MARCO VARGAS CORREIA

MODERN TECHNIQUES FOR CONSTRAINT SOLVING
THE CASPER EXPERIENCE

Dissertação apresentada para obtenção do Grau de Doutor em Engenharia Informática, pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Ciência.Inovação 2010

UNIÃO EUROPEIA
Fundo Social Europeu

LISBOA
2010

Acknowledgments

First I would like to thank Pedro Barahona, the supervisor of this work. Besides his valuable scientific advice, his general optimism kept me motivated throughout the long, and sometimes convoluted, years of my PhD. Additionally, he managed to find financial support for my participation on international conferences, workshops, and summer schools, and even for the extension of my grant for an extra year and an half. This was essential to keep me focused on the scientific work, and ultimately responsible for the existence of this dissertation.

Designing, developing, and maintaining a constraint solver requires a lot of effort. I have been lucky to count on the help and advice of colleagues contributing directly or indirectly to the design and implementation of CASPER, to whom I am truly indebted:

Francisco Azevedo introduced me to constraint solving over set variables, and was the first to point out the potential of incremental propagation. The set constraint solver integrating the work presented in this dissertation is based on his own solver CARDINAL, which he has thoroughly explained to me.

Jorge Cruz was very helpful for clarifying many issues about real valued arithmetic, and propagation of constraints over real valued variables in general. The adoption of CaSPER as an optional development platform for his lectures boosted the development of the module for real valued constraint solving.

Olivier Perriquet was directly involved in porting the PSICO library to CASPER, an application of constraint programming to solve protein folding problems.

Ruben Viegas developed a module for constraint programming over graph variables, which eventually led to a joint research on new representations for the domains of set domain variables. Working with Ruben was a pleasure and very rewarding. His work marks an important milestone in the development of CaSPER - the first contributed domain reasoning module.

Sérgio Silva has contributed a parser for the MINIZINC language that motivated an improved design of the general solver interface to other programming languages.

Several people have used the presented work for their own research or education. Many times they had to cope with bugs, lack of documentation, and an ever evolving architecture. Although this was naturally not easy for them, they were always very patient and helped improving my work in many ways. I'm very grateful to João Borges, Everardo Barcenas, Jean Christoph Jung, David Buezas, and the students of the "Search and Optimization" and "Constraints on Continuous Domains" courses.

I would like to thank to my colleagues at CENTRIA for all the informal discussions and

mostly for being a constant source of encouragement and motivation. I specially thank Armando Fernandes and Cecilia Gomes for their concern and advice on the time schedule and contents of this dissertation.

A significant part of the material presented in this dissertation is based on the formalism developed by Guido Tack. I am very grateful to him for his willingness to answer my questions on some aspects of his approach, and for his valuable comments on the work presented in chapters 6 and 7.

Finally I thank my family and friends for all the emotional support and for being a motivation in the quest for success. This dissertation is dedicated to them.

No idea is so antiquated that it was not once modern; no idea is so modern that it will not someday be antiquated.

Ellen Glasgow

Sumário

A programação por restrições é um modelo adequado à resolução de problemas combinatoriais com aplicação a problemas industriais e académicos importantes. Ela é realizada com recurso a um resolvedor de restrições, um programa de computador que tenta encontrar uma solução para o problema, i.e. uma atribuição de valores a todas as variáveis que satisfaça todas as restrições.

Esta dissertação descreve um conjunto de técnicas utilizadas na implementação de um resolvedor de restrições. Estas técnicas tornam um resolvedor de restrições mais extensível e eficiente, duas propriedades que dificilmente são integradas em geral, e em particular em resolvedores de restrições. Mais especificamente, esta dissertação debruça-se sobre dois problemas principais: propagação incremental genérica, e propagação de restrições decomponíveis arbitrárias. Para ambos os problemas apresentamos um conjunto de técnicas que são originais, correctas, e que se orientam no sentido de tornar a plataforma mais eficiente, extensível, ou ambos.

A matéria apresentada nesta dissertação surgiu da resolução dos problemas encontrados no desenho e implementação de um resolvedor de restrições genérico. O resolvedor CASPER (Constraint Solving Platform for Engineering and Research) não serve apenas de protótipo integrando todas as técnicas apresentadas, mas é também a plataforma experimental comum aos vários modelos teóricos discutidos. Para além do trabalho relacionado com o desenho e implementação de um resolvedor de restrições, esta dissertação apresenta também a primeira aplicação bem sucedida da plataforma na abordagem de um problema importante em aberto, nomeadamente a caracterização de heurísticas que direccionem a pesquisa rapidamente para uma solução.

Abstract

Constraint programming is a well known paradigm for addressing combinatorial problems which has enjoyed considerable success for solving many relevant industrial and academic problems. At the heart of constraint programming lies the constraint solver, a computer program which attempts to find a solution to the problem, i.e. an assignment of all the variables in the problem such that all the constraints are satisfied.

This dissertation describes a set of techniques to be used in the implementation of a constraint solver. These techniques aim at making a constraint solver more extensible and efficient, two properties which are hard to integrate in general, and in particular within a constraint solver. Specifically, this dissertation addresses two major problems: generic incremental propagation and propagation of arbitrary decomposable constraints. For both problems we present a set of techniques which are novel, correct, and directly concerned with extensibility and efficiency.

All the material in this dissertation emerged from our work in designing and implementing a generic constraint solver. The CASPER (Constraint Solving Platform for Engineering and Research) solver does not only act as a proof-of-concept for the presented techniques, but also served as the common test platform for the many discussed theoretical models. Besides the work related to the design and implementation of a constraint solver, this dissertation also presents the first successful application of the resulting platform for addressing an open research problem, namely finding good heuristics for efficiently directing search towards a solution.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1.

Introduction

1.1. Constraint reasoning

Constraint reasoning may be introduced with a simple example. Consider the following well known combinatorial object:

Definition 1.1 (Magic Square)**.** An order n magic square is a $n \times n$ matrix containing the numbers 1 to n^2, where each row, column, and main diagonal equal the same sum (the magic constant).

Magic squares were known to Chinese mathematicians as early as 650 BC. They were often regarded as objects with magical properties connected to diverse fields such as astronomy, mythology and music [Swaney 2000]. Figure 1.1 shows one of the earliest known squares, part of Albrecht Dürer's engraving *Melancholia*.

Problems involving magic squares range from completing an empty or partially filled magic square, or counting the number of magic squares with a given order or other mathematical properties. Solving these problems presents various interesting challenges: while filling an empty magic square of odd order may be accomplished in polynomial time, completing a partially filled square is NP-complete, and finding the exact number of squares with some dimension is #P-complete (see e.g. [Spitznagel 2010]).

The combinatorial structure inherent to this puzzle together with its simple declarative description makes it a good candidate for a constraint programming solving approach. Figure 1.2 shows a constraint program which finds a magic square where the numbers 15 and 14 (the date of the engraving) are already placed as in Dürer's original. The program embodies the following outstanding features of this technology:

Completeness Constraint programs provide completeness guarantees. This contrasts with other combinatorial solving methods, such as tabu search or genetic algorithms, which do not explore the solution space exhaustively. Consequently, these methods are not adequate for a number of problems, e.g. proving that there is no square having a given sequence of numbers, or counting the number of squares of a given order.

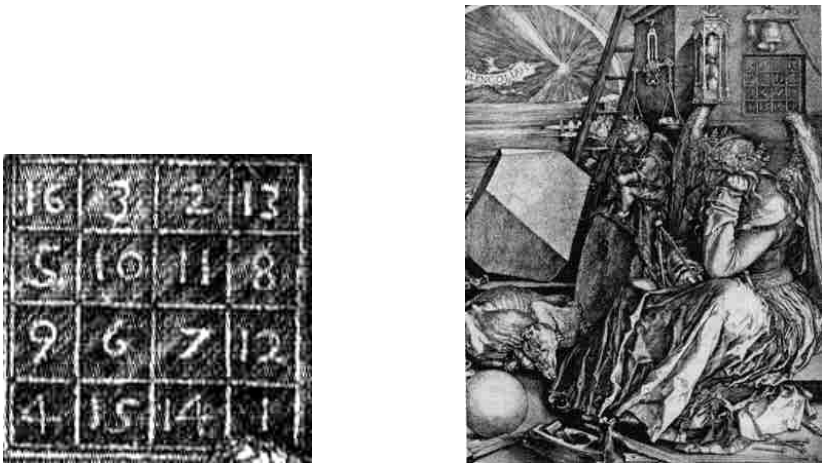


Figure 1.1.: Magic square found in Albrecht Dürer's engraving *Melancholia* (1514).

```
 1 void magic(Int n) {
 2   const Int k = n*(n*n+1)/2.0;
 3   Solver solver;
 4   DomVarArray<Int,2> square(solver,n,n,1,n*n);
 5   solver.post(distinct(square));
 6   MutVar<Int> i(solver);
 7   for (Int j = 0; j < n; j++) {
 8     solver.post(sum(all(i,range(1,n),square[i][j])) == k);
 9     solver.post(sum(all(i,range(1,n),square[j][i])) == k);
10   }
11   solver.post(sum(all(i,range(1,n),square[i][i])) == k);
12   solver.post(sum(all(i,range(1,n),square[i][n-i-1])) == k);
13   solver.post(square[3][1]==15 and square[3][2]==14);
14   if (solver.solve(label(square)))
15     cout << square << endl;
16 }
```

Figure 1.2.: Constraint program (in C++ using CaSPER) for finding a magic square.

Declarativeness Constraint programs are compact and highly declarative, promoting a clear separation between modeling the problem (the user's task: lines 2-13) and solving the problem (the solver's task: line 14). Moreover, imposing additional constraints to the problem can be done incrementally in a declarative fashion.

Efficiency Constraint programs are efficient. The above program solves the puzzle instantaneously, and when adapted for counting is able to enumerate all the 7040 possible magic squares of order 4 in a couple of seconds.

Constraint programs explicitly or implicitly make use of a constraint solver. The constraint program of fig. 1.2 references a set of objects which are responsible for the constraint solving process. In this case, the constraint solver is implemented as a C++ library. In contrast, some constraint programs are written in a language different from the language implementing the constraint solver, in which case a conversion phase is required. We will use the term constraint solver to denominate the set of algorithms and data structures that ultimately implement the constraint programming approach.

Constraint programming embraces a rich set of techniques and modeling protocols targeting general combinatorial problem solving. While our puzzle illustrates this class of problems, the prominence of constraint programming arises from its application to solve real world problems in diverse areas such as scheduling, planning, computer graphics, circuit design, language processing, database systems, and biology, among many others.

Related work

- The magic square, Latin square, sudoku, and other related problems mostly arise as recreational devices, although Latin squares, a special case of a multipermutation, have found application in cryptography [Laywine and Mullen 1998].
- The annual conference of the field, *Principles and Practice of Constraint Programming (CP)*, uncovers a myriad of applications of constraint programming to solve real world problems. The proceedings of its 2008's edition features, among others, examples of CP applied to planning and scheduling [Moura et al. 2008], packing [Simonis and O'Sullivan 2008], and biology [Dotu et al. 2008].
- A very complete essay on all aspects of constraint programming is [Rossi et al. 2006].

1.2. This dissertation

1.2.1. Motivation

The material in this dissertation emerged from the process of developing a general purpose constraint solver for use in a research environment. The main motivation that led us to con-

sider this endeavor was the lack of a constraint solver which was both competitive, extensible, open-source, and written in a popular, preferably object-oriented, programming language. We found these are necessary attributes for a constraint solver aiming to be a general constraint solving research platform.

Achieving the optimal balance between efficiency and extensibility is challenging for any large software project in general, and in particular for a constraint solver. Our hypothesis was that one does not necessarily sacrifices the other if the solver is based on a solid architecture, specifically designed with these concerns in mind.

1.2.2. Contributions

Committing ourselves to this project presented us with many interesting problems. Many of them have been solved by others, often in different ways, since practical constraint solvers have been around since the 80's. However, until very recently the constraint programming comunity has partially disregarded implementation aspects. The architecture and design decisions used in most of these solvers is thus many times not fully described, discussed, or justified. Therefore, for many problems we had to find our own solution given that the solutions found by others were either (a) not published, (b) jeopardized efficiency or extensibility, or (c) did not fit well with the rest of our architecture.

Presenting, explaining and evaluating all decisions behind the design of a constraint solver would be an enormous task, and perhaps not very interesting since many of these decisions condition each other. Instead, we have deliberately chosen to present what we believe are the most interesting and original ideas in our solver, hoping that they may be useful to others as well. Additionally, we also included our work on look-ahead search heuristics, which is the first application of our solver fulfilling its purpose: to be a research platform on constraint programming.

The four major contributions of this dissertation may thus be summarized as follows:

Techniques for incremental propagation We introduce a framework for integrating incremental propagation in a general purpose constraint solver. The contribution is twofold: a generalized propagation algorithm assisting domain agnostic incremental propagation, and its application for incremental propagation of finite set constraints, showing how the framework efficiently supports domain specific models of incremental propagation.

Efficient propagation of decomposable constraints We extend the theoretical model of Tack [2009] for the case of propagation of arbitrary decomposable constraints involving multiple variables. We prove that the generalized propagation model is correct, and provide an algorithm for approximating its completeness guarantees. We show how arbitrary decomposable constraints may be automatically compiled and efficiently propagated using this model for a special class of propagation algorithms.

Look-ahead heuristics We present a family of variable and value search heuristics based on look-ahead information, i.e. information collected while performing a limited amount of search and propagation. In particular, we describe how to integrate these heuristics with propagation algorithms achieving a specific form of consistency, namely singleton consistency, adding a negligible performance overhead to the global algorithm. We show that the resulting combination compares favorably with other popular heuristics in a number of standard benchmarks.

CaSPER We developed a new constraint solver implementing the techniques discussed in this dissertation. The solver was designed with efficiency, simplicity and extensibility as primary concerns, aiming to fulfill the need of a flexible platform for research on constraint programming. Its flexibility and competitive performance is attested throughout this dissertation, either when used for implementing and evaluating the specific techniques discussed, but also when compared globally with other state-of-the-art solvers.

Most of the material in this dissertation has appeared on the following publications, although with a different, less uniform presentation.

Marco Correia, Pedro Barahona, and Francisco Azevedo (2005). CaSPER: A Programming Environment for Development and Integration of Constraint Solvers. *Workshop on Constraint Programming Beyond Finite Integer Domains, BeyondFD'05* (proceedings).

Marco Correia and Pedro Barahona (2006). Overview of an Open Constraint Library. *ERCIM Workshop on Constraint Solving and Constraint Logic Programming, CSCLP'06* (proceedings), pp. 159–168.

Marco Correia and Pedro Barahona (2007). On the integration of singleton consistency and look-ahead heuristics. *Recent Advances in Constraints*, volume 3010 of *Lecture Notes in Artificial Intelligence*, pp 62–75. Springer.

Marco Correia and Pedro Barahona (2008). On the Efficiency of Impact Based Heuristics. *Principles and Practice of Constraint Programming, CP'08* (proceedings), volume 5202 of *Lecture Notes in Computer Science*, pp. 608–612. Springer.

Ruben Duarte Viegas, Marco Correia, Pedro Barahona, and Francisco Azevedo (2008). Using Indexed Finite Set Variables for Set Bounds Propagation. *Ibero-American Conference on Artificial Intelligence, IBERAMIA'08*, volume 5290 of *Lecture Notes in Artificial Intelligence*, pp. 73–82. Springer.

Marco Correia and Pedro Barahona (2009). Type parametric compilation of algebraic constraints. *Progress in Artificial Intelligence*, volume 5816 of *Lecture Notes in Artificial Intelligence*, pp. 201–212. Springer.

1.2.3. Overview

This dissertation is organized as follows.

Chapter 2 Presents the conceptual and operation models behind constraint solving and introduces the formalism used throughout this dissertation.

The first part covers incremental propagation, and is composed of chapters 3-5:

Chapter 3 Summarizes the major design decisions and techniques used in state-of-the-art constraint solvers, in particular its main constraint propagation algorithm, commonly used techniques for maintaining state, and other less frequently discussed, nevertheless important architectural elements. The chapter provides the necessary background for the first part of the dissertation.

Chapter 4 Describes two standard propagation models, namely variable and propagator centered and discusses a set of commonly used optimizations. Shows that, when compared to variable centered algorithms, the use of a propagator centered algorithm is advantageous in a number of aspects, including performance. Introduces a new generalized propagator centered model which brings to propagator centered models a feature originally unique to variable centered models - support for incremental propagation.

Chapter 5 Shows that incremental propagation can be more efficient than non-incremental propagation, in particular for constraints over set domains. Describes and compares two distinct models for maintaining the information required by incremental propagators for constraints over sets. Provides an efficient implementation of these models that takes advantage of the generic incremental propagation kernel introduced in the previous chapter.

The second part of the dissertation focuses on propagation of decomposable constraints, and is composed of chapters 6-9:

Chapter 6 Presents the formal model used for representing propagators over arbitrary decomposable constraints, which is used extensively throughout the second part of the dissertation. Extends the notation introduced in [Tack 2009] to accommodate constraints involving an arbitrary number of variables. Additionally, shows how sound and complete propagators may be obtained for this type of constraints.

Chapter 7 Considers the case of incomplete propagation of decomposable constraints by extending the material presented in the previous chapter and moving closer to a practical implementation. Formalizes sound and incomplete propagators, and presents an algorithm that provides an approximation to the problem of deciding the completeness of the propagators obtained from this model.

Chapter 8 Details a realization of the theoretical model introduced in the previous chapters for obtaining propagators with specific type of completeness. Shows how these propagators may be efficiently compiled for arbitrary decomposable constraints. Performs a theoretical comparison of the compilation and propagation algorithms with other algorithms for propagating arbitrary decomposable constraints, in particular the popular method based on the introduction of auxiliary variables and propagators.

Chapter 9 Describes the implementation of the compilation and propagation algorithms for decomposable constraints discussed in the previous chapter. Performs a set of experiments for evaluating the performance of such implementations.

The third part of the dissertation aims at evaluating the CaSPER solver as a general research platform and consists of chapters 10-11:

Chapter 10 Describes a set of search heuristics which explore look-ahead information. Shows how to efficiently integrate these heuristics with strong consistency propagation algorithms. Evaluates the performance of the solver in a number of benchmarks using these and other popular search heuristics.

Chapter 11 Compares the performance of CaSPER with other state-of-the-art constraint solvers that competed on the third international CSP solver competition.

Chapter 2.

Constraint Programming

This chapter will introduce several important concepts used in constraint programming, and provide an overview of the two major actors of the constraint solving process, namely constraint propagation and search. Simultaneously, it will present the notation and formal model used throughout this dissertation for describing many aspects of constraint solving.

2.1. Concepts and notation

2.1.1. Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is traditionally defined by a set of variables modeling the unknowns of the problem, a set of domains which define the possible values the variables may take, and a set of constraints that express the relations between variables. Before we formalize CSPs, let us detail these concepts.

Definition 2.1 (Assignment)**.** An assignment a is a mapping from variables X to values V. A *total assignment* maps every variable in X to some value, while a *partial assignment* involves only a subset of X. We represent an assignment using a set of expressions of the form $x \longmapsto v$, meaning that variable $x \in X$ takes value $v \in V$.

Definition 2.2 (Constraint)**.** A constraint c describes a set of (partial) assignments specifying the possible assignments to a set of variables in the problem. We may represent a constraint by extension by providing the full set of allowed assignments, or by intension in which case we will write the constraint expression in square brackets. A partial assignment a is consistent with some constraint c if a belongs to the set of assignments allowed by c.

Example 2.3 (Assignment,Constraint)**.** The assignment $a_1 = \{x_1 \longmapsto 1, x_2 \longmapsto 3\}$ assigns 1 to variable x_1 and 3 to variable x_2. The assignment $a_2 = \{x_1 \longmapsto 2, x_2 \longmapsto 3\}$ assigns 2 to variable x_1 and 3 to variable x_2. The constraint $c = \{\{x_1 \longmapsto 2, x_2 \longmapsto 1\}, \{x_1 \longmapsto 1\}\}$ may also be represented as $c = [(x_1 = 2 \wedge x_2 = 1) \vee x_1 = 1]$. The assignment a_1 is consistent with the constraint c, while the assignment a_2 is not.

In theory, variables and constraints are sufficient to model a CSP, since constraints may be used to specify all the domains of the variables in the problem. However, most textbook definitions of CSPs explicitly specify an initial set of values for the variables in the problem, which will be refered to as *variable domains*.

Definition 2.4 (Variable domain)**.** A variable domain $d_{\mathscr{D}} \subseteq \mathscr{D}$ represents the set of allowed values for some variable. Common variable domains are $d_{\mathbb{Z}}$ for integer variables, $d_{2^{\mathbb{Z}}}$ for integer set variables, and $d_{\mathbb{R}}$ for real-valued variables. When $\mathscr{D}$ is omitted we assume $d_{\mathbb{Z}}$.

We may now define constraint satisfaction problems.

Definition 2.5 (CSP)**.** A Constraint Satisfaction Problem is a triple $\langle X, D, C \rangle$ where X is a finite set of variables, D is a finite set of variable domains, and C is a finite set of constraints. We will denote by $D(x)$ the domain of some variable $x \in X$. Similarly, we will refer to the set of constraints involving some variable $x \in X$ as $C(x)$. The set of variables in some constraint $c \in C$ may be selected with $X(c)$.

The task of solving a CSP consists of finding a solution, i.e. one total assignment which is consistent with all constraints in the problem, or proving that no such assignment exists.

Example 2.6 (Magic square as a CSP)**.** We can easily formalize the problem of filling a n-order magic square, introduced in the previous section. The CSP consists of n^2 integer variables, one for each cell. Each variable $x_{i,j} \in X$ represents the unknown figure corresponding to the cell at position (i, j) in the square, where $1 \leq i \leq n$, $1 \leq j \leq n$, and have the initial domain $D(x_{i,j}) = \{1, \ldots, n^2\}$. This corresponds to line 4 in the program of figure 1.2 on page 2. Let $k = n(n^2+1)/2$ denote the magic constant. The CSP's constraint set C is composed of the following constraints:

$$\bigwedge_{1 \leq i,j,k,l \leq n : i \neq k \vee j \neq l} [x_{i,j} \neq x_{k,l}] \qquad \text{all cells take distinct values (line 5)}$$

$$\bigwedge_{1 \leq i \leq n} \sum_{1 \leq j \leq n} [x_{i,j} = k] \qquad \text{the sum of cells in the same row equals } k \text{ (line 8)}$$

$$\bigwedge_{1 \leq i \leq n} \sum_{1 \leq j \leq n} [x_{j,i} = k] \qquad \text{the sum of cells in the same column equals } k \text{ (line 9)}$$

$$\begin{matrix} \sum_{1 \leq i \leq n} [x_{i,i} = k] \\ \sum_{1 \leq i \leq n} [x_{i,n-i} = k] \end{matrix} \qquad \text{the sum of cells in the main diagonals equals } k \text{ (lines 11,12)}$$

2.1.2. Tuples and tuple sets

Tuples and sets of tuples are central concepts in constraint programming and in this dissertation in particular. They are used to model constraints, and will form the basis for defining domains (not to be confused with variable domains described earlier). We will use the following notation when referring to tuples and tuple sets.

Definition 2.7 (Tuple)**.** An n-tuple is a sequence of n elements, denoted by angle brackets. We make no restriction on the type of elements in a tuple, but tuples of integers will be most often used. Tuples will be refered by using bold lowercase letters, optionally denoting the number of elements in superscript.

Definition 2.8 (Element projection)**.** We will write t_j to refer to the j-th element of tuple $\mathbf{t}$, in which case we consider tuples as 1-based arrays. Similarly, we extend the notation to allow multiple selection, writing $t_J = \langle t_j \rangle_{j \in J}$ to specify the tuple of elements at positions given by set J.

Example 2.9 (Tuple, Element projection)**.** Considering $\mathbf{t}^3 = \langle 2,3,1 \rangle$, a 3-tuple, we have $t_2 = \langle 3 \rangle$ (or simply $t_2 = 3$), and $t_{\{1,3\}} = \langle 2,1 \rangle$.

Definition 2.10 (Tuple set)**.** A tuple set $S^n \subseteq \mathbb{Z}^n$ is a set of n-tuples, also referred to as table. When needed, we will refer to the size of the tuple set, the number of tuples, as $|S^n|$.

Definition 2.11 (Tuple projection)**.** Let $\text{proj}_j(S^n) = \{t_j : \mathbf{t} \in S^n\}$ the projection operator. We generalize the operator for projections over a set of indexes, $\text{proj}_J(S^n) = \{t_J : \mathbf{t} \in S^n\}$.

Example 2.12 (Tuple set, Tuple projection)**.** $S^3 = \{\langle 1,2,3 \rangle, \langle 3,1,2 \rangle\}$ is a 3-tuple set, with size $|S^3| = 2$. Then $\text{proj}_2(S^3) = \{\langle 2 \rangle, \langle 1 \rangle\}$ and $\text{proj}_{\{2,3\}}(S^3) = \{\langle 2,3 \rangle, \langle 1,2 \rangle\}$.

Definition 2.13 (Assignments as tuples)**.** Throughout this dissertation we will implicitely use tuples to represent assignments. A given assignment $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ may be represented by an n-tuple $\mathbf{t}^n = \langle v_1, \ldots, v_n \rangle$. If an assignment $\mathbf{t}^i$ is a partial assignment, i.e. covers only a subset X' of X, then the size i of the tuple equals the number of variables in X', that is $i = |X'|$, in which case the set X' will be made explicit.

Definition 2.14 (Constraints as tuple sets)**.** We may also represent constraints as tuple sets. The notation $\text{con}(c)$ specifies the set of tuples corresponding to the partial assignments allowed by the constraint. It is assumed that the set of partial assignments allowed by the constraint affects the same set of variables. This does not restrict the expressiveness of the notation since any partial assignment may be extended to cover more variables by taking the Cartesian product of the allowed values in the domains of the remaining variables.

Example 2.15. Let $D(x_1) = \{1,2\}$, and $D(x_2) = \{1,2,3\}$. The assignment $a = \{x_1 \longmapsto 1, x_2 \longmapsto 3\}$ may be represented as $\mathbf{a} = \langle 1,3 \rangle$. Let c be a constraint defined as $c = \{\{x_1 \longmapsto 2, x_2 \longmapsto 1\}, \{x_1 \longmapsto 1\}\}$. Then $\text{con}(c) = \{\langle 2,1 \rangle, \langle 1,1 \rangle, \langle 1,2 \rangle, \langle 1,3 \rangle\}$. The fact that the assignment a is consistent with the constraint c is equivalent to the expression $a \in \text{con}(c)$.

2.1.3. Domain approximations

Before we define domains, let us introduce two important tuple set operators, which will be referred to as *domain approximations*.

Definition 2.16 (Cartesian approximation)**.** The Cartesian approximation $[\![S^n]\!]^\delta$ of a tuple set $S^n \subseteq \mathbb{Z}^n$ is the smallest Cartesian product which contains S^n, that is:

$$[\![S^n]\!]^\delta = \text{proj}_1\left(S^n\right) \times \ldots \times \text{proj}_n\left(S^n\right)$$

Definition 2.17 (Box approximation)**.** Given an ordered set $S \subseteq \mathscr{D}$, let $\text{conv}(S)$ be the convex set of S, i.e.

$$\text{conv}_{\mathscr{D}}(S) \quad = \quad \{z \in \mathscr{D} : \min(S) \leq z \leq \max(S)\}$$

The box approximation $[\![S^n]\!]^{\beta(\mathscr{D})}$ of a tuple set $S^n \subseteq \mathscr{D}^n$ is the smallest n-dimensional box containing S^n, that is:

$$[\![S^n]\!]^{\beta(\mathscr{D})} = \text{conv}_{\mathscr{D}}\left(\text{proj}_1\left(S^n\right)\right) \times \ldots \times \text{conv}_{\mathscr{D}}\left(\text{proj}_n\left(S^n\right)\right)$$

We will be referring to integer and real box approximations exclusively, i.e. $[\![\cdot]\!]^{\beta(\mathbb{Z})}$ or $[\![\cdot]\!]^{\beta(\mathbb{R})}$. Whenever $\mathscr{D}$ is omitted it will be assumed that $\mathscr{D} = \mathbb{Z}$.

We introduce one more operator, the identity operator, which will be used mostly for simplifying notation:

Definition 2.18 (Identity approximation)**.** The identity operator $[\![\cdot]\!]^\varphi$ transforms a tuple set in itself, i.e. $[\![S^n]\!]^\varphi = S^n$.

Let $S^n, S_1^n, S_2^n \subseteq \mathbb{Z}^n$ be arbitrary n-tuple sets and $\Phi \in \{\varphi, \delta, \beta\}$. We note the following properties of these operators:

Property 2.19 (Idempotence)**.** The $[\![\cdot]\!]^\Phi$ operator is idempotent, i.e. $[\![S^n]\!]^\Phi = [\![[\![S^n]\!]^\Phi]\!]^\Phi$.

Property 2.20 (Monotonicity)**.** The $[\![\cdot]\!]^\Phi$ operator is monotonic, i.e. $S_1^n \subseteq S_2^n \implies [\![S_1^n]\!]^\Phi \subseteq [\![S_2^n]\!]^\Phi$.

Property 2.21. The $[\![\cdot]\!]^\Phi$ operator is closed under intersection, i.e. $[\![S_1^n]\!]^\Phi \cap [\![S_2^n]\!]^\Phi = [\![[\![S_1^n]\!]^\Phi \cap [\![S_2^n]\!]^\Phi]\!]^\Phi$.

These domain approximation operators will be used extensively to specify particular types of tuple sets called *domains*.

2.1.4. Domains

Definition 2.22 (Φ-domain)**.** Let S^n be an arbitrary n-tuple set and $\Phi \in \{\varphi, \delta, \beta\}$. We call S^n a Φ-domain if and only if $S^n = [\![S^n]\!]^\Phi$.

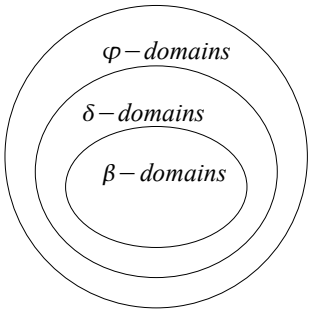


Figure 2.1.: Domain taxonomy

Domains will be used for multiple purposes. First we note that, by definition, any tuple set is a φ-domain. For any CSP $\langle X, D, C\rangle$ we can also observe the following: The set D of variable domains is a δ-domain capturing the initial set of variable assignments. For any constraint $c \in C$, $\mathrm{con}(c)$ is a domain specifying the possible assignments to $X(c)$, and the conjunction of all constraints $\mathrm{con}(\bigwedge_{c\in C} c)$ in the problem is also a domain specifying the solutions to the problem.

The following concepts define a partial order on domains.

Definition 2.23. A domain $S_1^n \subseteq \mathbb{Z}^n$ is *stronger* than a domain $S_2^n \subseteq \mathbb{Z}^n$ (or equivalently S_2^n is *weaker* than S_1^n), if and only if $S_1^n \subseteq S_2^n$. S_1^n is *strictly stronger* than S_2^n (or equivalently S_2^n is *strictly weaker* than S_1^n) if and only if $S_1^n \subset S_2^n$.

The following lemma shows how the previously defined approximations are ordered for a given tuple set.

Lemma 2.24. *Let $S^n \subseteq \mathbb{Z}^n$ be an arbitrary tuple set. Then,*

$$S^n = \llbracket S^n \rrbracket^{\varphi} \subseteq \llbracket S^n \rrbracket^{\delta} \subseteq \llbracket S^n \rrbracket^{\beta}$$

Note that this contrasts with the relation between all possible Φ-domains with $\Phi \in \{\varphi, \delta, \beta\}$, as depicted in figure 2.1.

Related work

- The classic view of CSP's was initially developed by Montanari [1974] and Mackworth [1977a]. The formalization described above extends their work by accommodating intensional constraints (initially constraints were exclusively given by extension), and broadening the notion of domain to include not only the traditional Cartesian domains, but also box domains and any arbitrary tuple set.

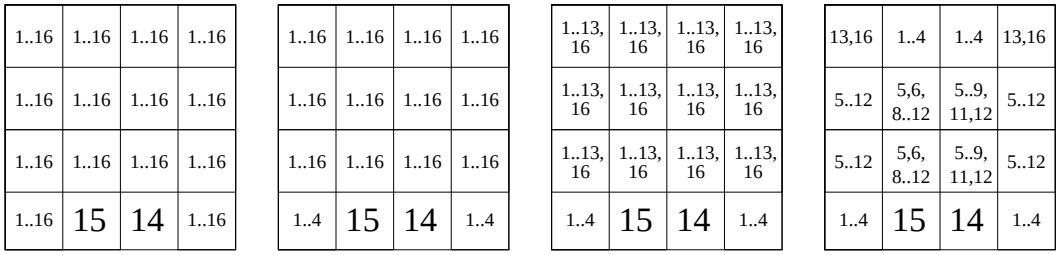


Figure 2.2.: Partially filled magic square of order 4: without any filtering (a); where some inconsistent values were filtered (b,c); with no inconsistent values (d).

- The Cartesian approximation was defined in [Ball et al. 2003], while box approximation is defined in [Benhamou 1995]. Domain approximations in the context of constraint programming were further refined by Benhamou [1996], and Maher [2002], and reconciled with the classical notions of consistency in [Tack 2009].

2.2. Operational model

Solving constraint satisfaction problems involves two main ingredients: propagation and search. Propagation infers sets of assignments which are not solutions to the problem and excludes them from the current domain. Search finds assignments which are possible within the current domain. The solving process consists of interleaving the execution of these two procedures, exploring the problem search space exhaustively until a solution is found.

2.2.1. Propagation

Let us revisit the magic square problem introduced earlier:

Example 2.25 (Magic Square filtering)**.** Imagine Dürer's task of finding a magic square with the figures 14 and 15 already placed. He probably began by writing an empty square, where all values are possible in all cells except in those which are already assigned (similar to the square of fig. 2.2 a). Soon he must have realized that some values were impossible in some cells, namely in the two inferior corners, which must add to 5 (fig. 2.2 b), and in all remaining cells, which cannot take 14 nor 15 since they are already placed (fig. 2.2 c).

The inference process just described is called *constraint propagation*, or *constraint filtering*. By removing figures that cannot be part of a magic square, Dürer was discarding an exponential number of inconsistent assignments. With some time and patience, he could have gone

further and removed all inconsistent values from the initial problem, ending with the partial magic square shown in fig. 2.2 (d).

The amount of filtering performed on a CSP is related to a property known as *consistency*. Before we give the classical definition of consistency regarding CSPs, let us first focus on consistencies of an arbitrary δ-domain with respect to a single constraint, which basically tell us how well the domain approximates the constraint.

Definition 2.26 (Domain consistency)**.** A δ-domain S^n is domain consistent for a constraint $c \in C$ if and only if $S^n \subseteq \llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^\delta \rrbracket^\delta$.

Definition 2.27 (Bounds($\mathbb{Z}$) consistency)**.** A δ-domain S^n is bounds($\mathbb{Z}$) consistent for a constraint $c \in C$ if and only if $S^n \subseteq \llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^\beta \rrbracket^\beta$.

Definition 2.28 (Bounds($\mathbb{R}$) consistency)**.** A δ-domain S^n is bounds($\mathbb{R}$) consistent for a constraint $c \in C$ if and only if $S^n \subseteq \llbracket \mathrm{con}(c_\mathbb{R}) \cap \llbracket S^n \rrbracket^{\beta(\mathbb{R})} \rrbracket^{\beta(\mathbb{R})}$.

Definition 2.29 (Bounds(D) consistency)**.** A δ-domain S^n is bounds(D) consistent for a constraint $c \in C$ if and only if $S^n \subseteq \llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^\delta \rrbracket^\beta$.

Definition 2.30 (Range consistency)**.** A δ-domain S^n is range consistent for a constraint $c \in C$ if and only if $S^n \subseteq \llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^\beta \rrbracket^\delta$.

The above concepts define different consistencies by requiring that all members of the domain lie within some neighborhood of the constraint. Intuitively, for a given consistency the inner approximation operator $\llbracket \cdot \rrbracket$ tell us which solutions to the constraint are taken into consideration, while the outer $\llbracket \cdot \rrbracket$ defines the set of non-solutions which are acceptable.

Example 2.31 (Consistency)**.** Consider the sum constraint c (corresponding to the bottom row of the magic square in fig. 2.2), and the tuple set $S^4 = \{1,2,4\} \times \{15\} \times \{14\} \times \{1,2,3,4\}$. The set of solutions that domain and bounds(D) consistency must approximate is $\mathrm{con}(c) \cap \llbracket S^n \rrbracket^\delta = \{\langle 1,15,14,4\rangle, \langle 2,15,14,3\rangle, \langle 4,15,14,1\rangle\}$. Since $\llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^\delta \rrbracket^\delta = \{1,2,4\} \times \{15\} \times \{14\} \times \{1,3,4\} \subset S^4$, then S^4 is not domain consistent to constraint c. Similarly, since $\llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^\delta \rrbracket^\beta = \{1,2,3,4\} \times \{15\} \times \{14\} \times \{1,2,3,4\} \supset S^4$, then S^4 is bounds(D) consistent to constraint c.

The notion of consistency applies naturally to a given set of constraints and domain of a CSP, in which case it may be referred to as *local consistency* or *global consistency* of the constraint network as explained below.

Definition 2.32 (Local consistency)**.** A CSP $\langle X, D, C\rangle$ is locally domain consistent (respectively locally bounds($\mathbb{Z}$), bounds($\mathbb{R}$), bounds(D), or range consistent), if and only if D is domain consistent (respectively bounds($\mathbb{Z}$), bounds($\mathbb{R}$), bounds(D), or range consistent) for every constraint $c \in C$.

Definition 2.33 (Global consistency)**.** A CSP $\langle X, D, C\rangle$ is globally domain consistent (respectively globally bounds($\mathbb{Z}$), bounds($\mathbb{R}$), bounds(D), or range consistent), if and only if D is domain consistent (respectively bounds($\mathbb{Z}$), bounds($\mathbb{R}$), bounds(D), or range consistent) for the constraint $\bigwedge_{c\in C} c$.

Example 2.34 (Consistency of a CSP)**.** The CSP corresponding to the magic square of figure 2.2 (b) is locally bounds($\mathbb{Z}$) and bounds(D) consistent, but not domain nor range consistent. The square shown in (c) is locally domain consistent (and consequently also locally bounds($\mathbb{Z}$), bounds($\mathbb{R}$), bounds(D), and range consistent). The square shown in (d) is globally domain consistent.

The computational cost of achieving local and global consistency on a given CSP depends on the constraint network structure, on the semantics of the constraints involved, and on the size of the domains. Achieving global consistency is usually intractable except for CSPs with a very specific network structure, but polynomial time algorithms exist that achieve local consistency for a number of important constraints. Perhaps because it is easier to reason with independent constraints rather than with their conjunction, constraint propagation has been traditionally implemented in modular, independent components called propagators, which achieve some form of local consistency on specific constraints.

Definition 2.35 (Propagator)**.** A propagator (or filter) implementing a constraint $c \in C$ is a function $\pi_c : \wp(\mathbb{Z}^n) \to \wp(\mathbb{Z}^n)$ which is contracting, i.e. $\pi_c(S^n) \subseteq S^n$ for any tuple set $S^n \subseteq \mathbb{Z}^n$. A propagator is sound if and only if it never removes tuples which are allowed by the associated constraint, i.e. $\mathrm{con}(c) \cap S^n \subseteq \pi_c(S^n)$ for any tuple set $S^n \subseteq \mathbb{Z}^n$.

Traditionally, propagators were also required to be monotonic, i.e. $\pi_c\left(S_1^n\right) \subseteq \pi_c\left(S_2^n\right)$ if $S_1^n \subseteq S_2^n$, and idempotent, i.e. $\pi_c(\pi_c(S^n)) = \pi_c(S^n)$, however these additional restrictions are not mandatory in modern constraint solvers, as shown in Tack [2009]. Throughout this dissertation we will consider propagators to be monotonic and will not make any assumption about its idempotency unless explicitely stated. Moreover, we will use the following notation concerning idempotency.

Definition 2.36 (Idempotent propagator)**.** Let π_c be a propagator for a constraint $c \in C$. Let $\pi_c^\star$ represent the iterated function $\pi_c^\star = \pi_c \circ \ldots \circ \pi_c$ such that $\pi_c\left(\pi_c^\star(x)\right) = \pi_c^\star(x)$. Propagator π_c is an idempotent propagator for S^n if and only if $\pi_c(S^n) = \pi_c^\star(S^n)$. In such case we also say that $\pi_c(S^n)$ is a fixpoint for π_c, or equivalently that π_c is at fixpoint for $\pi_c(S^n)$. Propagator π_c is an idempotent propagator if and only if $\pi_c(S^n)$ is a fixpoint for π_c for any $S^n \subseteq \mathbb{Z}^n$. Finally, we remark that we always have $\pi_c(S^n) \subset S^n$ unless S^n is a fixpoint for π_c. This is a consequence of π_c being deterministic, and is independent of the idempotency of π_c.

The contracting condition alone sets a very loose upper bound on the output of a propagator. Many functions meet these requirements without performing any useful filtering, as

for example the identity function. Useful propagators are complete with respect to some domain, which translates in achieving some consistency on the constraint associated with the propagator. According with our previous definitions of consistency, we now enumerate the corresponding completeness guarantees provided by propagators.

Definition 2.37 (Domain completeness)**.** A propagator π_c implementing constraint $c \in C$ is domain complete if and only if $\pi_c^\star(S^n)$ is domain consistent for the constraint c, for any δ-domain S^n. In such case we say the propagator achieves domain consistency for the constraint c.

Definition 2.38 (Bounds completeness)**.** A propagator π_c implementing constraint $c \in C$ is bounds($\mathbb{Z}$) (respectively bounds($\mathbb{R}$), bounds(D), or range) complete if and only if $\pi_c^\star(S^n)$ is bounds($\mathbb{Z}$) (respectively bounds($\mathbb{R}$), bounds(D), or range) consistent for the constraint c, for any δ-domain S^n.

Example 2.39 (Propagator completeness taxonomy)**.** Domain completeness and the different types of bounds completeness described above are ordered as shown in figure 2.3. Each arrow reflects a *strictly stronger than* relation between two completeness classes.

Additionally, we labeled arrows with a tuple set S for which the propagator at the start of the arrow is able to prune more values (marked as striked out) than the propagator at the end of the arrow. For this, we considered a propagator π_c for the constraint $c = \left[2x + 3y = z\right]$ and a tuple set $S = S_x \times S_y \times S_z$. For example, when applied to a domain $S = \{\langle 0,1\rangle, \langle 0\rangle, \langle 0,1,2\rangle\}$, a domain complete propagator is able to prune tuples $\langle 0,0,1\rangle$ and $\langle 1,0,1\rangle$ whereas a bounds(D) cannot. Note that bounds(D) complete and range complete propagators are incomparable.

Related work

- Historically, the notion of consistency was associated with CSPs involving extensional constraints. Mackworth [1977a] presented an algorithm for achieving domain consistency on CSPs involving binary extensional constraints, and later generalized for non-binary constraints [Mackworth 1977b]. The algorithm was referred to as arc consistency in the case of binary relations, and generalized arc consistency for n-ary relations. Although the term is still commonly used, we will use domain consistency throughout this dissertation. The presented bounds consistency definitions are described in more detail in chapter 3 of [Rossi et al. 2006]. The formalization of domain and bounds consistency and completeness using domain approximations is due to Tack [2009], which is in turn based on [Maher 2002] and [Benhamou 1996].

