. Springer, 2005.

[8] Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In Wilson and Lane [88], pages 525–530.

[9] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

[10] Christian Bessiere. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapter Propagation, pages 29–83. Elsevier Science Inc., New York, NY, USA, 2006.

[11] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. Acquiring constraint networks using a sat-based version space algorithm. In *AAAI'06: proceedings of the 21st national conference on Artificial intelligence*, pages 1565–1568. AAAI Press, 2006.

[12] Christian Bessière, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Veloso [85], pages 50–55.

[13] Christian Bessière, Remi Coletta, and Thierry Petit. Learning implied global constraints. In Veloso [85], pages 44–49.

[14] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Range and roots: Two common patterns for specifying and propagating counting and occurrence constraints. *Artif. Intell.*, 173(11):1054–1078, 2009.

[15] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[16] Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors. *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2006.

[17] Alan Bundy. A science of reasoning. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 178–198, 1991.

[18] John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In Brewka et al. [16], pages 73–77.

[19] Choco. Choco constraint programming system, 2009. http://choco.emn.fr.

[20] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, 1970.

[21] Maria Garcia de la Banda and Enrico Pontelli, editors. *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*. Springer, 2008.

[22] Minh Binh Do and Subbarao Kambhampati. Solving planning-graph by compiling it into csp. In *AIPS*, pages 82–91, 2000.

[23] Henry Ernest Dudeney. Send more money puzzle. *Strand Magazine*, 68:97, 214, 1924.

[24] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press / Brooks/Cole Publishing Company, 2002.

[25] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ansi c. *C. SIGPLAN Notices*, 26:29–43, 1991.

[26] Alan Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[27] Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In Kaelbling and Saffiotti [45], pages 109–116.

[28] Alan M. Frisch, Ian Miguel, and Toby Walsh. Cgrass: A system for transforming constraint satisfaction problems. In O'Sullivan [59], pages 15–30.

[29] Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In Gent [31], pages 367–382.

[30] I. P. Gent, I. Miguel, A. Rendl, and P. Gregory. Enhancing constraint models of planning problems by common subexpression elimination. In V. Bulitko and J. C. Beck, editors, *Proceedings of the Syposium on Abstraction, Reformulation and Approximation*, pages 128–135, 2009.

[31] Ian P. Gent, editor. *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*. Springer, 2009.

[32] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Brewka et al. [16], pages 98–102.

[33] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000, 2008.

[34] Ian P. Gent, Ian Miguel, and Andrea Rendl. Tailoring solver-independent constraint models: A case study with essence' and minion. In Miguel and Ruml [56], pages 184–199.

[35] Ian P. Gent, Karen Petrie, and Jean-Francois Puget. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapter Symmetry in Constraint Programming, pages 329–376. Elsevier Science Inc., New York, NY, USA, 2006.

[36] Ian P. Gent and Toby Walsh. Csplib: A benchmark library for constraints. Technical Report APES-09-1999, 1999.

[37] Matthew L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res. (JAIR)*, 1:25–46, 1993.

[38] Georg Gottlob and Toby Walsh, editors. *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*. Morgan Kaufmann, 2003.

[39] P. Gregory and A. Rendl. A constraint model for the settlers planning domain. In R. Aylett, editor, *Proceeding of the UK PlanSIG*. Herriot Watt University, December 2008.

[40] Warwick Harvey and Peter J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003.

[41] Brahim Hnich. *Function variables for Constraint Programming*. PhD thesis, Uppsala University, 2003.

[42] Alan J. Hu and Andrew K. Martin, editors. *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*. Springer, 2004.

[43] C. Jefferson, A. Miguel, I. Miguel, and A. Tarim. Modelling and solving english peg solitaire. *Computers and Operations Research*, 33(10):2935–2959, 2006.

[44] Peter J.Stuckey, Ralph Becket, Sebastian Brand, Mark Brown, Thibaut Feydy, Julien Fischer, Maria Garcia de la Banda, Kim Marriott, and Marc Wallace. The evolving world of minizinc. In *Workshop on Constraint Modelling and Reformulation*, pages 156–170, 2009.

[45] Leslie Pack Kaelbling and Alessandro Saffiotti, editors. *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*. Professional Book Center, 2005.

[46] Donald E. Knuth. *The Art of Computer Programming 3. Sorting and Searching: The Classic Work Newly Updated and Revised*. Addison-Wesley Longman, Amsterdam, 2. a. edition, June 1998.

[47] Thorsten Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004. ZIB-Report 04-58.

[48] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[49] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple bounded ltl model checking. In Hu and Martin [42], pages 186–200.

[50] Christophe Lecoutre. Xcsp 2.1 benchmarks, December 2009. `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`.

[51] James Little, Cormac Gebruers, Derek G. Bridge, and Eugene C. Freuder. Using case-based reasoning to write constraint programs. In Rossi [66], page 983.

[52] Andrew Loewenstern. Scheduling the cb1000 nanoproteomic analysis system with python, tailor, and minion. In Gent [31], pages 65–72.

[53] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *Journal of AI Research*, 20:1–59, 2003.