- The definition of propagator given above (def. 2.35 on the preceding page) considers arbitrary tuple sets while traditionally propagators are defined over Cartesian products. This

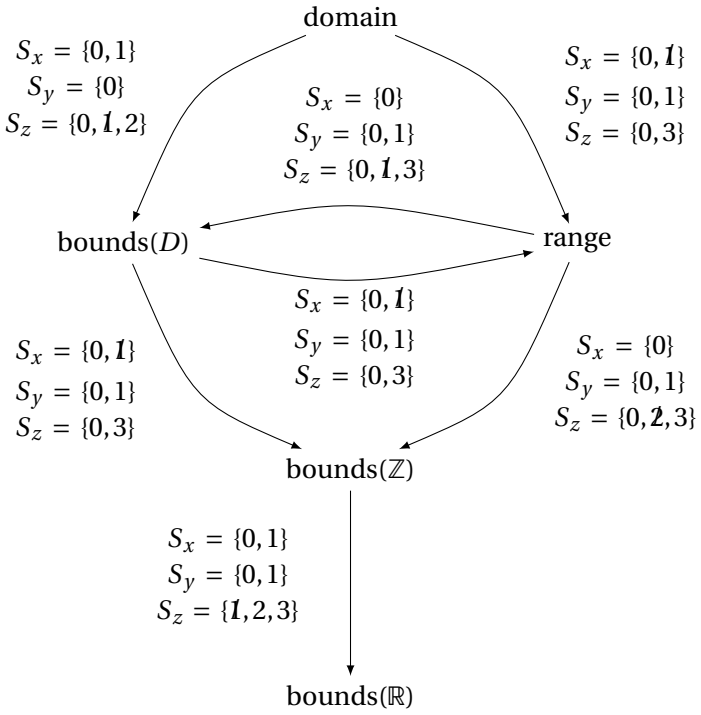


Figure 2.3.: Taxonomy of constraint propagation strength. Each arrow specifies a *strictly stronger than* relation between two consistencies (see example 2.39).

Function `GenerateAndTest(`d, C`)`

Input: A domain d and a set of constraints C
Output: A subset $S \subseteq d$ satisfying all constraints in C, i.e. $S \subseteq \text{con}(c) : \forall c \in C$

```
1 if d = ∅ then
2 |   return ∅
3 if |d| = 1 ∧ ∀c ∈ C, d ⊆ con(c) then
4 |   return {d}
5 ⟨d1, d2⟩ ← Branch(d)
6 return GenerateAndTest(d1,C) ∪ GenerateAndTest(d2,C)
```

generalization is intentional, since it will be used in part II. However, we kept the classical notions of consistency and propagation completeness associated to Cartesian products. Applying the same generalization there would make comparing to existing propagation algorithms more complex.

- A line of research in constraint programming is devoted to identifying tractable constraint network structures, where global consistency may be achieved in polynomial time. See for example chapter 7 of [Rossi et al. 2006]. For a complete characterization of tractable CSPs for 2-element and 3-element domains see [Schaefer 1978; Bulatov 2006].

- A number of polynomial algorithms have been identified that achieve domain or bounds consistency in a number of useful constraints. A notable example is the constraint that enforces all variables to take distinct values (used in the magic square example) for which an algorithm exists that achieves domain consistency in time $O\left(n^{2.5}\right)$ by Régin [1994], another achieving bounds($\mathbb{Z}$) on time $O\left(n \log n\right)$ by Puget [1998], and a range complete propagator which runs in time $O\left(n^2\right)$ by Leconte [1996]. Algorithms achieving bounds(D) consistency are rarely found in practice.

- Consistencies stronger than domain consistency have been proposed, namely path consistency [Montanari 1974], and k-consistency [Freuder 1978]. These consistencies approximate constraints using domains stronger than δ-domains, whose representation requires exponential space, and are disregarded in most constraint solvers.

2.2.2. Search

Generate and test is a brute force method that generates all possible combinations of values and then selects those that satisfy all constraints in the problem. The method is easily implemented using recursion (function `GenerateAndTest`).

Line 1 tests for the case where an empty domain is passed to the function, which trivially has no solutions. Line 3 tests if the current domain is singleton, i.e. all variables are instantiated,

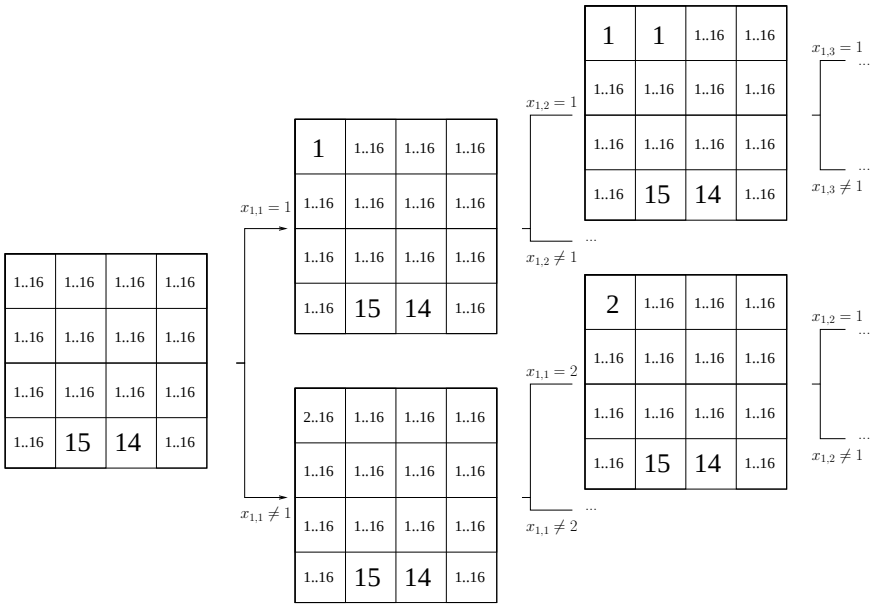


Figure 2.4.: Partial search tree obtained by `GenerateAndTest` on the magic square problem.

and satisfy all constraints in the problem. Otherwise, line 5 calls a function `Branch` that splits the domain d in two smaller domains d_1, d_2, such that $d = d_1 \cup d_2$ and $d_1 \cap d_2 = \emptyset$, and line 6 calls the function recursively on each of these domains.

Figure 2.4 shows a trace, or *search tree*, of the algorithm applied to the magic square problem. The leftmost square represents the initial domain, and the direct children of each square are the domains resulting from the `Branch` function.

The `GenerateAndTest` algorithm may generate an exponential number of domains that do not contain any solution. An example is the topmost square in fig. 2.4 that corresponds to a non-singleton domain which does not contain any solutions since the same value is used in the first two cells of the first row. An improved version of this algorithm is given by function `Solve`.

The difference is the inclusion of a call to the `Propagate` function, which filters inconsistent values from the input domain, as discussed in the previous section. The search tree of the `Solve` algorithm now depends on the consistency achieved by constraint propagation. For example, if `Propagate` enforces local domain consistency on this problem, we obtain the partial search tree shown in figure 2.5, where each square corresponds to the domain obtained after propagation (line 1). Note that all the upper children which cannot lead to any solution are immediately rejected, hence the algorithm avoids exploring an exponential number of nodes.

The `Solve` algorithm is complete for any CSP where all variables have finite domains - given

Function `Solve`(d, C)

Input: A domain d and a set of constraints C
Output: A subset $S \subseteq d$ satisfying all constraints in C, i.e. $S \subseteq \text{con}(c) : \forall c \in C$

```
1 d ← Propagate(d, C)
2 if d = ∅ then
3 |   return ∅
4 if |d| = 1 ∧ ∀c ∈ C, d ⊆ con(c) then
5 |   return {d}
6 ⟨d1, d2⟩ ← Branch(d)
7 return Solve(d1,C) ∪ Solve(d2,C)
```

enough time it will find a solution or prove that no solution exists, which is in fact one of the most appealing aspects of the constraint programming approach. Additionally, it is also space efficient since only one child of each node must be in memory at all times, and the depth of the search tree is bounded by the number of variables and values in their domains. Unfortunately, the time efficiency of this algorithm greatly depends on the way the search tree is explored, i.e. on the order in which the child nodes of a given node are visited (line 7). A number of heuristics have been proposed that try to direct search towards the solution, with some success for many combinatorial problems. However, the algorithm will, in the worst case, explore the full search tree before finding the solution or proving that it does not exist.

Although other complete search algorithms exist, they are essentially variations of the `Solve` algorithm. Therefore, in this dissertation we will assume that search is performed using this algorithm.

Related work

- The `Solve` algorithm is also known as *backtracking*. Other popular complete algorithms that may be used with constraint propagation are *backmarking*, *forward checking*, *backjumping*, among others [Dechter 2003].

- Most search heuristics are correlated with either a first-fail or a best-promise policy [Haralick and Elliott 1980]. First-fail consists in trying first the domains which are more likely to contain no solutions. While it seems contradictory, this will make the search algoritm visit smaller search paths first, which increases the number of search paths visited for a given amount of time. Best-promise does the opposite, it visits first the domains that are more likely to contain solutions. First-fail heuristics are usually connected with the selection of the next variable to enumerate: the variable with smaller domain, more constraints attached [Bessière and Régin 1996], more constraints to instantiated variables, and variations [Brélaz 1979; Boussemart et al. 2004]. Best-promise is usually connected to the choice

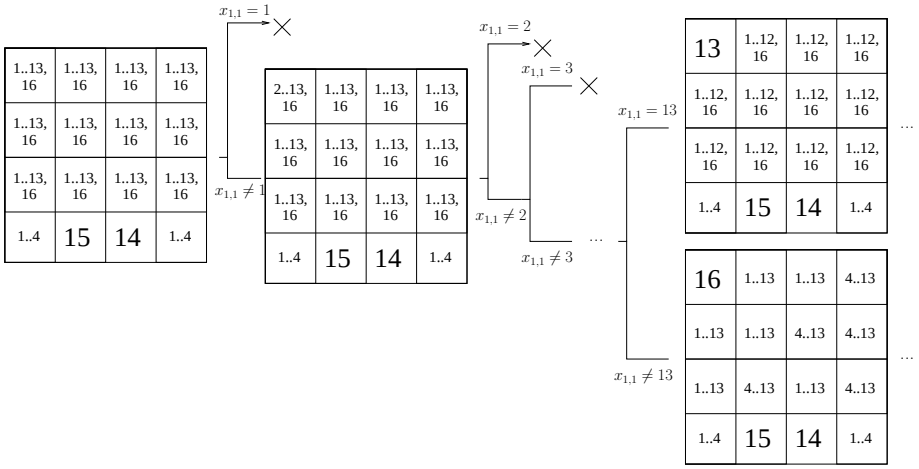


Figure 2.5.: Partial search tree obtained by SOLVE on the magic square problem while maintaining local domain consistency.

of value to enumerate, and is often obtained by integrating some knowledge about the structure of the problem [Geelen 1992].

2.3. Summary

This chapter presented the necessary notation for formalizing constraint satisfaction problems and the constraint solving process. We have seen that constraint propagation operates on tuple sets, and how their approximations may be used to characterize the completeness of propagation algorithms. Finally, we introduced the main algorithm of a constraint solver that integrates constraint propagation with a general brute-force search algorithm.

Part I.

Incremental Propagation

Chapter 3.

Architecture of a Constraint Solver

This chapter will describe a number of important techniques and decisions behind the design and implementation of a modern constraint solver. It will focus on two main components: a propagation kernel implementing generalized constraint propagation (§3.1), and the state manager, responsible for maintaining the state of all elements of a constraint solver synchronized with the search procedure (§3.2). Additionally, this chapter will present other important structural resources of constraint solvers, such as the support for domain specific reasoning, global constraints, and language interfaces (§3.3). Whenever possible, we will provide pointers to state-of-the-art constraint solvers which advertise the use of a particular method. In this same spirit, this chapter will also perform a global overview of CaSPER, the constraint solver used for implementing and evaluating all the techniques described in this dissertation.

3.1. Propagation kernel

The propagation kernel forms the core of a constraint solver. Because propagation takes the largest share of time spent on solving a given problem, designing a propagation kernel involves decisions that affect the global performance of a constraint solver.

3.1.1. Propagation loop

Most constraint solvers do not maintain a specific type of consistency in particular, but instead achieve an hybrid form of local consistency. They associate one or more propagators with specific strengths with each constraint of the problem, and combine their propagation by repeatedly executing propagators until a fixpoint is achieved. The existence of a fixpoint, hence the termination of the algorithm, is guaranteed since propagators are contracting and domains are finite, as we have observed in the previous chapter. A very simple implementation of this algorithm is given by function `Propagate1`.

This algorithm and its variants will be thoroughly revised in the next chapter.

Function `Propagate1(`d, C`)`

Input: A domain d and a set of constraints C
Output: A subset of d consistent with C

```
1  while ¬ fixpoint do
2      fixpoint ← true
3      foreach c ∈ C do
4          t ← d
5          d ← π_c(d)
6          if d ⊂ t then
7              fixpoint ← false
8
9
10 return d
```

3.1.2. Subscribing propagators

Most constraints typically involve a small subset of variables in the problem. For a given constraint $c \in C$, its associated propagator π_c reasons exclusively with $X(c)$ instead of the full set of variables X. Consequently, changes to the domains of the variables in $X \setminus X(c)$ are always ignored by π_c - the propagator π_c needs to be executed only when the domains of variables in $X(c)$ are updated. Algorithm `Propagate1` does not take this observation into account, and simply executes all propagators associated with all constraints in the problem, and that is, of course, very inefficient.

Example 3.1. Consider a CSP with variables $x_1, \ldots, x_n$, domains $D(x_i) = \{1, \ldots, b\}$, $1 \leq i \leq n$, and a set of constraints $C = c_1, \ldots, c_n$ where $x_1, x_2 \notin X(c_i)$, $1 \leq i \leq n-2$, and $c_{n-1} = [x_1 > x_2]$, $c_n = [x_1 < x_2]$. Note that this CSP does not have any solutions. Calling `Propagate1` on this CSP detects inconsistency after $O(nb)$ propagator executions (line 5), but only $O(2b)$ propagator executions are effective, i.e. not idempotent.

To mitigate this problem, constraint solvers maintain a queue of pending propagators, that is, a set of propagators that can effectively prune the current domain. This is implemented by subscribing propagators to their relevant variables, so that updating a variable's domain can trigger the addition of all subscribed propagators to the pending queue, or in other words, *schedule* the propagators for execution. Let us formalize the set of propagators subscribed to variable x:

Definition 3.2. For a given variable $x \in X$ let $\textsc{props}(x) = \bigcup_{c \in C(x)} \pi_c$.

Selecting which propagators must be scheduled requires a mechanism for identifying the set of variables whose domain changes from a given fixpoint to another.

Function `Propagate2(`d, P`)`

Input: A domain d and a set of propagators P implementing constraints C
Output: A subset of d consistent with the constraints in C

1 **while** $P \neq \emptyset$ **do**
2 $\quad \pi_c \leftarrow$ `SelectPropagator(`P`)`
3 $\quad P \leftarrow P \setminus \{\pi_c\}$
4 $\quad d' \leftarrow \pi_c(d)$
5 $\quad P \leftarrow P \cup \bigcup_{x \in \text{VARS}(d,d')} \text{PROPS}(x)$
6 $\quad d \leftarrow d'$
7 **return** d

Definition 3.3 (Pruned variables)**.** Let d_1^n, d_2^n, where $d_2^n \subseteq d_1^n$ denote two different domains of a given CSP involving n variables. Let $\text{VARS}\left(d_1^n, d_2^n\right) \subseteq X$ be defined as follows

$$\text{VARS}\left(d_1^n, d_2^n\right) = \left\{x_i \in X : d_2^n(x_i) \subset d_1^n(x_i)\right\}$$

The optimization is integrated in the `Propagate1` function by modifying the inner loop (lines 3-8) to iterate over the queue of pending propagators instead of the full set of propagators, as shown by function `Propagate2`. Line 2 selects one propagator from the queue of pending propagators P, line 3 removes the propagator from the queue, and line 4 executes the propagator. Line 5 adds to the queue P all propagators subscribed to the domains which were updated. The mutual fixpoint of all propagators is implied by an empty queue P.

3.1.3. Event driven propagation

Determining the exact set of propagators whose execution will prune a given domain is not straightforward. Variable subscription, described above, is often too conservative in the sense that it may still schedule propagators that are at fixpoint for the current domain.

Example 3.4. Consider again the previous example, but where the set of constraints is now $C = c_1, \ldots, c_n$ where $c_i = [x_1 + x_2 \neq x_{i+2}]$, $1 \leq i \leq n-2$, and $c_{n-1} = [x_1 > x_2]$, $c_n = [x_1 < x_2]$. Note that executing propagators π_{c_i}, $1 \leq i \leq n-2$, does not prune any domain whenever $|D(x_i)| > 1$ for all i. Consequently, calling `Propagate2` still requires $O(nb)$ propagator executions to detect inconsistency on this CSP, even if only $O(2b)$ propagator executions are effective.

In fact, determining if a propagator can prune a given domain is co-NP-complete in general. Fortunately, an approximation that is less conservative than variable subscription but still efficient to implement can be devised using events.

In an event driven solver, propagators are subscribed to events representing propagation

conditions. For example, propagators reasoning with integer domain variables usually subscribe to one of the following events:

- ◇ DOM (x_i) : the domain of variable x_i has been updated
- ◇ BND (x_i) : the bounds of the domain of variable x_i have been updated
- ◇ VAL (x_i) : variable x_i has become instantiated

Updating the domain of a variable triggers the corresponding events, adding subscribed propagators to the pending queue.

Example 3.5. In an event driven solver, a propagator for a constraint $c_i = [x_1 + x_2 \neq x_{i+2}]$ would subscribe to VAL (x_1), VAL (x_2), and VAL (x_{i+2}), since propagation of this constraint requires that either x_{i+2} or both x_1 and x_2 are instantiated. Likewise, propagators for $c_{n-1} = [x_1 > x_2]$, and $c_n = [x_1 < x_2]$ would subscribe to BND (x_1) and BND (x_2), since propagation for this constraint reasons exclusively with the bounds of the domains.

Note that even event driven solvers may still schedule propagators which are at fixpoint for the current domain. An example is the propagator for a constraint c_i, described above. In this case, the propagator may be executed needlessly because the exact propagation condition, i.e. (VAL $(x_1) \wedge$ VAL $(x_2)) \vee$ VAL (x_{i+2}), cannot be expressed as a disjunction of events. In chapter 4 we propose an alternative propagation model that may be used to circumvent this problem.

Finally, we remark that during search, propagators may cancel or alter the set of events which they are subscribed to. An example is the propagator for an n-ary boolean disjunction.

Example 3.6. Consider a propagator for the constraint $c = [\bigvee_{i \in 1 \ldots n} x_i]$, where n is an integer greater than two. This constraint is usually propagated using the *watched literal* scheme originally introduced for SAT solving [Gent et al. 2006b]. The propagator initially subscribes to two arbitrary non-ground variables x_i, x_j, more specifically to VAL (x_i), and VAL $\left(x_j\right)$. When any of these events occur, the algorithm cancels the subscription to the event, searches for another non-ground variable x_k where $k \neq i$ and $k \neq j$, and subscribes to VAL (x_k). It repeats this procedure until there are no more non-ground variables to replace the last instantiated variable, say x_i, which mean that x_j must be set to true.

Most modern constraint solvers use events to optimize propagation, as will be detailed in the next chapter.

3.1.4. Signaling fixpoint

Another inefficiency of both versions of the `Propagate` function given above arises from the way fixpoint is detected. The presented approach assumes that a propagator is not at fixpoint if its last execution was able to prune the current domain. While this is a safe way to

Function `Propagate3`(d, P)

Input: A domain d and a set of propagators P implementing constraints C
Output: A subset of d consistent with the constraints in C

```
1 while P ≠ ∅ do
2     π_c ← SelectPropagator(P)
3     ⟨ d', fixpoint ⟩ ← π_c(d)
4     P ← P ∪ ⋃_{x∈VARS(d,d')} PROPS(x) \ {π_c}
5     if ¬ fixpoint then
6         P ← P ∪ {π_c}
7     d ← d'
8 return d
```

detect fixpoint, it implies that a propagator that updates the domains of its own set of subscribed variables is always scheduled one additional time after achieving its fixpoint. A solution to this problem is to force propagators to report whether they are currently at fixpoint. The pseudocode for the algorithm integrating this optimization is given by function `Propagate3`. This function assumes that when the execution of π_c does not change any domain it returns fixpoint= true.

A constraint solver may have to handle propagators that are idempotent for any domain, and propagators that are idempotent for some domains.

Example 3.7. Consider a propagator for the constraint $a = [x_1 \geq x_2]$, where x_1, x_2, are integer domain variables, defined as follows,

$$\pi_a(d) = \begin{cases} d(x_1) & \leftarrow d(x_1) \setminus \{-\infty \ldots \lfloor d(x_2) \rfloor - 1\} \\ d(x_2) & \leftarrow d(x_2) \setminus \{\lceil d(x_1) \rceil + 1 \ldots + \infty\} \end{cases}$$

It easy to see that $\pi_a(d)$ is sound and a fixpoint for π_a for any domain d, i.e. the propagator is an idempotent propagator (def. 2.36 on page 16). Now, consider a similar propagator π_b for the constraint $b = [x_2 \geq x_1]$, and a propagator for $c = [x_1 = x_2]$ defined as,

$$\pi_c(d) \quad = \quad \pi_b(\pi_a(d))$$

Although π_c is sound, it is no longer an idempotent propagator, as witnessed when setting $d_1 = \{v : v \text{ is odd}\}$, and $d_2 = \{v : v \text{ is even}\}$.

The above example shows propagators which are idempotent for any domain, π_a, π_b, and a propagator π_c which may or may not be at fixpoint after execution. An optimized constraint solver could schedule π_a and π_b to a dedicated queue of idempotent propagators, but for π_c it has no other option than to query its idempotency status after every execution and proceed

accordingly.

Since the time spent on redundant propagator executions seems not to be significantly larger than the time spent in computing the idempotency status (see [Schulte and Stuckey 2004]), this optimization is not used in most constraint solvers. As far as we know, Gecode [Gecode 2010] is the only constraint solver implementing fixpoint signaling [Tack 2009].

3.1.5. Subsumption

Another common optimization used by most constraint solvers explores a relation between constraints and domains called entailment. The goal is the same as before: to avoid scheduling propagators unnecessarily.

Definition 3.8 (Entailnment,Subsumption)**.** A constraint c is said to be entailed for a domain d if and only if $d \subseteq \mathrm{con}(c)$. A propagator π_c for a constraint c is said to be subsumed for a domain d if and only if $\pi_c(d') = d'$ for any $d' \subseteq d$.

A propagator which is subsumed for d cannot contribute to any further pruning and therefore should never be scheduled again. Subsumed propagators can therefore be canceled. Canceling does the opposite of subscribing - it unsubscribes the propagators of all previously subscribed events, so that they may not be scheduled again.

Example 3.9. Consider a constraint $c = [x < y]$ and a domain $d = \{1,2\} \times \{3,4\}$. Then c is entailed for d, and a bounds($\mathbb{Z}$) complete propagator π_c is subsumed for d.

Modern constraint solvers typically cancel subsumed propagators. In CaSPER, cancellation is optional, i.e. is not integrated in the propagation kernel, but instead is left to the specific propagator implementation. This allows each propagator to choose to cancel itself if the effort spent in checking subsumption compensates the extra redundant executions.

3.1.6. Scheduling

We have not discussed so far the order in which propagators are executed. The `Propagate3` function is non-deterministic assuming that `SelectPropagator` selects an arbitrary propagator from the propagation queue. However, the order in which propagators are executed affects the number of executions and consequently the runtime required until the mutual fixpoint is achieved.

Example 3.10. Consider a CSP with variables x, y, and z, and domains $D(x) = \{-n \dots 1\}$, $D(y) = D(z) = \{1 \dots n\}$, where n is an arbitrary large number, and the constraints $c_1 = [x \geq y]$ and $c_2 = [x + y = z]$. Consider also domain complete propagators for both constraints. Propagating π_{c_1} only requires updating the bounds of the domains of x and y, which may be done in $O(1)$ time. Propagating π_{c_2} costs $O(n^3)$ since all values in the domains of x, y, and z must

be considered. Therefore, executing π_{c_2} and then π_{c_1} takes $O(n^3)$. However, if π_{c_1} is executed first, the domains of x and y are pruned to $D(x) = D(y) = \{1\}$ in $O(1)$, which can lower the cost of executing π_{c_2} to $O(1)$ since variables x and y are instantiated. Hence, in this case the total cost of executing both propagators is $O(1)$.

The above example suggests an heuristic for scheduling propagators: those with a lower cost should be executed first than those with higher cost. The rationale is that executing cheaper propagators first may prune the current domain thus simplifying the task of costly propagators.

Most modern constraint solvers employ priority based filter scheduling policies for scheduling their propagators. SICStus Prolog [Carlsson et al. 1997] uses two priority levels where indexicals have higher priority and global constraints have lower priority. Gecode and CaSPER uses respectively seven and ten priority levels based on estimated cost of execution. Eclipse Prolog [ECLiPSe 2010] supports twelve priority levels, but its finite domains solver uses only two. Choco [Laburthe and the OCRE project team 2008] uses cost and event based scheduling with seven priority levels.

- Entailment information may be obtained for special cases of *indexical constraints*, as shown in [Carlson et al. 1994].
- Cost based filter scheduling and an experimental comparison of different filter scheduling policies is detailed in [Schulte and Stuckey 2004].

3.2. State manager

Propagation eliminates inconsistent values from the domains. However, since propagation must be complemented by some form of non-deterministic search, any update to the current domain is tentative and thus may have to be undone. In fact, not only variable domains, but the state of propagators, their internal data structures and subsumption status, may have to be restored later due to a wrong guess in the search procedure. In practice, even good heuristics imply a large number of wrong decisions, which makes state handling a central element in a constraint solver.

Example 3.11. Consider a CSP with variables x, y, and z, domains $D(x) = D(y) = \{0,1\}$, $D(z) = \{0\ldots3\}$, and constraints $c_1 = [2x + y = z]$, $c_2 = [x \neq z]$, and $c_3 = [y \neq z]$. Figure 3.1 shows a possible search tree for this problem. Each numbered node represents a state while arcs denote search decisions. Arcs are labeled with a tentative assignment, inside boxes, and the result of propagating the assignment. The two leftmost leaves are not solutions to the problem since z cannot be equal to x or y (constraints c_2 and c_3). Both children of node 3 satisfy all the constraints and therefore are solutions to the problem.

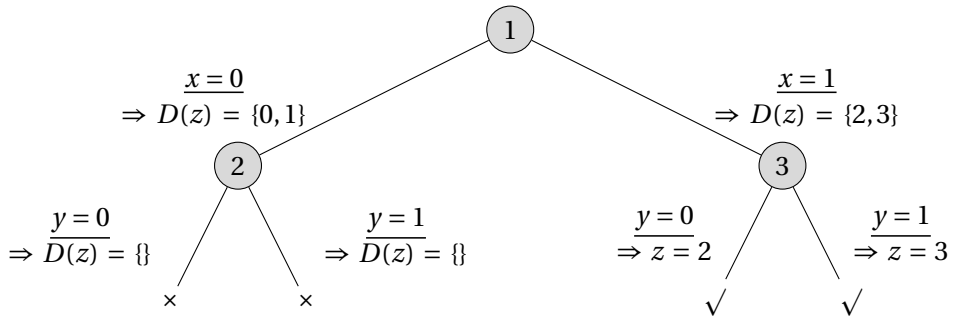
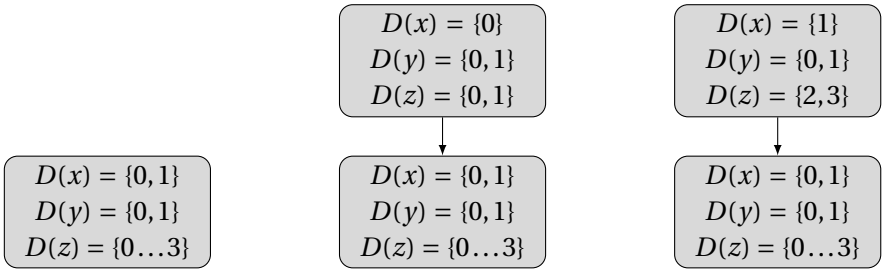


Figure 3.1.: A possible search tree for the CSP described in example 3.11. Search decisions are underlined.



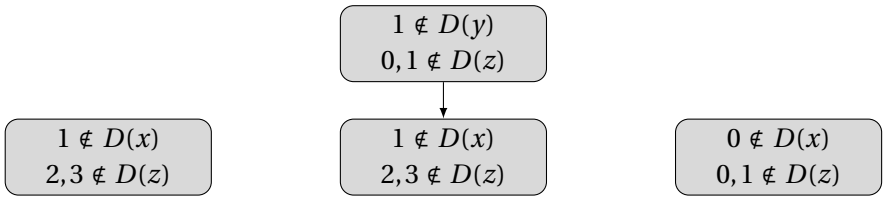
(a) at state 1 (b) at state 2 (c) at state 3

Figure 3.2.: Contents of the stack used by the copying method for handling state.

3.2.1. Algorithms for maintaining state

There are three popular methods for handling state of a constraint solver: copying, trailing, and recomputation. Copying saves the current state before each search decision, by pushing all stateful data to a stack. In case the search decision leads to a dead end, the current state may be restored by copying back all data from the top of the stack. Search then proceeds by committing to a different decision.

Example 3.12. Figure 3.2 shows the contents of the stack used by the copying method while solving the CSP of the previous example. The contents of all domains are pushed to the stack at the initial state (fig. 3.2 a), and again after the first search decision and propagation (fig. 3.2 b). After assigning $y = 0$ and propagating the solver finds an inconsistency, and restores state 2 from the top of the stack. Then it assigns $y = 1$, propagates, again finds an inconsistency and backtracks to state 1. This is done by poping state 2 from the stack and restoring state 1 from the top of the stack (fig. 3.2 a). It then assigns $x = 1$, propagates, and pushes state 3 to the stack (fig. 3.2 c). Finally, it assigns $y = 0$, propagates, and finds a solution.



(a) at state 2 (b) at first failed state (c) at state 3

Figure 3.3.: Contents of the stack used by the trailing method for handling state.

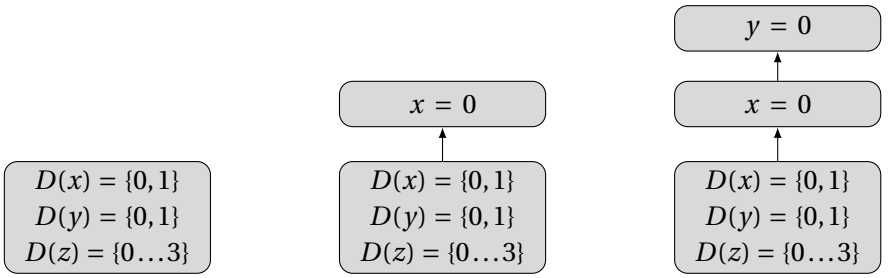
While copying backups all stateful data before any modifications, trailing stores only the data that is going to be modified. Since most of the times is hard to know beforehand which data is going to be updated, trailing must be embedded in any procedure that changes stateful data. Any update operation must therefore check whether the data that is going to change is already saved to the stack, and if not it must push it. On backtrack the solver transverses the list of saved data (the trail) on the top of the stack and restores it.

Example 3.13. The stack used by trailing on the CSP of example 3.11 is shown in figure 3.3. Unlike the stack used in copying, the trailing stack is initially empty. All the domain updates that are a consequence of the first search decision and propagation, leading state 1 to state 2, are then pushed to stack (fig. 3.3 a). When the solver finds the first inconsistency, the contents of the stack are as shown in fig. 3.3 b). It then restores state 2 by undoing the updates on the top of the stack and poping the top trail, leaving the stack as shown in (fig. 3.3 a). When the second inconsistency is found, the solver restores state 1 by undoing the contents of both trails in the stack, and emptying it. The contents of the stack at state 3 are shown in (fig. 3.3 c).

Unlike copying or trailing, recomputation does not store the full current state to memory, but instead only keeps track of the search decisions that lead to the current state. For restoring a given state, recomputation resets the original domains and other stateful data and repeats all operations since the root of the search tree leading to the state to be restored.

Example 3.14. Recomputation uses a queue for recording the original domains, and the search decisions leading to the current state. Figure 3.4 a) shows the initial state of the queue holding the original domains. At state 2 the search decision $x = 0$ is at the end of the queue (fig. 3.4 b). Then, $y = 0$ is added and the solver finds the first inconsistency (fig. 3.4 c). For restoring state 2, the solver first resets the domains from the backup stored in the first position of the queue, and then propagates all the assignments in the queue except the last ($y = 0$). The solver is now on state 2 again, with the queue as shown in (fig. 3.4 b), and thus search may proceed to explore the right branch.

The above methods for handling state may be categorized by expectation. Copying and trailing always backup the current state anticipating that it will be proven inconsistent and will



(a) at state 1 (b) at state 2 (c) at first failed state

Figure 3.4.: Contents of the queue used by the recomputation method for handling state.

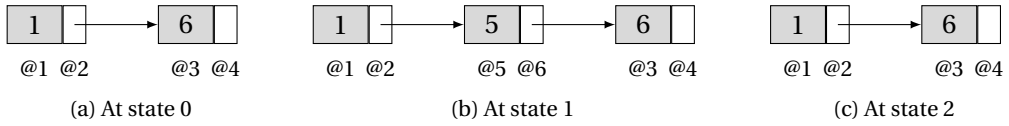
have to be restored: both are pessimistic. Copying is still more pessimistic than trailing since it assumes that everything will change before finding an inconsistency, hence it is more efficient to push the full state to memory beforehand. Unlike copying or trailing, recomputation never stores the current state hoping that it contains a solution and thus will not have to be restored - it is optimistic.

Trailing sacrifices modularity due to the fact that all operations and data structures must be *reversible*, i.e. must be synchronized with the trail. On the other hand, both copying and recomputation are non-intrusive - an existing data structure or algorithm does not need to be modified for integration in a constraint solver handling state using any of these methods. We will detail a simple framework for efficient and correct implementations of reversible data structures maintained with trailing in the next section.

Trailing has been widely adopted by most constraint solvers, namely CaSPER, ILOG Solver [ILOG 2003a], Choco, Minion [Gent et al. 2006a], and Prolog based solvers in detriment of copying or recomputation, showing a robust expectation balance in different kinds of search trees. One notable exception is Gecode which uses an hybrid approach combining copying and recomputation. Choco allows the user to select whether to use copying or trailing for maintaining state.

3.2.2. Reversible data structures

Solvers that use trailing for managing state typically provide a library of reversible data structures, i.e. data structures which automatically record all updates on the trail and thus are able to restore their previous state when required. A common framework for a clean implementation of these data structures builds on the concept of a *backtrackable* or *reversible* pointer. From the client's point of view, a reversible pointer is essentially a standard pointer which keeps a record of the addresses it was pointing to in any previous choice point, so that they can be restored on demand.



(a) At state 0 (b) At state 1 (c) At state 2

Figure 3.5.: Reversible single-linked list. Memory addresses of the values and pointers composing the data structure are shown below each cell.

Implementing a reversible pointer is straightforward - for any update of the reversible pointer a tuple composed of two standard pointers is stored in the trail: the address of the pointer variable, and the address of the location the pointer was pointing to before the update. Reversible pointers allows a complex reversible data structure to be implemented just like the corresponding non-reversible data structure, the only difference is that reversible pointers are used instead of *standard* pointers.

Example 3.15. Figure 3.5 shows a reversible single linked list, initially in state 0. Consider that the solver creates a choice point and inserts value 5 in the list, leaving the structure at state 1. For trailing this operation, the tuple $t_1 = \langle 2,3 \rangle$ is pushed in the trail, representing the update of a pointer at memory address 2 previously pointing to a memory location with address 3. Then, assume it creates another choice point and removes value 5 from the list, leading to state 2. This last operation triggers the insertion of a tuple $t_2 = \langle 2,5 \rangle$ in the trail reflecting the update of the same pointer (at memory address 2) previously pointing to a location with address 5. If the solver later backtracks one step and needs to undo the last operation then knowing t_2 is enough to restore the list back to state 1, just by copying the old value 5 to the memory location at address 2.

Using reversible pointers is also an efficient method of implementing most reversible data structures. For example, each element insertion or removal in the above list requires four constant time copy operations, two for saving and two when restoring. In fact, in the particular case of lists, inserting or removing a contiguous range of elements also takes a constant number of trail operations, since only the pointers connecting the first and last cells in the range have to be trailed. Additionally, this trailing method works very well with lists that must be kept sorted, as those used for maintaining variable domains, since the same cells are restored to the same positions.

The technique just described may be optimized to make the number of save and restore operations depend only on the number of choicepoints, and not on the number of updates to the data structure (see related work below). However, even the basic version described above is much more efficient than, for example, a method that keeps track of which elements have been inserted or removed, which would take linear time for restoring a single update to a sorted list.

Most Prolog engines provide a library of reversible structures. Eclipse Prolog for example provides reversible lists, trees, ordered sets, hash tables, among others. Reversible structures

are also available in Choco, referred to as *backtrackable structures*. In CaSPER the concept of reversible pointer is generalized to any type, providing the user with a myriad of truly transparent reversible data types, of which reversible pointers are a special case. The implementation explores parametric polymorphism so that the type of the basic data type is not abstracted, which is useful for the compiler, and in particular the optimizer. CaSPER additionally provides a library of several reversible structures, such as stacks, lists, arrays, matrices, sets, maps, and tuples among others. ILOG Solver also provides reversible pointers and some plain reversible data types such as reversible integers or booleans whose implementation seems to be based on subtype polymorphism.

3.2.3. Memory pools

The speculative nature of depth search algorithms makes memory management a crucial element in a constraint solver. Most constraints solvers implement an elaborated memory manager that essentially provides fast allocation and deallocation of free memory with a confined scope of existence. There are at least two good reasons to explicitly limit the lifetime of allocated memory. Firstly, it frees the programmer from the difficult task of keeping track of all allocated memory - memory will be automatically reclaimed by the memory manager when its period of existence is over. Secondly, it allows optimizing the number of calls required to request and release memory to the operating system since in many cases the provided memory will be used for data structures which share the same scope of existence. Memory with specific lifetime is usually associated with a dedicated *memory pool*.

A memory manager of a constraint solver may implement several memory pools, but it certainly provides a pool of *backtrackable memory*, i.e. memory which is available from the current choice point node to the leaf of the current search path. This kind of memory is useful for storing variables and propagators added during search, maintaining the incremental state of a propagator, and in general for any data which only exists below the current choice point. We will not detail an implementation of a backtrackable memory pool, but we note that it typically allows constant time allocation of new memory, and constant time deallocation of all memory allocated on the same choice point.

Example 3.16. In the previous example we implicitly assumed that the described single-linked list is allocated on backatrable memory. This means that erasing value 5 does not frees the memory locations at addresses 3 and 4, which would be the natural thing to do with a standard list. Storing the cell holding value 5 in backtrackable memory is essential to allow it to be safely restored just by resetting the values of the pointers, as explained above. The memory is only effectively freed when the solver backtracks to state 0, i.e. before the choice point where cell holding value 5 was created.

Other memory pools may exist in a constraint solver, all with the same goal: simplified and efficient deallocation of memory. Gecode additionally maintains a pool for storing equally-

sized blocks of memory, and a pool for temporary storage. Minion has backtrackable and non-backtrackable memory pools, and even stores some data structures across different memory pools. For example, the storage of domain of binary domain variables or contiguous interval variables is split in two: one part stored in backtrackable memory and another in non-backtrackable memory [Gent et al. 2006a]. CaSPER additionally provides a pool for memory which is a available exclusively for the current choice point, which is useful for storing temporary data used for computing the current fixpoint.

Related work

- For a detailed comparison of trailing methods versus copying and recomputation see [Reischuk et al. 2009].
- The Warren's Abstract Machine [Aït-Kaci 1991] pioneered the presented techniques for maintaining state on a trailing based system, namely reversible pointers and memory pools.
- Memory pools described above may be seen as dedicated *garbage collectors* (see e.g. [Jones and Lins 1996]). However, they are much simpler than general purpose garbage collectors and also much more efficient for the task since they are tightly coupled with the solver's execution model.

3.3. Other components

In the previous sections we have described the elements that form the core of a constraint solver. In this section we will discuss the remaining aspects of a constraint solver, which are orthogonal to the core architecture, and usually integrated in a modular, extensible way.

3.3.1. Constraint library

One of the most appealing aspects of a constraint solver is the span of supported constraints since it reflects its ability to deal with specific applications efficiently. The global constraint catalog [Beldiceanu et al. 2010] currently lists 348 global constraints, for which several practical propagation algorithms often exist. Creating, debugging and maintaining propagators for such a large number of constraints is a daunting task, considering that implementing an efficient propagator for many global constraints is not straightforward. This makes the set of supported constraints one of the most valuable resources of a constraint solver.

Most, if not all constraint solvers either allow adding propagators for existing constraints, or for new constraints, or more frequently both. The propagation algorithm given in section 3.1 promotes this extensibility by establishing a clear separation between the propagation loop, and the propagator task - the propagation kernel is blind with respect to the specific constraint being propagated or the algorithm used for the propagation. However, the constraint kernel

may provide services for simplifying the task of designing new propagators. Reversible structures and dedicated memory pools, introduced above, are one example. In the next chapter we will detail a kernel service that reports the set of changes in the domain of variables between two consecutive executions of a given propagator, which can be very useful for designing propagators.

All constraint solvers mentioned above provide propagators for a large number of constraints. Compared with the remaining solvers, Choco and CaSPER offer a smaller set of specialized domain consistent propagators. In these solvers, constraints for which no specialized propagator exist are compiled to an extensional representation and propagated using a more general but less efficient algorithm. CaSPER additionally implements a technique for achieving bounds consistency on arbitrary constraints, behaving as if specialized propagators were available, described in the second part of this dissertation.

3.3.2. Domain modules

Constraint solving has historically addressed integer domain problems, and more recently, set domain problems. A number of representations and algorithms for reasoning with other domains have also been proposed, namely for addressing multiset problems [Walsh 2003], graph problems [Viegas 2008], real-valued problems [Benhamou 1995], or for specific applications such as protein folding [Krippahl and Barahona 2002]. Support for integrating domain-specific reasoning is therefore an imperative design condition for a general purpose constraint solver, perhaps as important as the ability to integrate new constraint propagators. General purpose constraint solvers are Choco and ILOG Solver, supporting integer, set, and real-valued domain variables, Gecode, handling integer and set domain variables, and Prolog based solvers such as SICStus, Eclipse, and B-Prolog which additionally provide support for graph, tree, and rational domain variables. CaSPER fully supports integer and set domain variables, and provides experimental support for real-valued, graph and coordinate domain variables.

There are also a number of constraint solvers designed for addressing specific domains or special kind of constraints. These solvers are often significantly different than the aforementioned general purpose solvers. Examples are SAT solvers that address problems involving boolean variables, or solvers which work exclusively with finite domain variables, such as MDDC-Solv [Zhou 2009], Abscon [Merchez et al. 2001], CSP4J [Julien Vion 2007], or Sugar [Tamura et al. 2009], among others. These solvers often apply specialized techniques such as compilation of constraints to extensional representation or SAT.

3.3.3. Interfaces

Constraint programs are conceptually highly declarative - the user declares the constraints and variables of the problem, and the solver finds a satisfiable assignment. In practice, however, the user often has to provide specific instructions to the solving process for obtaining a so-

lution in reasonable time. A common solver interface gives the user control over the model, the propagation and search algorithms used for solving the problem. This is the case of Prolog based constraint solvers, Choco, Gecode and CaSPER. Notable exceptions are Minion, and recent versions of ILOG Solver, which provide self-parametrized propagation and search algorithms, thus following the *black-box* approach [Puget 2004].

Recent efforts have been made to establish interfaces for constraint solvers. The JAVA Constraint API is a JAVA interface for constraint solving [ACP 2010]. Numberjack interfaces constraint solvers from the Python language [Hebrard et al. 2010]. XCSP defines a XML based problem specification format, useful for interchanging problems between different solvers [Roussel and Lecoutre 2009]. Minizinc is also a problem specification language, but additionally allows parametrizing the propagation and search algorithms, although currently only in a limited form [Nethercote et al. 2007]. Many constraint solvers are now able to solve constraint problems written in XCSP and Minizinc. CaSPER currently fully supports the former, and offers partial support for the latter, as described in [Silva 2010].

3.4. Summary

This chapter presented an overview of the architecture and techniques used in state-of-the-art constraint solvers. The first part covered a generalized propagation kernel, optimized using variable and event subscription, fixpoint reasoning, subsumption checking, and filter scheduling. Then we have introduced the most popular algorithms for maintaining the state of a constraint solver, namely trailing, copying, and recomputation. We have also described the use of reversible data structures and memory pools, two important techniques for making state managing efficient and transparent. Finally, we have mentioned other important components of a constraint solver, in particular extensible support for new propagators, specific domain reasoning, and standard interfaces.

Chapter 4.

A Propagation Kernel for Incremental Propagation

This chapter will discuss a central component of a constraint solver - the propagation engine. It will describe two models for scheduling propagators, namely variable and propagator centered architectures, highlight their inherent advantages and disadvantages, and show how to integrate standard optimizations such as events or priorities on both models (§4.1). It will be shown that the propagator centered model is theoretically more efficient than the variable centered model. Then we will present a modified propagation centered algorithm that integrates a feature originally exclusive of the variable centered model - support for incremental propagation (§4.2). The implementation of this algorithm in an object-oriented environment will be discussed in §4.3. Finally, we will present a set of experiments performed for comparing several propagator and event scheduling policies using the new propagator-centered model and the variable-centered model (§4.4), and discuss the results obtained providing empirical evidence of the superiority of the former (§4.5).

4.1. Propagator and variable centered propagation

The following concept allows us to analyze and compare fixpoint computations.

Definition 4.1 (Propagation sequence)**.** A propagation sequence describes the domains and propagator executions involved in the computation of a fixpoint. The sequence consists on a series of nodes $d_1, \ldots, d_n$ representing domains, and labeled arrows representing propagator executions. There is a labeled arrow from a domain d_i to domain d_j, denoted $d_i \underset{\pi}{\rightarrow} d_j$, if and only if the execution of propagator π filtered domain d_i to domain d_j during the computation of the fixpoint. Although propagation is non-deterministic, we will assume a deterministic implementation on one processor, which consequently allow us to represent propagation sequences of the form

$$d_1 \underset{\pi_1}{\rightarrow} \ldots \underset{\pi_t}{\rightarrow} d_n$$

Function `PropagatePC(`d, P`)`

Input: A domain d and a set of propagators P implementing constraints C
Output: A subset of d consistent with the constraints in C

```
1 while P ≠ ∅ do
2     π_c ← SelectPropagator(P)
3     P ← P \ {π_c}
4     d' ← π_c(d)
5     P ← P ∪ ⋃_{x∈VARS(d,d')} PROPS(x)
6     d ← d'
7 return d
```

Function `PropagatePC`, introduced in the last chapter, implements what is commonly referred to as *propagator centered* propagation, due to its main data structure: a set of propagators.

Example 4.2. Consider a CSP with variables x_1, x_2, x_3, with domains $D(x_1) = D(x_2) = D(x_3) = \{1,2,3\}$, and the constraints $a = [x_2 \neq 3]$, $b = [x_1 \leq x_2]$, and $c = [\text{DISTINCT}(x_1, x_2, x_3)]$. Invoking the function `PropagatePC` with $P = \{\pi_a\}$, assuming that `SelectPropagator` follows a FIFO policy, and all propagators are domain complete, produces the following propagation sequence,

$$d_1 \underset{\pi_a}{\rightarrow} d_2 \underset{\pi_b}{\rightarrow} d_3 \underset{\pi_c}{\rightarrow} d_4 \left(\underset{\pi_b}{\rightarrow} d_4 \underset{\pi_c}{\rightarrow} d_4 \right)$$

A trace of the execution of algorithm `PropagatePC` for this example is shown below, where each row shows the state of the contents of the data structures at line 1,

0	$d_1(x_1) = \{1,2,3\}$	$d_1(x_2) = \{1,2,3\}$	$d_1(x_3) = \{1,2,3\}$	$P = \{\pi_a\}$
1	$d_2(x_1) = \{1,2,3\}$	$d_2(x_2) = \{1,2\}$	$d_2(x_3) = \{1,2,3\}$	$P = \{\pi_b, \pi_c\}$
2	$d_3(x_1) = \{1,2\}$	$d_3(x_2) = \{1,2\}$	$d_3(x_3) = \{1,2,3\}$	$P = \{\pi_c, \pi_b\}$
3	$d_4(x_1) = \{1,2\}$	$d_4(x_2) = \{1,2\}$	$d_4(x_3) = \{3\}$	$P = \{\pi_b, \pi_c\}$
4				$P = \{\pi_c\}$
5				$P = \{\}$

Note that the second execution of π_b and π_c (iterations 4 and 5) is redundant and could be avoided using the techniques for signaling fixpoint discussed in the previous chapter.

Some constraint solvers take a distinct approach, denoted *variable centered* propagation. This algorithm maintains a queue V of variables whose domain has changed and executes all propagators subscribed for each variable $x \in V$ in turn until there are no more changes in the domains (function `PropagateVC`).

Example 4.3. Consider again the CSP of the previous example. Invoking function `PropagateVC`

Function `PropagateVC(`d, V`)`

Input: A domain d and a set of variables $V \subseteq X$
Output: A subset of d consistent with all constraints in C

```
1  while V ≠ ∅ do
2      x ← SelectVariable(V)
3      V ← V \ {x}
4      P ← PROPS(x)
5      while P ≠ ∅ do
6          π_c ← SelectPropagator(P)
7          P ← P \ {π_c}
8          d' ← π_c(d)
9          V ← V ∪ VARS(d, d')
10         d ← d'
11
12 return d
```

with $V = \{x_2\}$, assuming that both `SelectVariable` and `SelectPropagator` follow a FIFO policy is described by the following propagation sequence,

$$d_1 \underset{\pi_a}{\rightarrow} d_2 \underset{\pi_b}{\rightarrow} d_3 \underset{\pi_c}{\rightarrow} d_4 \left(\underset{\pi_a}{\rightarrow} d_4 \underset{\pi_b}{\rightarrow} d_4 \underset{\pi_c}{\rightarrow} d_4 \underset{\pi_b}{\rightarrow} d_4 \underset{\pi_c}{\rightarrow} d_4 \underset{\pi_c}{\rightarrow} d_4 \right)$$

A trace of the execution of algorithm `PropagateVC` for this example is shown below, where each row shows the contents of the data structures at line 5,

0	$d_1(x_1) = \{1,2,3\}$	$d_1(x_2) = \{1,2,3\}$	$d_1(x_3) = \{1,2,3\}$	$x = x_2, V = \emptyset, P = \{\pi_a, \pi_b, \pi_c\}$
1	$d_2(x_1) = \{1,2,3\}$	$d_2(x_2) = \{1,2\}$	$d_2(x_3) = \{1,2,3\}$	$x = x_2, V = \{x_2\}, P = \{\pi_b, \pi_c\}$
2	$d_3(x_1) = \{1,2\}$	$d_3(x_2) = \{1,2\}$	$d_3(x_3) = \{1,2,3\}$	$x = x_2, V = \{x_2, x_1\}, P = \{\pi_c\}$
3	$d_4(x_1) = \{1,2\}$	$d_4(x_2) = \{1,2\}$	$d_4(x_3) = \{3\}$	$x = x_2, V = \{x_2, x_1, x_3\}, P = \emptyset$
4				$x = x_2, V = \{x_1, x_3\}, P = \{\pi_a, \pi_b, \pi_c\}$
5				$x = x_2, V = \{x_1, x_3\}, P = \{\pi_b, \pi_c\}$
6				$x = x_2, V = \{x_1, x_3\}, P = \{\pi_c\}$
7				$x = x_2, V = \{x_1, x_3\}, P = \emptyset$
8				$x = x_1, V = \{x_3\}, P = \{\pi_b, \pi_c\}$
9				$x = x_1, V = \{x_3\}, P = \{\pi_c\}$
10				$x = x_1, V = \{x_3\}, P = \emptyset$
11				$x = x_3, V = \emptyset, P = \{\pi_c\}$
12				$x = x_3, V = \emptyset, P = \emptyset$

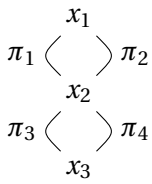


Figure 4.1.: Example of a CSP with variables x_1, x_2, x_3, and propagators $\pi_1, \ldots, \pi_4$.

As seen in the above example, variable centered propagation may lead to more propagator executions compared to propagator centered propagation. This is due to the fact that propagator centered propagation will never schedule a propagator if the propagator is already in the queue, which might not be true for variable centered propagation. The problem can perhaps be best explained using the graph of figure 4.1, which shows a possible structure for propagating a CSP. Each variable is represented as a node in the graph, and an arc between two nodes represents a propagator implementing some constraint between the corresponding variables. Assume that each propagator can modify the domains of both associated variables and is awaken each time the domain of any of its variables changes.

Consider applying variable centered propagation to propagate this CSP. If the domain of variable x_2 is updated, all propagators $\pi_1 \ldots \pi_4$ are added to the queue P. Now, imagine that π_1 is executed and changes variable x_1. Then propagator π_2 will be scheduled for execution when variable x_1 is processed even if it is already on the current propagation queue. The order in which propagators are scheduled does not solve this problem - if π_2 was executed first, π_1 would then be executed twice. The number of extra executions of variable centered propagation compared with propagator centered propagation could be estimated based on the number of loops in the constraint graph of the problem: in the example CSP, there are always two propagators which are executed twice each time variable x_2 changes.

We note that signaling fixpoint would allow both methods to achieve fixpoint using the same number of propagator executions. As discussed previously, this method is untractable in general, but for some constraints it is straightforward to determine fixpoint. For example, iterations 4-7 of the previous example could be easily avoided since the propagator for the constraint $a = [x_2 \neq 3]$ is idempotent.

4.1.1. Incremental propagation

A popular algorithm for achieving domain consistency on the DISTINCT constraint of the previous example is given in [Régin 1994]. The algorithm incrementally maintains a data structure which is a function of the domains of the variables involved in the constraint. For every execution of the propagator, this data structure must be updated to reflect the current domain, which may have changed since the last execution due to the execution of other propagators.

There are two options for performing this update, either the current data structure is discarded and rebuilt from the current domain, or the current data structure is updated by integrating the changes occurred on the domains since the last execution of the propagator, referred throughout this dissertation as *domain delta*. Apparently the best method to apply is constraint specific or perhaps domain specific, but for some situations using domain deltas may improve propagation significantly, as we will see in the next chapter.

An approximation for providing the domain delta to incremental propagators is simply to inform the propagator which variable has triggered propagation. The propagator for the DISTINCT constraint, for example, may use this information to optimize the update of its internal data structure since it knows which variable domain changed. Variable centered propagation naturally integrates this kind of incremental propagation simply by changing line 8 of function `PropagateVC` to $d' \leftarrow \pi_c(d, x)$, where x is the variable that triggered propagation (line 2). Since the propagator is executed once for every variable domain that has changed, its internal data structure will eventually reflect the current domain.

Example 4.4. Consider again the previous example, where π_b is designed to perform incremental propagation as explained above. The propagation sequence when applying incremental variable centered propagation is as follows,

$$d_1 \underset{\pi_a}{\rightarrow} d_2 \underset{\pi_b}{\rightarrow} d_3 \left(\underset{\pi_c}{\rightarrow} d_3 \underset{\pi_a}{\rightarrow} d_3 \underset{\pi_b}{\rightarrow} d_3 \underset{\pi_c}{\rightarrow} d_3 \underset{\pi_b}{\rightarrow} d_3 \right) \underset{\pi_c}{\rightarrow} d_4 \left(\underset{\pi_c}{\rightarrow} d_4 \right)$$

0	$d_1(x_1) = \{1,2,3\}$	$d_1(x_2) = \{1,2,3\}$	$d_1(x_3) = \{1,2,3\}$	$x = x_2, V = \emptyset, P = \{\pi_a, \pi_b, \pi_c\}$
1	$d_2(x_1) = \{1,2,3\}$	$d_2(x_2) = \{1,2\}$	$d_2(x_3) = \{1,2,3\}$	$x = x_2, V = \{x_2\}, P = \{\pi_b, \pi_c\}$
2	$d_3(x_1) = \{1,2\}$	$d_3(x_2) = \{1,2\}$	$d_3(x_3) = \{1,2,3\}$	$x = x_2, V = \{x_2, x_1\}, P = \{\pi_c\}$
3				$x = x_2, V = \{x_2, x_1\}, P = \emptyset$
4				$x = x_2, V = \{x_1\}, P = \{\pi_a, \pi_b, \pi_c\}$
5				$x = x_2, V = \{x_1\}, P = \{\pi_b, \pi_c\}$
6				$x = x_2, V = \{x_1\}, P = \{\pi_c\}$
7				$x = x_2, V = \{x_1\}, P = \emptyset$
8				$x = x_1, V = \emptyset, P = \{\pi_b, \pi_c\}$
9				$x = x_1, V = \emptyset, P = \{\pi_c\}$
10	$d_4(x_1) = \{1,2\}$	$d_4(x_2) = \{1,2\}$	$d_4(x_3) = \{3\}$	$x = x_1, V = \{x_3\}, P = \emptyset$
11				$x = x_3, V = \emptyset, P = \{\pi_c\}$
12				$x = x_3, V = \emptyset, P = \emptyset$

Note that the first and second executions of π_c (iterations 3 and 7) do not prune the domain. This is because the algorithm for propagating π_c is not fully aware of the current domain - it only knows that x_2 has changed, but not x_1. Only on its third execution (iteration 10), π_c is informed about the update of the domain of x_1, and is finally able to prune the domain of x_3.

This shows that using variable centered propagation for incremental propagation may lead to a different propagation sequence compared to non-incremental variable centered propagation. While the fact that an extra number of propagations may be required to synchronize the internal state of the incremental propagators with the current domain may suggest that adding incrementality to variable centered propagation leads to more propagator executions, we cannot confirm that this is always the case due to the complex nature of fixpoint computations (in our example they take the same number of executions).

We also remark that integrating incremental propagation with propagator centered propagation is not as simple as with variable centered propagation, since the information about which variable triggered the propagator execution is not directly available. Later on, we will see how to address this problem. Before that, let us show how the two approaches may be improved using events and priority queues.

4.1.2. Improving propagation with events

Both propagator and variable centered propagation may be improved using events to signal specific propagator conditions. Events have been informally introduced in the previous chapter. Let us now formalize them.

Definition 4.5. Let $d_1(x)$, and $d_2(x)$ be two domains of a variable $x \in X$, such that $d_2(x) \subseteq d_1(x)$. The set of events produced by the update of $d_1(x)$ to $d_2(x)$ is defined by

$$\text{events}(d_1(x), d_2(x)) = \text{dom}(d_1(x), d_2(x)) \cup \text{bnd}(d_1(x), d_2(x)) \cup \text{val}(d_1(x), d_2(x))$$

where

$$\text{dom}(d_1(x), d_2(x)) = \begin{cases} \{\text{DOM}(x)\} & \Leftarrow d_1(x) \supset d_2(x) \\ \{\} & \text{otherwise} \end{cases}$$

$$\text{bnd}(d_1(x), d_2(x)) = \begin{cases} \{\text{BND}(x)\} & \Leftarrow \lceil d_1(x) \rceil > \lceil d_2(x) \rceil \vee \lfloor d_1(x) \rfloor < \lfloor d_2(x) \rfloor \\ \{\} & \text{otherwise} \end{cases}$$

$$\text{val}(d_1(x), d_2(x)) = \begin{cases} \{\text{VAL}(x)\} & \Leftarrow |d_1(x)| > 1 \wedge |d_2(x)| = 1 \\ \{\} & \text{otherwise} \end{cases}$$

Similar to the $\text{VARS}(d, d')$ function which captures the set of variables whose domain has changed from d to domain d', we introduce the $\text{EVENTS}(d, d')$ function used for obtaining the set of events that describe the update of domain d to domain d'.

Definition 4.6. Let d_1, d_2 be two domains of the same CSP such that $d_1 \subseteq d_2$. Then

$$\text{EVENTS}(d_1, d_2) = \bigcup_{x \in X} \text{events}(d_1(x), d_2(x))$$

Function `PropagatePCEvents(`d, E`)`

Input: A domain d and a set of events E
Output: A subset of d consistent with the constraints in C

```
1 P ← ⋃_{e∈E} PROPS(e)
2 while P ≠ ∅ do
3     π_c ← SelectPropagator(P)
4     P ← P \ {π_c}
5     d' ← π_c(d)
6     P ← P ∪ ⋃_{e∈EVENTS(d,d')} PROPS(e)
7     d ← d'
8 return d
```

Since propagators are now associated with events instead of variables, we naturally extend the function PROPS(e) to provide us the set of propagators associated with event e.

Function `PropagatePCEvents` corresponds to an event based implementation of propagator centered propagation. The algorithm now accepts a set E of events signaling domain modifications since its last execution and uses the EVENTS function (line 6) which always inserts a set of propagators in the P queue which is a subset of those that would be inserted with the previous version.

Function `PropagateVCEvents` applies the same optimization to variable centered propagation. Compared with the basic version described earlier (function `PropagateVC`) there are some changes to note. First the algorithm now traverses a queue E of events, instead of a queue of variables - the algorithm is now performing *event centered* propagation. Again the optimization comes from the selection of propagators to execute (line 5) which will be more accurate than with the basic version. Compared with event based propagator centered propagation (function `PropagatePCEvents`), this algorithm may perform even more redundant executions, since the probability that a propagator may be scheduled twice is increased. Finally, we remark that incremental propagation may still be integrated easily, since the event that triggered propagation is available, describes an update of a single domain, and may be provided to the propagator to execute as before.

4.1.3. Improving propagation with priorities

As mentioned earlier, a propagator may be scheduled for execution using distinct policies. For the above algorithms, propagator scheduling may be controlled by modifying the semantics of `SelectPropagator` in functions `PropagatePC` and `PropagatePCEvents`, `SelectVariable` and `SelectPropagator` in function `PropagateVC`, and `SelectEvent` and `SelectPropagator` in function `PropagateVCEvents`. As mentioned earlier, popular propagator scheduling policies are FIFO or cost based scheduling, but these are mainly suited for propagator centered

Function `PropagateVCEvents(`d, E`)`

Input: A domain d and a set of events E
Output: A subset of d consistent with all constraints in C

```
1  P ← ∅
2  while E ≠ ∅ do
3      e ← SelectEvent(E)
4      E ← E \ {e}
5      P ← PROPS(e)
6      while P ≠ ∅ do
7          π_c ← SelectPropagator(P)
8          P ← P \ {π_c}
9          d' ← π_c(d)
10         E ← E ∪ EVENTS(d, d')
11         d ← d'
12
13 return d
```

propagation since scheduling propagators for variable or event centered propagation is affected also by the order in which variables or events are processed. Cost based scheduling for event centered propagation, for example, only affects the queue of propagators of the event being processed (line 7 of function `PropagateVCEvents`) which may potentially compromise its performance compared with propagator based scheduling which always maintains all pending propagators ordered by its cost. On the other hand, variable and event centered propagation offer other opportunities for scheduling - we may choose to first process variables which are more constrained or events which correspond to stronger propagation conditions. We will perform an empirical comparison of propagator scheduling using the presented algorithms in section 4.5.

4.2. The NOTIFY-EXECUTE algorithm

We have seen that propagator centered propagation does not directly allow incremental propagation. To specifically address this problem we designed a new propagation algorithm supporting incremental propagation which is still propagator centered, thus avoiding all the problems inherent to variable or event centered propagation pointed out previously.

The algorithm assumes a global queue P and is composed of two interleaved tasks, EXECUTE and NOTIFY, which break propagator execution from propagator scheduling. The execute phase (function `Execute`) processes all pending propagators, thus emptying the queue P. It is very similar to the basic propagator-centered algorithm given by function `PropagatePC`,

Function `Execute(`d`)`

Data: P and 'failed' are global variables
Input: A domain d
Output: A subset of d consistent with the constraints in C

1 **while** $P \neq \emptyset \wedge \neg$failed **do**
2 $\quad \pi_c \leftarrow$ `SelectPropagator(`P`)`
3 $\quad P \leftarrow P \setminus \{\pi_c\}$
4 $\quad d \leftarrow \pi_c(d)$
5 **return** d

except in that it does not insert propagators into the queue. This is done in the notify phase (procedure `Notify`) which basically processes an event corresponding to a domain update and informs all propagators attached to that event that they should update their internal state. Additionally, this phase is responsible for inserting pending propagators into the queue P.

For this model to work, a call to the `Notify` procedure must be made after every atomic domain update. What consists an atomic domain update is domain specific - for integer and set domains we have chosen to consider the removal of contiguous subset of values as a domain update[1]. This allows propagators to obtain exact information about the current domain and update their internal state incrementally. It also allows delta domains to be maintained in constant space since only the description of the last atomic update is required at any moment. Further details about delta domains will be given in the next chapter.

The `NotifyPropagator` function called from line 2 of the `Notify` procedure is therefore propagator specific, and returns one of `fail`, `schedule`, or `ignore`. The `fail` condition is returned when the propagator discovers inconsistency while updating its internal state, which is not uncommon, and allows the solver to backtrack immediately. The `ignore` condition is signaled if the propagator finds that it is idempotent for the current domain, which may also occur when updating its state. This avoids redundant executions of the propagator which cannot be captured by events. Otherwise, the `schedule` condition is returned to specify that the propagator must be executed. Note that idempotent propagators may still return `schedule` implying one redundant execution but otherwise not compromising the integrity of the system - this is required since finding if a propagator is idempotent may be costly as already discussed.

Priority scheduling may be integrated in this algorithm in two different ways. Propagators may be scheduled by modifying the `SelectPropagator` function called from line 2 of function `Execute`. For example, FIFO based scheduling would always return the first propagator in the queue, and cost based filter scheduling would always return the propagator with lowest cost. Although this simple modification would work for both cases, it would be too costly for

[1] By contiguous subset of values we mean values which are contiguous in the set, e.g. $\{1,3\}$ is a contiguous subset of $\{1,3,5\}$.

Procedure `Notify(`e`)`

Data: P and 'failed' are global variables
Input: An event e

```
1  foreach π_c ∈ PROPS(e) do
2      n ← NotifyPropagator(π_c)
3      switch n do
4          case n = fail
5              failed ← true
6          case n = schedule
7              P ← P ∪ {π_c}
8          otherwise
9              continue
10
11
12
```

the latter case, since the `SelectPropagator` function would potentially need to traverse the entire queue of propagators for finding the propagator with highest priority, thus requiring additional $O(|P|)$ time for each propagator execution. For cost based filter scheduling, it is better to maintain a queue of propagators associated with each cost, which requires the following modification to the above algorithms.

Definition 4.7. Let w be the number of possible costs, $\text{COST}(\pi_c) \in 1 \dots w$ be the cost associated with propagator π_c, and P denote an array of w propagation queues.

Then, consider replacing line 7 of function `Notify` by

$$P_{\text{COST}(\pi_c)} \leftarrow P_{\text{COST}(\pi_c)} \cup \{\pi_c\}$$

Now the `SelectPropagator` function, called from line 2 of function `Execute`, must first find the non-empty queue with highest priority, which can be done in $O(w)$, and then return any propagator from such queue, which can be accomplished in $O(1)$. We note that it is possible to decrease the average time complexity by maintaining P as a queue of non-empty queues. This architecture is similar to the *bucket-queue* described in [Tack 2009; Schulte and Tack 2010].

Additionally, we remark that this model allows nesting several filter scheduling policies. In fact, each queue P_i, where $i \in 1 \dots w$ may be itself a priority queue such as FIFO, LIFO, or even cost-based. This would allow a finer grain of priority levels with a logarithmic worst case complexity of access time. Our implementation supports selecting the type of queue for each level, which already found application for speeding up fixpoint computation in problems with

Procedure `Notify(`e`)`

Data: 'failed' is a global variable
Input: An event e

```
1 foreach v ∈ NOTIFIABLES(e) do
2     n ← v.Call
3     if n = fail then
4         failed ← true
5         return
6
7
```

a particular constraint structure [Jung 2008]. However, we have not yet performed a detailed analysis on the effect of changing priorities at this level in general.

The NOTIFY-EXECUTE algorithm also provides opportunity for event scheduling. This may be accomplished by ordering the events and corresponding calls to the `Notify` function for each domain update.

4.3. An object-oriented implementation

Our implementation of the NOTIFY-EXECUTE algorithm does not follow the model presented above literally, but instead takes advantage of the object-oriented paradigm for achieving more flexibility. The most important concept in this setting is the *notifiable* object, which generalizes and extends the function `NotifyPropagator` presented above by additionally maintaining a state, which is not possible using a simple function.

Definition 4.8 (Notifiable)**.** A notifiable v is a function object (see e.g. [Kühne 1997]) exposing a single method called `Call`. The method takes no arguments, and returns either `ok`, or `failed`, to signal if the solver should proceed or backtrack, respectively.

In practice, constructors for notifiable objects are given a set of objects required by the specific notifiable, in particular but not necessarily, an associated propagator. Once notifiable objects become part of the architecture, some modifications to procedure `Notify` are required, as shown on the current page.

The first visible modification is that now notifiables are associated with events, in the same way propagators were associated with events in the previous version - the NOTIFIABLES(e) function in the code returns the set of notifiables associated with event e. This allows several notifiables to be associated with a single propagator, which is very useful as will be shown shortly. The main `Notify` function is now blind with respect to propagators, it only calls notifiable objects, and therefore it is no longer responsible for adding propagators to the queue P.

Method v_i.Call()

Preconditions: x_i is instantiated
Input: An event e

1 $\pi_c.S.\text{PUSH}(i)$
2 $P \leftarrow P \cup \{\pi_c\}$
3 **return** `ok`

The task of scheduling propagators is now delegated to the notifiable objects, which therefore must have access to P. Let us illustrate with a practical example.

Example 4.9. Consider the global constraint $c = [\text{DISTINCT}(\mathbf{x})]$. Designing a propagator for achieving node consistency for c is trivial for the case $|\mathbf{x}| = 2$ - the propagator waits until one of the variables is instantiated and then removes its corresponding value from the domain of the other variable. When $|\mathbf{x}| = n$ and n is an arbitrary number greater than 2, then naively a set of $n(n-1)/2$ binary DISTINCT propagators are posted between each pair of variables to enforce the constraint. Achieving the fixpoint of this set of propagators takes $O(n^2)$ propagator executions, thus the overall time complexity is $O(n^2)$ assuming each execution takes constant time.

Another solution is to design a global node-consistent propagator for c which, although having the same overall complexity and achieving the same pruning than the previous decomposition, may decrease the number of executions to $O(n)$, one for each variable that gets instantiated. This may be significant since operations in the propagation queue have an associated, albeit small cost. Note that for maintaining the overall complexity we need that each execution of the global propagator takes at most $O(n)$.

A naïve, non-incremental, implementation of this propagator would, on each execution, search for the variables that became ground since the last execution and then prune their values from the domains of the remaining variables. Since each execution of the propagator would take $O(n^2)$ time, this solution is not admissible. A more efficient version of this propagator requires an internal state that is updated incrementally. It works as follows.

In the propagator π_c we maintain a stack S of the indexes of the variables that became ground since the last execution of the propagator. An execution of this propagator simply pops each index i from S and removes value x_i from the domains of the remaining variables. Since each variable can become instantiated only once, the number of executions of this propagator is bounded by $O(n)$, and thus the overall time complexity will be $O(n^2)$. In order to maintain the stack S we need a set N of n notifiable objects, one for each variable, where each notifiable object $v_i \in N$ is subscribed to $\text{VAL}(x_i)$, and has local variables P, π_c, and i, passed as arguments to its constructor. Then, the `Call` method of each notifiable v_i is as shown on this page.

Note that there are other subtleties for maintaining state in the previous example, such as the effect of a possible backtrack before the propagator is executed, but after its internal state is

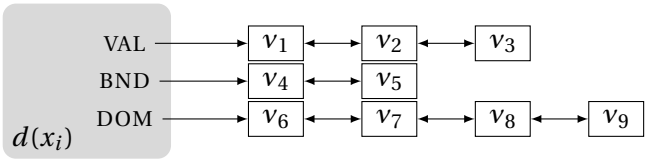


Figure 4.2.: Suspension list

updated. In practice this can be handled straightforwardly by using *reversible* data structures, as explained in the previous chapter.

4.3.1. Dependency lists

We did not yet address how to implement function NOTIFIABLES(e), which returns the set of notifiables subscribed to a specific event e. The common solution to this problem is to use *dependency lists*, also called *suspension lists*. This data structure essentially associates a queue of notifiables with each variable and event type, as shown in fig. 4.2. The object representing the domain of each variable x_i maintains three lists of notifiables (in case of a finite integer domain), one for each event type. Any method performing an update on the domain may therefore directly access the list of relevant notifiables in constant time. The implementation of the suspension lists may be more complex depending on the type of events allowed by the system, and on the kind of operations allowed. A doubly linked queue, as shown in the figure, makes it possible to insert and remove notifiables in constant time, which is mandatory for systems with support for subscribing and canceling propagators, as described in the previous chapter.

Note that the figure presents a simplified representation of a finite domain object - in practice it also maintains other data structures, e.g. for storing the current domain.

An interesting feature of this architecture is that it allows propagators to be subscribed to complex propagation conditions, which are otherwise not captured by basic event subscription, as illustrated in example 3.5 on page 28. With this propagation model, notifiables have the last word for avoiding the scheduling of idempotent propagators. This means that propagation conditions are no longer restricted to disjunctions of events, but may be expressed using an arbitrarily complex function.

4.3.2. Performance

An important feature of the NOTIFY-EXECUTE algorithm is that it only adds a negligible time and memory overhead to the standard propagator-centered model when there are no incremental propagators involved. In the non-incremental propagation-centered algorithm (function `PropagatePCEvents`), for each raised event it is necessary to add the set of relevant prop-

agators to the queue, which takes linear time on the number of propagators subscribed to that event. Likewise, in the NOTIFY-EXECUTE algorithm, for each event it is necessary to call all relevant notifiables. When no incremental propagators are involved, then the suspension list holds exactly one notifiable for each propagator subscribed to that event, whose task is simply to add the corresponding propagator to the queue. Therefore, for each notifiable the incremental model only performs one extra virtual call, and stores one extra pointer, compared with the non-incremental model.

When incremental propagators are involved, then the incremental propagation algorithm may be significantly more costly since a call for each notifiable may not only schedule the associated propagator, but also perform extra operations for updating the propagator's internal state, as seen in the previous example. This means that incremental propagators must be used wisely - a good balance must be achieved between the effort spent to maintain the propagator's internal state and the effort saved by exploring that state. The good news is that the NOTIFY-EXECUTE algorithm itself does not favour any of these methods in particular and allows mixing both incremental and non-incremental propagators. The choice of which propagation model to use is ultimately left to the designer of each propagator.

4.4. Experiments

In this section we perform an empirical evaluation of the previous described models. Since we are interested in propagation algorithms with support for incremental propagation, we will consider only the algorithm for event-centered propagation (function `PropagateVCEvents`) and the NOTIFY-EXECUTE algorithm introduced in the previous section. Recall that the former is essentially an improved version of the variable centered propagation algorithm, while the latter is a propagator centered propagation algorithm modified to support incremental propagation.

Besides directly comparing these two algorithms, the selected set of experiments also assesses the impact that different priority scheduling policies have on the global algorithm. More specifically, we will compare the following models.

4.4.1. Models

EC,EV:COST,PR:COST This model implements the event centered propagation algorithm (function `PropagateVCEvents`) using a cost based policy for scheduling both events and propagators. The cost of each event is associated with the event type: VAL events were given the highest priority, 0, BND events, 1, and DOM events the lowest priority, 2. The hypothesis is that events which signal stronger propagation conditions should be processed first since it should lead to more effective pruning. In this model propagators are scheduled also using their cost, which is associated with its worst-case performance time.

NE,EV:COST,PR:COST This model represents the notify-execute propagation algorithm given in the previous section. Propagators and events were scheduled based on their cost, as in the previous model. Note that in this particular model the impact of scheduling events affects only the order of execution of filters having the same cost.

EC,EV:COST,PR:FIFO Same as EC,EV:COST,PR:COST but scheduling filters using a simple FIFO policy, which means that the impact of scheduling events (using their cost) is increased.

NE,EV:COST,PR:FIFO Like the previous model, but implemented with the notify-execute algorithm.

EC,EV:ANTICOST,PR:COST Same as EC,EV:COST,PR:COST but scheduling events using the inverse cost function, i.e. DOM events are given the highest priority and VAL events the lowest.

NE,EV:ANTICOST,PR:COST Like the previous model, but implemented with the notify-execute algorithm.

EC,EV:ANTICOST,PR:FIFO Same as EC,EV:COST,PR:FIFO but with inverse cost event scheduling.

NE,EV:ANTICOST,PR:FIFO Like the previous model, but implemented with the notify-execute algorithm.

4.4.2. Benchmarks

For testing the above models we used the XCSP benchmark database, used for the CSP solver competitions, and available from [Lecoutre 2010]. From the full set of benchmarks we selected the subset of benchmarks involving global constraints, since this is the kind of problems where scheduling propagators plays an important role. Then, we further restricted this subset to those that were solved in more than 0.2 seconds and less than 120 seconds by the fastest algorithm, resulting in a benchmark set involving a total of 72 benchmarks. The set of global constraints expressible in XCSP is currently limited to the WEIGHTEDSUM, DISTINCT, ELEMENT, and CUMULATIVE constraints. Table B.1 in the appendix enumerates these benchmarks and specifies the number of global constraints of each kind present in each benchmark.

4.4.3. Setup

The code for all experiments was compiled with the gcc-4.4.3 C++ compiler and executed on an Intel Core 2 Duo @ 2.20GHz, using Linux-2.6.32.9. All algorithms were implemented in

CaSPER, revision 583. Each benchmark was repeated until the standard deviation of the propagation time was below 2% of the average propagation time, and then the minimum propagation time was used. Propagation time accumulates the CPU time used while executing propagators.

4.5. Discussion

The results of the experiments described above, shown in tables B.2 and B.3 in the appendix, and summarized in table 4.1, provide the ground for the following conclusions.

Scheduling filters based on cost is more efficient than using a simple FIFO policy when propagating with the notify-execute algorithm. In this case, propagation time using cost based scheduling takes 86% on average of the propagation time if using FIFO scheduling, and for some benchmarks it can be as low as 31% (see first row of table 4.1). On the other hand, event-centered propagation does not benefit from this optimization in general - the average propagation time using cost based scheduling takes 99% of the average propagation time using FIFO (second row of the same table).

The second conclusion is that scheduling events plays an important role for event-centered propagation - scheduling events using their cost reduces propagation time to 80% of the propagation time spent when scheduling events using the inverse of their cost, and is not sensible to the order filters are scheduled (third and fourth row). Symmetrically, scheduling events does not affect performance of the notify-execute algorithm, since the average propagation time when solving using cost based event scheduling is approximately the same propagation time when solving using the anticost based event scheduling, independently of the filter scheduling policy used (fifth and sixth rows).

Therefore, we may conclude that filter scheduling policies only affect the performance of the notify-execute algorithm, and event scheduling policies only affect the performance of event-centered propagation.

Finally, when comparing NE,EV:COST,PR:COST and EC,EV:COST,PR:COST, which are the best models using the notify-execute or the event-centered algorithm respectively, we observe that the notify-execute is much more efficient, which was expected due to the limitations of event-centered propagation described previously. On average, the solver implementing the notify-execute algorithm performs propagation in 15% of the time used by the solver implementing event-centered propagation, and was consistently better across all benchmarks, sometimes two orders of magnitude faster (see last row of the table).

4.6. Summary

In this chapter we have introduced a propagator-centered algorithm supporting incremental propagation. We have seen that this algorithm compares favorably, both theoretically and

	mean	stddev	min	max
NE,EV:COST,PR:COST / NE,EV:COST,PR:FIFO	0.86	1.28	0.31	1.17
EC,EV:COST,PR:COST / EC,EV:COST,PR:FIFO	0.99	1.06	0.83	1.17
EC,EV:COST,PR:FIFO / EC,EV:ANTICOST,PR:FIFO	0.81	1.42	0.2	1.29
EC,EV:COST,PR:COST / EC,EV:ANTICOST,PR:COST	0.8	1.42	0.2	1.04
NE,EV:COST,PR:FIFO / NE,EV:ANTICOST,PR:FIFO	0.99	1.03	0.91	1.03
NE,EV:COST,PR:COST / NE,EV:ANTICOST,PR:COST	1	1.02	0.95	1.04
NE,EV:COST,PR:COST / EC,EV:COST,PR:COST	0.15	3.06	0.01	0.86

Table 4.1.: Geometric mean, standard deviation, minimum and maximum of ratios of propagation times when solving the set of benchmarks using implementations of the models described above.

experimentally, with variable-centered propagation algorithms, which are the traditional approach for incremental propagation. We have also shown how event and filter cost based priorities interact with both models, in particular that scheduling events does not affect propagator centered algorithms.

Related work

- The distinction between variable and propagator centered propagation is made several times in the literature, see for example [Schulte and Tack 2010; Tack 2009; Lagerkvist 2008], but we could not find an experimental comparison of these models.

- We are not sure about the propagation model used in ILOG Solver. Puget and Leconte [1995] mention it is based on AC-5 [Deville and Hentenryck 1991], which would suggest an event-centered propagation algorithm. We know it allows incremental propagation through *demons*, which are similar to notifiables described above.

- The Choco solver [Laburthe and the OCRE project team 2008] allows both event-centered, propagator-centered and the hybridization of both types of propagation. For each domain update, the solver inserts the triggered events in a cost-based priority queue. Later the event queue is processed and propagators associated with that event are added to the propagation priority queue. Events with higher priority are raised immediately, adding the corresponding propagators to the propagation queue, and thus making the solver behave as propagator-centered. Otherwise the events are sorted according to their cost, much like as discussed above for event-centered propagation, but using a different cost map - VAL<DOM<BND, where VAL has the highest priority. We could not find if this solver supports incremental propagation.

- Schulte and Stuckey [2008] perform an extensive analysis of a constraint propagation engine. In particular, they discuss propagator-centered propagation, with all the optimizations described above, namely events, priorities, and other mentioned in the previous chapter, such as checking for subsumption and fixpoint. They do not compare with variable-centered nor event-centered propagation, and do not show how their algorithm may be adapted for incremental propagation. Additionally, they also do not consider event scheduling policies.

- Tack [2009] presents a formal analysis of the Gecode constraint programming system, a propagator-centered solver. His work describes an efficient implementation of the optimizations mentioned in [Schulte and Stuckey 2008]. It discusses variable-centered systems and highlights some of its advantages and disadvantages compared with propagation-centered systems. It does not perform an empirical comparison of these systems, does not consider event scheduling policies, and does not discuss support for incremental propagation.

- Incremental propagation for the Gecode propagator-centered solver is described in [Lagerkvist and Schulte 2007]. The presented algorithm is similar to the NOTIFY-EXECUTE algorithm, where the execution of a notifiable is replaced by the execution of an *advisor*, which essentially performs the same task. However, there are significant differences between the two algorithms. Firstly, in our case notifiables are associated with events, whether advisors are associated with variables. Additionally, advisors require a log of the update, that is, information of which variable changed and how it changed. In our case, the information about which variable has changed may be retrieved by subscribing several specific notifiables to each variable, as shown in section 4.3. Perhaps the most important difference is that the contract between domain updates and notifiables is different from the contract between domain updates and advisors. In our case, notifiables are executed for each atomic update, and have access to exact delta information. In contrast, advisors are executed less often, for complex kind of updates for which the corresponding deltas are often too costly to represent, and thus are approximated.

Chapter 5.

Incremental Propagation of Set Constraints

In this chapter we will analyze incremental propagation of constraints involving set domain variables. It will be shown that exploring incrementality in such constraints can lead to more efficient propagators (§5.3), and presents two distinct solutions for maintaining the information essential to an incremental propagator - the list of what has changed since last propagation, i.e. the domain deltas (§5.4). The first solution stores the domain deltas in each propagator, while the second maintains them in a shared data structure associated with each variable domain. We will show both theoretically and experimentally that the second solution is more efficient and more robust (§5.5-5.6).

5.1. Set constraint solving

Many interesting combinatorial problems involve finding sets of elements satisfying a number of set relations.

Example 5.1 (Steiner triples)**.** Let n, t be two positive integers such that $t = n(n-1)/6$. The Steiner triples problem consists in finding t triples of elements chosen from $1 \dots n$ such that any pair of triples share at most one element. A solution for $n = 7$, $t = 7$, is for example $\{\{1,2,3\},\{1,4,5\},\{1,6,7\},\{2,4,6\},\{2,5,7\},\{3,4,7\},\{3,5,6\}\}$.

Set problems may be naturally modeled using set variables and set constraints.

Example 5.2 (Steiner triples as a CSP)**.** Let $x_1 \dots x_t$ be set variables with domain $2^{1\dots n}$, i.e. the powerset of the set $1 \dots n$. The following constraints model the Steiner triples problem:

$$[\#x_i = 3], \forall i \in 1 \dots t \qquad \text{(each set is a triple)}$$
$$\left[\#\left(x_i \cap x_j\right) \leq 1\right], \forall i, j : 1 \leq i < j \leq t \qquad \text{(any pair of triples share at most one element)}$$

Constraint solvers with set abstraction facilities have been developed to address these problems using search and propagation, fully integrated within the constraint programming paradigm [Puget 1992; Gervet 1994; Azevedo 2007].

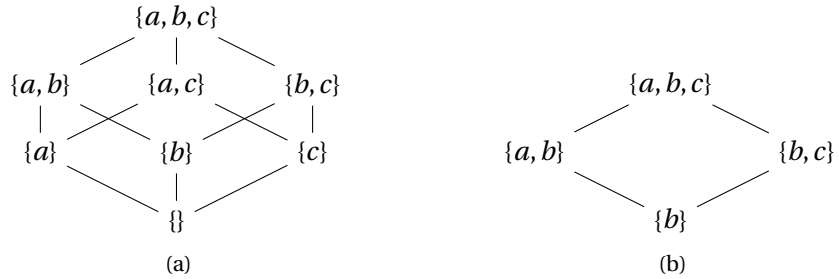


Figure 5.1.: a) Powerset lattice for $\{a,b,c\}$, with set inclusion as partial order. b) Lattice corresponding to the set domain $[\{b\},\{a,b,c\}]$.

5.1.1. Set domain variables

Most set constraint solvers maintain the domain of set domain variables as an interval $d = [l,u]$ where l, u, are known sets ordered by set inclusion, representing the greatest lower bound, GLB, and least upper bound, LUB, of d, respectively. Set intervals define a lattice of sets. Figure 5.1 a) illustrates the powerset lattice for the set domain $\{a,b,c\}$, where a line connecting set S_1 to underneath set S_2 means $S_2 \subset S_1$.

The two bounds (GLB and LUB) define a set interval (e.g.$[\{b\},\{a,b,c\}]$) which is the domain of a set variable x, meaning that x is one of the sets defined by its interval (lattice); all other sets are excluded from its domain. Thus, b is definitely an element of x, while a and c are the only other possible elements. Hence x may be instantiated to one of the sets in the lattice of fig. 5.1 b).

Some set solvers, notably CARDINAL [Azevedo 2007], also explicitly maintain the cardinality of the set variable. This allows for extra inference when propagating set constraints, which has proven very efficient in a number of benchmarks. A set domain therefore consists of a set interval and an integer domain representing the cardinality allowed for the set. For example the set domain $[\{b\},\{a,b,c\}] : [1\ldots2]$ may be instantiated to $\{b\}$, $\{a,b\}$, or $\{b,c\}$, but not to $\{a,b,c\}$ since its cardinality must be at most 2.

5.1.2. Set constraints

Set constraint solvers have specialized propagators for a number of useful constraints and operations over set variables. Examples of common relations are set membership ($i \in x$), set inclusion ($x_1 \subseteq x_2$), equality ($x_1 = x_2$), distinctness ($x_1 \neq x_2$), and disjointness ($x_1 \cap x_2 = \emptyset$). Set operations includes set intersection ($x_1 \cap x_2$), set union ($x_1 \cup x_2$), set difference ($x_1 \setminus x_2$), and symmetric difference ($x_1 \bigtriangleup x_2$). Additionally, a number of constraints involving sets of set variables, commonly referred to as global constraints, are usually available in a set solver.

These include the alldisjoint, alldistinct, or partition constraints, among others. Finally, some set solvers also implement propagators for a number of constraints over integer set variables which occur often in practice, such as minimum or maximum element in a set (resp. $\min(x)$, $\max(x)$), or the sum of all elements in a set ($\sum i : i \in x$).

Propagators typically explore a small set of primitives for updating the domain of the variables involved in the constraint. In the case of set domain variables there are three main operations: insert elements in the domain, remove elements from the domain, and update the cardinality of the domain.