[54] Adriana Lopez and Fahiem Bacchus. Generalizing graphplan by formulating planning as a csp. In Gottlob and Walsh [38], pages 954–960.

[55] Darko Marinov, Sarfraz Khurshid, Suhabe Bugrara, Lintao Zhang, and Martin C. Rinard. Optimizations for compiling declarative models into boolean formulas. In Bacchus and Walsh [7], pages 187–202.

[56] Ian Miguel and Wheeler Ruml, editors. *Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings*, volume 4612 of *Lecture Notes in Computer Science*. Springer, 2007.

[57] B.A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5, issue 3:16–23, 1990.

[58] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[59] Barry O'Sullivan, editor. *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming, Cork, Ireland, June 19-21, 2002. Selected Papers*, volume 2627 of *Lecture Notes in Computer Science*. Springer, 2003.

[60] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[61] Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the clp scheme. *J. Log. Program.*, 16(3):319–359, 1993.

[62] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *AAAI/IAAI*, pages 359–366, 1998.

[63] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994.

[64] Jean-Charles Régin and Michel Rueher, editors. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*. Springer, 2004.

[65] Andrea Rendl. TAILOR: Tailoring constraint models to solvers, December 2009. http://www.cs.st-andrews.ac.uk/~andrea/tailor.

[66] Francesca Rossi, editor. *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*. Springer, 2003.

[67] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[68] Olivier Roussel and Christophe Lecoutre. Xml representation of constraint networks format XCSP 2.1. `http://www.cril.univ-artois.fr/CPAI08/XCSP2_1Competition.pdf`, 2008.

[69] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*, chapter Inference in First-Order Logic, pages 272–319. Prentice Hall, 2003.

[70] Oscar Sapena, Eva Onaindia, Antonio Garrido, and Marlene Arangu. A distributed csp approach for collaborative planning systems. *Eng. Appl. Artif. Intell.*, 21(5):698–709, 2008.

[71] Hermann Schichl and Arnold Neumaier. Interval analysis on directed acyclic graphs for global optimization. *J. of Global Optimization*, 33(4):541–562, 2005.

[72] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space? *ACM Trans. Program. Lang. Syst.*, 27(3):388–425, 2005.

[73] Christian Schulte and Peter J. Stuckey. Dynamic analysis of bounds versus domain propagation. In de la Banda and Pontelli [21], pages 332–346.

[74] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling with gecode, November 2009. `http://www.gecode.org/doc-latest/modeling.pdf`.

[75] Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, April 1994.

[76] Barbara M. Smith, Karen E. Petrie, and Ian P. Gent. Models and symmetry breaking for 'peaceable armies of queens'. In Régin and Rueher [64], pages 271–286.

[77] Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Modelling the golomb ruler problem. In *IJCAI-99 Workshop on Non-binary Constraints*, 1999.

[78] Java Sun. Java api 1.5.0, November 2009. `http://java.sun.com/j2se/1.5.0/docs/api/`.

[79] Guido Tack. Gecode-flatzinc interpreter, December 2009. `http://www.gecode.org/flatzinc.html`.

[80] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[81] Peter van Beek. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapter Backtracking Search Algorihthms, pages 85–133. Elsevier Science Inc., New York, NY, USA, 2006.

[82] Peter van Beek and Xinguang Chen. Cplan: a constraint programming approach to planning. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference*

on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence, pages 585–590, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[83] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[84] Willem-Jan van Hoeve and Irit Katriel. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapter Global Constraints, pages 169–207. Elsevier Science Inc., New York, NY, USA, 2006.

[85] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.

[86] Vincent Vidal and Héctor Geffner. Branching and pruning: an optimal temporal pocl planner based on constraint programming. *Artif. Intell.*, 170(3):298–335, 2006.

[87] Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming, 1997.

[88] David Wilson and H. Chad Lane, editors. *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008, Coconut Grove, Florida, USA*. AAAI Press, 2008.

[89] Pieter Wuille and Tom Schrijvers. Monadic constraint programming with gecode. In *Workshop on Constraint Modelling and Reformulation*, pages 171–185, 2009.

Appendix A

The Syntax of Essence′

Essence′ is a solver-independent constraint modelling language, which is a subset of the abstract specification language Essence [26]. Hence Essence′ can be used to

- formulate constraint problem models

- specify parameter values

- summarise problem solutions

This grammar specification outlines each of these different specifications in the way they are supported by the Essence′ translator Tailor [34].

A.1 Grammar Specification

An Essence′ problem model consists of two separate specifications: a problem specification defining decision variables, domains and constraints, and a parameter specification giving parameter values to specify the problem instance. The solution(s) of a problem instance can then be summarised by a solution specification. Hence we have three different types of specifications:

1. problem model

2. parameter specification

3. solution specification

Comments in Essence′ are preceded by $, which can be placed everywhere in the grammar. $ is a line-wise comment. Before giving a concise grammar for each part, we want to give an overview of the notation that is used.

A.1.1 Notation

- Terms written in *italic font* are non-terminals and terms written in `typewriter font` or special characters that are underlined (such as <u>:</u>) are terminals.