Example 5.3 (Propagator for set inclusion)**.** Let $\text{GLB}(x)$, and $\text{LUB}(x)$, denote respectively the greatest lower bound and least upper bound of the domain of the set variable x. The following set of operations propagates the constraint $x_1 \subseteq x_2$ where x_1 and x_2 are set variables:

$$\begin{aligned}
\text{GLB}(x_2) &\leftarrow \text{GLB}(x_2) \cup \text{GLB}(x_1) \\
\text{LUB}(x_1) &\leftarrow \text{LUB}(x_1) \cap \text{LUB}(x_2) \\
\lceil \#x_1 \rceil &\leftarrow \min(\lceil \#x_1 \rceil, \lceil \#x_2 \rceil) \\
\lfloor \#x_2 \rfloor &\leftarrow \max(\lfloor \#x_1 \rfloor, \lfloor \#x_2 \rfloor)
\end{aligned}$$

The last two operations update the cardinalities of the sets and correspond to propagating an inequality constraint over the pair of integer domain variables representing the cardinalities of the sets: $\#x_1 \leq \#x_2$. Inference on the cardinality of the sets will always be accomplished through propagation of integer constraints, and thus will not be described in this chapter.

Example 5.4. Let x_1, x_2, be two set domain variables with domains $D(x_1) = [\{a,b\}, \{a,b,c\}] : [2 \ldots 3]$, $D(x_2) = [\{b\}, \{a,b\}] : [1 \ldots 2]$. Enforcing the constraint $x_1 \subseteq x_2$ by applying the operations described above instantiates the variables to $x_1 = x_2 = \{a,b\}$.

5.2. Domain primitives

The operations required for propagating the set inclusion constraint are rather straightforward. Other constraints such as set intersection or set union often require a larger set of operations, see [Azevedo 2007] for a comprehensive description. Nevertheless, for all set constraints, propagation is performed by a set of updates to the GLB, LUB, and cardinalities of the involved variables, as shown above for the set inclusion constraint. Let us formalize these operations.

Definition 5.5 (Set domain primitives)**.** Let x be a set variable, and R denote a set of values, possibly with just one element. We will consider two atomic operations for updating the domain of set domain variables, defined as follows

$$\begin{aligned}
\text{INSERTINGLB}(x,R) &= \{\text{GLB}(x) \leftarrow \text{GLB}(x) \cup R\} \\
\text{REMOVEFROMLUB}(x,R) &= \{\text{LUB}(x) \leftarrow \text{LUB}(x) \setminus R\}
\end{aligned}$$

primitive	cost
INSERTINGLB (x, R)	$O(\|\text{GLB}(x)\| + \|R\|)$
REMOVEFROMLUB (x, R)	$O(\|\text{POSS}(x)\| + \|R\|)$
INSERTINGLB* (x, R)	$O(\|\text{LUB}(x)\|)$
REMOVEFROMLUB* (x, R)	$O(\|\text{LUB}(x)\|)$

Table 5.1.: Worst-case runtime for set domain primitives when performing non-incremental propagation.

We note that the above operations are not *safe*. There is an implicit relation between a set's GLB and LUB due to the lattice structure of the domain described above: $\text{GLB}(x) \subseteq \text{LUB}(x)$. All updates on the domain GLB or LUB must fail if this constraint is not satisfied. For making this explicit we redefine the above set of primitives which return a boolean indicating the success status of the operation.

Definition 5.6 (Safe set domain primitives)**.** Let x and R be defined as above and r be a boolean value indicating the return status of the operation, and

$$\begin{aligned} \text{INSERTINGLB}^\star(x, R) &= \{r \leftarrow R \subseteq \text{LUB}(x) ; \text{INSERTINGLB}(x, R)\} \\ \text{REMOVEFROMLUB}^\star(x, R) &= \{r \leftarrow (R \cap \text{GLB}(x) = 0) ; \text{REMOVEFROMLUB}(x, R)\} \end{aligned}$$

Example 5.7. The propagator for the set inclusion constraint $c = [x_1 \subseteq x_2]$ given in example 5.3 may be expressed using the above primitives,

$$\begin{aligned} \pi_c = &\text{INSERTINGLB}^\star(x_2, \text{GLB}(x_1)) \\ &\wedge \text{REMOVEFROMLUB}^\star(x_1, \text{LUB}(x_1) \setminus \text{LUB}(x_2)) \end{aligned}$$

There are a number of important optimizations that we should remark. Firstly, we note that π_c does not always call both primitives, but instead only calls those for which R has changed since last execution. Most set solvers use events, as discussed in chapter 3, that implement this optimization. Secondly, for efficiency reasons, set solvers do not explicitly maintain both the GLB and LUB sets of a set domain variable. Instead, they maintain the GLB and the POSS set, where for a given set variable x, $\text{POSS}(x) = \text{LUB}(x) \setminus \text{GLB}(x)$, representing the set of values that can be possibly added to $\text{GLB}(x)$.

Assuming this representation, that both the GLB and POSS sets of a set domain variable, which are totally ordered, are maintained in a sorted data structure, and that R is a sorted set, then the runtime cost of the set domain primitives is given by table 5.1.

5.3. Incremental propagation

Constraint propagation is accomplished by executing propagators until a fixpoint is reached. Recall that a specific propagator may therefore be executed multiple times during the same fixpoint computation. For helping us in the detailed analysis of incremental propagation over set constraints, let us refine the previously introduced notation.

Definition 5.8 (Propagation sequence)**.** A propagation sequence describes the domains and propagator executions involved in the computation of a fixpoint. The sequence consists on a series of nodes $d_1, \ldots, d_n$ representing domains, and labeled arrows representing propagator executions. There is a labeled arrow from a domain d_i to domain d_j, denoted $d_i \underset{\pi^t}{\rightarrow} d_j$, if and only if the t'th execution of propagator π filtered domain d_i to domain d_j during the computation of the fixpoint. Although propagation is non-deterministic, we will assume a deterministic implementation on one processor, which consequently allow us to represent propagation sequences of the form

$$d_1 \underset{\pi^1}{\rightarrow} \ldots \underset{\pi^t}{\rightarrow} d_n$$

Example 5.9. Consider set variables x_1, x_2, with domains $D(x_1) = D(x_2) = [1 \ldots n, 1 \ldots m]$, where $m > n+1$. Consider also the constraints $a = [x_1 \subseteq x_2]$, $b = [n+1 \in x_1]$ and $c = [m \notin x_2]$. Let us assume the following sequence of propagations until a fixpoint is reached:

$$d_1 \underset{\pi^1_a}{\rightarrow} d_2 \underset{\pi^1_b}{\rightarrow} d_3 \underset{\pi^2_a}{\rightarrow} d_4 \underset{\pi^1_c}{\rightarrow} d_5 \underset{\pi^3_a}{\rightarrow} d_6$$

$$\begin{array}{ll}
d_1(x_1) = [1 \ldots n, 1 \ldots m] & d_1(x_2) = [1 \ldots n, 1 \ldots m] \\
d_2(x_1) = d_1(x_1) & d_2(x_2) = d_1(x_2) \\
d_3(x_1) = [1 \ldots n+1, 1 \ldots m] & d_3(x_2) = d_2(x_2) \\
d_4(x_1) = d_3(x_1) & d_4(x_2) = [1 \ldots n+1, 1 \ldots m] \\
d_5(x_1) = d_4(x_1) & d_5(x_2) = [1 \ldots n+1, 1 \ldots m-1] \\
d_6(x_1) = [1 \ldots n+1, 1 \ldots m-1] & d_6(x_2) = d_5(x_2)
\end{array}$$

Definition 5.10. Let $\text{GLB}_{<\pi^t}(x)$ (resp. $\text{LUB}_{<\pi^t}(x)$) denote the GLB (resp. LUB) of x immediately *before* the t'th execution of π. Similarly, let $\text{GLB}_{>\pi^t}(x)$ (resp. $\text{LUB}_{>\pi^t}(x)$) denote the GLB (resp. LUB) of x immediately *after* the t'th execution of π. For instance, in the example above we have $\text{GLB}_{<\pi^2_a}(x_1) = 1 \ldots n+1$ and $\text{LUB}_{>\pi^3_a}(x_2) = 1 \ldots m-1$.

According to the propagation rules for the constraint $a = [x_1 \subseteq x_2]$ given in the previous section, the set of primitives involved in the several executions of π_a in the propagation sequence

of the above example are

$$\begin{aligned}\pi_a^i &= \text{INSERTINGLB}^\star\left(x_2, \text{GLB}_{<\pi_a^i}(x_1)\right)\\ &\wedge \text{REMOVEFROMLUB}^\star\left(x_1, \text{LUB}_{<\pi_a^i}(x_1) \setminus \text{LUB}_{<\pi_a^i}(x_2)\right)\end{aligned}$$

Although the above propagation is correct, the involved operations are somewhat redundant. For example all elements in $\text{GLB}_{<\pi_a^2}(x_1)$ except one have already been inserted in $\text{GLB}(x_2)$ by π_a^1 when π_a^2 is executed. Similarly, all elements in $\text{LUB}_{<\pi_a^3}(x_1) \setminus \text{LUB}_{<\pi_a^3}(x_2)$ except one were already removed from $\text{LUB}(x_1)$ by π_a^1 when π_a^3 is executed. In fact, π_a^2 and π_a^3 could have been simplified to

$$\pi_a^2 = \text{INSERTINGLB}^\star(x_2, \{n+1\}) \tag{5.1}$$
$$\pi_a^3 = \text{REMOVEFROMLUB}^\star(x_1, \{m\}) \tag{5.2}$$

In this case, the set R to insert or remove from the domain is computed from the set of domain updates since the last execution of the propagator, hence the term *incremental* propagation. Let us formalize this concept.

Definition 5.11 (Set domain deltas)**.** The GLB delta for a variable x with respect to the i'th execution of a propagator π, written $\Delta_x^{\text{GLB}}(\pi^i)$, is the set of values *inserted* in $\text{GLB}(x)$ since the execution π^{i-1}. Similarly, the LUB delta for a variable x with respect to the i'th execution of a propagator π, written $\Delta_x^{\text{LUB}}(\pi^i)$, is the set of values *removed* from $\text{LUB}(x)$ since the execution π^{i-1}. Formally,

$$\begin{aligned}\Delta_x^{\text{GLB}}\left(\pi^i\right) &= \text{GLB}_{<\pi^i}(x) \setminus \text{GLB}_{>\pi^{i-1}}(x)\\ \Delta_x^{\text{LUB}}\left(\pi^i\right) &= \text{LUB}_{>\pi^{i-1}}(x) \setminus \text{LUB}_{<\pi^i}(x)\end{aligned}$$

Example 5.12. An incremental propagator for the set inclusion constraint $a = [x_1 \subseteq x_2]$ may be defined as

$$\begin{aligned}\pi_a^i =& \text{INSERTINGLB}^\star\left(x_2, \Delta_{x_1}^{\text{GLB}}\left(\pi_a^i\right)\right)\\ &\wedge \text{REMOVEFROMLUB}^\star\left(x_1, \Delta_{x_2}^{\text{LUB}}\left(\pi_a^i\right)\right)\end{aligned}$$

Using this propagator for computing the fixpoint of example 5.9 would perform exactly the incremental operations shown in eq. 5.1 and 5.2,

$$\begin{aligned}\pi_a^2 &= \text{INSERTINGLB}^\star\left(x_2, \Delta_{x_1}^{\text{GLB}}\left(\pi_a^2\right)\right) = \text{INSERTINGLB}^\star(x_2, \{n+1\})\\ \pi_a^3 &= \text{REMOVEFROMLUB}^\star\left(x_1, \Delta_{x_2}^{\text{LUB}}\left(\pi_a^3\right)\right) = \text{REMOVEFROMLUB}^\star(x_1, \{m\})\end{aligned}$$

Note that these operations are still linear in the worst case (see table 5.1), which makes incremental propagation advantageous only if $\left|\Delta_x^{\text{GLB}}\right|$ and $\left|\Delta_x^{\text{LUB}}\right|$ are small on average. It turns out that this occurs frequently in practice, as we will see later.

5.4. Implementation

Incremental propagators require access to the set of updates to the domains of the associated variables since their last execution - the domain deltas. In this section we investigate two solutions for making this information available to propagators, with distinct efficiency tradeoffs.

5.4.1. Propagator-based deltas

For each incremental propagator π_c and variable x in their associated constraint c we explicitly maintain the delta sets $\Delta_x^{\text{GLB}}\left(\pi_c^{i+1}\right)$ and $\Delta_x^{\text{LUB}}\left(\pi_c^{i+1}\right)$, where i corresponds to the last execution of propagator π_c.

Definition 5.13. Let $G(x)$ and $L(x)$ denote the set of delta sets associated with variable x, and $G(\pi_c)$ and $L(\pi_c)$ denote the set of delta sets associated with propagator π_c, i.e.

$$G(x) = \left\{\Delta_x^{\text{GLB}}\left(\pi_c^{i+1}\right) : \forall \pi_c\right\} \quad G(\pi_c) = \left\{\Delta_x^{\text{GLB}}\left(\pi_c^{i}\right) : \forall x \in X\right\}$$
$$L(x) = \left\{\Delta_x^{\text{LUB}}\left(\pi_c^{i+1}\right) : \forall \pi_c\right\} \quad L(\pi_c) = \left\{\Delta_x^{\text{LUB}}\left(\pi_c^{i}\right) : \forall x \in X\right\}$$

During search and propagation the solver maintains all delta sets by implementing the following operations:

- Each domain update of a variable x is stored in all relevant delta sets:

$$\text{INSERTINGLB}(x, R) \quad \Rightarrow \quad \Delta \leftarrow \Delta \cup (R \cap \text{POSS}(x)), \forall \Delta \in G(x) \tag{5.3}$$
$$\text{REMOVEFROMLUB}(x, R) \quad \Rightarrow \quad \Delta \leftarrow \Delta \cup (R \cap \text{POSS}(x)), \forall \Delta \in L(x) \tag{5.4}$$

- All delta sets associated with a propagator are cleared after its execution or when the solver backtracks:

$$\Delta \leftarrow \emptyset \quad , \forall \Delta \in G(\pi_c) \tag{5.5}$$
$$\Delta \leftarrow \emptyset \quad , \forall \Delta \in L(\pi_c) \tag{5.6}$$

Example 5.14. Consider set variables x_1, x_2, x_3, with domains $D(x_1) = D(x_2) = D(x_3) = [\{\}, \{1,2\}]$. Consider also the constraints $a = [1 \in x_1]$, $b = [2 \notin x_1]$, and the relation $x_1 = x_2 = x_3$ enforced with a set of binary constraints $c = [x_1 \subseteq x_2]$, $d = [x_2 \subseteq x_1]$, $e = [x_1 \subseteq x_3]$, and $f = [x_3 \subseteq x_1]$. Let us assume that the propagators for the constraint $x_1 = x_2 = x_3$ have been executed once and

thus are at fixpoint. Then, consider that constraints a, and b, are added to constraint store, triggering the following sequence of propagations until the fixpoint is reached:

$$d_1 \underset{\pi_a^1}{\rightarrow} d_2 \underset{\pi_b^1}{\rightarrow} d_3 \underset{\pi_c^2}{\rightarrow} d_4 \underset{\pi_d^2}{\rightarrow} d_5 \underset{\pi_e^2}{\rightarrow} d_6 \underset{\pi_f^2}{\rightarrow} d_7$$

	$d_1(x_1) = [\{\}, \{1,2\}]$	$d_1(x_2) = [\{\}, \{1,2\}]$	$d_1(x_3) = [\{\}, \{1,2\}]$
π_a^1:	$d_2(x_1) = [\{1\}, \{1,2\}]$	$d_2(x_2) = d_1(x_2)$	$d_2(x_3) = d_1(x_3)$
π_b^1:	$d_3(x_1) = \{1\}$	$d_3(x_2) = d_2(x_2)$	$d_3(x_3) = d_2(x_3)$
π_c^2:	$d_4(x_1) = d_3(x_1)$	$d_4(x_2) = [\{1\}, \{1,2\}]$	$d_4(x_3) = d_3(x_3)$
π_d^2:	$d_5(x_1) = d_4(x_1)$	$d_5(x_2) = \{1\}$	$d_5(x_3) = d_4(x_3)$
π_e^2:	$d_6(x_1) = d_5(x_1)$	$d_6(x_2) = d_5(x_2)$	$d_6(x_3) = [\{1\}, \{1,2\}]$
π_f^2:	$d_7(x_1) = d_6(x_1)$	$d_7(x_2) = d_6(x_2)$	$d_7(x_3) = \{1\}$

There are 12 delta sets involved in the incremental propagation of the previous example: $\Delta_{x_i}^{\mathrm{GLB}}(\pi_j)$ and $\Delta_{x_i}^{\mathrm{LUB}}(\pi_j)$ for all $i \in 1\ldots3$ and $j \in \{c,d,e,f\}$. They are maintained according to the rules given above as follows:

π_a^1:	$\Delta_{x_1}^{\mathrm{GLB}}(\pi_c) = \Delta_{x_1}^{\mathrm{GLB}}(\pi_e) = \{1\}$	(rule 5.3)
π_b^1:	$\Delta_{x_1}^{\mathrm{LUB}}(\pi_d) = \Delta_{x_1}^{\mathrm{LUB}}(\pi_f) = \{2\}$	(rule 5.4)
π_c^2:	$\Delta_{x_2}^{\mathrm{GLB}}(\pi_d) = \{1\}$	(rule 5.3)
	$\Delta_{x_1}^{\mathrm{GLB}}(\pi_c) = \Delta_{x_2}^{\mathrm{LUB}}(\pi_c) = \emptyset$	(rule 5.5)
π_d^2:	$\Delta_{x_2}^{\mathrm{LUB}}(\pi_c) = \{2\}$	(rule 5.4)
	$\Delta_{x_2}^{\mathrm{GLB}}(\pi_d) = \Delta_{x_1}^{\mathrm{LUB}}(\pi_d) = \emptyset$	(rules 5.5, 5.6)
π_e^2:	$\Delta_{x_3}^{\mathrm{GLB}}(\pi_f) = \{1\}$	(rule 5.3)
	$\Delta_{x_1}^{\mathrm{GLB}}(\pi_e) = \Delta_{x_3}^{\mathrm{LUB}}(\pi_e) = \emptyset$	(rule 5.5)
π_f^2:	$\Delta_{x_3}^{\mathrm{LUB}}(\pi_e) = \{2\}$	(rule 5.4)
	$\Delta_{x_3}^{\mathrm{GLB}}(\pi_f) = \Delta_{x_1}^{\mathrm{LUB}}(\pi_f) = \emptyset$	(rules 5.5, 5.6)

In our implementation we represent each delta set with a stack. This allows the operations described by eqs. 5.3 and 5.4, to be performed in $O(R)$ and those described by eqs. 5.5 and 5.6 in $O(1)$. However, since propagation may update domains in an arbitrary order, a stack associated with a specific delta set is not necessarily sorted. Unfortunately, sorted delta sets are required for a linear runtime cost of the insert and remove operations, as we have seen in the previous section. This means that, in our implementation, incremental propagation for set solving has a higher worst-case runtime cost compared to non-incremental propagation. We will return to this matter later.

5.4.2. Variable-based deltas

An incremental propagator must access the set of changes on the domains of their associated variables since *its own* last execution. The propagator-based solution described above solves this problem by storing a copy of each domain update in all relevant propagators. This means that for each update R of a domain of a variable x participating in n constraints, there are potentially n copies of R stored in n delta sets. The variable-based solution avoids this problem by storing only one copy of the domain update in a shared location (the variable's domain). Propagators are then given access to their own relevant set of updates by a sophisticated mechanism based on iterators.

Definition 5.15 (Shared FIFO data structure)**.** A shared FIFO data structure Q is any FIFO data structure with additional support for the following primitives: BEGIN (Q) which returns an iterator, i.e. a pointer, to a special element called BEGIN, END (Q) which returns a pointer to the last element in the queue, and TAIL (Q, i) which returns the queue of elements of Q starting after the element pointed by i, and ending in END (Q) .

For every variable x in the problem we have a pair of shared FIFO data structures: Δ_x^{GLB}, associated with the set of updates to GLB (x), and Δ_x^{LUB}, associated with the set of updates to LUB (x). For any incremental propagator π_c and relevant variable x, we define a pair of iterators to Δ_x^{GLB}, Δ_x^{LUB}, denoted $I_x^{\text{GLB}}(\pi_c)$ and $I_x^{\text{LUB}}(\pi_c)$ respectively. The delta sets and the iterators are maintained during search and propagation by applying the following operations:

- Each domain update of a variable x is stored in all relevant delta sets:

$$\text{INSERTINGLB}(x, R) \quad \Rightarrow \quad \text{PUSHBACK}\left(\Delta_x^{\text{GLB}}, R \cap \text{POSS}(x)\right) \tag{5.7}$$
$$\text{REMOVEFROMLUB}(x, R) \quad \Rightarrow \quad \text{PUSHBACK}\left(\Delta_x^{\text{LUB}}, R \cap \text{POSS}(x)\right) \tag{5.8}$$

- All delta sets are cleared after each fixpoint computation, or when the solver backtracks:

$$\Delta_x^{\text{GLB}} \leftarrow \{\text{BEGIN}\} \qquad , \forall x \in X \tag{5.9}$$
$$\Delta_x^{\text{LUB}} \leftarrow \{\text{BEGIN}\} \qquad , \forall x \in X \tag{5.10}$$

- All iterators associated with a propagator π_c are assigned to the BEGIN position when the solver backtracks, and before the initial propagation:

$$I_x^{\text{GLB}}(\pi_c) \leftarrow \text{BEGIN}\left(\Delta_x^{\text{GLB}}\right) \qquad , \forall x \in X(c) \tag{5.11}$$
$$I_x^{\text{LUB}}(\pi_c) \leftarrow \text{BEGIN}\left(\Delta_x^{\text{LUB}}\right) \qquad , \forall x \in X(c) \tag{5.12}$$

A propagator π_c may obtain the delta set of a variable x at any moment during propagation using $\text{TAIL}\left(\Delta_x^{\text{GLB}}, I_x^{\text{GLB}}(\pi_c)\right)$ or $\text{TAIL}\left(\Delta_x^{\text{LUB}}, I_x^{\text{LUB}}(\pi_c)\right)$. In practice the delta set is obtained by iterating $I_x(\pi_c)$ throughout the queue - it defines the set of deltas of x already processed by the

propagator π_c. Usually, when the propagator exits $I_x(\pi_c)$ is pointing to the last element in the queue, but this not required by the model.

Computing the fixpoint of example 5.14 involving the set of operations described above is shown in figure 5.2.

All the above operations may be performed in time $O(1)$. Compared to the previous approach, the variable-based solution is more efficient in memory and runtime: the propagator-based approach takes runtime and memory $O(n|R|)$ for each domain update, where R is the set of elements inserted or removed from the domain and n is the number of propagators involving the variable being updated, while the variable-based approach takes runtime and memory $O(|R|)$. However, accessing the (sorted) delta set is still $O\big(|R|\log(|R|)\big)$ since the queue obtained by the TAIL primitive is not sorted, similarly to what happens with the previous approach.

Finally, we remark that the propagator-based solution still has an advantage over the present approach. If no incremental propagators are used for solving a given problem then no time is spent in saving deltas: it follows a pay-per-use philosophy. On the other hand, the variable-based solution always stores one copy of any domain update, regardless of the existence or absence of propagators which will effectively make use of this information. Particularly for set solving there is no real disadvantage on the present solution since the vast majority of propagators are (or can be made) incremental, however for integer domains, the propagator based solution is perhaps preferable.

5.4.3. Optimizations

Both approaches can be made more efficient by exploring the fact that a domain update involving a contiguous subset may be performed in constant time.

Definition 5.16 (Contiguous subset)**.** Let $S = \{e_1, \ldots, e_n\}$ be a totally ordered set, $e_1 < e_2 < \ldots < e_n$. A contiguous subset C of S is a totally ordered set such that for any element e in $\min(C) \leq e \leq \max(C)$, e is in S if and only if e is in C. For example $\{2,5\}$ is a contiguous subset of $\{1,2,5,7\}$, while $\{2,7\}$ is not.

Let C be a contiguous subset of $\text{POSS}(x)$. One should recall that both INSERTINGLB(x, C) and REMOVEFROMLUB(x, C) must remove C from $\text{POSS}(x)$. If the POSS set of set domain variables is stored in a data structure for which list splicing takes constant time, e.g. a doubly linked list, storing the domain delta corresponding to the update can be achieved in constant time (see fig. 5.3).

Instead of storing the individual elements being removed, both approaches may store lists of contiguous subsets. Consider a domain update involving a non-contiguous subset R of $\text{POSS}(x)$, and let c denote the minimum number of contiguous subsets of $\text{POSS}(x)$ that cover R. Then, this optimization allow us to decrease the runtime and memory cost involved in storing

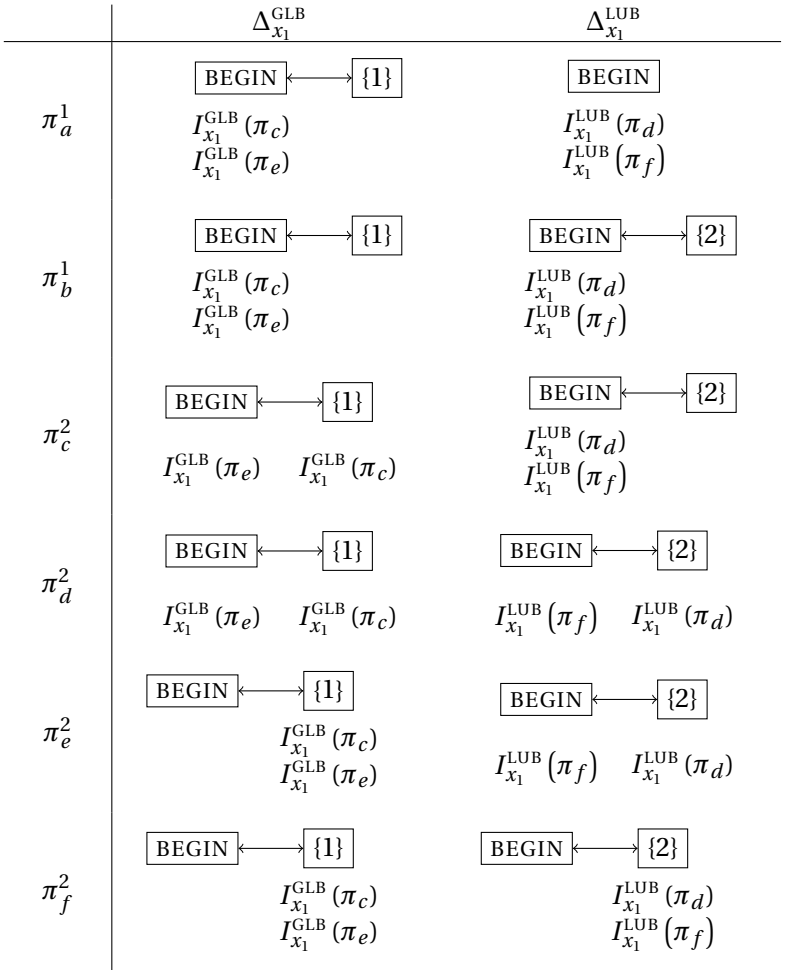
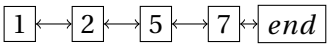
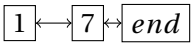


Figure 5.2.: $\Delta^{\text{GLB}}_{x_1}$ and $\Delta^{\text{LUB}}_{x_1}$ delta sets and associated iterators after each propagation during the computation of fixpoint for the problem described in example 5.14.

Initial POSS (x):



POSS (x) after INSERTINGLB (x, C) or REMOVEFROMLUB (x, C) with $C = \{2, 5\}$:



Δ_x^{GLB} or Δ_x^{LUB}:

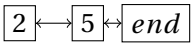


Figure 5.3.: Updating the POSS set and storing the corresponding delta using a list splice operation assuming a doubly linked list representation.

primitive	cost incremental pre-sorting	cost incremental contiguous
INSERTINGLB (x, R)	$O\left(\lvert\text{GLB}(x)\rvert + \lvert R\rvert \log(\lvert R\rvert)\right)$	$O(c \lvert\text{GLB}(x)\rvert)$
REMOVEFROMLUB (x, R)	$O\left(\lvert\text{POSS}(x)\rvert + \lvert R\rvert \log(\lvert R\rvert)\right)$	$O(c \lvert\text{POSS}(x)\rvert)$
INSERTINGLB* (x, R)	$O\left(\lvert\text{LUB}(x)\rvert + \lvert R\rvert \log(\lvert R\rvert)\right)$	$O(c \lvert\text{LUB}(x)\rvert)$
REMOVEFROMLUB* (x, R)	$O\left(\lvert\text{LUB}(x)\rvert + \lvert R\rvert \log(\lvert R\rvert)\right)$	$O(c \lvert\text{LUB}(x)\rvert)$

Table 5.2.: Worst-case runtime for set domain primitives when performing incremental propagation.

deltas to $O(cn)$ and $O(c)$ when using the propagator and variable based approach respectively, where n is the number of propagators involving x.

Since any contiguous subset of a totally ordered set is also totally ordered, we may use this optimization for mitigating the problem of the unsorted delta set already mentioned. For this, we replace any call to INSERTINGLB(x, R) or REMOVEFROMLUB(x, R) where R is a non-contiguous subset of POSS (x) to a set of c calls to INSERTINGLB(x, C_i) or REMOVEFROMLUB(x, C_i) where C_i is the i'th contiguous subset of POSS (x) covering R. The cost of the primitive update operations becomes as shown in table 5.2.

Note that this table cannot be directly compared with table 5.1 since domain update primitives are called with different parameter R in both cases. However, the cost of incremental propagation using these optimizations is asymptotically worst than the cost of non-incremental propagation since it depends on c while non-incremental propagation does not. However, in practice c is very small, typically 1, as we will see in the next section.

5.5. Experiments

In this section we evaluate the benefits of using incremental propagation compared with non-incremental propagation. Additionally, we test the presented implementations for incremental propagation, namely propagator and variable deltas, and a third hybrid approach explained below.

The experiments consist on solving a set of known benchmarks using different solvers, where each solver implements a different propagation model. The search tree visited by each solver is the same for a given experiment since the search heuristics and the applied propagation are the same.

5.5.1. Models

NON-INCREMENTAL This model uses a non-incremental propagator for each constraint of the problem.

INCREMENTAL-PROPAGATOR This model uses incremental propagators implemented using propagator deltas, as described in section 5.4.1, for most constraints in the problem.

INCREMENTAL-VARIABLE This model uses incremental propagators implemented using propagator deltas, as described in section 5.4.2, for most constraints in the problem.

INCREMENTAL-HYBRID This model uses an hybrid propagator combining the above two models for each constraint. When the propagator is executed we test whether $c = 1$, and if so it is propagated incrementally. Otherwise, we simply ignore the delta set and propagate it non-incrementally. This assures that the propagator has the same asymptotic runtime cost of the non-incremental variant.

GECODE The above models were implemented in CaSPER. In order to assess their competitiveness, we also implemented all the problems in the Gecode solver[Gecode 2010] (see also the related work section below).

5.5.2. Problems

Social golfers (prob10 in CSPLib)

The Social golfers problem consists in scheduling a golf tournament. The golf tournament lasts for a specified number of weeks w, organizing g games each week, each game involving s players. There is therefore a total of $g \times s$ players participating in the tournament. The goal is to come up with a schedule where each pair of golfers plays in the same group at most once.

This problem may be modeled using a 2-dimensional matrix x of $w \times g$ integer set domain variables, where each variable describes the g'th group playing in week w. For more information see [Gent and Walsh 1999].

Hamming codes

This is a particular case of the Fixed-length error correcting codes (problem 36 in CSPLib). The problem involves generating a set of binary strings which satisfy a pairwise minimum distance. Each instance is defined by a tuple $\langle n, l, d \rangle$ where n is the number of strings, l is the string length, and d is the minimum distance allowed between any two strings. For measuring the distance between two strings the Hamming distance is used.

This problem may be implemented using two arrays a, b, of n integer set variables each. Each integer set variable a_i denotes the set of positions set to 0 in the string i. Conversely, the set variable b_i denotes the set of positions set to 1 in the string i. They are related through the following set of constraints:

$$\forall_{1 \leq i \leq n} \text{PARTITION}(a_i, b_i)$$

Then, the following constraint ensures that each string is at least d bits apart from any other string:

$$\forall_{1 \leq i < j \leq n} \left| a_i \cap b_j \right| + \left| b_i \cap a_j \right| \geq d$$

Balanced partition

The balanced partition problem consists in finding a partition of the set of values $1 \ldots v$ through the sets $S_1, \ldots, S_n$ which minimizes m, where

$$m = \max_{1 \leq i \leq n} \left(\sum_{j \in S_i} j \right)$$

For the decision version of this problem the value m is given and the goal is to find the corresponding partition. This problem has a direct model using n set domain variables corresponding to $S_1, \ldots, S_n$ constrained by a PARTITION constraint.

Metabolic pathways

The metabolic pathways problem consists in searching for a particular path in a graph representing a metabolic network. Metabolic networks are biochemical networks which encode information about molecular compounds and reactions which transform these molecules into substrates and products. This problem was modeled in GRASPER, a graph-based constraint solver built on top of the set solver available in CASPER. Both the problem, the model, and the graph-based solver are detailed in [Viegas 2008; Viegas and Azevedo 2007].

Winner determination problem

In a combinatorial auction, the auctioneer makes available a set $M = 1 \ldots m$ of items and there is a set $K = 1 \ldots k$ of bidders participating in the auction. A set of bids $B = 1 \ldots b$ is submitted

by the bidders, where each bid corresponds to a subset of M. Let p_i be the price agreed to pay for the items of bid i, and G_j the set of bids contemplating item j. The winner determination problem consists in labeling each bid as winning ($x_i = 1$) or losing ($x_i = 0$) so as to maximize the auctioneer's revenue given that no item can belong to more than one winning bid:

$$\max \sum_{i \in B} p_i x_i \qquad \text{s.t.} \forall_{j \in M} \sum_{i \in G_j} x_i \leq 1$$

Like the previous benchmark, this problem was modeled in GRASPER. The specific model and further details are given in [Viegas 2008].

5.5.3. Setup

The code for all experiments was compiled with the gcc-4.4.3 C++ compiler and executed on an Intel Core 2 Duo @ 2.20GHz, using Linux-2.6.32.9. The versions of the CaSPER and Gecode solvers were the most recently available at the moment, respectively revision 583 and version 3.3.1. Each benchmark was repeated until the standard deviation of the runtime was below 2% of the average time, and then the minimum runtime was used.

5.6. Discussion

The runtimes obtained for all the experiments are detailed in section B.2 (in the appendix). Each table shows the number of domain updates, average size of domain update, the time and number of propagations for solving the benchmark. Additionally, for each instance we show the number of variables involved in the problem (v), the ratio constraints/variable (c/v) and number of fails (f).

The runtimes for solving the first three benchmarks are summarized in table 5.3. The table provides some statistical measures concerning the ratio of the runtime using the corresponding implementation over the runtime using an implementation with non-incremental propagation. For example, solving a problem performing incremental propagation using propagator deltas takes on (geometric) average 88% of the time spent solving the same problem with non-incremental propagation.

The table shows that incremental propagation using variable deltas is better on average than any other method, although only slightly better than incremental propagation using propagator deltas. Additionally, it is always better than non-incremental propagation, and also the most robust alternative. The hybrid approach does not pay off in practice probably due to the overhead involved in selecting the best approach at every propagation. Finally, the last row shows that all tested solvers implemented in CaSPER slightly outperform the Gecode solver.

The results concerning the last two benchmarks are summarized in table 5.4. The table provides a statistical analysis on the ratio of the solver runtime using incremental propaga-

	mean	stddev	min	max
INCREMENTAL-PROPAGATOR	0.88	1.14	0.67	1.07
INCREMENTAL-VARIABLE	0.83	1.12	0.66	0.96
INCREMENTAL-HYBRID	0.93	1.14	0.73	1.15
GECODE	1.14	1.13	0.91	1.33

Table 5.3.: Geometric mean, standard deviation, minimum and maximum of ratios of runtime for solving the first set of benchmarks described above using the presented implementations of incremental propagation, compared to non-incremental propagation.

	mean	stddev	min	max
INCREMENTAL-VARIABLE	0.7	1.27	0.46	0.91

Table 5.4.: Geometric mean, standard deviation, minimum and maximum of ratios of runtime for solving the second set of benchmarks (graph problems) using the variable delta implementation of incremental propagation, compared to propagator deltas.

tion based on variable deltas over the solver runtime using incremental propagation based on propagator deltas. In these benchmarks using variable deltas instead of propagator deltas for implementing incremental propagation has a greater impact on runtime compared with the first three benchmarks (about 13% better on average). This discrepancy is probably due to the fact that some graph constraints used involve an exponential number of set propagators, which is the case where using variable deltas is more efficient.

5.7. Summary

We have seen that incremental propagation can make constraints involving set domain variables significantly more efficient. Incremental propagation requires knowing the set of changes between propagator executions, for which we have introduced two distinct techniques: propagator and variable based deltas. We have shown that the latter is theoretically more efficient in both time and memory, and that it performs slightly better in practice on problems involving set domain variables. Additionally, on problems involving graph domain variables it can achieve an improvement on runtime of about 30% on average compared to propagator based deltas.

Related work

- The set solver used in the experiments is based on and extends the Cardinal set solver [Azevedo 2007]. This set solver is implemented in Eclipse prolog [ECLiPSe 2010] and does not support incremental propagation.
- We have focused exclusively on propagators achieving bounds consistency for set constraints. Domain consistent propagators for set constraints have been introduced in [Lagoon and Stuckey 2004]. These propagators maintain a compacted extensional representation of the set of possible values (which are themselves sets) for each set domain variable. An incremental domain consistent propagator for a constraint involving set variables would require access to a number of domain updates which is exponential on the set cardinality, instead of linear as in the case of incremental bounds consistent propagators. Therefore, we believe that our method would not be easily adapted for domain complete propagators.
- In this chapter we presented incremental propagation for constraints involving set domain variables. CaSPER also provides incremental propagators for constraints over integer domain variables, but they are implemented differently due to the following two facts. Firstly, most propagators involving integer domain variables have a very efficient non-incremental implementation, specifically those for arithmetic constraints achieving bounds consistency. Secondly, incremental domain consistent propagators such as the alldifferent, extensional or regular constraint propagators, achieve their incrementality by querying and modifying its internal state according to the specific domain update, as illustrated in the previous chapter. Therefore, the information about most domain updates is not used and maintaining it in delta sets would often be a waste of time. The current implementation associates a demon to each propagator which is able to access and update the internal state of incremental propagators. For those propagators which require the full delta information, the specific demon may store it in a data structure which is local to the propagator, similar to what is done with propagator based deltas. This technique is not new, it is used for example in Choco [Choco 2010], ILOG Solver [ILOG 2003b], and Minion [Gent et al. 2006b].
- Incremental propagation in Gecode solver is implemented through an abstraction called *Advisor* [Lagerkvist and Schulte 2007; Lagerkvist 2008]. Advisors are associated with propagators and besides providing delta information can be used to alter the internal state of each propagator, similar to the incremental propagators involving integer domain variables available in CaSPER explained above. We did not find a detailed description of the advisors used on propagators for simple set constraints but their implementation should map to propagator based deltas.
- Throughout this chapter we assumed a set domain representation based on a doubly linked list. This is indeed the most common implementation, but others are possible. For example in [Viegas et al. 2008] we introduced an hash set representation of the domain of a set

domain variable. This representation has interesting advantages, namely $O(1)$ inserts and updates, but disallows the optimization which explores updates of contiguous sets of values described in section 5.4.3.

Part II.

Efficient Propagation of Arbitrary Decomposable Constraints

Chapter 6.

Propagation of Decomposable Constraints

This chapter formalizes propagation of decomposable constraints. More specifically, it describes propagation of decomposable constraints as a function of *views*, an abstraction representing the pointwise evaluation of a function over a given (tuple) set. It is organized as follows: First we characterize a decomposable constraint as a functional composition (§6.1). Then, we introduce views (§6.2), and show that they may be used to express sound and complete propagators for arbitrary constraints (§6.3). Finally, we detail how views may be obtained from composition of other views, and that this property can be used to model complete propagators for complex decomposable constraints (§6.4).

6.1. Decomposable constraints

Most constraints used in practice are decomposable. Below are some examples of constraints which may be decomposed, as will be detailed later.

Example 6.1 (Arithmetic constraints)**.** This is probably the most common type of decomposable constraints. Examples are constraints involving a sum of variables, e.g. $\left[\sum_i x_i = 0\right]$, a linear combination, e.g. $\left[\sum_i a_i x_i \geq 0\right]$, or a product, e.g. $\left[\prod_i x_i = 0\right]$. Arithmetic constraints may include an arbitrary combination of arithmetic operators, such as for example $[|x_1 - x_2| = 2x_3]$.

Example 6.2 (Boolean constraints)**.** Boolean constraints involve Boolean domain variables or expressions, for example a disjunction of variables $[\bigvee_i x_i]$, or more complex expressions on Boolean constraints such as disjunctions $[\bigvee_i (x_i > 0)]$, logical implications $\left[x > 0 \Rightarrow y < 0\right]$ or equivalences $\left[x > 0 \Leftrightarrow y\right]$.

Example 6.3 (Counting constraints)**.** Counting constraints restrict the number of occurrences of some values within a collection of variables, for example the EXACTLY($\mathbf{x}, v, c$) constraint $\left[\sum_i (x_i = v) = c\right]$, or the AMONG($\mathbf{x}, V, c$) constraint $\left[\sum_i (x_i \in V) = c\right]$.

Example 6.4 (Data constraints)**.** Also known as *ad-hoc* constraints, they represent an access to an element of a data structure (a table, a matrix, a relation) [Beldiceanu et al. 2010]. The most common constraint in this class is the ELEMENT$\left(\mathbf{x}, i, y\right)$ constraint enforcing $\left[x_i = y\right]$ where i

is a variable. Decomposable constraints involving this constraint are for example $[x_i \geq y]$ or $[x_i - x_j \geq 0]$.

6.1.1. Functional composition

We propose to express decomposable constraints as a composition of functions. For this purpose we will make extensive use of functions that evaluate to tuples, i.e. $f : \mathbb{Z}^n \to \mathbb{Z}^k$ where $k \geq 1$, together with the operations (and conventions):

Definition 6.5 (Functional composition)**.** Functional composition is denoted by operator $\circ$ as usual: $(f \circ g)(x) = f(g(x))$.

Definition 6.6 (Cartesian product of functions)**.** Cartesian product of functions is denoted by operator $\times$ as follows: $(f \times g)(x) = \langle f(x), g(x) \rangle$. If x is a tuple we may write

$$(f \times g)(x_1, \ldots, x_n) = \langle f(x_1, \ldots, x_n), g(x_1, \ldots, x_n) \rangle$$

These operators may of course be combined thus providing a very effective way to represent decomposable constraints. To exercise this terminology, we will exemplify with a possible decomposition of some of the constraints given above. In the following examples let x, y represent variables, a a constant, and $p_i(\mathbf{x}) = x_i$ the projection operator.

Example 6.7 (Equality constraint)**.** Let constraint $c_{\text{eq}}(\mathbf{x}) = [x_1 = x_2]$ be the binary constraint stating that variable x_1 and x_2 must take the same value. A unary equality constraint $c_{\text{eqc}}(x, a) = [x = a]$ may be obtained by $c_{\text{eqc}} = [c_{\text{eq}} \circ (p_1 \times f_a)]$, where $f_a(\mathbf{x}) = a$.

Example 6.8 (Sum constraint)**.** Let $f(\mathbf{x}) = x_1 + x_2$. A sum of three variables is represented by $f \circ (f \times p_3)$. The generalization to a sum of n variables is defined as

$$g = f \circ (f \times p_3) \circ (f \times p_3 \times p_4) \circ \ldots \circ (f \times p_3 \times p_4 \times \ldots \times p_n)$$

Therefore we may decompose a constraint for a sum of n variables $c_{\text{sum0}}(\mathbf{x}^n) = [\sum_i^n x_i = 0]$ as $c_{\text{sum0}} = [c_{\text{eq0}} \circ g]$, where $c_{\text{eq0}}(\mathbf{x}) = c_{\text{eqc}}(\mathbf{x}, 0)$ defined in the previous example.

Example 6.9 (Linear constraint)**.** A linear constraint $c_{\text{lin0}}(\mathbf{a}^n, \mathbf{x}^n) = [\sum_{i=1}^n a_i x_i = 0]$ may be composed as follows. Let $f_1, \ldots, f_n : \mathbb{Z}^n \to \mathbb{Z}$, where $f_i(\mathbf{x}^n) = a_i x_i$. Then, $c_{\text{lin0}} = [c_{\text{sum0}} \circ (f_1 \times \ldots \times f_n)]$, where c_{sum0} is defined in the previous example.

Example 6.10 (Arbitrary arithmetic constraint)**.** Let $f(\mathbf{x}) = x_1 - x_2$, $g(x) = |x|$, $h(x) = 2x$. The arithmetic constraint $c(\mathbf{x}^3) = [|x_1 - x_2| = 2x_3]$ may be represented as

$$c = [c_{\text{eq}} \circ ((g \circ f \circ (p_1 \times p_2)) \times (h \circ p_3))]$$

Example 6.11 (EXACTLY constraint)**.** This constraint, represented by $\left[\sum_i (x_i = v) = c\right]$, may be obtained similarly to example 6.9 but where $f_i(\mathbf{x}^n) = [x_i = v]$ and the constraint c_{eqc} is used instead of c_{eq0}.

Example 6.12 (Data constraints)**.** The constraints introduced in example 6.4 may be decomposed into

$$\begin{aligned} [x_i \geq y] &= [\text{ELEMENT}(\mathbf{x}, i, a_i) \wedge a_i \geq y] \\ [x_i - x_j \geq 0] &= [\text{ELEMENT}(\mathbf{x}, i, a_i) \wedge \text{ELEMENT}(\mathbf{x}, j, a_j) \wedge a_i - a_j \geq 0] \end{aligned}$$

6.2. Views

This section presents an informal introduction to *views* as a tool for analyzing the propagation of decomposable constraints.

Example 6.13. Consider a CSP defined by variables x_1, x_2, x_3, domains $D(x_1) = D(x_2) = D(x_3) = \{1,2,3\}$, and constraints $c_1 = [x_1 = x_2]$, and $c_2 = [x_1 + x_2 = x_3]$. A propagator for constraint c_1 performs the following operations:

$$\begin{aligned} D(x_1) &\leftarrow D(x_1) \cap D(x_2) \\ D(x_2) &\leftarrow D(x_2) \cap D(x_1) \end{aligned}$$

Similarly, a propagator for the constraint c_2 may be described by the following operations:

$$\begin{aligned} D(x_1 + x_2) &\leftarrow D(x_1 + x_2) \cap D(x_3) \\ D(x_3) &\leftarrow D(x_3) \cap D(x_1 + x_2) \end{aligned}$$

where $D(x_1 + x_2) = [2 \ldots 6]$ represents the possible values (i.e. the domain) of the subexpression $x_1 + x_2$.

In most constraint solvers, domains of subexpressions occurring in constraints are not directly available to propagators, which are designed to work exclusively with variable domains. In these solvers, an offline modeling phase is responsible for obtaining an equivalent CSP where all constraints are *flattened*, that is where each subexpression appearing in a constraint is replaced by an auxiliary variable whose domain is the domain of the subexpression. Nevertheless, we may develop a conceptual model that considers explicit subexpression domains while abstracting on how they are computed and maintained. This will allow a theoretical analysis of the propagation on the constraint decomposition, regardless of the method used for representing the subexpression domains. In chapter 8 we evaluate some practical instantiations of the model, namely when performing the transformation detailed above.

We begin by defining an abstraction which captures the domain of a subexpression in a constraint: a view.

Definition 6.14 (View)**.** A view over a function $f : \mathbb{Z}^n \to \mathbb{Z}^k$ is a pair $\varphi = \left\langle \varphi_f^+, \varphi_f^- \right\rangle$ of two functions, the *image function* $\varphi_f^+ : \wp(\mathbb{Z}^n) \to \wp\left(\mathbb{Z}^k\right)$, and the *object function* $\varphi_f^- : \wp\left(\mathbb{Z}^k\right) \to \wp(\mathbb{Z}^n)$, defined as

$$\begin{aligned} \varphi_f^+\left(S^n\right) &= \left\{ f\left(\mathbf{x}^n\right) \in \mathbb{Z}^k : \mathbf{x}^n \in S^n \right\}, \forall S^n \subseteq \mathbb{Z}^n \\ \varphi_f^-\left(S^k\right) &= \left\{ \mathbf{x}^n \in \mathbb{Z}^n : f\left(\mathbf{x}^n\right) \in S^k \right\}, \forall S^k \subseteq \mathbb{Z}^k \end{aligned}$$

A view φ_f is therefore defined by considering the pointwise application of f over a given set. The image function computes a set of images of f, that is the set resulting from applying f to all the points in the given set. The object function does the inverse transformation, it computes the set of objects of f for which its image is in the given set.

Example 6.15. Applying a view over function $f(x) = x + 1$ to a random set:

$$\begin{aligned} \varphi_f^+(\{1,2,3\}) &= \{2,3,4\} \\ \varphi_f^-(\{2,3,4\}) &= \{1,2,3\} \end{aligned}$$

As discussed earlier, propagating a constraint may require consulting and updating the domain of a subexpression. A view over the subexpression defines these operations:

Example 6.16. Consider a view φ_g over function $g\left(x,y\right) = x + y$, and the CSP of example 6.13 on the preceding page. Function φ_g^+ may be used to compute the subexpression domain $D(x_1 + x_2)$ from the variable domains $D(x_1)$ and $D(x_2)$, while function φ_g^- may be used to specify the set of values for the variables x_1, x_2 that satisfy constraint c_2.

$$\begin{aligned} \varphi_g^+(D(x_1) \times D(x_2)) &= [2 \ldots 6] \\ \varphi_g^-(D(x_3)) &= \left\{ \left\langle x,y \right\rangle \in \mathbb{Z}^2 : x + y \in \{1,2,3\} \right\} \end{aligned}$$

Propagation consists in removing inconsistent values from the domain of the variables. In this sense, the computation of φ_f^-, i.e. the set of consistent assignments, will always precede an intersection with the original domain to guarantee that the propagation is *contracting*. The following definition captures exactly that.

Definition 6.17 (Contracting object function)**.** Let $S^n \in \mathbb{Z}^n$, $S^k \in \mathbb{Z}^k$, be two arbitrary tuple sets and $f : \mathbb{Z}^n \to \mathbb{Z}^k$ an arbitrary function. Then,

$$\widehat{\varphi}_f\left(S^k, S^n\right) = \varphi_f^-\left(S^k\right) \cap S^n$$

Example 6.18. The result of evaluating and updating the domain $D(x_1 + x_2)$, as defined in the

previous example, may be formalized using views as follows:

$$\begin{aligned} \varphi_g^+ \left(D\left(x_1\right) \times D\left(x_2\right) \right) &= [2 \ldots 6] \\ \widehat{\varphi}_g \left(D\left(x_3\right), D\left(x_1\right) \times D\left(x_2\right) \right) &= \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle\} \end{aligned}$$

6.3. View-based propagation

A constraint may be seen as a Boolean function $c : \mathbb{Z}^n \to \{0,1\}$, mapping allowed tuples (i.e. tuples satisfying the constraint) to 1 and forbidden tuples to 0. In this sense, views may represent not only subexpression domains, but the domains of the constraints as well. Let us then formally explore the relation among views, constraint domains, and the most common operations occurring in the process of constraint solving: checking and propagating constraints.

6.3.1. Constraint checkers

A constraint checker decides if the constraint c is satisfied (or entailed), or unsatisfied (resp. disentailed) for some tuple set. Constraint checkers are used in practice for many purposes, such as implementing reification [Hentenryck and Deville 1993], optimizing constraint propagation by detecting subsumed propagators [Schulte and Stuckey 2004], or as part of weaker search algorithms which do not require propagation, e.g. pure backtracking.

Definition 6.19 (Constraint checker)**.** A constraint checker is a monotonic function $\chi_c : \wp\left(\mathbb{Z}^n\right) \to \wp\left(\{0,1\}\right)$. The constraint checker is *sound* if and only if

$$\begin{aligned} \operatorname{con}(c) \cap S^n \neq \emptyset \Rightarrow 1 \in \chi_c\left(S^n\right) \wedge \\ \operatorname{con}(c) \cap S^n \neq S^n \Rightarrow 0 \in \chi_c\left(S^n\right) \end{aligned}$$

and *complete* if and only if

$$\begin{aligned} \operatorname{con}(c) \cap S^n \neq \emptyset \Leftarrow 1 \in \chi_c\left(S^n\right) \wedge \\ \operatorname{con}(c) \cap S^n \neq S^n \Leftarrow 0 \in \chi_c\left(S^n\right) \end{aligned}$$

Example 6.20 (Constraint checker)**.** Consider set $S = \{\langle 2,2\rangle, \langle 3,3\rangle\}$ and constraints $c_1 = \left[x = y\right]$, $c_2 = \left[x \neq y\right]$. Then χ_{c_1} is sound iff $1 \in \chi_{c_1}(S)$ and complete iff $\chi_{c_1}(S) = \{1\}$. Similarly, χ_{c_2} is sound iff $0 \in \chi_{c_2}(S)$ and complete iff $\chi_{c_2}(S) = \{0\}$.

Constraint checkers may be obtained directly using the image view function.

Definition 6.21 (View-based constraint checker)**.** A view-based constraint checker for a constraint $c \in C$ is a function $\check{\chi}_c : \wp\left(\mathbb{Z}^n\right) \to \wp\left(\{0,1\}\right)$ such that $\check{\chi}_c\left(S^n\right) = \varphi_c^+\left(S^n\right)$.

The following proposition states that view based constraint checkers are sound and complete constraint checkers.

Proposition 6.22. *A view-based constraint checker $\check{\chi}_c$ is a sound and complete constraint checker.*

6.3.2. Propagators

We have previously defined propagators (def. 2.35 on page 16) implementing a constraint $c \in C$ as a function $\pi_c : \wp(\mathbb{Z}^n) \to \wp(\mathbb{Z}^n)$ having the following properties:

- $\text{con}(c) \cap S^n \subseteq \pi_c(S^n)$ (soundness)
- $\pi_c(S^n) \subseteq S^n$ (contraction)

A propagator π_c which satisfies these conditions and also $\pi_c^\star(S^n) = \text{con}(c) \cap S^n$ corresponds to the strongest propagator for the constraint c - the propagator that, at fixpoint, is able to remove all tuples not satisfying the constraint. Let us represent such propagator using views.

Definition 6.23 (View-based propagator)**.** A view-based complete propagator (or filter) implementing a constraint $c \in C$ is a function $\check{\pi}_c : \wp(\mathbb{Z}^n) \to \wp(\mathbb{Z}^n)$ such that

$$\check{\pi}_c^\star\left(S^n\right) = \varphi_c^-(\{1\}) \cap S^n = \widehat{\varphi}_c\left(\{1\}, S^n\right)$$

This definition supports the following proposition.

Proposition 6.24. *A view-based complete propagator for a constraint c is a sound and complete propagator for c.*

6.4. Views over decomposable constraints

In the previous section we have shown how constraint checkers and propagators map to their view-based counterparts when the constraint is represented as a single view. This does not yet bring any practical advantages in terms of modularity or code reuse - an arbitrary constraint would still require a specific view for that constraint, as it requires a specific propagator. However, when the constraint is decomposable, a view over the constraint may be obtained from composition of other views. First we present some properties of views:

Property 6.25 (Distributivity)**.** Views distribute over set union and with set intersection. Let S_1^n, S_2^n be two arbitrary sets and $\oplus \in \{\cup, \cap\}$,

$$\begin{aligned} \varphi_f^+\left(S_1^n \oplus S_2^n\right) &= \varphi_f^+\left(S_1^n\right) \oplus \varphi_f^+\left(S_2^n\right) \\ \varphi_f^-\left(S_1^n \oplus S_2^n\right) &= \varphi_f^-\left(S_1^n\right) \oplus \varphi_f^-\left(S_2^n\right) \end{aligned}$$

Property 6.26 (Monotonicity)**.** φ_f^+ and φ_f^- are monotonic for any arbitrary function f.

Property 6.27 (Identity)**.** Let $S^n \subseteq \mathbb{Z}^n$, $S^k \subseteq \mathbb{Z}^k$, and $f : \mathbb{Z}^n \to \mathbb{Z}^k$ an arbitrary function. The following equivalences are true,

$$\begin{aligned} S^n &\subseteq \varphi_f^- \circ \varphi_f^+ \left(S^n\right) \\ S^n &= \widehat{\varphi}_f \left(\varphi_f^+ \left(S^n\right), S^n\right) = \varphi_f^- \circ \varphi_f^+ \left(S^n\right) \cap S^n \\ S^k &= \varphi_f^+ \circ \varphi_f^- \left(S^k\right) \end{aligned}$$

6.4.1. Composition of views

Views may be composed either vertically using functional composition, or horizontally using the Cartesian product of functions. A view over a composition of functions is equivalent to the composition of views over the individual functions:

Proposition 6.28. *Let $S^i \subseteq \mathbb{Z}^i$, $S^k \subseteq \mathbb{Z}^k$ denote arbitrary tuple sets, $f : \mathbb{Z}^j \to \mathbb{Z}^k$, $g : \mathbb{Z}^i \to \mathbb{Z}^j$ arbitrary functions (possibly composition of functions). Then,*

$$\begin{aligned} \varphi_{f\circ g}^+ \left(S^i\right) &= \varphi_f^+ \circ \varphi_g^+ \left(S^i\right) \\ \varphi_{f\circ g}^- \left(S^k\right) &= \varphi_g^- \circ \varphi_f^- \left(S^k\right) \end{aligned}$$

Corollary 6.29. *Let $S^i \subseteq \mathbb{Z}^i$, $S^k \subseteq \mathbb{Z}^k$ denote arbitrary tuple sets, $f : \mathbb{Z}^j \to \mathbb{Z}^k$, $g : \mathbb{Z}^i \to \mathbb{Z}^j$ arbitrary functions (possibly Cartesian product of functions). Then,*

$$\widehat{\varphi}_{f\circ g} \left(S^k, S^i\right) = \widehat{\varphi}_g \left(\widehat{\varphi}_f \left(S^k, \varphi_g^+ \left(S^i\right)\right), S^i\right)$$

Example 6.30 (Composition of views)**.** Consider expression $\left|x - y\right|$ expressed as a functional composition $f \circ g$, where $g(x_1, x_2) = x_1 - x_2$, and $f(x) = |x|$. Let $S^2 = \{\langle 2,2\rangle, \langle 1,3\rangle\}$, and $S^1 = \{2\}$. Then $\varphi_{f\circ g}^+ (S^n)$ may be obtained by $\varphi_f^+ \circ \varphi_g^+ \left(S^i\right) = \varphi_f^+ (\{0,-2\}) = \{0,2\}$. The above corollary allow us to compute $\widehat{\varphi}_{f\circ g} \left(S^1, S^2\right)$ as

$$\begin{aligned} \widehat{\varphi}_{f\circ g} \left(S^1, S^2\right) &= \widehat{\varphi}_g \left(\widehat{\varphi}_f \left(S^1, \varphi_g^+ \left(S^2\right)\right), S^2\right) \\ &= \widehat{\varphi}_g \left(\widehat{\varphi}_f \left(S^1, \{0,-2\}\right), S^2\right) \\ &= \widehat{\varphi}_g \left(\{-2\}, S^2\right) \\ &= \{\langle 1,3\rangle\} \end{aligned}$$

Figure 6.1 illustrates these operations.

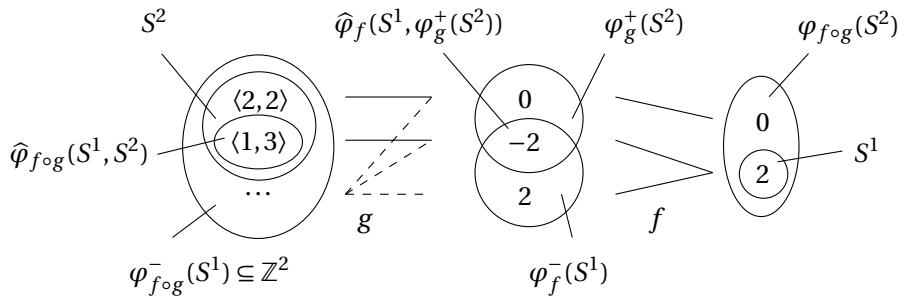


Figure 6.1.: Computations involved in the composition of views described in example 6.30.

A view over a Cartesian product of functions is not generally equivalent to the Cartesian product of views, although it is under some circunstances as we will see later. For now we will only stress that applying a view over a Cartesian product of functions is well defined:

Example 6.31 (View over Cartesian product of functions)**.** Consider set $S^2 = \{\langle 1,1\rangle, \langle 1,2\rangle\}$ and functions $f(x_1,x_2) = x_1+x_2$, $g(x_1,x_2) = x_1-x_2$. The view image function $\varphi^+_{f\times g}$ may be obtained by

$$\begin{aligned}
\varphi^+_{f\times g}\left(S^2\right) &= \left\{\left(f\times g\right)(\mathbf{x}) \in \mathbb{Z}^2 : \mathbf{x} \in S^2\right\} && \text{(def. 6.14)}\\
&= \left\{\left\langle f(\mathbf{x}), g(\mathbf{x})\right\rangle \in \mathbb{Z}^2 : \mathbf{x} \in S^2\right\} && \text{(def. 6.6)}\\
&= \{\langle 2,0\rangle, \langle 3,-1\rangle\}
\end{aligned}$$

6.4.2. Checking and propagating decomposable constraints

Constraint checkers and propagators for composed constraints may be obtained by exploring composition of views introduced in the previous section and the fact that constraint checkers and propagators may be defined using views over their associated constraints.

Proposition 6.32. *A view-based constraint checker for the constraint $c\circ f$ may be obtained by*

$$\check{\chi}_{c\circ f}\left(S^n\right) = \varphi^+_c \circ \varphi^+_f\left(S^n\right)$$

Proposition 6.33. *A complete view-based propagator for the constraint $c\circ f$ may be obtained by*

$$\check{\pi}_{c\circ f}\left(S^n\right) = \widehat{\varphi}_f\left(\pi^\star_c\left(\varphi^+_f\left(S^n\right)\right), S^n\right)$$

In this case $\check{\pi}_{c\circ f}(S^n)$ is also idempotent, i.e. $\check{\pi}_{c\circ f}(S^n) = \check{\pi}^\star_{c\circ f}(S^n)$.

Example 6.34. Consider the view-based propagator for the constraint $c\circ\left(f\times p_3\right) = [x_1+x_2 = x_3]$ obtained by the above formula, where $c = [x_1 = x_2]$, $f(\mathbf{x}) = x_1+x_2$. Let $S^3 = \{\langle 1,2,3\rangle, \langle 4,5,6\rangle\}$.

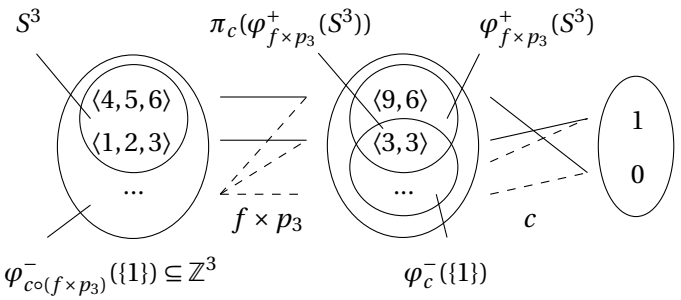


Figure 6.2.: Application of the view-based propagator for the composed constraint described in example 6.34.

Function $\varphi^+_{f\times p_3}$ applies the addition and identity operations to the original set, $\varphi^+_{f\times p_3}\left(S^3\right) = \{\langle 3,3\rangle, \langle 9,6\rangle\}$. The resulting set is filtered by propagator $\pi_c(\{\langle 3,3\rangle, \langle 9,6\rangle\}) = \{\langle 3,3\rangle\}$, and transformed back into $\widehat{\varphi}_{f\times p_3}(\{\langle 3,3\rangle\}, \{\langle 1,2,3\rangle, \langle 4,5,6\rangle\}) = \{\langle 1,2,3\rangle\}$. Figure 6.2 illustrates these operations.

The above proposition defines a complete idempotent view-based propagator $\pi_{c\circ f}$ from a composition of views with an idempotent propagator π_c. It turns out that if we relax the condition of idempotency of π_c we may still obtain a complete view-based propagator.

Proposition 6.35. *A complete constraint propagator (not necessarily idempotent) for the constraint $c \circ f$ may be obtained by*

$$\check{\pi}_{c\circ f}\left(S^n\right) = \widehat{\varphi}_f\left(\pi_c\left(\varphi^+_f\left(S^n\right)\right), S^n\right)$$

6.5. Conclusion

We have seen how views may be used to obtain sound and complete propagators and constraint checkers for decomposable constraints. Moreover, we have shown that a view over a decomposable constraint may be modeled by a composition of views. In chapter 8 we explore this property to define a repository of views over primitive expressions and a method that instantiates a view-based propagator for any decomposable constraint. In the next chapter we will focus on an important issue not yet addressed: the efficiency of view-based propagation.

Related work

- Hentenryck et al. [1992]; Carlson [1995] introduced indexicals as a tool for creating propagators for arithmetic constraints. An indexical roughly correspond to the image function φ^+ in the sense that it computes the domain of an expression. However, unlike views, indexicals

do not define the inverse transformation φ^{-} and therefore are less powerful - representing a decomposable constraint using indexicals requires the additional definition of the projection of the object function for each variable in the constraint.

- Constrained expressions [ILOG 2003a] are apparently similar to views. We are not aware of a formalization of these, but the implementation of propagators using constraint expressions share some similarities to the implementation of view-based propagators, as we will see later.

- The idea of defining view-based propagators for arbitrary constraints is incipient in [Correia et al. 2005] and [Schulte and Tack 2005]. The former presents a general overview of a constraint solver incorporating type polymorphism and its application for creating propagators from arbitrary expressions. The latter coins the term *view* (we originally called them *polymorphic constraints*), and describes how it can be used for creating generic propagator implementations, that is, propagators which can be reused for different constraints.

- Rina Dechter [Rossi et al. 2006] approaches decomposition of constraints from a different perspective. There, the full constraint network is taken into consideration in order to obtain generic global propagation algorithms with theoretical performance guarantees. The propagation is still time or space exponential but depends exclusively on specific graph-based parameters of the graph describing the constraint network. In contrast, we focus on the decomposition of a single constraint into simpler constraints for which specific propagation algorithms are applied independently. Our decomposition does not provide performance guarantees for the global search algorithm.

- The problem of creating propagators for arbitrary expressions has been approached in a different way using knowledge compilation techniques [Gent et al. 2007; Cheng and Yap 2008]. These methods create a compact extensional representation of the set of solutions to the constraint and apply a general propagation algorithm which filters tuples not found in this set. The propagation algorithm completely discards the semantics of the expression, contrasting to our approach that exploits them. It is therefore most used for constraints without any particular semantics, such as databases. Applying these algorithms for arithmetic expressions is possible, and not so inefficient in practice concerning runtime when expressions have small domains. However even the most compact extensional representation is exponential in the number of variables, which makes the method unpractical for most arithmetic constraints.

- The formalization presented above is due to Tack [2009], but has been adapted and extended in several ways to accommodate views over functions involving multiple variables. Firstly, we associate views with (sub)expressions, while in its original formalization views are associated with variables. Secondly, views were originally required to be over injective

functions, a condition we found too restrictive (in particular it excludes most multiple variable functions) and consequently discarded in our formalization. We have also introduced the Cartesian product of functions to express multiple variable views, the contracting object function $\widehat{\varphi}_f$ to guarantee that view-based propagators are contracting (which is always the case if f is injective), and view-based constraint checkers which were not contemplated in the original formalization.

Chapter 7.

Incomplete View-Based Propagation

In the previous chapter we have seen how to obtain complete view-based propagators for arbitrary constraints. Unfortunately, complete propagation algorithms for most constraints are intractable, and therefore such propagators have limited, or no practical application. Most practical propagators are incomplete, in the sense that they only filter a subset of the inconsistent values for a given constraint and domain. In this chapter we will see how sound incomplete propagators may be obtained from a composition of views with the domain approximation operators introduced in chapter 2 (§7.1). Then, it will be shown that the strength of the resulting propagator is a function of these approximations and consequently that propagators with various completeness guarantees may be obtained from views (§7.2). Furthermore, we will present an algorithm for comparing the strength of two (incomplete) view-based propagators (§7.3). For this we use a rewriting system that maps any view-based propagator into a lattice of consistencies that includes the well-known cases of domain consistency as well as the different forms of bounds consistency. This algorithm allows us to select the most efficient view-based propagator for a certain consistency type. We evaluate its adequacy on section 7.4.

7.1. $\Phi\Psi$-complete propagators

As seen in chapter 2, filtering algorithms are categorized by the consistency they are able to enforce on a constraint network - propagators which find and remove more inconsistent values are usually more expensive in terms of memory, time, or both. Practical propagators are incomplete to some degree, that is, they only guarantee that their output is stronger than some upper bound. Let us recall the definition of propagator completeness.

Definition 7.1 ($\Phi\Psi$-completeness)**.** Let S^n be an arbitrary δ-domain, $c \in C$ an arbitrary constraint, and $\Phi, \Psi \in \{\varphi, \delta, \beta\}$. A propagator π_c is $\Phi\Psi$-complete, denoted as $\pi_c^{\Phi\Psi}$, if and only if

$$\pi_c^\star\left(S^n\right) \subseteq \left[\!\left[\mathrm{con}\left(c\right) \cap \left[\!\left[S^n\right]\!\right]^\Phi\right]\!\right]^\Psi \cap S^n$$

We have already introduced some common completeness classes, namely domain, range, bounds(D), bounds($\mathbb{Z}$), and bounds($\mathbb{R}$). All these classes except the latter map to instantiations of the above parametrized expression as summarized in table 7.1.

Upper bound	Propagator strength	$\Phi\Psi$-completeness
$\llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^{\delta} \rrbracket^{\delta} \cap S^n$	domain complete	$\delta\delta$-complete
$\llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^{\delta} \rrbracket^{\beta} \cap S^n$	bounds(D) complete	$\delta\beta$-complete
$\llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^{\beta} \rrbracket^{\delta} \cap S^n$	range complete	$\beta\delta$-complete
$\llbracket \mathrm{con}(c) \cap \llbracket S^n \rrbracket^{\beta} \rrbracket^{\beta} \cap S^n$	bounds($\mathbb{Z}$) complete	$\beta\beta$-complete

Table 7.1.: Constraint propagator completeness.

7.2. View models

View models provide an approximation to complete view based propagation as introduced in the previous chapter.

Definition 7.2 (View approximations)**.** Let $f : \mathbb{Z}^n \to \mathbb{Z}^k$ be an arbitrary function, S^n an arbitrary tuple set, and $\Phi, \Psi \in \{\varphi, \delta, \beta\}$. We define view functions that are always input a given domain type, and that are always relaxed to a certain domain type, and denote them as follows:

$$\begin{aligned} \Phi_f^+ \underset{\Psi}{\rightarrow} &= \llbracket \varphi_f^+ \left(\llbracket S^n \rrbracket^{\Phi} \right) \rrbracket^{\Psi} \\ \underset{\Phi}{\rightarrow} \Psi_f^- &= \llbracket \varphi_f^- \left(\llbracket S^n \rrbracket^{\Phi} \right) \rrbracket^{\Psi} \\ \underset{\Phi}{\rightarrow} \widehat{\Psi}_f &= \llbracket \widehat{\varphi}_f \left(\llbracket S^k \rrbracket^{\Phi}, S^n \right) \rrbracket^{\Psi} \end{aligned}$$

We define an arbitrary approximation for a view-based propagator $\check{\pi}_{c \circ f}$, called a *view model* for $c \circ f$, by applying domain approximations to the result of every operation involved in a complete view-based propagator defined in the previous chapter,

$$\check{\pi}_{c \circ f}\left(S^n\right) = \widehat{\varphi}_f \left(\pi_c^{\star} \circ \varphi_f^+ \left(S^n\right), S^n \right)$$

Specifically, since both the input and output of the view functions φ_f^+ and $\widehat{\varphi}_f$ may be approximated, and we may use a $\Phi\Psi$-complete propagator for c instead of a complete propagator, a view model has at least six approximation operators involved,

$$\langle \Phi_1, \ldots, \Phi_6 \rangle^{c \circ f} = \llbracket \widehat{\varphi}_f \left(\llbracket \pi_c^{\Phi_3 \Phi_4 \star} \left(\llbracket \varphi_f^+ \left(\llbracket S^n \rrbracket^{\Phi_1} \right) \rrbracket^{\Phi_2} \right) \rrbracket^{\Phi_5}, \llbracket S^n \rrbracket^{\Phi_1} \right) \rrbracket^{\Phi_6} \cap S^n$$

which is more conveniently represented using the notation introduced by definition 7.2:

$$\langle \Phi_1, \ldots, \Phi_6 \rangle^{c \circ f} = \Phi_{1f}^+ \underset{\Phi_2}{\rightarrow} \pi_c^{\Phi_3 \Phi_4 \star} \underset{\Phi_5}{\rightarrow} \widehat{\Phi}_{6f}$$

Additionally, a view model is parametrized by two extra approximation operators guaranteeing that the input and output tuple sets of the full propagator have a known domain type.

This takes into account the fact that common propagators work with specific domain types (e.g. δ-domains) and not arbitrary tuple sets. Let us then formalize view models.

Definition 7.3 (View model)**.** Let $\Phi_0,\ldots,\Phi_7 \in \{\varphi,\delta,\beta\}$ be a set of approximation operators. Any instance of the following functional composition is called a view model for the constraint $c\circ f$,

$$\langle\Phi_0,\ldots,\Phi_7\rangle^{c\circ f} \quad = \quad \underset{\Phi_0}{\rightarrow}\,\Phi^{+}_{1f}\,\underset{\Phi_2}{\rightarrow}\,\pi_c^{\Phi_3\Phi_4\star}\,\underset{\Phi_5}{\rightarrow}\,\widehat{\Phi}_{6f}\,\underset{\Phi_7}{\rightarrow}$$

The operational semantics of a view model should be clear: given a Φ_0-domain S^n, the propagator first applies the approximated image function $\Phi^{+}_{1f}\underset{\Phi_2}{\rightarrow}$, then the idempotent propagator $\pi_c^{\Phi_3\Phi_4\star}$ for the constraint c and finally function $\underset{\Phi_5}{\rightarrow}\,\widehat{\Phi}_{6f}$ whose result is intersected with S^n and relaxed to a Φ_7-domain.

Example 7.4. Consider the constraint $c\circ f = \left[2x+3y=z\right]$ composed from constraint $c = \left[x=y\right]$ and function $f\left(x,y,z\right) = \left\langle 2x+3y,z\right\rangle$. Let $S = \{0,1\}\times\{0,1\}\times\{0,3\}$, and m be a view model for $c\circ f$ defined as follows,

$$m \quad = \quad \underset{\delta}{\rightarrow}\,\delta^{+}_{f}\,\underset{\varphi}{\rightarrow}\,\pi_c^{\beta\beta\star}\,\underset{\varphi}{\rightarrow}\,\widehat{\beta}_{f}\,\underset{\delta}{\rightarrow}$$

Propagating m involves the following computations,

$$\begin{aligned}
\left[\!\left[\varphi^{+}_{f}\left(\left[\!\left[S\right]\!\right]^{\delta}\right)\right]\!\right]^{\varphi} &= \{0,2,3,5\}\times\{0,3\}\\
\pi_c^{\beta\beta\star}\left(\{0,2,3,5\}\times\{0,3\}\right) &= \{0,2,3\}\times\{0,3\}\\
\left[\!\left[\widehat{\varphi}_f\left(\left[\!\left[\{0,2,3\}\times\{0,3\}\right]\!\right]^{\varphi},S\right)\right]\!\right]^{\beta}\cap S &= \{0,1\}\times\{0,1\}\times\{0,3\}
\end{aligned}$$

leading to no filtering.

7.2.1. Soundness

A view model is a relaxation of a complete view-based propagator. Since complete view-based propagators are sound propagators then view models must also be sound propagators.

Proposition 7.5. *Let $\Phi_0,\ldots,\Phi_7 \in \{\varphi,\delta,\beta\}$ be a set of approximation operators. Any view model $\langle\Phi_0,\ldots,\Phi_7\rangle^{c\circ f}$ is a sound propagator for $c\circ f$.*

7.2.2. Completeness

Deciding completeness of an arbitrary view model is not trivial. However, the completeness of a specific set of view models can be easily mapped to one of the completeness classes introduced in section 7.1. Let us first focus on view models belonging to this set, called $\Phi\Psi$ view models.

Definition 7.6 ($\Phi\Psi$ view model)**.** Let $\Phi, \Psi \in \{\varphi, \delta, \beta\}$. A $\Phi\Psi$ view model for a constraint $c \circ f$ is any view model m such that

$$\begin{aligned} m &= \llbracket \widehat{\varphi}_f \left(\pi_c^\star \circ \varphi_f^+ \left(\llbracket S^n \rrbracket^\Phi \right), \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n \\ &= \underset{\delta}{\rightarrow} \Phi_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \widehat{\Psi}_f \underset{\delta}{\rightarrow} \end{aligned}$$

In other words, a $\Phi\Psi$ view model is a model for propagating δ-domains which only approximates the input of the image view function and the output of the object view function. When a view model is a $\Phi\Psi$ view model, then its completeness is given by the following proposition.

Proposition 7.7. *A $\Phi\Psi$ view model for a constraint $c \circ f$ is a $\Phi\Psi$-complete propagator for $c \circ f$. Moreover, it is also an idempotent propagator, i.e.*

$$\underset{\delta}{\rightarrow} \Phi_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \widehat{\Psi}_f \underset{\delta}{\rightarrow} \quad = \quad \check{\pi}_{c\circ f}^{\Phi\Psi} = \check{\pi}_{c\circ f}^{\Phi\Psi\star}$$

Example 7.8. Consider the constraint and tuple set S from the previous example and the following view model,

$$n \quad = \quad \underset{\delta}{\rightarrow} \delta_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \widehat{\beta}_f \underset{\delta}{\rightarrow}$$

Propagating n involves the following computations,

$$\begin{aligned} \llbracket \varphi_f^+ \left(\llbracket S \rrbracket^\delta \right) \rrbracket^\varphi &= \{0,2,3,5\} \times \{0,3\} \\ \pi_c^{\varphi\varphi\star} \left(\{0,2,3,5\} \times \{0,3\} \right) &= \{0,3\} \times \{0,3\} \\ \llbracket \widehat{\varphi}_f \left(\llbracket \{0,3\} \times \{0,3\} \rrbracket^\varphi, S \right) \rrbracket^\beta \cap S &= \{0\} \times \{0,1\} \times \{0,3\} \end{aligned}$$

This view model is $\delta\beta$-complete (it is one of the examples illustrating the propagator strength taxonomy given in fig. 2.3 on page 18), unlike the view model of the previous example since $n(S) \subset m(S)$.

By definition, a $\Phi\Psi$ view model for a constraint $c \circ f$ uses an idempotent propagator for c. The idempotency of the propagator for c is a sufficient condition for guaranteeing $\Phi\Psi$-completeness of a $\Phi\Psi$ view model for $c \circ f$. The following definition relaxes the idempotency condition.

Definition 7.9 ($\Phi\Psi$ relaxed view model)**.** Let $\Phi, \Psi \in \{\varphi, \delta, \beta\}$. A $\Phi\Psi$ relaxed view model for a constraint $c \circ f$ is any view model m such that

$$\begin{aligned} m &= \llbracket \widehat{\varphi}_f \left(\pi_c \circ \varphi_f^+ \left(\llbracket S^n \rrbracket^\Phi \right), \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n \\ &= \underset{\delta}{\rightarrow} \Phi_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi} \underset{\varphi}{\rightarrow} \widehat{\Psi}_f \underset{\delta}{\rightarrow} \end{aligned}$$

A $\Phi\Psi$ relaxed view model for $c \circ f$ may or may not be $\Phi\Psi$-complete, according to the following proposition.

Proposition 7.10. *Let S be an arbitrary tuple set and m a $\Phi\Psi$ relaxed view model for $c \circ f$. Then m is $\Phi\Psi$-complete if and only if any pruning achieved by the propagator for c is preserved by all involved view functions and approximations, formally*

$$\pi_c\left(\varphi_f^+\left(\llbracket S \rrbracket^\Phi\right)\right) \quad \supseteq \quad \varphi_f^+\left(\llbracket m^\star(S) \rrbracket^\Phi\right)$$

Example 7.11. Consider the constraint $c \circ f = [x \geq 4]$ where $c = [x \geq 4]$ and $f(x) = x$. Additionally, let π_c be a non-idempotent propagator defined as follows:

$$\pi_c(S) = \{x \in S : x \geq 4 \vee x < \max(S \setminus [4 \ldots \infty])\}$$

To see that propagator π_c is sound and complete consider for example $S = [1 \ldots 4]$. Then $\pi_c(S) = \{1,2,4\}$, $\pi_c \circ \pi_c(S) = \{1,4\}$ and $\pi_c \circ \pi_c \circ \pi_c(S) = \pi_c^\star(S) = \{4\}$.

Let p be a $\beta\beta$ view model and q a $\beta\beta$ relaxed view model for the constraint $c \circ f$ obtained respectively from $\pi_c^\star$ and π_c:

$$\begin{aligned} p(S) &= \llbracket \widehat{\varphi}_f\left(\pi_c^\star \circ \varphi_f^+\left(\llbracket S \rrbracket^\beta\right), \llbracket S \rrbracket^\beta\right) \rrbracket^\beta \cap S \\ q(S) &= \llbracket \widehat{\varphi}_f\left(\pi_c \circ \varphi_f^+\left(\llbracket S \rrbracket^\beta\right), \llbracket S \rrbracket^\beta\right) \rrbracket^\beta \cap S \end{aligned}$$

Propagator p is bounds($\mathbb{Z}$) complete by proposition 7.7, and infers $p(S) = \{4\}$. Propagator q is not bounds($\mathbb{Z}$) complete by proposition 7.10:

$$\begin{aligned} \pi_c\left(\varphi_f^+\left(\llbracket S \rrbracket^\beta\right)\right) \quad &\not\supseteq \quad \varphi_f^+\left(\llbracket \check{\pi}_{c\circ f}^{\beta\beta\star}(S) \rrbracket^\beta\right) \\ \{1,2,4\} \quad &\not\supseteq \quad [1 \ldots 4] \end{aligned}$$

In particular it does not do any filtering for S, i.e. $q(S) = S$.

Finally we remark that soundness of $\Phi\Psi$ view models is not affected by idempotency since for any $S^n \subseteq \mathbb{Z}^n$,

$$\llbracket \widehat{\varphi}_f\left(\pi_c^\star \circ \varphi_f^+\left(\llbracket S^n \rrbracket^\Phi\right), \llbracket S^n \rrbracket^\Phi\right) \rrbracket^\Psi \cap S^n \quad \subseteq \quad \llbracket \widehat{\varphi}_f\left(\pi_c \circ \varphi_f^+\left(\llbracket S^n \rrbracket^\Phi\right), \llbracket S^n \rrbracket^\Phi\right) \rrbracket^\Psi \cap S^n$$

7.2.3. Idempotency

Let us go back to non-relaxed view models, that is view models which use an idempotent propagator for c. Proposition 7.7 tells us that any $\Phi\Psi$ view model for $c \circ f$ that uses an idempotent propagator π_c is also idempotent. Unfortunately, this is not true for a general view model for $c \circ f$.

Proposition 7.12. *Let $m = \langle \Phi_0, \ldots, \Phi_7 \rangle$ be a (non-relaxed) view model for an arbitrary constraint $c \circ f$, and S an arbitrary Φ_0-domain. Then m may be not idempotent for S.*

This can be shown by the following example.

Example 7.13. Let $c \circ f = [2x_1 = 2x_2 + 1]$, where $f(x_1, x_2) = \langle 2x_1, 2x_2 + 1 \rangle$, and $c = [x_1 = x_2]$. Consider the δ-domain $S = [1 \ldots 4] \times [1 \ldots 4]$ and the following view model for propagating $c \circ f$,

$$m \quad = \quad \underset{\delta}{\rightarrow} \beta_f^+ \underset{\beta}{\rightarrow} \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow} \widehat{\beta}_f \underset{\delta}{\rightarrow}$$

Applying m once gives us $m(S) = [2 \ldots 4] \times [1 \ldots 3]$ which is not a fixpoint for m (the constraint is unsatisfiable and m is able to prove it at fixpoint).

Knowing whether a propagator is idempotent for a given domain is used on some systems for optimizing propagator execution scheduling, as seen in chapter 3. We note that a propagator scheduler that takes into account idempotency of a view model for a constraint $c \circ f$ by querying the idempotency of the propagator for c is safe but may loose propagation - it could possibly fail to reexecute a non-idempotent view model.

7.2.4. Efficiency

We have not yet addressed the efficiency of view-based propagation. In general, computing and representing views over many interesting functions is intractable in time and space. However, when the input or output (or both) sets of a view have some special structure, computing and representing views is polynomial, as is the case for many useful views.

Example 7.14 (Tractable view functions)**.** Let $f(\mathbf{x}^n) = \Sigma_{i=1}^n x_i$. The runtime cost of computing $\varphi_f^+(S^n)$ for an arbitrary tuple set $S^n \subseteq \mathbb{Z}^n$, and the space cost for representing $\widehat{\varphi}_f(c, S^n)$ for some constant are exponential in n. However, the cost of computing $\varphi_f^+\left(\llbracket S^n \rrbracket^\beta\right)$ and representing $\llbracket \widehat{\varphi}_f\left(c, \llbracket S^n \rrbracket^\beta\right) \rrbracket^\beta$ is linear in n.

The fact that some view-based propagators are more efficient than others is not surprising since the same is true for ordinary (i.e. non-view-based) propagators. Given that the set of view models is a superset of $\Phi\Psi$-complete propagators, view models indeed provide an extra degree of freedom for achieving a tradeoff between completeness and efficiency. Additionally, the declarative definition of a view model may expose a performance bottleneck which may be avoided by using an alternative view approximation.

Example 7.15. A bounds($\mathbb{Z}$) complete propagator for the constraint $x_1 + x_2 = x_3$ may be represented by the following view model,

$$\underset{\delta}{\rightarrow} \beta_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow}$$

where $c(x) = [x_1 = x_2]$, $f(x) = \langle x_1 + x_2, x_3 \rangle$. It turns out that it may also be obtained by an alternative view model, which makes use of a much more efficient propagator for c,

$$\underset{\delta}{\rightarrow} \beta_f^+ \underset{\beta}{\rightarrow} \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow}$$

The fact that both models are equivalent is easy to see in this case. Although propagator $\pi_c^{\beta\beta}$ is not equivalent to $\pi_c^{\varphi\varphi}$ in general, the equivalence stands when the input domain is already a β-domain. This is the case here since the output of $\beta_f^+ \underset{\varphi}{\rightarrow}$ is always a β-domain, that is,

$$\beta_f^+ \underset{\varphi}{\rightarrow} \quad = \quad \beta_f^+ \underset{\beta}{\rightarrow}$$

The above example shows a view model which happens to be a $\Phi\Psi$ view model, hence $\Phi\Psi$-complete. However, example 7.4 has shown us that view models are not $\Phi\Psi$-complete in general. The problem of quantifying the strength of a view model, or deciding if a given model is stronger than another is not trivial for many view models. The next section addresses this problem.

7.3. Finding stronger models

Both the problem of comparing view models and the problem of quantifying the strength of a given view model m may be addressed by a procedure which finds the sets L_m, U_m, of view models stronger and weaker than m:

$$\begin{aligned} L_m &= \{b : b \preceq m\} \\ U_m &= \{b : m \preceq b\} \end{aligned}$$

Then, comparing two view models m_1, m_2, may be accomplished by testing $m_1 \in L_{m_2}$, $m_1 \in U_{m_2}$ or vice-versa. The problem of quantifying the strength of a view model m may be reduced to this problem by comparing m with a $\Phi\Psi$ view model, which as we have seen are $\Phi\Psi$-complete propagators. While finding stronger view models is not straightforward in general, a large number of models stronger than a given view model may be easily found.

7.3.1. Trivially stronger models

Definition 7.16. Given two arbitrary functions $f, g : \wp(\mathbb{Z}^n) \to \wp\left(\mathbb{Z}^k\right)$, f is *stronger* than g, written $f \preceq g$, if and only if $f(S^n) \subseteq g(S^n)$ for any domain $S^n \subseteq \mathbb{Z}^n$.

Proposition 7.17. *Let $f_1, \ldots, f_k, g : \wp(\mathbb{Z}^n) \to \wp(\mathbb{Z}^n)$ be arbitrary monotonic functions. Then for*

any $1 \leq i \leq j \leq k$,

$$f_i \circ \ldots \circ f_j \preceq g \quad \Rightarrow \quad f_1 \circ \ldots \circ f_k \preceq f_1 \circ \ldots \circ f_{i-1} \circ g \circ f_{j+1} \circ \ldots \circ f_k$$

A view model is by definition a functional composition involving only monotonic functions and operators: φ^+, φ^-, π, $[\![\cdot]\!]^\Phi$, and $\cap$. A consequence of the above proposition is that we will obtain a stronger view model if we replace any subset of these functions by a stronger subset. We will use this result in two ways. Firstly, to establish a natural partial ordering among view models.