- A *letter* is an alphabetic character. An *identifier* is a string whose first character is a *letter* and the rest of its characters are alphanumeric or "_". Identifier recognition is case sensitive.

- A *number* is any string whose elements are the numeric characters.

- $\{a\}$ stands for a non-empty list of as.

- $\{a\}$' stands for a non-empty list of as separated by commas.

- $[a]$ stands for one or zero occurences of a.

A.1.2 Grammar: Problem Specification

Below is the specification for both problem model and parameter specification syntax. Note however, that only constant definitions are extracted from parameter specifications - objectives/variable declarations/constraints will be ignored.

Model

Model	::=	*Header*
		[{*Declaration*}']
		[*Objective*]
		[`such that` { *RelationalExpression* }']
Header	::=	`ESSENCE`' *number* <u>.</u> *number*
Declaration	::=	`given` { *DomainIdentifier* }' |
		`where` { *Expression* }' |
		`letting` { *Constant* }' |
		`find` { *DomainIdentifier* }'
Objective	::=	`maximising` *ArithmeticExpression* |
		`minimising` *ArithmeticExpression*
DomainIdentifier	::=	{ *identifier* }' <u>:</u> *Domain*
Constant	::=	*identifier* `be domain` *Domain* |
		identifier [<u>:</u> *Domain*] `be` *Expression*

Domains

SimpleDomain	::	`bool` |
		`int` ({ *RangeDomain* }') |
		identifier
Domain	::=	(*Domain*) |
		SimpleDomain |
		`matrix indexed by` [{ *SimpleDomain* }'] `of` *SimpleDomain*
RangeDomain	::=	*BoundedDomain* |
		..*Expression* | *Expression* ..
BoundedDomain	::=	*Expression* | { *Expression* }' |
		Expression .. *Expression* |

Constraint Expressions

Expression	::=	(*Expression*) |
		RelationalExpression |
		ArithmeticExpression
RelationalExpression	::=	`false` | `true` |
		AtomExpression |
		! *RelationalExpression* |
		Expression Relop Expression |
		RelationalExpression Boolop RelationalExpression |
		QuantifiedExpression |
		GlobalConstraint
ArithmeticExpression	::=	*number* |
		AtomExpression |
		− *ArithmeticExpression* |
		| *ArithmeticExpression* |
		ArithmeticExpression Mulop ArithmeticExpression |
		QuantifiedSum |
		(`min` | `max`) (*ArithmeticExpression* , *ArithmeticExpression*)
AtomExpression	::=	*identifier* | *ArrayElement*
ArrayElement	::=	*identifier* [{ *IndexRangeExpression* }']
QuantifiedExpression	::=	(`forall` | `exists`) { *identifier* } ' : *BoundedDomain* .
		RelationalExpression
QuantifiedSum	::=	`sum` { *identifier* } ' : *BoundedDomain* .
		ArithmeticExpression
GlobalConstraint	::=	`alldiff` (*AtomExpression*) |
		`element` (*AtomExpression* , *AtomExpression* , *AtomExpression*) |
		`table` ([{ *AtomExpression* }'] , [{ [{ *number* }'] }']) |
		`atleast` (*AtomExpression* , *ConstantList* , *ConstantList*) |
		`atmost` (*AtomExpression* , *ConstantList* , *ConstantList*)
MulOp	::=	+ | − | / | `*` | ^ | %
BoolOp	::=	\/ | /\ | => | <=>
RelOp	::=	= | != | <= | < | >= | > |
		`<lex` | `<=lex` | `>lex` | `>=lex`

Further Restrictions

- Quantifications may not range over decision variables, i.e. *Expression*s in *Bounded-Domain*s may not contain decision variables

Operator	Functionality	Associativity
,	comma	Left
:	colon	Left
( )	left and right parenthesis	Left
[ ]	left and right brackets	Left
!	not	Right
/\	and	Left
\/	or	Left
=>	if (implication)	Left
<=>	iff (logical equality)	Left
−	unary minus	Right
^	power	Left
* / %	multiplication, integer division, modulo	Left
+ −	addition, substraction	Left
< <= > >= <lex <=lex >lex >=lex	(lex)less, (lex)less or equal, (lex)greater, (lex)greater or equal	none
= !=	equality, disequality	none
.	dot	Right

Table A.1: Operator precedence in ESSENCE′

A.1.3   Grammar: Solution Specification

SolutionSpecification	::=	*Header*
		[{*Solution*}′]
Header	::=	ESSENCE′ *number* . *number*
Solution	::=	variable *identifier* is { *SolutionExpression*}′
SolutionExpression	::=	*number* |
		ConstantArray |
ConstantArray	::=	[{ *number* }′] |
		[{ *ConstantArray* }′]

A.2   Operator Precedence

Table 1 describes the precedence of the operators that are arranged by decreasing order of precedence (the operators on top have highest precedence)

A.3 Examples

To illustrate the grammar specified above, we give some examples. These examples can be found at TAILOR's website at `http://www.cs.st-and.ac.uk/~andrea/tailor`.