Corollary 7.18. *For any constraint c, function f, and approximation operators* $\Phi_1, \ldots, \Phi_k$, *and* Θ,

$$\Phi_i \preceq \Theta \Rightarrow \langle \Phi_1, \ldots, \Phi_n \rangle \preceq \langle \Phi_1, \ldots, \Phi_{i-1}, \Theta, \Phi_{i+1}, \ldots, \Phi_n \rangle$$

Recall that approximation operators are totally ordered: $\varphi \preceq \delta \preceq \beta$ (see lemma 2.24). Therefore for a given view model we may obtain a set of trivially stronger (resp. weaker) models just by replacing approximation operators by stronger (resp. weaker) approximation operators.

Definition 7.19. Let $m_1 = \langle \Phi_1, \ldots, \Phi_8 \rangle$, $m_2 = \langle \Theta_1, \ldots, \Theta_8 \rangle$ be two view models. m_1 is *trivially stronger* than m_2, written $m_1 \preceq^T m_2$, if and only if $\Phi_i \preceq \Theta_i$, $1 \leq i \leq 8$. Additionally, we will denote the set of trivially stronger (resp. weaker) models of a given model m by $\lfloor m \rfloor^T = \{b : b \preceq^T m\}$ (resp. $\lceil m \rceil^T = \{b : m \preceq^T b\}$).

The trivially stronger relation orders all possible view models for a constraint in a lattice, shown in figure 7.1.

Proposition 7.20. *The set of view models for a constraint ordered by the trivially stronger relation is a bounded lattice where* $\top = \langle \beta, \beta, \ldots, \beta \rangle$ *and* $\bot = \langle \varphi, \varphi, \ldots, \varphi \rangle$.

Example 7.21. The $\beta\beta$-complete propagator for the constraint $[x_1 + x_2 = x_3]$ of example 7.15 is trivially stronger than the presented alternative view model but the converse is not true, that is

$$\begin{array}{lll} \underset{\delta}{\rightarrow} \beta_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow} & \preceq^T & \underset{\delta}{\rightarrow} \beta_f^+ \underset{\beta}{\rightarrow} \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow} \\ \underset{\delta}{\rightarrow} \beta_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow} & \not\preceq^T & \underset{\delta}{\rightarrow} \beta_f^+ \underset{\beta}{\rightarrow} \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow} \end{array}$$

This means that the model in the left hand side of the equation may be reached in the lattice from the node corresponding to the model in the right hand side by moving downwards into the trivially stronger lattice (or vice-versa). However, for this specific constraint we have

$$\underset{\delta}{\rightarrow} \beta_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow} \quad \succeq \quad \underset{\delta}{\rightarrow} \beta_f^+ \underset{\beta}{\rightarrow} \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow} \beta_f^- \underset{\delta}{\rightarrow}$$

which is not captured in the lattice since it is not a trivially stronger relation.

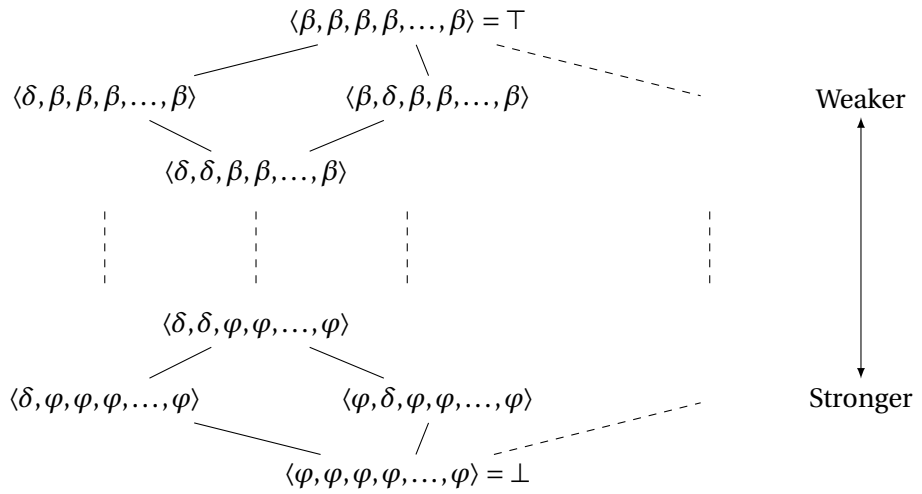


Figure 7.1.: View model lattice.

7.3.2. Relaxing the problem

For a given view model m we are interested in the sets of view models stronger and weaker than m. We will present a procedure that does not compute L_m, U_m exactly, but instead provides an approximation.

Definition 7.22 (Approximation bound)**.** Let m be an arbitrary view model. Then the approximated lower bound of m, $\widetilde{L}_m$, is a set of view models where $b \in \widetilde{L}_m \Rightarrow b \preceq m$. The approximated upper bound of m, $\widetilde{U}_m$ is similarly defined as $b \in \widetilde{U}_m \Rightarrow b \succeq m$.

The sets $\widetilde{L}_m = \{\bot\}$ and $\widetilde{U}_m = \{\top\}$ are trivial bounds for any view model, but are of course too loose. A procedure which finds tighter upper bounds for view models of many useful constraints is detailed in the following section. The algorithm for computing the lower bound is analogous.

7.3.3. Computing an upper bound

Function `FindUb` takes as input a view model m for a constraint $c \circ f$ and returns a set of models $\widetilde{U}_m$. The method makes use of a rule database $R_{c\circ f}$.

Definition 7.23 (Rule database)**.** A rule database $R_{c\circ f}$ for a constraint $c \circ f$ is a set of rules where each rule $\langle m_1, m_2\rangle \in R_{c\circ f}$ is a tuple representing a non-trivial order relation between two view models, that is $m_1 \preceq m_2$ although $m_1 \not\preceq^T m_2$. The rule database is sound iff $\langle m_1, m_2\rangle \in R_{c\circ f} \Rightarrow m_1 \preceq m_2$. It is complete iff $m_1 \preceq m_2 \Rightarrow \textsc{HasPath}\left(R_{c\circ f}, m_1, m_2\right)$ for any non-trivial order relation

Function `FindUb(`m`)`

Data: $R_{c\circ f}$ is a rule database for the constraint $c \circ f$
Input: A view model m for the constraint $c \circ f$
Output: A set of view models $\widetilde{U}_m$ for the constraint $c \circ f$

```
1  U ← ⌈m⌉^T
2  repeat
3  |   changed ← false
4  |   foreach ⟨m1, m2⟩ ∈ R_{c∘f} do
5  |   |   if m1 ∈ U then
6  |   |   |   U ← U ∪ ⌈m2⌉^T
7  |   |   |   changed ← true
8  |   |
9  |
10 until ¬changed
11 return U
```

$m_1 \preceq m_2$, where HASPATH is defined as

$$\begin{aligned}\textsc{hasPath}\left(R_{c\circ f}, m_1, m_2\right) \quad &\Leftrightarrow \quad m_1 = m_2 \\ &\vee \quad \exists m_i : m_1 \preceq^T m_i \wedge \textsc{hasPath}\left(R_{c\circ f}, m_i, m_2\right) \\ &\vee \quad \exists m_i : \langle m_1, m_i\rangle \in R_{c\circ f} \wedge \textsc{hasPath}\left(R_{c\circ f}, m_i, m_2\right)\end{aligned}$$

The algorithm computes the transitive closure of the given set of rules $R_{c\circ f}$ unioned with the set of trivial rules. It performs a breadth first search from the node representing the given view model, traversing the lattice of models downwards (i.e. in the *stronger than* direction), until there are no more applicable rules in the database. The algorithm is basically an exhaustive rule rewriting procedure, for which the following is easy to see.

Proposition 7.24. *Algorithm `FindUb` is sound if the rules database $R_{c\circ f}$ is sound. It is complete if the rules database $R_{c\circ f}$ is complete.*

7.3.4. Rule databases

Algorithm `FindUb` requires a database of rules $R_{c\circ f}$ which is specific to the constraint $c \circ f$. Creating a rule database for any constraint is of course unpractical. Instead we create rule databases for primitive constraints R_c and expressions R_f and use them to obtain $R_{c\circ f}$.

Definition 7.25 (Constraint rule database)**.** Let c be an arbitrary constraint and R_c be a constraint rule database for c. Then each rule $\langle m_1, m_2\rangle \in R_c$ specifies a non-trivial order relation between two propagators for c, $m_1 = \underset{\Phi_1}{\rightarrow} \pi_c^{\Phi_2\Phi_3} \underset{\Phi_4}{\rightarrow}$ and $m_2 = \underset{\Theta_1}{\rightarrow} \pi_c^{\Theta_2\Theta_3} \underset{\Theta_4}{\rightarrow}$.

Integrating a constraint rule database in a rule database is straightforward due to the following corollary to proposition 7.17,

Corollary 7.26. *For any constraint c, function f, and approximation operators* Φ_i,Θ_i,

$$\begin{array}{rcl} \underset{\Phi_3}{\rightarrow} \pi_c^{\Phi_4\Phi_5} \underset{\Phi_6}{\rightarrow} & \preceq & \underset{\Theta_3}{\rightarrow} \pi_c^{\Theta_4\Theta_5} \underset{\Theta_6}{\rightarrow} \\ & \Rightarrow & \\ \underset{\Phi_1}{\rightarrow} \Phi_{2f}^{+} \underset{\Phi_3}{\rightarrow} \pi_c^{\Phi_4\Phi_5} \underset{\Phi_6}{\rightarrow} \Phi_{7f}^{-} \underset{\Phi_8}{\rightarrow} & \preceq & \underset{\Phi_1}{\rightarrow} \Phi_{2f}^{+} \underset{\Theta_3}{\rightarrow} \pi_c^{\Theta_4\Theta_5} \underset{\Theta_6}{\rightarrow} \Phi_{7f}^{-} \underset{\Phi_8}{\rightarrow} \end{array}$$

Definition 7.27 (Function rule database)**.** Let f be an arbitrary function and R_f be a function rule database for f. Then each rule $\langle m_1, m_2\rangle \in R_f$ specifies a non-trivial order relation between two view models, $m_1 = \langle \Phi_1, \ldots, \Phi_8\rangle^{x\circ f}$ and $m_2 = \langle \Theta_1, \ldots, \Theta_8\rangle^{x\circ f}$, where x is an arbitrary constraint.

Note that a function rule database abstracts the semantics of the constraint used with the view - rules must be applicable to any constraint. For obtaining a tight upper bound, the algorithm benefits from the specificity of the rules, which is increased when the function and constraint rule databases for the given view model are combined.

Proposition 7.28. *Let R_c be a constraint rule database for a constraint c, R_f a function rule database for a function f , let*

$$\begin{array}{rcl} R_c^{+} & = & \{\langle t_1, t_2\rangle : \\ & & t_1 = \langle \Phi_1, \Phi_2, m_1, \Phi_3, \Phi_4\rangle, t_2 = \langle \Phi_1, \Phi_2, m_2, \Phi_3, \Phi_4\rangle, \\ & & \Phi_1, \ldots, \Phi_4 \in \{\varphi, \beta, \delta\}, \\ & & \langle m_1, m_2\rangle \in R_c\} \end{array}$$

Then $R_{c\circ f} = R_c^{+} \cup R_f$ is a sound rule database for the constraint $c \circ f$.

Example 7.29. Consider the constraint $c\circ f = [ax_1 \geq x_2]$, where $c = [x_1 \geq x_2]$ and $f(\mathbf{x}) = \langle ax_1, x_2\rangle$, and a is a positive constant. Let R_c be a rule database for c defined as follows:

$$R_c = \begin{array}{rcll} \underset{\beta}{\rightarrow} \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow} & \preceq & \underset{\beta}{\rightarrow} \pi_c^{\beta\delta\star} \underset{\beta}{\rightarrow} & \text{(rule 1)} \\ \underset{\beta}{\rightarrow} \pi_c^{\beta\delta\star} \underset{\beta}{\rightarrow} & \preceq & \underset{\beta}{\rightarrow} \pi_c^{\beta\delta\star} \underset{\delta}{\rightarrow} & \text{(rule 2)} \\ \underset{\delta}{\rightarrow} \pi_c^{\beta\delta\star} \underset{\delta}{\rightarrow} & \preceq & \underset{\delta}{\rightarrow} \pi_c^{\beta\varphi\star} \underset{\delta}{\rightarrow} & \text{(rule 3)} \end{array}$$

The first rule states that a $\beta\beta$ complete propagator for c achieves the same pruning that a $\beta\delta$-complete propagator for c when its input is a β-domain and its output is stored as a β-domain. The second rule specifies that the output of $\pi_c^{\beta\delta\star}(S)$ when S is a β-domain is always a

β-domain. The third rule states that a $\beta\delta$ (range)-complete propagator for c achieves the same pruning that a $\beta\varphi$-complete propagator for c.

Let R_f be a rule database for f defined as follows:

$$R_f = \begin{array}{lcll} \underset{\delta}{\rightarrow}\beta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\beta\star}\underset{\beta}{\rightarrow}\beta^-_f\underset{\delta}{\rightarrow} & \preceq & \underset{\delta}{\rightarrow}\delta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\beta\star}\underset{\beta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(rule 1)} \\ \underset{\delta}{\rightarrow}\delta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\delta\star}\underset{\delta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \preceq & \underset{\delta}{\rightarrow}\delta^+_f\underset{\delta}{\rightarrow}\pi^{\beta\delta\star}\underset{\delta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(rule 2)} \\ \underset{\delta}{\rightarrow}\delta^+_f\underset{\delta}{\rightarrow}\pi^{\beta\varphi\star}\underset{\delta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \preceq & \underset{\delta}{\rightarrow}\delta^+_f\underset{\varphi}{\rightarrow}\pi^{\varphi\varphi\star}\underset{\varphi}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(rule 3)} \end{array}$$

The first rule specifies that if the output of the image view function and the input of the object view function are β-domains then the input of the image view function and the output of the object view function may be relaxed to β-domains with no loss of representation. The second rule states that when using a $\pi^{\beta\delta\star}$ propagator then the output of the image view function for f may be relaxed to a β-domain with no loss of representation. The third rule specifies that a domain complete propagator for $\pi_{x\circ f}$ for any constraint x achieves the same pruning that a view model using a $\beta\varphi$-complete propagator for x.

Now consider the following model m for propagating $c\circ f$,

$$m = \underset{\delta}{\rightarrow}\beta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\beta\star}_c\underset{\beta}{\rightarrow}\beta^-_f\underset{\delta}{\rightarrow}$$

Finding the transitive closure of this model under the combined set of rules $R_{c\circ f}$ proves that m is a domain complete propagator for $c\circ f$. A possible trace of the algorithm is the following,

$$\begin{array}{lr} m = \underset{\delta}{\rightarrow}\beta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\beta\star}_c\underset{\beta}{\rightarrow}\beta^-_f\underset{\delta}{\rightarrow} & \\ = \underset{\delta}{\rightarrow}\delta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\beta\star}_c\underset{\beta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(from rule 1 in } R_f\text{)} \\ = \underset{\delta}{\rightarrow}\delta^+_f\underset{\beta}{\rightarrow}\pi^{\beta\delta\star}_c\underset{\delta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(from rules 1 and 2 in } R^+_c\text{)} \\ = \underset{\delta}{\rightarrow}\delta^+_f\underset{\delta}{\rightarrow}\pi^{\beta\delta\star}_c\underset{\delta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(from rule 2 in } R_f\text{)} \\ = \underset{\delta}{\rightarrow}\delta^+_f\underset{\delta}{\rightarrow}\pi^{\beta\varphi\star}_c\underset{\delta}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(from rule 3 in } R^+_c\text{)} \\ = \underset{\delta}{\rightarrow}\delta^+_f\underset{\varphi}{\rightarrow}\pi^{\varphi\varphi\star}_c\underset{\varphi}{\rightarrow}\delta^-_f\underset{\delta}{\rightarrow} & \text{(from rule 3 in } R_f\text{)} \\ = \pi^{\delta\delta\star}_{c\circ f} & \text{(prop. 7.7)} \end{array}$$

7.3.5. Multiple views (functional composition)

As seen in the previous chapter, view-based propagators for composed constraints may be obtained by exploring composition of views. In such cases, the corresponding view models

will use multiple views, contrasting with the single view models we have been considering so far.

Definition 7.30. Let f, g be arbitrary functions, and c a constraint. A view model for the constraint $c \circ f \circ g$ is characterized by the following functional composition,

$$\langle \Phi_1, \ldots, \Phi_{12} \rangle \quad = \quad \underset{\Phi_1}{\rightarrow} \Phi^+_{2g} \underset{\Phi_3}{\rightarrow} \Phi^+_{4f} \underset{\Phi_5}{\rightarrow} \pi_c^{\Phi_6\Phi_7} \underset{\Phi_8}{\rightarrow} \Phi^-_{9f} \underset{\Phi_{10}}{\rightarrow} \Phi^-_{11g} \underset{\Phi_{12}}{\rightarrow}$$

The previous algorithm has therefore to be adapted for handling view models parametrized by a variable number of approximation operators. This may be accomplished for any constraint $c \circ f \circ g$ using an adequate rule database $R_{c\circ f\circ g}$. We will not provide the counterpart of proposition 7.28 for this case, but instead remark that it basically explores the following lemma to map view models involving multiple views to the simpler case presented previously.

Lemma 7.31. *Let f, g be arbitrary functions and c a constraint. Then,*

$$\begin{array}{c} \underset{\Phi_1}{\rightarrow} \Phi^+_{2g} \underset{\Phi_3}{\rightarrow} \Phi^+_{4f} \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi\star} \underset{\varphi}{\rightarrow} \Phi^-_{5f} \underset{\Phi_6}{\rightarrow} \Phi^-_{7g} \underset{\Phi_8}{\rightarrow} \\ = \\ \underset{\Phi_1}{\rightarrow} \Phi^+_{2g} \underset{\Phi_3}{\rightarrow} \pi_{c\circ f}^{\Phi_4\Phi_5\star} \underset{\Phi_6}{\rightarrow} \Phi^-_{7g} \underset{\Phi_8}{\rightarrow} \end{array}$$

7.3.6. Multiple views (Cartesian composition)

As discussed, constraints that involve Cartesian product of functions naturally express many useful relations. In this case, the corresponding view based propagator may be obtained from approximated views over a Cartesian product of functions. An approximated view over a Cartesian product of functions may in turn be defined from approximated views over these functions.

Proposition 7.32. *Let $S^n \subseteq \mathbb{Z}^n$, $S^i = proj_{1\ldots i}\left(S^k\right)$, and $S^j = proj_{i+1\ldots i+j}\left(S^k\right)$. Let $f : \mathbb{Z}^n \to \mathbb{Z}^i$, $g : \mathbb{Z}^n \to \mathbb{Z}^j$ be two arbitrary functions, and $\Phi, \Psi \in \{\varphi, \delta, \beta\}$. Then,*

$$\begin{array}{rcl} [\![\varphi^+_{f\times g}\left([\![S^n]\!]^\Phi\right)]\!]^\Psi & \subseteq & [\![\varphi^+_f\left([\![S^n]\!]^\Phi\right)]\!]^\Psi \times [\![\varphi^+_g\left([\![S^n]\!]^\Phi\right)]\!]^\Psi \\ [\![\varphi^-_{f\times g}\left([\![S^k]\!]^\Phi\right)]\!]^\Psi & \subseteq & [\![\varphi^-_f\left([\![S^i]\!]^\Phi\right)]\!]^\Psi \cap [\![\varphi^-_g\left([\![S^j]\!]^\Phi\right)]\!]^\Psi \\ [\![\widehat{\varphi}_{f\times g}\left([\![S^k]\!]^\Phi, S^n\right)]\!]^\Psi & \subseteq & [\![\widehat{\varphi}_f\left([\![S^i]\!]^\Phi, S^n\right)]\!]^\Psi \cap [\![\widehat{\varphi}_g\left([\![S^j]\!]^\Phi, S^n\right)]\!]^\Psi \end{array}$$

with equality if f and g are functionally independent and $\Phi \in \{\delta, \beta\}$.

The above proposition allows to obtain an approximated view over any Cartesian product of functions by reusing views over the elementary functions. Moreover, when these functions do not share variables and the input set is at least a Cartesian domain, then a propagator using

this decomposition will have the same completeness as the one obtained using the original (i.e. not decomposed) view.

7.3.7. Idempotency

The method for comparing propagation strength of view models described above does not require them to be idempotent. Constraint rule databases are integrated through corollary 7.26 which does not require idempotency. In fact, constraint rule databases for idempotent view models of a constraint c are a superset of constraint rule databases for non-idempotent view models for c, since

$$\begin{array}{lcl} \underset{\Phi_1}{\rightarrow} \pi_c^{\Phi_2\Phi_3} \underset{\Phi_4}{\rightarrow} & \preceq & \underset{\Theta_1}{\rightarrow} \pi_c^{\Theta_2\Theta_3} \underset{\Theta_4}{\rightarrow} \\ & \Rightarrow & \\ \underset{\Phi_1}{\rightarrow} \pi_c^{\Phi_2\Phi_3\star} \underset{\Phi_4}{\rightarrow} & \preceq & \underset{\Theta_1}{\rightarrow} \pi_c^{\Theta_2\Theta_3\star} \underset{\Theta_4}{\rightarrow} \end{array}$$

The same occurs with function rule databases given that

$$\begin{array}{lcl} \underset{\Phi_1}{\rightarrow} \Phi^+_{2g} \underset{\Phi_3}{\rightarrow} \pi_{c\circ f}^{\Phi_4\Phi_5} \underset{\Phi_6}{\rightarrow} \Phi^-_{7g} \underset{\Phi_8}{\rightarrow} & \preceq & \underset{\Theta_1}{\rightarrow} \Theta^+_{2g} \underset{\Theta_3}{\rightarrow} \pi_{c\circ f}^{\Theta_4\Theta_5} \underset{\Theta_6}{\rightarrow} \Theta^-_{7g} \underset{\Theta_8}{\rightarrow} \\ & \Rightarrow & \\ \underset{\Phi_1}{\rightarrow} \Phi^+_{2g} \underset{\Phi_3}{\rightarrow} \pi_{c\circ f}^{\Phi_4\Phi_5\star} \underset{\Phi_6}{\rightarrow} \Phi^-_{7g} \underset{\Phi_8}{\rightarrow} & \preceq & \underset{\Theta_1}{\rightarrow} \Theta^+_{2g} \underset{\Theta_3}{\rightarrow} \pi_{c\circ f}^{\Theta_4\Theta_5\star} \underset{\Theta_6}{\rightarrow} \Theta^-_{7g} \underset{\Theta_8}{\rightarrow} \end{array}$$

This means that, by using rule databases that consider non-idempotent propagators exclusively, we are sure that the algorithm is correct for the case where such propagators are executed until fixpoint. However, rule databases that are specific to idempotent propagators may be used to improve the approximations obtained for models that are indeed idempotent, in particular for view models with multiple views since corollary 7.31 requires idempotency. This is a consequence of the direct relation between specificity of rules and inference inherent to rule rewriting algorithms.

Integrating idempotency information in our algorithm is a matter of selecting specific rules (i.e. for idempotent or non-idempotent propagators) according to the model we are testing. This assumes of course that there is a way to know the idempotency of the model. In fact this may not be a problem, either because it is known beforehand as with auxiliary variables view models (more on this later), or because it may be inferred (see section 7.2.3).

Finally, we recall that $\Phi\Psi$-completeness of a view model for a constraint $c \circ f$ is guaranteed by requiring propagator for c to be idempotent. Therefore, for proving $\Phi\Psi$-completeness of a view model that uses a non-idempotent propagator for c the rule database must contain some rules comparing idempotent with non-idempotent propagators. Although this is possible in theory, in our experiments we opted to use view models based on idempotent propagators exclusively.

7.3.8. Complexity and optimizations

Let n be the number of functions in a given view model $\langle \Phi_1, \ldots, \Phi_{4(n+1)} \rangle^{c \circ f_1 \circ \ldots \circ f_n}$. The algorithm `FindUb` may explore an exponential number of models, more specifically $O\left(3^{4(n+1)}\right)$. However, since practical constraints usually involve a small n, this is seldom a problem in practice. For constraints with large n we may use an incremental version of the algorithm which first computes the upper bound for $\langle \Phi_1, \ldots, \Phi_8 \rangle^{c \circ f_1}$, uses proposition 7.7 to select the stronger $\Phi\Psi$-complete propagator $\widehat{\pi}^{\Phi\Psi}_{c \circ f}$ in that set, and then repeat the process iteratively from $\left(c \circ f_1\right) \circ f_2$ to $\left(c \circ f_1 \circ \ldots\right) \circ f_n$. In this case the algorithm will loose in completeness, but the runtime complexity decreases to $O\left(n 3^8\right)$.

The lattice structure of trivial order relations also provides opportunities for optimization. The algorithm may maintain only the non-trivially stronger models of the upper bound, that is $\left\lfloor \widetilde{U} \right\rfloor^T$ instead of the full set $\widetilde{U}$. This may be done by replacing line 2 by $U \leftarrow \{m\}$, the condition in line 6 by $m_1 \in \lceil U \rceil^T$ and line 7 by a routine which stores the model m_2 in $\lfloor U \rfloor^T$ only if m_2 is not already a member, in which case it first removes $\lceil m_2 \rceil^T$ from $\lfloor U \rfloor^T$. We remark that in this case the rule rewriting algorithm may still explore an exponential number of models in the worst scenario, but the performance of the algorithm for the average case is increased.

Regarding the space complexity we note that although proposition 7.28 has shown how to compute $R_{c \circ f}$ from a union of explicit sets R^+_c, and R_f, this was intended for making the description of the algorithm more clear, and that an implementation of the `FindUb` algorithm does not require the existence of a pre-computed $R_{c \circ f}$ (not even R^+_c), but instead infers the combined rules dynamically from R_c and R_f. In such case, the space complexity is in $O\left(n \times 3^{16}\right)$ since there are n rule databases each having a quadratic number of rules. We may still decrease this number by optimizing the size of the rule databases, which also plays a role on the algorithm runtime. Small rule databases may be obtained using the transitive reduction of all rules, which may be computed offline, in polynomial time [Aho et al. 1972].

7.4. Experiments

There are two important applications for the algorithm described above, both of them central in the modeling phase of a constraint program. On one hand we may use it to approximate the completeness guarantees of a given view model for a specific constraint. This information is helpful for reasoning globally about the constraint program and make an informed and concerted decision on the selection of propagators for the constraints in the problem. In this case we have an arbitrary view model $m = \langle \Phi_1, \ldots, \Phi_n \rangle^c$, and test if a $\Phi\Psi$-complete propagator $\pi^{\Phi\Psi}_c$ belongs to `FindUb`(m), `FindLb`(m), or both.

On the other hand, we may already have decided upon a specific strength for a given constraint propagator, but wish to select the most efficient view model that implements it. This problem is the inverse of the former: we are given a $\Phi\Psi$-complete propagator $m = \pi^{\Phi\Psi}_c$ and

then select the most efficient view model $\langle \Phi_1, \ldots, \Phi_n \rangle^c$ from `FindUb`(m).

For evaluating our algorithm, we simulated an instance of the first scenario. Specifically, we have a view model m for a constraint $c \circ f_1 \circ \ldots \circ f_2$ consisting of only β approximation operators except the initial and final approximation, that is $m = \langle \delta, \beta, \beta, \ldots \beta, \delta \rangle^{c \circ f_1 \circ \ldots \circ f_2}$, and wish to know the strength of the resulting propagator. This problem occurs often in practice since these $\beta\beta$ view approximations are very efficient to compute.

We have generated propagator rule databases for $c = [x = y]$, and $c = [x \leq y]$, and view rule databases for the functions $f(x) = ax$, $f(x) = |x|$, $f(x, y) = x + y$, and $f(x, y) = x \times y$, where x, y are variables and a, b are constants. The algorithm was then tested in a number of arithmetic constraints created by composition of these functions.

Table 7.2 shows the results. For each function we show the obtained upper bound (actually only the trivially stronger models) and their intersection with the obtained lower bound, which was $\lceil \widetilde{L}_m \rceil^T = \{\text{bounds}(\mathbb{Z})\}$ for all constraints. Set $\widetilde{E}_m = \widetilde{U}_m \cap \widetilde{L}_m$ represents the set of view models known to be equivalent to the given model. In the last column we show the difference between the exact set of equivalent models and the set of equivalent models computed by our algorithm. An empty set in this column means that the algorithm was exact.

Recall that a $\top$ represents the weakest view model in the lattice. We have found a number of view models which are strictly stronger than $\top$ and strictly weaker than bounds$(\mathbb{Z})$ which we did not classify (see the last row in the table). The propagator strength of those models should be bounds$(\mathbb{R})$, a class strictly weaker than bounds$(\mathbb{Z})$, however we are still unsure how this class maps into the view model lattice.

7.5. Incomplete constraint checkers

Like propagators, constraint checkers may also be approximated. View models for constraint checkers are reasonably simpler than view models for propagators since constraint checkers may be defined exclusively from the image view function φ^+. We may define an incomplete constraint checker by approximating its input or output:

Definition 7.33 (Incomplete constraint checker)**.** Let $\Phi_1, \Phi_2 \in \{\varphi, \delta, \beta\}$. An incomplete constraint checker for the constraint c is defined as

$$\langle \Phi_1, \Phi_2 \rangle^c \quad = \quad \llbracket \varphi^+ \left(\llbracket S \rrbracket^{\Phi_1} \right) \rrbracket^{\Phi_2}$$

Proposition 7.34. *Let $\Phi_1, \Phi_2 \in \{\varphi, \delta, \beta\}$ and c be an arbitrary constraint. An incomplete constraint checker $\langle \Phi_1, \Phi_2 \rangle^c$ is a sound constraint checker for c.*

Additionally, we may rank constraint checkers according to its completeness similarly to what was done with incomplete constraint propagators. Likewise, an incomplete constraint checker for a functional composition may be created by combining the constraint checkers for

	$\lfloor U_m \rfloor^T$	$\widetilde{E}_m$	$E_m \setminus \widetilde{E}_m$
$ax = c$ $ax \geq y$	{domain}	{bounds($\mathbb{Z}$), bounds(D), range, domain}	{} {}
$x + y \geq z$ $x + y = c$	{range, bounds(D)}	{bounds($\mathbb{Z}$), bounds(D), range}	{domain} {}
$\sum_{i=1}^{n>2} x_i \geq z$ $\sum_{i=1}^{n>2} x_i = c$	{range}	{range, bounds($\mathbb{Z}$)}	{domain, bounds(D)} {}
$ax = y$ $ax + by = c$ $ax + by \geq z$	{bounds(D)}	{bounds(D), bounds($\mathbb{Z}$)}	{} {} {domain, range}
$\sum_{i=1}^{n} x_i = z$ $x \times y = c$ $x \times y \geq z$	{bounds($\mathbb{Z}$)}	{bounds($\mathbb{Z}$)}	{} {} {bounds(D)}
$\prod_{i=1}^{n} x_i = z$ $\prod_{i=1}^{n>2} x_i = c$ $\sum_{i=1}^{n} a_i x_i = z$ $\prod_{i=1}^{n>2} x_i \geq z$	{⊤}	{}	{} {} {} {bounds($\mathbb{Z}$), bounds(D)}

Table 7.2.: Strength of the view model m for a set of arithmetic constraints.

the individual functions, and its overall completeness may be assessed by an adapted version of the rule rewriting algorithm described above.

7.6. Summary

In this chapter we have characterized view models as a tool for defining incomplete propagators with different strength. For assessing the strength of a view model we have presented a rule rewriting algorithm that maps the model into a lattice of consistencies including the well-known cases of domain consistency as well as the different forms of bounds consistency. The algorithm may be used to select the most efficient view-based propagator for a certain consistency type or to inform the consistency achieved by some predefined model.

An implementation of the algorithm and the rule databases for the experiments in this chapter may be obtained from the author upon request.

Related work

- Yuanlin and Yap [2000] present an analysis on the complexity of propagating some arithmetic constraints. In particular, they show that a bounds complete propagation algorithm is also domain complete when the constraints are linear.
- In [Apt and Zoeteweij 2003] the decomposition of arithmetic expressions using idempotent propagators is presented formally. The propagation complexity is analyzed for different decomposition models, although the focus is on symbolic manipulation of expressions, in particular replacing common subexpressions and factorizing.
- A related problem is to compare the strength of two different sets of propagators in a given CSP. This problem was approached in Schulte and Stuckey [2005] which presented an algorithm that detects when two sets of propagators are equivalent, in particular bounds and domain complete propagators. The algorithm performs a different, although related analysis, and provides exact results for cases where our algorithm does not, specifically constraints involving inequalities. There are other significant differences, namely that no distinction is made among several bounds complete classes, and that it works with propagators, and not arbitrary view models.
- Schulte and Tack [2008] provide a set of conditions which guarantees the strength of a view-based propagator, which is the problem we address in section 7.3. In their work, the strength of a view model is computed as a function of two properties of the involved views, namely *injectivity* and *surjectivity*, and the strength of the propagator being derived. These properties essentially guarantee that a known set of equivalent models exists for the functions φ_f^+, φ_f^- and π_f individually. It is then proved that when these models exist the obtained view model is a propagator with a certain strength. The simplicity of this method is appealing,

but it has some limitations. Firstly, f is required to be an injective function, which excludes many interesting functions for which efficient view models have been found, namely functions involving multiple variables (see examples in section 6.1). Secondly, the method is less committed to the semantics of the involved functions and consequently is less exact in general. It cannot show for example that the view model for the constraint $[ax = z]$ mentioned in table 7.2 is a bounds($\mathbb{Z}$) propagator. Finally, our algorithm may be used to order view models that are not (equivalent to) $\Phi\Psi$-propagators, and that is not possible using their method.

Future work

The fact that we are using a blind, brute force algorithm for approximating the strength of a view model suggests several developments:

- ▷ Compare models which use different functions. It has been shown that symbolic manipulation such as factorizing improves propagation in some constraints [Apt and Zoeteweij 2003]. It would be interesting to see if we can adapt our algorithm to order different representations of the same constraint.
- ▷ Create rule databases for specific non-idempotent propagators. In our experiments we used rule databases solely for idempotent propagators. However, as discussed in section 7.3.7, non-idempotent propagators may be integrated as well using specific rules.
- ▷ Improve the treatment of inequalities by considering *open* β-operators, i.e. box domains which are not finite. These operators fit nicely in the operator hierarchy since they are weaker than β-operators, and would allow for more expressive rules and therefore provide stronger inference in general, and in particular for view models involving inequalities.

Chapter 8.

Type Parametric Compilation of Box View Models

This chapter details an efficient realization of the formerly described view models. More specifically, it presents a set of algorithms for propagating a view model consisting exclusively of box approximations, and shows how to apply them for an arbitrary decomposable constraint. The set of algorithms described cover the classical propagation scheme for decomposable constraints based on the introduction of auxiliary variables, and also two frameworks based on an abstraction called a *view object*. We show that the kind of algorithm used for propagating a view model is deeply connected with its compilation scheme.

The chapter is organized as follows: we first discuss a class of view models which are inherently very efficient to compute, namely box view models (§8.1). Then we introduce box view objects, explain how they naturally map to box view models (§8.2), and discuss the connection between practical implementations of view objects and the kind of polymorphism available in the subjacent programming language (§8.3). Section 8.4 shows that view models may also be implemented using auxiliary variables and extra propagators. Then, we describe a compilation algorithm for each of the presented methods for propagating view models (§8.5) and provide a theoretical comparison of these methods (§8.6). Finally, we discuss views over useful non-arithmetic expressions (§8.7).

8.1. Box view models

Bounds propagators are widely used in constraint programs. Consider the following example.

Example 8.1. Let $c = [x_1 \geq x_2]$ and S^2 an arbitrary δ-domain. A bounds($\mathbb{Z}$) propagator for c (which happens to be also domain complete) may be defined as follows

$$\begin{aligned} \pi_c^{\beta\beta}(S) &= \llbracket \mathrm{con}(c) \cap \llbracket S \rrbracket^{\beta} \rrbracket^{\beta} \cap S \\ &= \left(\left[\lfloor \mathrm{proj}_2(S) \rfloor \ldots \lceil \mathrm{proj}_1(S) \rceil\right] \times \left[\lfloor \mathrm{proj}_2(S) \rfloor \ldots \lceil \mathrm{proj}_1(S) \rceil\right]\right) \cap S \end{aligned}$$

The fact that the above bounds($\mathbb{Z}$) propagator is defined exclusively from operations involving the lower and upper bounds of the input set makes it very efficient to compute. In general,

computing view operations involving box approximations is also efficient: β-domains may be stored in constant space since the domain is fully characterized by its lower and upper bound, and computing views which are applied and relaxed to β-domains does not depend on the size of the domain for most functions.

Example 8.2. Let $f(x) = |x|$ and $S_1, S_2 \subseteq \mathbb{Z}$ two arbitrary sets, and assume $\lfloor S_2 \rfloor \geq 0$. Consider the problem of computing the view functions $[\![\varphi_f^+([\![S_1]\!]^\beta)]\!]^\beta$ and $[\![\widehat{\varphi}_f([\![S_2]\!]^\beta, [\![S_1]\!]^\beta)]\!]^\beta$. Recall that, by definition,

$$\begin{aligned}
[\![\varphi_f^+\left([\![S_1]\!]^\beta\right)]\!]^\beta &= [\![\left\{|x| : x \in [\![S_1]\!]^\beta\right\}]\!]^\beta \\
[\![\widehat{\varphi}_f\left([\![S_2]\!]^\beta, [\![S_1]\!]^\beta\right)]\!]^\beta &= [\![\left\{x : |x| \in [\![S_2]\!]^\beta \wedge x \in [\![S_1]\!]^\beta\right\}]\!]^\beta
\end{aligned}$$

This definition may suggest that evaluating these functions would take linear time, but in fact they may be computed in constant time assuming that finding the minimum and maximum of sets S_1, S_2 takes constant time (i.e. the sets are ordered):

$$\begin{aligned}
[\![\varphi_f^+\left([\![S_1]\!]^\beta\right)]\!]^\beta &= \begin{cases} [\lfloor S_1 \rfloor \ldots \lceil S_1 \rceil] & \Leftarrow \lfloor S_1 \rfloor > 0 \\ [-\lceil S_1 \rceil \ldots -\lfloor S_1 \rfloor] & \Leftarrow \lceil S_1 \rceil < 0 \\ [0 \ldots \max(-\lfloor S_1 \rfloor, \lceil S_1 \rceil)] & \text{otherwise} \end{cases} \\
[\![\widehat{\varphi}_f\left([\![S_2]\!]^\beta, [\![S_1]\!]^\beta\right)]\!]^\beta &= \begin{cases} [\max(\lfloor S_2 \rfloor, \lfloor S_1 \rfloor) \ldots \min(\lceil S_2 \rceil, \lceil S_1 \rceil)] & \Leftarrow \lfloor S_1 \rfloor > 0 \\ [\max(-\lceil S_2 \rceil, \lfloor S_1 \rfloor) \ldots \min(-\lfloor S_2 \rfloor, \lceil S_1 \rceil)] & \Leftarrow \lceil S_1 \rceil < 0 \\ [\max(-\lceil S_2 \rceil, \lfloor S_1 \rfloor) \ldots \min(\lceil S_2 \rceil, \lceil S_1 \rceil)] & \text{otherwise} \end{cases}
\end{aligned}$$

Before we proceed to analyze the efficiency of view models, let us make the following observation concerning notation. In this chapter we will focus on tuple sets which are at least δ-domains. This allows us to express formulas like the above, which are based on the notation introduced in the previous chapter, with a more convenient notation which is also more common within constraint programming.

Example 8.3. Let D be the set of Cartesian product of variable domains in a CSP $\langle X, D, C \rangle$ as defined in chapter 2. A bounds($\mathbb{Z}$) propagator for $c = [x_1 \geq x_2]$, $x_1, x_2 \in X$, may be defined as follows

$$\begin{aligned}
\pi_c^{\beta\beta}(D) &= \left(\left[\lfloor \text{proj}_2(D) \rfloor \ldots \lceil \text{proj}_1(D) \rceil\right] \times \left[\lfloor \text{proj}_2(D) \rfloor \ldots \lceil \text{proj}_1(D) \rceil\right]\right) \cap D \\
&= \left([\lfloor D(x_2) \rfloor \ldots \lceil D(x_1) \rceil] \times [\lfloor D(x_2) \rfloor \ldots \lceil D(x_1) \rceil]\right) \cap D
\end{aligned}$$

Given that the output of the above propagator is also a Cartesian product, i.e.

$$\pi_c^{\beta\beta}(D) = \text{proj}_1\left(\pi_c^{\beta\beta}(D)\right) \times \text{proj}_2\left(\pi_c^{\beta\beta}(D)\right)$$

we may express the propagator as

$$\begin{aligned}\mathrm{proj}_1\left(\pi_c^{\beta\beta}(D)\right) &= [\lfloor D(x_2)\rfloor \ldots \lceil D(x_1)\rceil] \cap D(x_1)\\ \mathrm{proj}_2\left(\pi_c^{\beta\beta}(D)\right) &= [\lfloor D(x_2)\rfloor \ldots \lceil D(x_1)\rceil] \cap D(x_2)\end{aligned}$$

or more commonly,

$$\pi_c^{\beta\beta}(D) = \begin{cases} D(x_1) & \leftarrow [\lfloor D(x_2)\rfloor \ldots \lceil D(x_1)\rceil] \cap D(x_1)\\ D(x_2) & \leftarrow [\lfloor D(x_2)\rfloor \ldots \lceil D(x_1)\rceil] \cap D(x_2)\end{cases}$$

Since bounds($\mathbb{Z}$) propagators and view box approximations are efficient to compute, it is not surprising that view models resulting from the combination of these propagators and views are also efficient.

Example 8.4. Let $c \circ f = [|x_1| \geq x_2]$ where $f(x_1, x_2) = \langle |x_1|, x_2\rangle$, and $c = [x_1 \geq x_2]$. The view model

$$\underset{\delta}{\rightarrow}\, \beta_f^+ \underset{\beta}{\rightarrow}\, \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow}\, \widehat{\beta}_f \underset{\delta}{\rightarrow}$$

may be defined based on the $\pi_c^{\beta\beta}$ propagator and view operations presented above:

$$D(x_1) \leftarrow D(x_1) \cap \begin{cases} [\lfloor D(x_2)\rfloor \ldots \lceil D(x_1)\rceil] & \Leftarrow \lfloor D(x_1)\rfloor > 0\\ [\lfloor D(x_1)\rfloor \ldots -\lfloor D(x_2)\rfloor] & \Leftarrow \lceil D(x_1)\rceil < 0\\ D(x_1) & \text{otherwise}\end{cases}$$

$$D(x_2) \leftarrow D(x_2) \cap \begin{cases} [\lfloor D(x_2)\rfloor \ldots \lceil D(x_1)\rceil] & \Leftarrow \lfloor D(x_1)\rfloor > 0\\ [\lfloor D(x_2)\rfloor \ldots -\lfloor D(x_1)\rfloor] & \Leftarrow \lceil D(x_1)\rceil < 0\\ [\lfloor D(x_2)\rfloor \ldots \max(-\lfloor D(x_1)\rfloor, \lceil D(x_1)\rceil)] & \text{otherwise}\end{cases}$$

In this chapter we focus on the process of compiling efficient box view models like the above from the necessary views and propagators.

Definition 8.5 (Box view model)**.** A box view model is a view model consisting exclusively of box approximation operators, formally

$$\underset{\delta}{\rightarrow}\, \beta_f^+ \underset{\beta}{\rightarrow}\, \pi_c^{\beta\beta\star} \underset{\beta}{\rightarrow}\, \widehat{\beta}_f \underset{\delta}{\rightarrow}$$

8.2. View objects

To help us formalize view model implementations it is useful to think of views and domain variables as objects exposing a known interface. Note however that this does not imply an object oriented architecture for practical use of views, as it will be shown in the next chapter.

8.2.1. Typed constraints

Let us begin by introducing a type-aware representation of constraints and expressions. Specifically, we will express constraints as sentences of a first-order language $\mathscr{L}(\Sigma, X)$ involving a standard set of arithmetic and logical operators Σ, and domain variables X. The type of a variable occurring in a constraint is made explicit by a suffix : type after the reference to the variable. Making the type of the operands explicit allow us to specify constraints between objects of different types unambiguously.

Domain variables are objects of type dvar and their interface simply exposes the lower and upper bounds of the variable's domain. We assume throughout this chapter that accessing and updating these bounds takes constant time.

Example 8.6. The constraint $c \circ f$ of the previous example applied to the set of domain variables D may be represented by the following typed constraint $e \in \mathscr{L}(\Sigma, X)$,

$$e = [|x_1 : \mathsf{dvar}| \geq x_2 : \mathsf{dvar}]$$

Note that the function $c \circ f$ of the previous example uniquely maps to the above typed constraint $e \in \mathscr{L}(\Sigma, X)$ when applied to D.

8.2.2. Box view objects

A box view object implements the operations $[\![\varphi_f^+([\![S_1]\!]^\beta)]\!]^\beta$ and $[\![\widehat{\varphi}_f([\![S_2]\!]^\beta, [\![S_1]\!]^\beta)]\!]^\beta$ over a specific expression $e \in \mathscr{L}(\Sigma, X)$. We assume that both the function f and the δ-domain S_1 may be determined from e as in the previous example.

Box view objects are of type bnd, and have the following interface,

$$\mathsf{bnd} \quad : \quad \left\{ \begin{array}{l} \textsc{getMin} : \mathsf{func}[\rightarrow l], \textsc{getMax} : \mathsf{func}[\rightarrow u], \\ \textsc{updMin} : \mathsf{func}[l \rightarrow], \textsc{updMax} : \mathsf{func}[u \rightarrow] \end{array} \right\} \tag{8.1}$$

The GETMIN and GETMAX functions compute φ^+ and the UPDMIN and UPDMAX are used to

update $\widehat{\varphi}$, specifically

$$\begin{aligned}
\textsc{getMin} &: l \leftarrow \left\lfloor \llbracket \varphi_f^+ \left(\llbracket S_1 \rrbracket^\beta \right) \rrbracket^\beta \right\rfloor \\
\textsc{getMax} &: u \leftarrow \left\lceil \llbracket \varphi_f^+ \left(\llbracket S_1 \rrbracket^\beta \right) \rrbracket^\beta \right\rceil \\
\textsc{updMin}(l) &: S_1 \leftarrow S_1 \cap \llbracket \widehat{\varphi}_f \left([l \ldots +\infty], \llbracket S_1 \rrbracket^\beta \right) \rrbracket^\beta \\
\textsc{updMax}(u) &: S_1 \leftarrow S_1 \cap \llbracket \widehat{\varphi}_f \left([-\infty \ldots u], \llbracket S_1 \rrbracket^\beta \right) \rrbracket^\beta
\end{aligned}$$

Note that the parameter l, b of the UPDMIN and UPDMAX functions are respectively the lower and upper bounds of $\llbracket S_2 \rrbracket^\beta$.

Example 8.7. A box view object over a domain variable $x \in X$ is defined by the following set of methods

$$\mathsf{bnd}\,[x : \mathsf{dvar}] \quad = \quad \begin{cases} \textsc{getMin} = \{r \leftarrow \lfloor D(x) \rfloor\} \\ \textsc{getMax} = \{r \leftarrow \lceil D(x) \rceil\} \\ \textsc{updMin} = \{D(x) \leftarrow D(x) \setminus [-\infty \ldots i-1]\} \\ \textsc{updMax} = \{D(x) \leftarrow D(x) \setminus [i+1 \ldots +\infty]\} \end{cases}$$

8.3. View object stores

A view object provides a known interface to a specific expression $e \in \mathscr{L}(\Sigma, X)$. The set of view objects over expressions of a given language is called a view object store.

Definition 8.8 (View store)**.** A view store $\lambda\langle\mathscr{L}\rangle$ is an indexed collection of objects which provide a common interface (a view) to each element of the language $\mathscr{L}$ being viewed. We call $\lambda\langle\mathscr{L}\rangle$ a view store λ over $\mathscr{L}$, and $\lambda[e]$ the view object for the expression $e \in \mathscr{L}$.

A box view store over the language of typed constraints defined previously is conveniently denoted $\mathsf{bnd}\langle\mathscr{L}\rangle$. View stores indexing other kinds of view objects, i.e. views combined with other approximations, could be defined in the same way.

Given that one may create infinitely many expressions from most useful languages, in particular from our language $\mathscr{L}$, defining a view object for each possible expression is clearly infeasible. We now turn to this problem and present two solutions with a different compromise regarding efficiency as we will see later.

8.3.1. Subtype polymorphic stores

A possible solution explores the fact that each view object has a known interface, described above by eq. 8.1. As long as the interface defines the necessary operations, the true identity of

the object may be abstracted away. This approach, called *subtype polymorphism*, is a classical solution to the problem of allowing values of different data types to be handled using a uniform interface.

Subtype polymorphic view stores are defined over a subset $\mathscr{L}'$ of language $\mathscr{L}$ supporting a finite number of expressions. Specifically, finite view stores define view objects over domain variables and operations involving other view objects, formally

$$\mathscr{L}' = \{x : \mathsf{dvar}\} \cup \bigcup_{\oplus \in \Sigma} x : \mathsf{bnd} \oplus y : \mathsf{bnd}$$

Example 8.9. The following is the definition of a bnd view object over the addition of two bnd objects:

$$\mathsf{bnd}\left[x : \mathsf{bnd} + y : \mathsf{bnd}\right] : \begin{cases} \textsc{getMin} = \{r \leftarrow x.\textsc{getMin}() + y.\textsc{getMin}()\} \\ \textsc{getMax} = \{r \leftarrow x.\textsc{getMax}() + y.\textsc{getMax}()\} \\ \textsc{updMin} = \left\{ \begin{array}{l} x.\textsc{updMin}\left(i - y.\textsc{getMax}()\right) \\ y.\textsc{updMin}\left(i - x.\textsc{getMax}()\right) \end{array} \right\} \\ \textsc{updMax} = \left\{ \begin{array}{l} x.\textsc{updMax}\left(i - y.\textsc{getMin}()\right) \\ y.\textsc{updMax}\left(i - x.\textsc{getMin}()\right) \end{array} \right\} \end{cases}$$

Since a subtype polymorphic view store indexes a limited set of expressions, the problem of obtaining the view object corresponding to an expression $e \in \mathscr{L}$ such that $e \notin \mathscr{L}'$ remains to be solved. This problem will be addressed later when we focus on compilation of expressions.

8.3.2. Parametric polymorphic stores

Another possible solution is to design a store that indexes generic view objects, that is view objects whose definition does not depend on the actual type of the associated expression. This design idiom is commonly referred to as parametric polymorphism [Reynolds 1974]. Let us formalize expressions involving generic types.

Definition 8.10 (Expression template)**.** An expression template is an expression $t \in \mathscr{L}\left(\Sigma, X'\right)$ where X' is the set of domain variables X augmented with term variables (denoted by uppercase letters). It represents the largest language subset $\mathscr{L}^t \subseteq \mathscr{L}$ where for all $i \in \mathscr{L}^t$, i may be obtained from t by instantiating the term variables.

Unlike subtype polymorphic stores, parametric polymorphic stores are based on two important requirements: Firstly, view objects may be associated with partially defined expressions, i.e. expression templates, instead of concrete expressions $e \in \mathscr{L}$. Secondly, references to view stores may appear in the definition of view objects. By providing view objects for a careful selection of expression templates, we may obtain a view object for any expression $e \in \mathscr{L}$.

Example 8.11. The following is the definition of a bnd view object for the addition of two generic expressions:

$$\mathsf{bnd}[X+Y] = \left\{ \begin{array}{l} \textsc{getMin} = \{r \leftarrow \mathsf{bnd}[X].\textsc{getMin}() + \mathsf{bnd}[Y].\textsc{getMin}()\} \\ \textsc{getMax} = \{r \leftarrow \mathsf{bnd}[X].\textsc{getMax}() + \mathsf{bnd}[Y].\textsc{getMax}()\} \\ \textsc{updMin} = \left\{ \begin{array}{l} \mathsf{bnd}[X].\textsc{updMin}(i - \mathsf{bnd}[Y].\textsc{getMax}()) \\ \mathsf{bnd}[Y].\textsc{updMin}(i - \mathsf{bnd}[X].\textsc{getMax}()) \end{array} \right\} \\ \textsc{updMax} = \left\{ \begin{array}{l} \mathsf{bnd}[X].\textsc{updMax}(i - \mathsf{bnd}[Y].\textsc{getMin}()) \\ \mathsf{bnd}[Y].\textsc{updMax}(i - \mathsf{bnd}[X].\textsc{getMin}()) \end{array} \right\} \end{array} \right.$$

8.4. Auxiliary variables

The classical solution for creating propagators for arbitrary constraints does not make use of view objects at all. Instead, it consists in introducing a propagator and an auxiliary variable for each subexpression of the input expression. The propagator is responsible for channeling modifications in the auxiliary variable's domain to the domains of the variables in the subexpression and vice-versa.

Proposition 8.12. *Any box view model may be enforced by a set of propagators and a set of auxiliary domain variables.*

8.5. Compilation

We now detail the process of generating a propagator or set of propagators for a given arbitrary constraint. The compilation process will be defined declaratively by means of action rules:

Definition 8.13 (Action Rule, Rule repository)**.** An action rule associated with a language $\mathscr{L}$ takes the form

$$\frac{t}{e} \left[\begin{array}{c} a_1 \\ \vdots \\ a_n \end{array} \right]$$

and defines the rewriting of any input expression $i \in \mathscr{L}^t$ (i.e. matched by the template expression t) into output expression $e \in \mathscr{L}$, performing actions $a_1, \ldots, a_n$ (the rule actions) as a side effect. A rule repository $\mathrm{R}(\mathscr{L})$ is a collection of action rules associated with language $\mathscr{L}$.

Rule repositories will be used to perform derivations of expressions.

Definition 8.14 (Derivation)**.** A derivation (written $R[e]$) denotes the output of applying the most specific rule in the repository R matching e. A rule $r_1 \equiv \frac{t_1}{e_1}$ is more specific than a rule $r_2 \equiv \frac{t_2}{e_2}$ if $\mathscr{L}^{t_1} \subset \mathscr{L}^{t_2}$. For simplification, we will assume that there is no ambiguity for selecting the most specific rule for a given expression. In practice this can be enforced by adding extra rules to the repository.

8.5.1. Subtype polymorphic views

Compiling an arbitrary expression to subtype polymorphic view objects recursively rewrites an expression bottom-up, starting with higher precedence subexpressions. The introduction of an auxiliary variable is replaced by the introduction of a box view object, which means that all view object definitions may safely assume that all operands are $x : \mathsf{bnd}$ objects.

The compilation algorithm is fully described by the following rule repository:

$$\mathrm{R}_{\mathrm{BND}} = \begin{cases} \forall_{\oplus \in \Sigma} \dfrac{X \oplus Y}{z : \mathsf{bnd}} \left[\; z \leftarrow \mathsf{new\,bnd}\,[\mathrm{R}_{\mathrm{BND}}\,[X] \oplus \mathrm{R}_{\mathrm{BND}}\,[Y]] \; \right] \\ \dfrac{x : \mathsf{dvar}}{z : \mathsf{bnd}} \left[\; z \leftarrow \mathsf{new\,bnd}\,[x : \mathsf{dvar}] \; \right] \end{cases}$$

The above repository is incomplete: extra rules should be added for handling expressions with arity other than two, and creating bound objects from literals. We intentionally exclude these rules throughout this section for the sake of simplicity - they may be included straightforwardly without loss of correctness.

The decomposition process for an arbitrary constraint $e \in \mathscr{L}$ is triggered by the addition of the following propagator to the constraint store

$$\pi\,[X] = \{\mathrm{R}_{\mathrm{BND}}\,[X]\,.\textsc{updMin}\,(\texttt{true})\} \tag{8.2}$$

Propagator $\pi\,[e]$ instantiates the view for the constraint e and propagates it by setting its value to `true`.

Example 8.15. The compilation of the constraint $[a \times b + c \leq d]$ using repository $\mathrm{R}_{\mathrm{BND}}$ yields the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a:\mathsf{dvar}\times b:\mathsf{dvar}+c:\mathsf{dvar}\leq d:\mathsf{dvar}}{v_1:\mathsf{bnd}\times v_2:\mathsf{bnd}+c:\mathsf{dvar}\leq d:\mathsf{bnd}}}{v_3:\mathsf{bnd}+v_4:\mathsf{bnd}\leq v_6:\mathsf{bnd}}}{v_5:\mathsf{bnd}\leq v_6:\mathsf{bnd}}}{v_7:\mathsf{bnd}} \quad \left[\begin{array}{c} v_1 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[a:\mathsf{dvar}] \\ v_2 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[b:\mathsf{dvar}] \\ v_3 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[v_1:\mathsf{bnd}\times v_2:\mathsf{bnd}] \\ v_4 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[c:\mathsf{dvar}] \\ v_5 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[v_3:\mathsf{bnd}+v_4:\mathsf{bnd}] \\ v_6 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[d:\mathsf{dvar}] \\ v_7 \leftarrow \mathsf{new}\,\mathsf{bnd}\,[v_5:\mathsf{bnd}\leq v_6:\mathsf{bnd}]\,, \\ \pi = \{v_7.\textsc{updMin}\,(\mathtt{true})\} \\ \mathtt{post}\,\pi \end{array}\right]$$

8.5.2. Parametric polymorphic views

The decomposition model described above assume the existence of a finite indexed collection of views. Replacing subexpressions by view objects guarantees the existence of a view for each generated subexpression. Compiling an expression to parametric polymorphic view objects can be done simply by a top-down recursive instantiation of view objects, triggered by instantiating the view for the full expression:

$$\pi\,[X] = \{\mathsf{bnd}\,[X]\,.\textsc{updMin}\,(\mathtt{true})\} \tag{8.3}$$

Given that now the full expression is known to the view (i.e. operands are not abstractions like before), it is easy to design an algorithm which, for a specific instantiation, replaces all function calls with the corresponding definitions.

Example 8.16. Simplification of the GETMIN function of the bnd view over the $a+b+c$ expression:

$$\begin{aligned}
&\mathsf{bnd}\,[a:\mathsf{dvar}+b:\mathsf{dvar}+c:\mathsf{dvar}]\,.\textsc{getMin} = \\
&\qquad = \{r \leftarrow \mathsf{bnd}\,[a:\mathsf{dvar}+b:\mathsf{dvar}]\,.\textsc{getMin}\,() + \mathsf{bnd}\,[c:\mathsf{dvar}]\,.\textsc{getMin}\,()\} = \\
&\qquad = \{r \leftarrow \mathsf{bnd}\,[a:\mathsf{dvar}]\,.\textsc{getMin}\,() + \mathsf{bnd}\,[b:\mathsf{dvar}]\,.\textsc{getMin}\,() + \lfloor D\,(c)\rfloor\} = \\
&\qquad\qquad = \{r \leftarrow \lfloor D\,(a)\rfloor + \lfloor D\,(b)\rfloor + \lfloor D\,(c)\rfloor\}
\end{aligned}$$

In the present context, views are used very much like macros, effectively allowing creating specific propagators for arbitrary expressions which do not make use of function calls at execution.

Example 8.17. The compilation of the constraint $[a+b+c=d]$ using repository $\mathrm{R}_{\mathrm{BND}}$ yields

the following propagator:

$$\pi = \begin{cases} D(d) \leftarrow D(d) \setminus [-\infty \ldots \lfloor D(a) \rfloor + \lfloor D(b) \rfloor + \lfloor D(c) \rfloor - 1] \\ D(d) \leftarrow D(d) \setminus [\lceil D(a) \rceil + \lceil D(b) \rceil + \lceil D(c) \rceil + 1 \ldots + \infty] \\ D(a) \leftarrow D(a) \setminus [\lceil D(d) \rceil - \lfloor D(b) \rfloor - \lfloor D(c) \rfloor + 1 \ldots + \infty] \\ \qquad\qquad etc. \end{cases}$$

8.5.3. Auxiliary variables

The compilation process for the auxiliary variables model recursively rewrites an expression bottom-up, starting with higher precedence subexpressions, similarly to what is done for subtype polymorphic views. It is defined by the following repository:

$$\mathrm{R}_{\mathrm{AUX}}(\pi) = \begin{cases} \bigvee\limits_{\oplus \in \Sigma} \dfrac{X \oplus Y}{z : \mathsf{dvar}} \left[\begin{array}{c} z \leftarrow \mathsf{new}\,\mathsf{dvar}, \\ \mathtt{post}\,\pi\,[\mathrm{R}_{\mathrm{AUX}}[X] \oplus \mathrm{R}_{\mathrm{AUX}}[Y] = z] \end{array} \right] \\ \dfrac{x : \mathsf{dvar}}{x : \mathsf{dvar}}\,[] \end{cases}$$

The decomposition process for an arbitrary constraint $e \in \mathscr{L}$ is triggered by the addition of the following propagator to the constraint store:

$$\pi\,[\mathrm{R}_{\mathrm{AUX}}[e] = \mathtt{true}] \tag{8.4}$$

Note that in this decomposition process auxiliary variables are introduced for any subexpression, possibly using reification in the case of logical or comparison subexpressions.

Example 8.18. The compilation of the constraint $[a \times b + c \leq d]$ using repository $\mathrm{R}_{\mathrm{AUX}}$ yields the following derivation:

$$\frac{\dfrac{\dfrac{a : \mathsf{dvar} \times b : \mathsf{dvar} + c : \mathsf{dvar} \leq d : \mathsf{dvar}}{t_3 : \mathsf{dvar} + c : \mathsf{dvar} \leq d : \mathsf{dvar}}}{t_2 : \mathsf{dvar} \leq d : \mathsf{dvar}}}{t_1 : \mathsf{dvar}} \left[\begin{array}{c} t_1 \leftarrow \mathsf{new}\,\mathsf{dvar} \\ t_2 \leftarrow \mathsf{new}\,\mathsf{dvar} \\ t_3 \leftarrow \mathsf{new}\,\mathsf{dvar} \\ \mathtt{post}\,\pi\,[a : \mathsf{dvar} \times b : \mathsf{dvar} = t_3 : \mathsf{dvar}] \\ \mathtt{post}\,\pi\,[t_3 : \mathsf{dvar} + c : \mathsf{dvar} = t_2 : \mathsf{dvar}] \\ \mathtt{post}\,\pi\,[(t_2 : \mathsf{dvar} \leq d : \mathsf{dvar}) = t_1 : \mathsf{dvar}] \\ \mathtt{post}\,\pi\,[t_1 : \mathsf{dvar} = \mathtt{true}] \end{array} \right]$$

This method requires a predefined repository of propagators for basic expressions involving domain variables, more specifically it assumes the existence of a collection of propagators $\pi\langle\mathscr{L}'\rangle$ indexed by expressions of a bounded language $\mathscr{L}' \subset \mathscr{L}$, typically defined as

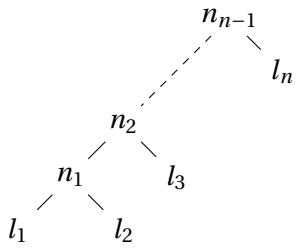


Figure 8.1.: An unbalanced expression syntax tree. The internal nodes $n_1 \dots n_{n-1}$ represent operators and leafs $l_1 \dots l_n$ represent variables.

$$\mathscr{L}' = \bigcup_{\oplus \in \Sigma} x : \mathsf{dvar} \oplus y : \mathsf{dvar} = z : \mathsf{dvar}$$

Since rules in $\mathrm{R}_{\mathrm{AUX}}$ may only generate $x : \mathsf{dvar}$ expressions, created propagators will always have $x : \mathsf{dvar}$ operands, i.e. $\mathrm{R}_{\mathrm{AUX}}(\pi)[e]$ only requires propagators in $\pi\langle\mathscr{L}'\rangle$, for any $e \in \mathscr{L}$.

Assuming a method like hashing is used to index elements of the rule and propagator repositories, the runtime cost of the compilation algorithm $\mathrm{R}_{\mathrm{AUX}}$ for an arbitrary constraint is linear on the number of binary operators in the constraint.

8.6. Model comparison

The adoption of views avoids introducing auxiliary variables and propagators for every subexpression. Conceptually, a view object over an expression serves the same purpose as the auxiliary variable introduced for that expression: to expose its domain. However, the presented models differ in many ways. We will refer to the decomposition model using auxiliary variables as VARS, subtype polymorphic view as SVIEWS, and parametric polymorphic views as PVIEWS.

For the memory and runtime analysis below we will consider an arithmetic constraint involving n variables with uniform domain size d, with an unbalanced syntax tree, i.e. where each operator in the expression involves at least one variable. Figure 8.1 shows a fragment of the expression syntax tree.

8.6.1. Memory

A view object can be designed to expose just the subset of the expression's domain which is required for the view's client (e.g. the bounds of the expression). In contrast, a variable maintains the full domain of the expression, possibly containing regions which will always be ignored for propagation. For an expression containing $n-1$ operators such as the expression in fig. 8.1, the

auxiliary variables memory overhead is in $O(nD)$ compared to any of the views model, where D is the size of the largest domain of an auxiliary variable. For some expressions, namely an expression containing only multiplications, we can have $D = d^{n-1}$. In practice, the use of intervals to store the auxiliary variable's domains (i.e. $D = 2$) eliminates this problem.

8.6.2. Runtime

There is a fundamental operational difference between both view models and the auxiliary variables model. A view object computes its domain *on demand*, that is, it will never update its domain before it is needed by the view's client. This is the opposite of what happens in the auxiliary variable model, as the posted propagators will act on the auxiliary variable's domains independently.

This analysis focuses the cost of accessing and updating the bounds of an expression, which may correspond to accessing/updating an auxiliary variable or a view object, depending on the model. The cost is measured in terms of number of propagators executed, number of function calls, number of arithmetic operations, and number of domain updates.

We first note that enforcing bounds(R) consistency to an algebraic expression requires $O(nd)$ steps [Yuanlin and Yap 2000]. As an example, consider a CSP with the constraint $x \times 2 = y \times 2 + 1$, where $D(x) = D(y) = [1 \dots d]$.

We will first focus on the costs of updating the bounds of an expression whose syntax tree resembles fig. 8.1. The auxiliary variables model associates a propagator and an auxiliary variable with each internal node n_i of the expression syntax tree. The propagator for a node n_1 involving two leafs may execute $O(2d)$ times and cause the same number of updates in the domain of the corresponding auxiliary variable. This means that the propagator for n_2 may execute $O(3d)$ times and so on for all the $n-1$ internal nodes. The number of propagators executed for the auxiliary variables model is therefore $O(n^2 d)$. Since each propagator execution requires a constant number of arithmetic operations, the number of operations for this model is also $O(n^2 d)$. Finally, given that each propagation implies the update of the domain of an auxiliary variable, the number of domain updates is $O(n^2 d)$.

Both subtype and parametric polymorphic views have only one propagator for the entire expression, which can be executed at most $O(nd)$ times. Unfortunately, a single update of the expression is now more costly because it may require evaluating the full tree, which is $O(n)$. This means that the total number of operations is still $O(nd \times n)$. However, given that there are no auxiliary variables in these models, the number of domain updates is only $O(nd)$. We remark that the only difference between both view models is in the number of function calls due to the simplification described in section 8.5.2.

Accessing the bounds of an expression is cheaper in the auxiliary variables model. This is because the domain of each subexpression is *cached* in the associated auxiliary variable. In contrast, both models involving views require evaluating the expression. Table 8.1 summarizes these results.

model	propagators		functions		operations		updates	
VARS	-	$O(n^2d)$	$O(1)$	-	$O(1)$	$O(n^2d)$	-	$O(n^2d)$
SVIEWS	-	$O(nd)$	$O(n)$	$O(n^2d)$	$O(n)$	$O(n^2d)$	-	$O(nd)$
PVIEWS	-	$O(nd)$	$O(1)$	-	$O(n)$	$O(n^2d)$	-	$O(nd)$

Table 8.1.: Cost of accessing and updating an arbitrary expression represented by each of the described models.

8.6.3. Propagation

The consistency achieved by view based propagators was detailed in the previous chapter. We have seen that idempotency of the propagator for a constraint c plays an important role in the strength of a view model for a constraint $c \circ f$. Unlike views, decomposing arbitrary expressions using auxiliary variables assures that non-idempotent propagators for subexpressions will always act as idempotent propagators. This is due to the fact that subexpressions are propagated independently and therefore will be added to the propagation queue whenever they are not at fixpoint. As a consequence of proposition 7.10, decomposing a constraint $c \circ f$ using auxiliary variables may improve propagation comparing to views when the propagator for c is not idempotent.

Example 8.19. Consider the constraint $c \circ f = [x_1 \times x_2 \times 2 = x_3]$, where $f(x_1, x_2, x_3) = \langle x_1 \times x_2, x_3 \rangle$ and $c = [x_1 \times 2 = x_2]$ and a box view model m for propagating the constraint $c \circ f$. Assume $D(x_1) = D(x_2) = [2 \dots 3]$, and $D(x_3) = [9 \dots 15]$. Propagating $m(D)$ leaves the domains of x_1 and x_2 unchanged, and prunes the domain of x_3:

$$\begin{aligned} \llbracket \varphi_f^+ \left(\llbracket D \rrbracket^\beta \right) \rrbracket^\beta &= [4 \dots 9] \times [9 \dots 15] \\ \pi_c^{\beta\beta\star}([4 \dots 9] \times [9 \dots 15]) &= [5 \dots 7] \times [10 \dots 14] \\ \llbracket \widehat{\varphi}_f \left([5 \dots 7] \times [10 \dots 14], \llbracket D \rrbracket^\beta \right) \rrbracket^\beta \cap D &= [2 \dots 3] \times [2 \dots 3] \times [10 \dots 14] \end{aligned}$$

Now consider the following standard (non-idempotent) propagator $\pi_c^{\beta\beta}$ for the constraint c:

$$\pi_c^{\beta\beta} = \begin{cases} D(x_3) & \leftarrow D(x_3) \cap [\lfloor D(x_1 \times x_2) \rfloor \times 2 \dots \lceil D(x_1 \times x_2) \rceil \times 2] \\ D(x_1 \times x_2) & \leftarrow D(x_1 \times x_2) \cap [\lfloor D(x_3) \rfloor / 2 \dots \lceil D(x_3) \rceil / 2] \end{cases}$$

If the constraint is decomposed using a view object for $D(x_1 \times x_2)$, then propagating m using

the above propagator gives:

$$\begin{aligned} \llbracket \varphi_f^+ \left(\llbracket D \rrbracket^\beta \right) \rrbracket^\beta &= [4\ldots9] \times [9\ldots15] \\ \pi_c^{\beta\beta}([4\ldots9] \times [9\ldots15]) &= [5\ldots7] \times [9\ldots15] \\ \llbracket \widehat{\varphi}_f \left([5\ldots7] \times [10\ldots14], \llbracket D \rrbracket^\beta \right) \rrbracket^\beta \cap D &= [2\ldots3] \times [2\ldots3] \times [9\ldots15] \end{aligned}$$

Since the domains of the variables did not change, the propagator is at fixpoint. If the constraint is decomposed using an auxiliary variable to represent $D(x_1 \times x_2)$, then the propagation engine would reexecute $\pi_c^{\beta\beta}$ since the domain of the auxiliary variable has changed. This would then cause m to be also reexecuted and infer $D(x_3) = [10\ldots14]$ as when using an idempotent propagator for c.

The above example may suggest that a decomposition of constraint $c \circ f$ using views would be equivalent to using auxiliary variables if view models simply allow the propagator π_c to run until fixpoint. However, converting non-idempotent propagators to idempotent propagators requires also that the changes made by π_c are persistent from one execution to another, which is always true when π_c involves domain variables exclusively, but might not be the case when π_c is applied to view objects. In fact, updating the domain of an auxiliary variable may be seen as a permanent operation - the domain effectively *remembers* the update. The same is not always true when updating view objects since they are not strictly monotonic in general, that is

$$S_1 \subset S_2 \not\Rightarrow \llbracket \widehat{\varphi} \left(\llbracket S_1 \rrbracket^\beta, \llbracket S \rrbracket^\beta \right) \rrbracket^\beta \cap S \subset \llbracket \widehat{\varphi} \left(\llbracket S_2 \rrbracket^\beta, \llbracket S \rrbracket^\beta \right) \rrbracket^\beta \cap S$$

Example 8.20. Consider again the previous example, but now assume that we use a view model that runs propagator $\pi_c^{\beta\beta}$ until fixpoint. The first execution of the view model would compute

$$\begin{aligned} D(x_3) &\leftarrow D(x_3) \cap [\lfloor D(x_1 \times x_2) \rfloor \times 2 \ldots \lceil D(x_1 \times x_2) \rceil \times 2] = [10\ldots14] \\ D(x_1 \times x_2) &\leftarrow D(x_1 \times x_2) \cap [\lfloor D(x_3) \rfloor / 2 \ldots \lceil D(x_3) \rceil / 2] = [5\ldots7] \end{aligned}$$

However, when $D(x_1 \times x_2)$ is associated with a view object, evaluating $D(x_1 \times x_2)$ after the second update above would still yield $D(x_1 \times x_2) = [4\ldots9]$, i.e. the update is not persistent. The reason for this is that the interval $[5\ldots7]$ cannot be obtained by the product of a unique pair of integer intervals representing $D(x_1)$ and $D(x_2)$. Therefore, reexecuting $\pi_c^{\beta\beta}$ would not perform any further pruning.

As we have discussed earlier, decompositions using auxiliary variables cache the domains of subexpressions across executions of the involved propagators. In contrast, the domain represented by box view objects is not maintained but recomputed in every propagation. A possible solution to the above problem would be to cache the domains of the box view objects when

the update cannot be expressed by pruning the domains of the variables involved in the corresponding subexpression. This solution is a compromise between recomputing and caching the domain of subexpressions and guarantees that decomposition based on views has the same strength as when using auxiliary variables even when using non-idempotent propagators. While caching domains introduces performance penalties as discussed above, we would still expect this solution to be more efficient than the auxiliary variable decomposition since domains would be cached much less frequently. In our experiments we found out that the amount of extra propagation earned when using auxiliary variables does not compensate the overall inefficiencies of the variable decomposition model, as will be shown in the following chapter.

8.7. Beyond arithmetic expressions

Although we have mostly focused on views over arithmetic expressions, views are also effective for accessing and updating other kind of expressions, namely the following.

8.7.1. Casting operator

Casting provides a type-safe mechanism to access or update polymorphic data types. A very common situation in Constraint Programming is treating Boolean variables as integer variables or vice-versa, as shown in the following example.

Example 8.21 (Casting)**.** Consider again the EXACTLY($\mathbf{x}, v, c$) constraint which constrains the number of occurrences of a given value in a collection of variables, i.e. $\left[\sum_i (x_i = v) = c\right]$. The expression in parenthesis is a Boolean expression, while the sum is treating it like a numeric expression, where $true$ corresponds to 1 and $false$ to 0. This is known as type casting. Defining a box view object over casting expressions allows a solver to integrate these kind of expressions in an efficient, type-safe manner.

8.7.2. Array access operator

The ELEMENT($\mathbf{x}, i, v$) constraint is used to access a position in an array of domain variables by constraining the element i of the array $\mathbf{x}$ to be equal to v. By forcing the x_i variable to equal variable v, using this constraint may introduce a superfluous variable in a number of constraints, for example $\left[x_i > y\right]$ which is typically decomposed as $\left[\text{ELEMENT}\,(\mathbf{x}, i, v) \wedge v > y\right]$. In contrast, a box view object over a x_i expression maintains the bounds of the variable at position i, and does not introduce an extra variable.

8.7.3. Iterated expressions

Sometimes we may want to aggregate n subexpressions in some global expression, and n is not known at compile time. For example in the Golomb ruler problem (see [Gent and Walsh 1999]), the constraint

$$c = \left[\text{DISTINCT}\left(x_i - x_j : 1 < i < j < m\right)\right]$$

enforces all pairwise differences to be distinct. Since m is typically given at runtime, the number of differences $(m \times (m-1)/2)$ is not known when compiling the program, and therefore we cannot state the constraint by enumerating all subexpressions literally. Even if we could, it would not be very convenient either since m can be arbitrarily large.

When decomposing using auxiliary variables or subtype polymorphic views, an auxiliary variable (resp. a subtype polymorphic view object) is created for each difference at runtime, and the DISTINCT constraint is posted on these variables (resp. sview objects),

$$\begin{aligned} c &= \bigwedge_{1<i<j<m} \left[a_{ij} = x_i - x_j\right] \\ &\wedge \quad [\text{DISTINCT}(a)] \end{aligned}$$

However, for compiling constraints to type parametric view objects the compiler must visit the full expression top-down at compile time, as explained above. We must therefore find a compact and readable description of a repeating occurrence of a subexpression. Based on the concept of *aggregator* [Hentenryck et al. 2000; Hentenryck and Michel 2005], we solved this problem using the $\text{ALL}(I, S, d, e)$ expression which represents an array where each position holds the expression e instantiated when each variable in I is assigned to a value of the set S and the condition d is satisfied.

Example 8.22 (ALL expression)**.** The constraint c may be represented as follows, where i, j are variables taking values in $\{1, \ldots, m\}$, satisfying $i < j$,

$$c = \left[\text{DISTINCT}\left(\text{ALL}\left(\{i, j\}, \{1, \ldots, m\}, i < j, x_i - x_j\right)\right)\right]$$

For this to work we defined view objects over iterated expressions, that is, expressions that depend on the value of the iterated variables i, j. In this particular case, view objects over iterated expressions are able to save a quadratic number of variables and still benefit from the efficiency of type parametric view objects.

8.8. Summary

This chapter introduced subtype and parametric polymorphic box view objects for efficient propagation of box view models for arbitrary decomposable constraints. We have established

a formal relation between decompositions using view objects and auxiliary variables. Specifically, we have shown under which circumstances using view objects or auxiliary variables for propagating a given view model lead to the same search space. We have also seen how to compile a propagation algorithm using a set of view objects or a set of propagators and auxiliary variables from an arbitrary constraint. Moreover, we have presented two variants of the propagation algorithm based on view objects, which suggest distinct implementations in polymorphic programming languages. In the next chapter we will see that this difference has a significant impact on performance.

Related work

There is a significant amount of work related to the material in this chapter, but since it is available mostly in the form of computer program implementations, we have chosen to discuss it in the end of the next chapter.

Future work

- We have intentionally restricted the instantiation of view models to view models consisting only of box approximations. This was partially because view objects may be implemented efficiently as discussed, but also because presenting an implementation of a generic view model would make the exposition much more complex. In fact, other view models may be propagated using a similar approach. The resulting propagators are correct, but creating efficient corresponding view objects is much more challenging in general. As an example, consider designing a domain view object, that is an object which computes

$$[\![\varphi_f^+([\![S_1]\!]^\delta)]\!]^\delta$$
$$S_1 \leftarrow S_1 \cap [\![\widehat{\varphi}_f([\![S_2]\!]^\delta, [\![S_1]\!]^\delta)]\!]^\delta$$

  A domain view object must be able to compute δ-domains, which cannot be done in constant time for most functions f as it is the case for β-domains. In fact, the above operations require time exponential in the arity of f, which renders domain view objects unpractical for most functions. It was found that for specific classes of functions, namely injective functions, computing δ-domain objects may be done in linear time [Tack 2009]. As future work it would be interesting to see if incrementality (discussed in part I of this dissertation) could be used to improve the runtime cost of computing domain view objects.

- Providing a domain view (or even φ-view) object over a compact extensional representation of the domain of the view is another possible development. Compact representations of arbitrary tuple sets have been successfully used for propagating extensional constraints [Gent et al. 2007; Cheng and Yap 2008]. Since such representations are exponential in the worst case, using views over subexpressions could be a way to achieve a finer space-time tradeoff.

Chapter 9.

Implementation and Experiments

This chapter is divided in two parts. The first provides some details on how view objects presented previously may be implemented, either in a logic programming setting (§9.1) or in a strongly typed imperative language (§9.2). The second part evaluates decomposition methods experimentally, summarizing the tested models and benchmarks (§9.3), and discussing the results obtained (§9.4).

9.1. View models in Logic Programming

Propagators for box view models may be implemented straightforwardly in Prolog. A view object for a given function f is defined by a set of predicates, one for each method, of the form METHOD(f, b) where b is either an input or output parameter depending on the method. Prolog's unification mechanism allows a literal implementation of parametric polymorphic view stores with respect to the previously described conceptual model. The rule repository is the Prolog predicate store, and rewriting rules are simply stated as Prolog predicates.

Example 9.1. Partial implementation of bnd$[x:\mathsf{dvar}]$, bnd$[X+Y]$, and $\pi[X]$ in (eclipse) Prolog:

```
% bnd[X+Y]
bnd_getmin(X+Y,R):- bnd_getmin(X,R1),
                    bnd_getmin(Y,R2),
                    R is R1+R2.
bnd_updmin(X+Y,I):- bnd_getmax(X,MX),
                    bnd_getmax(Y,MY),
                    IX is I-MY, IY is I-MX,
                    bnd_updmin(X,IX),
                    bnd_updmin(Y,IY).

% bnd[x:dvar]
bnd_getmin(X,R):- get_min(X,R).
bnd_updmin(X,I):- X #>= I.
```

```
% pi[X]
propagate(X):- bnd_updmin(X,1).
```

The above program implements type parametric box view objects. The optimization consisting of inlining the predicates that define the propagator, described in section 8.5.2, may be done either implicitly by the Prolog compiler, or explicitly if the Prolog engine supports it, which is not uncommon (e.g. expand_goal/2 in eclipse Prolog).

9.2. View models in strongly typed programming languages

Implementing our conceptual box view model in a strongly typed programming language is possible if some sort of type polymorphism support is available in the language. Most if not all popular strongly typed programming languages have built in support for subtype polymorphism, either by overloading or through the use of inheritance in the case of object oriented programming languages. Recently, parametric polymorphism has been introduced in some object oriented programming languages such as c++, java and c#.

We have implemented box view models in c++. Since c++ supports both subtype and parametric polymorphism, we were able to integrate the two variants of our model within the constraint solver engine, therefore obtaining a fair experimental platform. Let us detail these implementations.

9.2.1. Subtype polymorphism

Subtype polymorphism is available in C++, C#, through the use of inheritance. In this setting we need to define an abstract interface for box view objects, which is essentially the implementation of equation 8.1 on page 114.

```
class IBox {
    virtual int getMin()=0;
    virtual int getMax()=0;
    virtual void updMin(int i)=0;
    virtual void updMax(int i)=0;
};
```

A box view object for a specific function is a kind of a box view object, that is it implements the box view object interface.

Example 9.2. The following class defines bnd$[X+Y]$, the subtype polymorphic box view object for the addition of two box view objects:

```
class Add2 : IBox {
    Add2(IBox* x, IBox* y) : x(*x),y(*y) {}
    virtual int getMin() { return x.getMin()+y.getMin(); }
    virtual int getMax() { return x.getMax()+y.getMax(); }
    virtual void updMin(int i)
    { x.updMin(i-y.getMax()); y.updMin(i-x.getMax());}
    virtual void updMax(int i)
    { x.updMax(i-y.getMin()); y.updMax(i-x.getMin());}
    IBox& x;
    IBox& y;
};
```

Compiling a given constraint into subtype polymorphic box view objects is straightforward since in c++ expressions are evaluated bottom-up. Below are set of convenience functions which may be used to create subtype polymorphic view box objects for a binary addition.

```
IBox* add(IBox* x,IBox* y)
{ return new Add2(x,y); }

IBox* operator+(IBox* x,IBox* y)
{ return new Add2(x,y); }
```

The user may then create box view objects for arbitrary expressions in C++ using a clean syntax:

```
DomVar a,b,c;
a+b*c;
add(a,mul(b,c));
```

9.2.2. Parametric polymorphism

The fact that the c++ compiler evaluates expressions bottom-up makes the implementation of parametric polymorphic view objects slightly more complex, since as we have seen in the previous chapter, need to be compiled top-down. The solution we propose breaks the compilation algorithm in two phases. The first phase creates a syntactic representation of the expression, called a type parametric relation object, using the natural bottom-up evaluation order intrinsic in the language. Type parametric relations captures the data and the type of the objects and operations involved in the constraint. After the full constraint is compiled, we use the obtained relation object for instantiating the required view objects.

We will use templates for defining type parametric relations, since this is the language mechanism available in c++ to support type parametric polymorphism. The following template defines generic binary relations, where “Op” is a type describing the operator, and “X” and “Y” are types of the operands.

```
template<class Op, class X, class Y>
class Rel2 {
    Rel2(X x, Y y) : x(x),y(y) {}
    X x;
    Y y;
};
```

Since any expression may be transformed to a relation object with a unique type, we can create view objects over arbitrary expressions by defining templates over relation objects.

Example 9.3. The following template defines $\mathsf{bnd}[X+Y]$, the parametric polymorphic box view object for the addition of two arbitrary objects:

```
template<class X, class Y>
class Box<Rel2<Add,X,Y> > {
    Box(Rel2<Add,X,Y> r) : x(r.x),y(r.y) {}
    int getMin() { return x.getMin()+y.getMin(); }
    int getMax() { return x.getMax()+y.getMax(); }
    void updMin(int i)
    { x.updMin(i-y.getMax()); y.updMin(i-x.getMax());}
    void updMax(int i)
    { x.updMax(i-y.getMin()); y.updMax(i-x.getMin());}
    Box<X> x;
    Box<Y> y;
};
```

Parametric relation objects are created by a set of convenience functions, such as:

```
template<class X, class Y>
Rel2<Add,X,Y> add(X x,Y y)
{ return Rel2<Add,X,Y>(x,y); }

template<class X, class Y>
Rel2<Add,X,Y> operator+(X x,Y y)
{ return Rel2<Add,X,Y>(x,y); }
```

Note that the above functions only create the relation object for the expression, and not the corresponding view object. Creating the view object is accomplished by providing the relation object to the following function:

```
template<class T>
Box<T> box(T t)
{ return Box<T>(t); }
```

The following code instantiates two parametric box view objects for an expression using the above constructs.

```
DomVar a,b,c;
box(a+b*c);
box(add(a,mul(b,c)));
```

9.2.3. Advantages of subtype polymorphic views

We have seen in section 8.6.3 that propagators using type parametric view objects are theoretically more efficient than those using subtype polymorphic view objects, by assuming that the compiler is able to eliminate function calls. This is often the case when the constraint is stated directly in a strongly typed programming language supporting type parametricity and the code is compiled with a modern compiler. However, this may not be possible if the expression is given as input by the user at runtime, for example if the solver is embedded in an interpreter. In this case the expression was obviously not known when compiling the interpreter, and therefore there is no opportunity for compile time optimizations. We remark however that type parametric views may still be useful in this scenario for defining a set of propagators for a limited number of expressions, which may be enough for domain specific interpreters. An example is to create a propagator for an n-ary sum from a set of binary sums.

9.3. Experiments

In this section we evaluate the performance of previously described decomposition methods on a set of benchmarks. Specifically we are interested in comparing the following models.

9.3.1. Models

VARS This is the classical method for decomposing constraints into primitive propagators introducing one auxiliary variable for each subexpression, as discussed in section 8.4 on page 117.

VARS+GLOBAL This model is similar to the previous but uses global constraints for lowering the number of auxiliary variables. Only a subset of problems support this decomposition in which case we will specifically mention which global constraints are used.

PVIEWS The model that implements the decomposition based on parametric polymorphic view objects, the central topic in the previous chapter.

SVIEWS The decomposition based on subtype polymorphic view objects described in the previous chapter.

CPVIEWS This model is equivalent to the pviews model except that the computations of the view objects are cached in order to guarantee the same propagation as when using auxiliary variables. This was discussed in section 8.6.3 on page 123.

VIEWS+GLOBAL Like the VARS+GLOBAL model, this model uses a combination of some type of views and a global constraint propagator.

All the above decomposition models were implemented in CaSPER. Additionally, we also implemented the first two in Gecode [Gecode 2010] as an external reference, denoted GECODE-VARS and GECODE-VARS+GLOBAL respectively.

9.3.2. Problems

The set of benchmarks covers a total of 22 instances from 6 different problems. Before we present them in detail, we should make a few general considerations.

For each given instance we used the same labeling heuristics (or no heuristics at all) for testing the above models. This means that, for each instance, the solvers resulting from the implementation of the above models explore exactly the same search space, unless the decompositions have different propagation strength, which may occur as we have already seen.

In the following exposition we will mostly focus on the decomposable constraints for which our models apply. When describing the model, we may choose to ignore other necessary, redundant, or symmetry breaking constraints that we used in our implementations. These were kept constant across all implementations of the above models for each benchmark and therefore do not influence the conclusions. For additional information, we provide references to detailed descriptions of the problems in the online constraint programming benchmark database CSPLib [Gent and Walsh 1999]. The source code for all the solvers may be obtained from the author upon request.

Systems of linear equations

This experiment consists in solving a system of linear equations. Linear equations are usually integrated in Constraint Programming using a specific global constraint propagator. The goal

of the experiment is therefore to assess the overhead of decomposing expressions using the presented models compared to a decomposition which uses a special purpose algorithm, i.e. a global constraint.

Each system of linear equations is described by a tuple $\langle n, d, c, a, s|u\rangle$ where n is the number of variables in the problem, d is the uniform domain size, c is the number of linear equations, a is the number of terms in each equation, and the last term denotes if the problem is (s)atisfiable or (u)nsatisfiable. Each problem is defined by

$$\bigwedge_{i=1}^{c} \sum_{v \in p(i)} v = t$$

where $p(i)$ is a function returning a combination of a variables for the equation i, selected randomly from the full set of C_a^n possible combinations. The independent term t in each equation was selected randomly with a uniform probability from the interval $[a \dots a \times d]$. Different random seeds were experimented in order to generate difficult instances.

The a-ary sum constraints were decomposed using binary sums implementing a subset of the previously described models, namely VARS, SVIEWS, and PVIEWS. Model VARS+GLOBAL used global constraint propagators for the a-ary sums.

Systems of nonlinear equations

The second experiment considers systems of nonlinear equations. These problems arise often in practice, and since the decomposition to special purpose propagators is not so direct as in previous case, it provides a realistic opportunity to apply the previously discussed models.

A system of nonlinear equations is described by a tuple $\langle n, d, c, a_1, a_2, s|u\rangle$ where a_1 is the number of terms in each equation, each term is composed of a product of a_2 factors, and all remaining variables have the same meaning as before. Each system of nonlinear equations is formally defined as:

$$\bigwedge_{i=1}^{c} \sum_{j=1}^{a_1} \prod_{v \in p(i,j)} v = t$$

where $p(i, j)$ is a function returning a combination of a_2 variables for the term j of equation i, selected randomly from the full set of $C_{a_2}^n$ possible combinations.

The tested models consist on the decomposition into binary sums and products using auxiliary variables exclusively (VARS) and using view models (SVIEWS, PVIEWS, and CPVIEWS). We also tested two models where each product is decomposed using either auxiliary variables or views, projected to a variable x_i, and a sum propagator is used to enforce $\sum_{i=1}^{a_1} x_i$ for each equation (VARS+GLOBAL and PVIEWS+GLOBAL respectively).

Social golfers (prob10 in CSPLib)

The Social golfers problem consists in scheduling a golf tournament. The golf tournament lasts for a specified number of weeks w, organizing g games each week, each game involving s players. There is therefore a total of $g \times s$ players participating in the tournament. The goal is to come up with a schedule where each pair of golfers plays in the same group at most once.

This problem may be solved efficiently in Constraint Programming using a 3-dimensional matrix x of $w \times g \times s$ integer domain variables, where each variable identifies a golf player. For two groups of players G_1, G_2, the MEETONCE constraint ensures that any pair of players in one group does not meet in the other group,

$$\text{MEETONCE}(G_1, G_2) \quad = \quad \left[\sum_{x \in G_1, y \in G_1} (x = y) \leq 1 \right]$$

This constraint is then used to impose that each pair of players meets at most once during the entire tournament,

$$\bigwedge_{1 \leq w_i < w_j \leq w} \text{MEETONCE}\left(\{x_{w_i,g_i,s_i} : 1 \leq g_i \leq g, 1 \leq s_i \leq s\}, \{x_{w_j,g_j,s_j} : 1 \leq g_j \leq g, 1 \leq s_j \leq s\} \right)$$

We tested a subset of our models for propagating the MEETONCE constraint, namely PVIEWS+GLOBAL, SVIEWS+GLOBAL, CPVIEWS+GLOBAL, and VARS+GLOBAL. View based models implement the MEETONCE constraint using the above expression directly, using a global propagator for the sum constraint. This constraint translates almost literally to CaSPER,

$$\text{MEETONCE}(G_1, G_2) \quad = \quad \left[\text{SUM}\left(\text{ALL}\left(x \in G_1, y \in G_2, x = y \right) \right) \leq 1 \right]$$

The VARS+GLOBAL model implements the traditional decomposition using a set b of s^2 auxiliary boolean domain variables,

$$\text{MEETONCE}(G_1, G_2) \quad = \quad \bigwedge_{x \in G_1, y \in G_2} \left[b_i = (x = y) \right] \tag{9.1}$$

$$\wedge \quad \left[\sum_{i=1}^{s^2} b_i \leq 1 \right] \tag{9.2}$$

In this case we used the (reified) equality propagator for each equation in the conjunction of eq. 9.1, and a sum global propagator for equation 9.2.

Golomb ruler (prob6 in CSPLib)

A Golomb ruler of m marks and length x_m is a set of m integers,

$$0 = x_1 < x_2 < \ldots < x_m$$

such that the $m(m-1)/2$ differences $x_i - x_j$, $1 \leq j < i \leq m$ are distinct. The Golomb ruler problem is an optimization problem, where the goal is to find the smallest possible Golomb ruler with a given number of marks.

This problem makes use of a constraint of the form

$$\text{DISTINCT}\left(\{x_i - x_j : 1 \leq j < i \leq m\}\right)$$

enforcing that the pairwise differences $x_i - x_j$ are all distinct. The classical decomposition for this constraint (VARS+GLOBAL) introduces one auxiliary variable for each pairwise difference, and makes use of the distinct global propagator for the DISTINCT constraint,

$$\begin{array}{ll} \bigwedge_{1 \leq i < j \leq m} & \left[a_{i,j} = x_i - x_j\right] \\ \wedge & \left[\text{DISTINCT}\,(a)\right] \end{array}$$

Using views avoids introducing the set of auxiliary variables a. Instead, the constraint is used directly as follows,

$$\text{DISTINCT}\left(\text{ALL}\left(1 \leq i \leq m, 1 \leq j \leq m, j < i, x_i - x_j\right)\right)$$

and enforced with the bounds complete propagator introduced by Lopez-Ortiz et al. [2003]. In our benchmarks we solved the decision version of this problem, i.e. we provided the size of the ruler x_m as a parameter of the problem, and asked for the ruler satisfying the constraints.

Low autocorrelation binary sequences (prob5 in CSPLib)

The goal is to construct a binary sequence $S = x_1, \ldots, x_n$ of length n, where $D(x_i) = \{-1, 1\}$, $1 \leq i \leq n$, that minimizes the autocorrelations between bits, i.e. that minimizes the following expression,

$$m = \sum_{i=1}^{n-1} \left(\sum_{j=1}^{n-i-1} x_j \times x_{j+i+1} \right)^2 \tag{9.3}$$

The VARS+GLOBAL model decomposes the above expression using three sets of auxiliary variables a, b, and c, and sum constraints as follows,

$$\begin{array}{ll} \bigwedge_{i=1}^{n-1} \bigwedge_{j=1}^{n-i-1} & \left[x_j \times x_{j+i+1} = a_{i,j}\right] \\ \bigwedge_{i=1}^{n-1} & \left[\sum_{j=1}^{n-i-1} a_{i,j} = b_i\right] \\ \bigwedge_{i=1}^{n-1} & \left[b_i^2 = c_i\right] \\ \wedge & \left[m = \sum_{i=1}^{n-1} c_i\right] \end{array}$$

In this experiment we implemented three variants of the PVIEWS model. The PVIEWS-PARTIAL1 model implements the following decomposition which is just slightly more compact than the above,

$$\begin{array}{ll} \bigwedge_{i=1}^{n-1} & \left[\sum_{j=1}^{n-i-1} x_j \times x_{j+i+1} = b_i\right] \\ \bigwedge_{i=1}^{n-1} & \left[b_i^2 = c_i\right] \\ \wedge & \left[m = \sum_{i=1}^{n-1} c_i\right] \end{array}$$

The PVIEWS-PARTIAL2 model further decreases the number of auxiliary variables,

$$\begin{array}{ll} \bigwedge_{i=1}^{n-1} & \left[\sum_{j=1}^{n-i-1} x_j \times x_{j+i+1} = b_i\right] \\ \wedge & \left[m = \sum_{i=1}^{n-1} b_i^2\right] \end{array}$$

Finally, the PVIEWS, SVIEWS and CPVIEWS model implements expression 9.3 directly, without introducing any auxiliary variable.

Fixed-length error correcting codes (prob36 in CSPLib)

This problem involves generating a set of strings from a given alphabet which satisfy a pairwise minimum distance. Each instance is defined by a tuple $\langle a, n, l, d\rangle$ where a is the alphabet size, n is the number of strings, l is the string length, and d is the minimum distance allowed between any two strings. For measuring the distance between two strings we have used the Hamming distance on two instances and the Lee distance on the other two. For two arbitrary

strings x, y of length l, these measures are defined as follows,

$$\begin{aligned} \text{HAMMING}(x,y) &= \sum_{i=1}^{l} (x_i \neq y_i) \\ \text{LEE}(x,y) &= \sum_{i=1}^{l} \min\left(|x_i - y_i|, a - |x_i - y_i|\right) \end{aligned} \tag{9.4}$$

This problem was modeled by a matrix x of $n \times l$ integer domain variables, where each variable $x_{i,j}$ can take a value in $1 \ldots a$ corresponding to the symbol of string i in position j. Then, distance constraints are imposed between each pair of strings,

$$\bigwedge_{1 \leq i_1 < i_2 \leq n} \text{DISTANCE}\left(\{x_{i_1,j} : 1 \leq j \leq l\}, \{x_{i_2,j} : 1 \leq j \leq l\}\right) \geq d$$

The VARS+GLOBAL model decomposes distance constraints using auxiliary variables and sum constraints. Note that in the case of the Lee distance, a total of $4l$ auxiliary variables are introduced for each distance constraint. The SVIEWS, PVIEWS, and CPVIEWS models implement both distance functions without any extra variables.

9.3.3. Setup

The code for the first two experiments was compiled with the gcc-4.2.4 C++ compiler, while the remaining experiments were compiled with gcc-4.4.3. All experiments were executed on an Intel Core 2 Duo @ 2.20GHz, using Linux-2.6.31.6. The versions of the CaSPER and Gecode solvers were the most recently available, respectively revision 548 and version 3.3.1. Each benchmark was repeated until the standard deviation of the runtime was below 2% of the average time, and then the minimum runtime was used.

9.4. Discussion

The results of all benchmarks are detailed in appendix B.3 from which we drawn the following conclusions.

9.4.1. Auxiliary variables Vs Type parametric views

Table 9.1 compares the runtime of the best model using auxiliary variables, i.e. either VARS or VARS+GLOBAL, with the runtime of the best model that uses type parametric views, i.e. either PVIEWS or PVIEWS+GLOBAL. View objects do not intend to be a replacement for global constraint propagators, and therefore this table shows how much the runtime of a constraint program may be improved when using the best available tools.

	mean	stddev	min	max
Systems of linear equations	1.07	1.15	0.91	1.21
Systems of nonlinear equations	0.42	1.08	0.36	0.46
Social golfers	0.62	1.09	0.59	0.69
Golomb ruler	0.74	1.02	0.73	0.75
Low autocorrelation binary sequences	0.35	1.02	0.35	0.36
Fixed-length error correcting codes	0.44	1.38	0.28	0.57
All	0.56	1.51	0.28	1.21
All except linear	0.49	1.34	0.28	0.75

Table 9.1.: Geometric mean, standard deviation, minimum and maximum of the ratios defined by the runtime of best performing model using views over the runtime of the best performing model using auxiliary variables, on all benchmarks.

Before we take a global view on the results in this table, let us focus on the special case of the benchmark involving systems of linear equations. We recall that this benchmark should not be considered as part of a realistic application of views or auxiliary variables since it may be modeled using a global constraint propagator exclusively. However, modeling the global sum constraint using type parametric views over binary sums was only 7% worst on average, which is nevertheless remarkable.

For all other benchmarks, using type parametric views instead of auxiliary variables was consistently better, approximately twice as fast on (geometric) average, and always more than 25% faster. An interesting particular case are the two instances of the “Fixed length error correcting codes” problem using the Lee distance. These instances are in fact the only for which decomposing using auxiliary variables could be recommended. This is because there is a subexpression which occurs twice in the expression, and therefore can be represented by the same auxiliary variable (see equation 9.4), possibly leading to a smaller search tree. Even without this optimization, the solver using type parametric views was more than twice as fast on average on these instances.

Regarding propagation, even if using auxiliary variables may sometimes lead to smaller search trees, the difference was not so significant in our experiments. In fact, for those instances where using auxiliary variables increases propagation strength compared to views, the discrepancy in the number of fails was only of 6% on average, and never more than 20% (table 9.2). On the other hand, the number of propagations using auxiliary variables is on average an order of magnitude greater than when using views, which partially explains the superior performance of type parametric view models (shown in the appendix).

	mean	stddev	min	max
Systems of nonlinear equations	1.06	1.08	1.01	1.16
Golomb ruler	1.00	1.00	1.00	1.01
Fixed-length error correcting codes	1.20	1.00	1.20	1.20
All	1.06	1.08	1.00	1.20

Table 9.2.: Geometric mean, standard deviation, minimum and maximum of the ratios defined by the number of fails of the best performing solver using views over the number of fails of the best performing solver using auxiliary variables, on all instances of each problem where the number of fails differ.

	mean	stddev	min	max
Systems of linear equations	0.57	1.63	0.33	0.96
Systems of nonlinear equations	0.83	1.07	0.77	0.93
Social golfers	0.77	1.05	0.73	0.81
Golomb ruler	0.89	1.03	0.87	0.91
Low autocorrelation binary sequences	0.41	1.02	0.41	0.42
Fixed-length error correcting codes	0.55	1.43	0.4	0.77
All	0.67	1.4	0.33	0.96

Table 9.3.: Geometric mean, standard deviation, minimum and maximum of the ratios defined by the runtime of the solver implementing the PVIEWS model over the runtime of the solver implementing the SVIEWS model, on all benchmarks.

9.4.2. Type parametric views Vs Subtype polymorphic views

Recall from section 8.6.2 on page 122 that solvers using type parametric view objects are able to avoid a linear number of function calls for each access or update due to code inlining optimizations. Table 9.3 shows how this optimization improves performance in practice, in particular for problems involving a large number of subexpressions where it can be 66% times faster.

9.4.3. Caching type parametric views

Caching type parametric views is a way to overcome the loss of propagation strength in some problems compared to the decomposition based on auxiliary variables. In our experiments, cached parametric view objects increased runtimes considerably, even if exploring a smaller search space. Table 9.4 summarizes the results.

As we have seen in table 9.2, the search space visited by type parametric views is not much larger than when performing caching, which certainly influences these results. Note that caching

	mean	stddev	min	max
Systems of nonlinear equations	0.57	1.13	0.46	0.65
Social golfers	0.6	1.06	0.56	0.63
Golomb ruler	0.83	1.03	0.8	0.85
Low autocorrelation binary sequences	0.82	1	0.82	0.82
Fixed-length error correcting codes	0.57	1.44	0.33	0.72
All	0.64	1.27	0.33	0.85

Table 9.4.: Geometric mean, standard deviation, minimum and maximum of the ratios defined by the runtime of the solver implementing the PVIEWS model over the runtime of the solver implementing the CPVIEWS model, on all benchmarks.

type parametric views is still more efficient than using auxiliary variables (15% better on average).

9.4.4. Competitiveness

Modeling decomposable constraints using type parametric views makes CaSPER competitive with the state-of-the-art Gecode solver. This may be seen by comparing the results presented in table 9.5 with table 9.6. In the first table we compare the runtimes obtained by running the same model on both solvers, i.e. VARS+GLOBAL and GECODE-VARS+GLOBAL. The second table compares the PVIEWS model against GECODE-VARS+GLOBAL, which are the best models that can be implemented in both platforms using the available modeling primitives. While CaSPER is worse in all but one problem when using auxiliary variables and global propagators, it becomes very competitive when using type parametric views. We believe that the discrepancy observed in the "Fixed length error correcting codes" benchmark is related to aspects of the architecture of both solvers which are orthogonal to the tested models, in particular the performance of labeling which seems to vary with the number of variables in Gecode, while remaining approximately constant in CaSPER (see figure 9.1).

9.5. Summary

We have seen how box view objects may be implemented using several language paradigms, with a focus on strongly typed languages, namely c++. Decomposition models based on auxiliary variables, subtype polymorphic views, and type parametric views were implemented for a number of well known benchmarks, and the results were discussed. We observed that type parametric views are clearly more efficient than models resulting from the other decomposition/compilation methods for all benchmarks. Moreover, we have seen that this technique

	mean	stddev	min	max
Systems of linear equations	1.32	1.22	0.99	1.54
Systems of nonlinear equations	1.42	1.12	1.2	1.63
Social golfers	1.43	1.28	1.15	1.86
Golomb ruler	1.59	1.13	1.38	1.71
Low autocorrelation binary sequences	1.45	1.01	1.44	1.46
Fixed-length error correcting codes	0.19	1.93	0.09	0.33
All	0.99	2.34	0.09	1.86
All except fixed-length error correcting codes	1.43	1.16	0.99	1.86

Table 9.5.: Geometric mean, standard deviation, minimum and maximum of the ratios defined by the runtime of the CaSPER solver implementing the VARS+GLOBAL model over the runtime of the Gecode solver implementing the VARS+GLOBAL model, on all benchmarks.

	mean	stddev	min	max
Systems of linear equations	1.42	1.3	0.98	1.78
Systems of nonlinear equations	0.59	1.12	0.5	0.68
Social golfers	0.89	1.28	0.67	1.1
Golomb ruler	1.18	1.15	1	1.29
Low autocorrelation binary sequences	0.51	1.03	0.5	0.52
Fixed-length error correcting codes	0.08	2.47	0.03	0.19
All	0.55	2.87	0.03	1.78
All except fixed-length error correcting codes	0.85	1.54	0.5	1.78

Table 9.6.: Geometric mean, standard deviation, minimum and maximum of the ratios defined by the runtime of the CaSPER solver implementing the PVIEWS model over the runtime of the Gecode solver implementing the VARS+GLOBAL model, on all benchmarks.

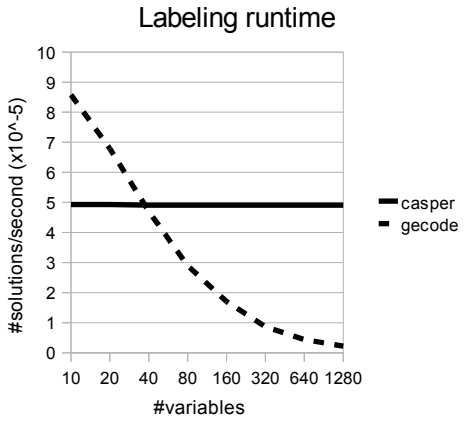


Figure 9.1.: Number of solutions per second when enumerating all solutions of a CSP with a given number of variables (in the xx axis), domain of size 8, and no constraints.

improves the performance of CaSPER to the point of being competitive with Gecode, which is regarded as one of the most efficient solvers available.

Related work

- Most popular Prolog systems [SICStus 2006; ECLiPSe 2010; GNUProlog 2008] automatically decompose constraints over arbitrary arithmetic expressions using auxiliary variables. The use of indexicals and goal expanded constraints [SICStus 2006] is advised as a way to avoid introducing auxiliary variables. We are not aware of a published description of the implementation of the propagator achieved in this case, but we suspect it should be comparable to subtype polymorphic views model, since Prolog, being an untyped interpreted language, should not support type parametric polymorphism. Given the simplicity of the Prolog implementation of this model (see section 9.1) it is unfortunate that it requires special features of the language and is not transparent to the user.

- ILog Solver is well known constraint solver written in C++ supporting posting constraints over arbitrary expressions. It processes an expression bottom-up by converting its subexpressions to *constrained expressions* [ILOG 2003a], which are exactly subtype polymorphic view objects.

- Choco solver [Choco 2010] is a constraint solver written in Java which also allows posting constraints over arbitrary expressions. In this system the user may ask the solver to either decompose the expression using the auxiliary variables model, or to compile the expression to an extensional representation.

- Gecode solver [Gecode 2010] is a very efficient constraint solver written in C++. At the time of this writing, Gecode does not support constraints over arbitrary expressions. Instead, it requires the user to provide the decomposition by means of auxiliary variables. However, this solver makes an extensive use of views very much as we describe in section 8.5.2. The main difference is that in Gecode, views are available as an extension mechanism for designing new propagators *explicitly*. That is, unlike our model, there is no automatic process for instantiating the required views for a given arbitrary expression. For more information see [Schulte and Tack 2005].

Part III.

Applications

Chapter 10.

On the Integration of Singleton Consistencies and Look-Ahead Heuristics

Complete constraint programming solvers have their efficiency dependent on two complementary components, propagation and search. Constraint propagation is a key component in constraint solving, eliminating values from the domains of the variables with polynomial algorithms. When propagation is effective, such algorithms reduce the search space by some combinatorial (i.e exponential) factor, with a polynomial cost. The other component, search, aims at finding solutions in the remaining search space, and is usually driven by heuristics both for selecting the variable to enumerate and the value that is chosen first.

Typically, these components are independent. In particular, heuristics take into account some features of the remaining search space, and some structure of the problem to take decisions. Clearly, the more information there is, the more likely it is to get a good (informed) heuristics. In many search problems addressed in Artificial Intelligence, additional information is often obtained by performing a limited amount of look-ahead, and assessing the state of search some steps ahead. For example in two players games, like chess or checkers, rather than considering the current state of the board to decide the move to make, a number of moves by both players can be simulated, such that board configurations with a more advanced phase of the game (thus better informed) can be taken into consideration.

Although better solver performances can be achieved by improving any of the two components (propagation and search) they are seldom, if ever, integrated. The main reason for this lies in that typical propagation procedures, such as maintenance of local bounds or domain consistency, are basically local filtering algorithms. Recently, a lot of attention has been given to a class of algorithms which analyze look-ahead what-if scenarios: what would happen if a variable x takes some value v? Such look-ahead analysis (typically done by subsequently maintaining some local consistency on the constraint network) may detect that value v is not part of any solution, and eliminate it from the domain of variable x. This is the purpose of the different variants of Singleton Consistency (SC) [Debruyne and Bessière 1997; Barták and Erben 2004; Bessière and Debruyne 2005; Lecoutre and Cardon 2005].

In this chapter we propose to go one step further of the above approaches. On the one hand, by recognizing that SC propagation is not very cost-effective [Prosser et al. 2000], we propose

to restrict it to those variables more likely to be chosen by the variable selection heuristics (§10.1). More specifically, we assume that there are often many variables that can be selected and for which no good criteria exists to discriminate them. This is the case with the first-fail (FF) heuristics, where often there are many variables with 2 values, all connected to the same number of other variables (as is the case with complete graphs). Hence the information gain obtained from SC propagation is used to break the ties between the pre-selected variables.

On the other hand, we attempt to better exploit the information made available by the look-ahead procedure, and use it not only to filter values but also to guide search. We thus investigate the possibility of integrating Singleton Consistency propagation procedures with variable and value selection heuristics (§10.2), and analyze the speedups obtained in a number of benchmark problems (§10.3-10.4).

10.1. Singleton consistencies

Let us consider a CSP $P = \langle X, D, C \rangle$ where X is the set of variables, D is the Cartesian product of the variables domains, and C is the set of constraints (see def. 2.5 on page 10). Let n be the number of variables in P, i.e. $n = |X|$.

Definition 10.1 (Singleton θ-consistency)**.** Let d be δ-domain such that $d \subseteq D$, and $d|_{x_i=a}$ be the domain obtained by setting $d(x_i) = \{a\}$, i.e.

$$d|_{x_i=a} = d(x_1) \times \ldots \times d(x_{i-1}) \times \{a\} \times d(x_{i+1}) \times \ldots \times d(x_n)$$

Additionally, let $d|^{\theta}_{x=a}$ be the subdomain $d|^{\theta}_{x=a} \subseteq d|_{x=a}$ which is θ-consistent for all constraints $c \in C$. A domain d is singleton θ-consistent (SC), iff for any value $a \in d(x)$ of every variable $x \in X$, $d|^{\theta}_{x=a}$ is not empty.

Cost-effective singleton consistencies are singleton arc-consistency (SAC) [Debruyne and Bessière 1997] and singleton generalized arc-consistency (SGAC) [Prosser et al. 2000].

To achieve SC in a CSP, procedure SC [Debruyne and Bessière 1997] instantiates each variable to each of its possible values in order to prune those that (after some form of propagation) lead to a domain wipe out (function `SC`).

Once some (inconsistent) value is removed, then there is a chance that a value in a previously revised variable has become inconsistent, and therefore SC must check these variables again. This can happen at most ns times, where n is the number of variables, and s the size of the largest domain, hence SC time complexity is in $O(n^2 s^2 \Theta)$, Θ being the time complexity of the algorithm that achieves θ-consistency on the constraint network. Variants of this algorithm with the same pruning power but yielding distinct space-time complexity trade-offs have been proposed [Barták and Erben 2004; Bessière and Debruyne 2004, 2005; Lecoutre and Cardon 2005]. A related algorithm that considers each variable only once (function `RSC`), has

Function $\mathtt{SC}_\theta(d,X,C)$

Input: A domain d, a set of variables X, and a set of constraints C
Output: A domain $d' \subseteq d$ which is singleton consistent with C

```
repeat
    modified ← false
    foreach x ∈ X do
        ⟨d, modified′⟩ ← SRevise_θ(x, d, C)
        modified = modified ∨ modified′
        if d(x) = ∅ then
            return ∅
until ¬ modified
return d
```

Function $\mathtt{RSC}_\theta(d,X,C)$

Input: A domain d, a set of variables X, and a set of constraints C
Output: A domain $d' \subseteq d$ which is restricted singleton consistent with C

```
foreach x ∈ X do
    ⟨d, modified⟩ ← SRevise_θ(x, d, C)
    if d(x) = ∅ then
        return ∅
return d
```

better runtime complexity $O(nsΘ)$, but achieves a weaker consistency, is called restricted singleton consistency (RSC) [Prosser et al. 2000].

Note that both algorithms use function `SRevise` which prunes the domain of a single variable by trying all of its possible instantiations.

10.2. Informed decision making

Another possible trade-off between run-time complexity and pruning power is to enforce singleton consistency on a subset of variables $S \subseteq X$. We identified two possible goals which condition the selection of S : filtering and decision making. From a filtering perspective, S should be the smallest subset where (restricted) singleton consistency can actually prune values, and although this is not known *a priori,* approximations are possible by exploring incrementality

Function $\texttt{SRevise}_\theta(x,d,C)$

Input: A variable x, domain d, and a set of constraints C
Output: A domain $d' \subseteq d$ where $d|^\theta_{x=a}\, \forall a \in d(x)$ is θ-consistent with C

```
1 modified ← false
2 foreach a ∈ d(x) do
3     if Propagate_θ(d, C ∪ [x = a]) = ∅ then
4         d(x) ← d(x) \ a
5         modified ← true
6
7 return ⟨d, modified⟩
```

and value support [Barták and Erben 2004; Bessière and Debruyne 2005]. On the other hand, S may be selected for improving the decision making process, in particular of variable selection heuristics that are based on the cardinality of the current domains. In this case, the pruning resulting from enforcing singleton consistency is used as a mechanism to break ties both in the selection of variable and in the choice of the value to enumerate.

Observing the general preference for variable heuristics which select smallest domains first, we propose defining S as the set of variables whose domain cardinality is below a given threshold t.

Definition 10.2. Let $X_{\leq t}$ be the subset of variables in X having domains with cardinality less or equal to t, i.e.

$$X_{\leq t} = \{x_i \in X : |d(x_1)| \leq t\}$$

We denote by $\text{RSC}_{\leq t}(d,X,C)$ and $\text{SC}_{\leq t}(d,X,C)$, respectively, the algorithms $\text{RSC}(d,X_{\leq t},C)$ and $\text{SC}(d,X_{\leq t},C)$.

A further step in integrating singleton consistencies with search heuristics is to explore information regarding the subproblems that are generated each time a value is tested for consistency. We propose a class of look-ahead heuristics (LA) for any CSP which reason over the number of solutions in the current domain $d \subseteq D$, given by a function $\sigma(d)$, collected while enforcing singleton consistency. Although there is no known polynomial algorithm for computing σ (finding the number of solutions of a CSP is a #P-complete problem), there exists a number of naive as well as more sophisticated approximation functions [Gent et al. 1996; Kask et al. 2004]. We conjecture that by estimating the size of the solution space for each possible instantiation, i.e. $\sigma\left(d|^\theta_{x=a}\right)$, there is an opportunity for making more informed decisions that will exhibit both better first-failness and best-promise behavior. Moreover, the class of approximations of σ presented below are easy to compute, do not add complexity to the cost of generating the subproblems, and only requires a slight modification of the `SRevise` function.

Function `SReviseInfo`$_\theta$`(`x`,`d`,`C`,`INFO`)`

Input: A variable x, domain d, a set of constraints C, and a reference to the INFO data strucure

Output: A domain $d' \subseteq d$ where $d|^{\theta}_{x=a} \forall a \in d(x)$ is θ-consistent with C

```
1 modified ← false
2 foreach a ∈ d(x) do
3     b ← Propagate_θ(d, C ∪ [x = a])
4     INFO[x, a] ← CollectInfo(b, C)
5     if b = ∅ then
6         d(x) ← d(x) \ a
7         modified ← true
8
9 return ⟨d, modified⟩
```

The `SReviseInfo` function stores in a table, denoted INFO, relevant information to the specific subproblem being considered in each loop iteration. In our case, INFO is an estimation of the subproblem solution space, more formally $\text{INFO}[x,a] = \sigma'\left(d|^{\theta}_{x=a}\right)$ where $\sigma' \approx \sigma$. The table is initialized before singleton consistency enforcement, computed after propagation, and handed to the `SelectVariable` and `SelectValue` functions as shown in function `Solve`.

There are several possible definitions for these functions associated with how they integrate the collected information. Regarding the selection of variable, we identified two FF heuristics which are cheap and easy to compute:

$$\text{VAR}_1(d) = \arg\min_{x \in X}\left(\sum_{a \in d(x)} \sigma'\left(d|^{\theta}_{x=a}\right)\right)$$

$$\text{VAR}_2(d) = \arg\min_{x \in X}\left(\max_{a \in d(x)} \sigma'\left(d|^{\theta}_{x=a}\right)\right)$$

Informally, VAR_1 gives preference for the variable with a smaller sum of the number of solutions for every possible instantiation, while VAR_2 selects the variable whose instantiation with maximum number of solutions is the minimum among all variables. For the selection of value for some variable $x \in X$, a possible BP heuristic is

$$\text{VAL}_1(d,x) = \arg\max_{a \in d(x)}\left(\sigma'\left(d|^{\theta}_{x=a}\right)\right)$$

which simply prefers the instantiation that prunes less solutions from the remaining search space.

Functions VAR_1 and VAL_1 correspond to the minimize promise variable heuristic and maxi-

Function $\texttt{Solve}_\theta(d,C,\text{INFO})$

Input: A domain d, and a set of constraints C
Output: A set of solutions in d satisfying all constraints in C

```
1 INFO ← ∅
2 d ← SC_θ(d, C, INFO)
3 if d = ∅ then
4 |   return ∅
5 if |d| = 1 ∧ ∀c ∈ C, d ⊆ con(c) then
6 |   return d
7 x ←SelectVariable(X, INFO)
8 a ←SelectValue(x, INFO)
9 return Solve_θ(d, C ∪ [x = a]) ∪Solve_θ(d, C ∪ [x ≠ a])
```

mize promise value heuristic defined in [Geelen 1992]. Note that we do not claim these are the best options for the estimation of the search space or the number of solutions. We have simply adopted them for simplicity and for testing the concept (more discussion on section 10.5).

10.3. Experiments

A theoretical analysis on the adequacy of these heuristics as FF or BP candidates is needed, but hard to accomplish. Alternatively, in this section we attempt to give some empirical evidence of the quality of these heuristics by presenting the results of using them combined with constraint propagation and backtracking search on a set of typical CSP benchmarks.

More specifically, the set of experiments presented in this section intend to answer the following three questions:

- Does performing any restricted form of singleton consistency improves the overall solving process compared with performing full singleton consistency? In this case we are interested in assessing the possible advantages of using a faster but less complete propagation algorithm, indepentently of the search heuristic used.

- When solving a problem using (restricted) singleton consistent propagation, does integrating lookahead information in the search heuristic present any advantage compared to using a search heuristic which discards this information?

- How does performing (restricted) singleton consistency and using the look-ahead information to guide search compares with not performing such costly propagation and using other good heuristics which do not require look-ahead information.

10.3.1. Heuristics

LA This heuristic implements the proposed functions VAR_1 and VAL_1. As a first attempt at measuring its potential, a simple measure was used for estimating the number of solutions in a given domain d:

$$\sigma'(d) = \sum_{x \in X} \log_2(|d(x)|)$$

which informally expresses that the number of solutions is correlated to the size of the subproblem search space[1]. Although this is a very rough estimate, we are assuming that it could nevertheless provide valuable information to compare alternatives (see section 10.5).

DOM+*value* This heuristic selects the variable with less values in its domain, and assigns it the value selected by the value heuristic *value*. The value heuristic *value* can be either MIN, which corresponds to selecting the minimum possible value in the domain of the selected variable, or MC that selects the value which minimizes the number of conflicts with the variables connected to the selected variable by some constraint.

DOM/WDEG+*value* This heuristic selects the variable with the smallest ratio between the number of values in its domain and the sum of the weighted degrees of all the constraints it participates. The weighted degree of a constraint is the the number of times it has been proven unsatisfiable since the begining of the search process (see [Boussemart et al. 2004]). The value selection is guided by *value* as described above.

IMPACTS The impact of an assignment $x \mapsto a$ is defined as,

$$I_{x \mapsto a}(d) = \frac{\sigma'(d) - \sigma'\left(d|^{\theta}_{x=a}\right)}{\sigma'(d)}$$

averaged over all domains d since the begining of search. This heuristic selects the variable and value with largest impact (see [Refalo 2004]).

10.3.2. Strategies

heuristic On every choice point uses *heuristic* without any kind of singleton consistency enforcement to make the decision on which variable/value to select, where *heuristic* is one of the heuristics defined in the previous section.

SC+*heuristic* On every choice point first achieves singleton consistency and then uses *heuristic* to make the decision on which variable/value to select.

[1] We use the logarithm since the size of search space can be a very large number.

RSC+*heuristic* On every choice point first achieves restricted singleton consistency and then uses *heuristic* to make the decision on which variable/value to select.

SC2+*heuristic* On every choice point first achieves singleton consistency further restricted to variables with only two values in its domain, i.e. $\mathrm{SC}_{\leq 2}(X, C)$ as discussed previously, and then uses *heuristic* to make the decision on which variable/value to select.

RSC2+*heuristic* On every choice point first achieves restricted singleton consistency further restricted to variables with only two values in its domain, i.e. $\mathrm{RSC}_{\leq 2}(X, C)$ as discussed previously, and then uses *heuristic* to make the decision on which variable/value to select.

10.3.3. Problems

Graph Coloring

Graph coloring consists of trying to assign n colors to m nodes of a given graph such that no pair of connected nodes have the same color. In this section we evaluate the performance of the presented heuristics in two sets of 100 instances of 10-colorable graphs, respectively with 50 and 55 nodes, generated using Joseph Culberson's k-colorable graph generator [Culberson 2010].

A CSP for solving the graph coloring problem was modeled by using variables to represent each node and values to define its color. Difference binary constraints were posted for every pair of connected nodes.

The average degree of a node in the graph d, i.e. the probability that each node is connected to every other node, has been used for describing the phase transition in graph coloring problems [Cheeseman et al. 1991]. In this experiment we started by determining empirically the phase transition to be near $d = 0.6$, and then generated 100 random instances varying d uniformly in the range $[0.5 \ldots 0.7]$.

Random CSPs

Randomly generated CSPs have been widely used experimentally, for instance to compare different solution algorithms. In this section we evaluate the look-ahead heuristics on several random n-ary CSPs. These problems were generated using model C [Gent et al. 2001] generalized to n-ary CSPs, that is, each instance is defined by a 5-tuple $\langle n, d, a, p_1, p_2 \rangle$, where n is the number of variables, d is the uniform size of the domains, a is the uniform constraint arity, p_1 is the density of the constraint graph, and p_2 the looseness of the constraints.

These tests evaluate the performance of the several heuristics in a set of random instances near the phase transition. For this task we used the constrainedness measure κ [Gent et al. 1996] for the case where all constraints have the same looseness and all domains have the

same size:

$$\kappa = \frac{-|\mathscr{C}|\log_2(p_2)}{n\log_2 d}$$

where $|\mathscr{C}|$ is the number of n-ary constraints.

We started by fixing n, d and a, and then computed 100 values for p_2 uniformly in the range $[0.1\ldots0.8]$. For each of these values, a value of p_1 was used such that $\kappa = 0.95$ (problems in the phase transition have typically $\kappa \approx 1$). The value of p_1, given by

$$p_1 = -\kappa\frac{n\log_2 d}{\log_2 p_2} \times \frac{a!\,(n-a)!}{n!}$$

is computed from the first formula and by noting that p_1 is the fraction of constraints over all possible constraints in the constraint graph, i.e.

$$p_1 = |\mathscr{C}|\,\frac{a!\,(n-a)!}{n!}$$

Solutions were stored as positive table constraints and a domain complete propagator implementing GAC-Schema [Bessière and Régin 1997] was used for filtering.

Partial Latin Squares

Latin squares is a well known benchmark which combines randomness and structure, already introduced in chapter 2. Recall that the problem consists in placing the elements $1\ldots n$ in a $n \times n$ grid, such that each element occurs exactly once on the same row or column. A partial Latin squares (or quasigroup completion) problem is a Latin squares problem with a number of preassigned cells, and the goal is to complete the puzzle.

The problem was modeled using the direct encoding, i.e. using an all-different constraint for every row and column, propagated using a domain complete propagator. The dual encoding model, as proposed in [Dotu et al. 2003], was also considered but never improved over the direct model on the presented instances. The value selection heuristic used in conjunction with the dom variable selection heuristic, denoted as mc (minimum-conflicts), selects the value which occurs less in the same row and column of the variable to instantiate. This is reported to be the best known value selection heuristic for this problem [Dotu et al. 2003].

We generated 200 instances of a satisfiable partial Latin squares of a given size, with approximately one third of cells preassigned, using lsencode-v1.1 [Kautz et al. 2001], a widely used random quasigroup completion problem generator.

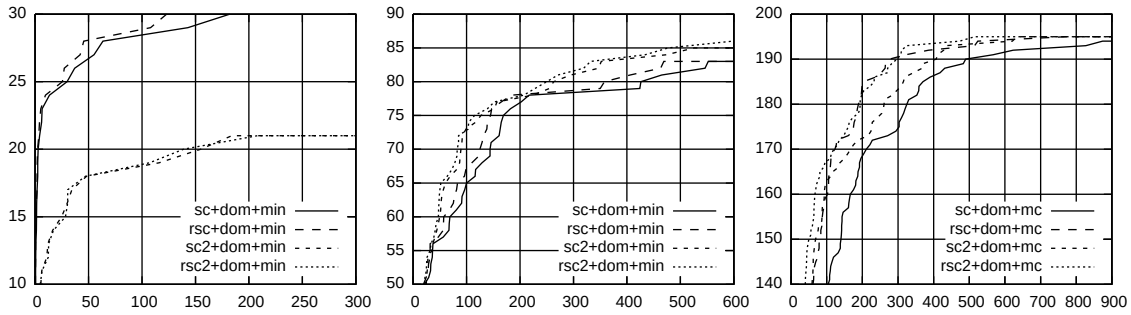


Figure 10.1.: Number of problems solved (yy axis) after a given time period (xx axis). The graphs show the results obtained for, from left to right, the graph coloring instances, the random instances, and latin square instances.

10.4. Discussion

In the following experiments all times are given in seconds, and represent the time needed for finding the first solution. The first set of experiments try to answer the two first questions presented in the previous section. These experiments considered the graph coloring instances with $n = 50$ nodes, the random instances with $n = 50$, $d = 5$ and $a = 3$, and the latin square problem instances of size $n = 30$. The experiments involving graph coloring and random instances were performed on a Pentium4, 3.4GHz with 1Gb RAM, while the experiments involving latin square instances were performed on a Pentium4, 1.7GHz with 512Mb RAM. Timeouts were set to 300, 600 and 900 seconds for the graph coloring, random and latin squares instances respectively. These experiments were implemented in CaSPER, revision 157.

Figure 10.1 compares the results obtained for solving the benchmarks described above when achieving some form of (restricted) singleton consistency, but using an heuristic that does not make use of look-ahead information to guide the search (we used DOM+MIN on the graph and random instances, and DOM+MC on the latin square instances). Note that this experiment only considers using singleton consistency for improving propagation.

We can observe that restricting singleton consistency to subsets of variables with a small domain size has an impact on performance, but it is not clear if it is positive in general - it is positive on the random and latin square instances, and negative on the graph instances. However, using restricted singleton consistency was consistently better than full singleton consistency across all instances, which confirms the results of [Prosser et al. 2000].

In figure 10.2 we assess the advantage of using look-ahead information to guide search in the case this information is available *for free*, that is as a consequence of the enforcement of some form of singleton consistency. For this we compare the performance of the solver using the LA heuristic with the performance of the solver using either DOM+MIN or DOM+MC as in the

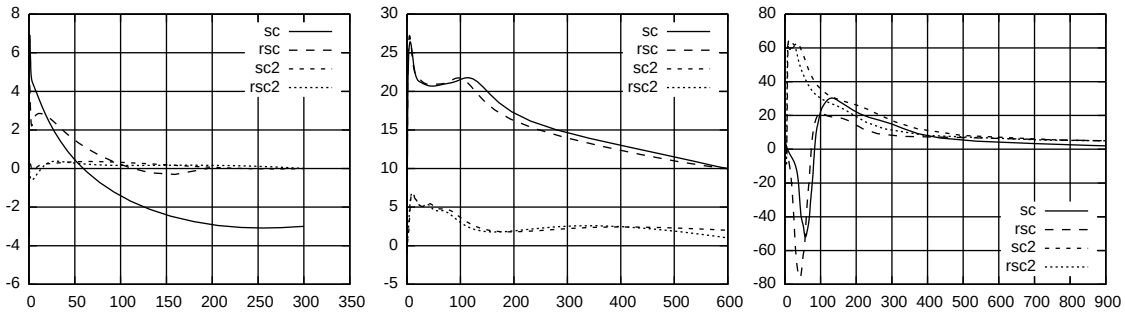


Figure 10.2.: Difference between the number of problems solved when using the LA heuristic and when using the DOM+MIN heuristic (yy axis) after a given time period (xx axis). The graphs show the results obtained for, from left to right, the graph coloring instances, the random instances, and latin square instances.

previous experiment, when both solvers already maintain some form of singleton consistency.

The figure shows that using look-ahead information to guide search when this information is available at no extra cost is significatively advantegeous on all instances except on those of the graph coloring problem, where it did not make much difference. We can also observe that this is true independently of the form of singleton consistency enforced, although the combination with some forms of singleton consistency seem to perform better on some problems. Finally, we note that the fact that the curves tend to zero with time is a consequence of the smaller number of hard problems compared with the number of easy problems (heavy tailed behaviour).

The second set of experiments address the third question presented in the previous section. These experiments considered the graph coloring instances with $n = 65$ nodes, the random instances with $n = 50$, $d = 5$ and $a = 4$, and the latin square problem instances of size $n = 35$. Experiments were performed on a Pentium4, 1.7GHz with 512Mb RAM. Timeout was set to 1800 seconds for any instance. These experiments were implemented in CaSPER, revision 333.

In these experiments we compared the performance of a solver implementing some form of singleton consistency and using the collected information to guide search with the performance of a solver implementing other good heuristic which does not required lookahead information.

The results (fig. 10.3) show that there is no clear winner across all problems. As possibly expected, for problems where propagation achieves less pruning, impact based heuristics are less effective. This is the case of both the random and the graph coloring problems, as illustrated in fig. 10.4, that shows, in log scale, the reduction of the size of the search space, subsequent to each enumeration. In random problems the propagation is poor given the lack of structure of the problem.

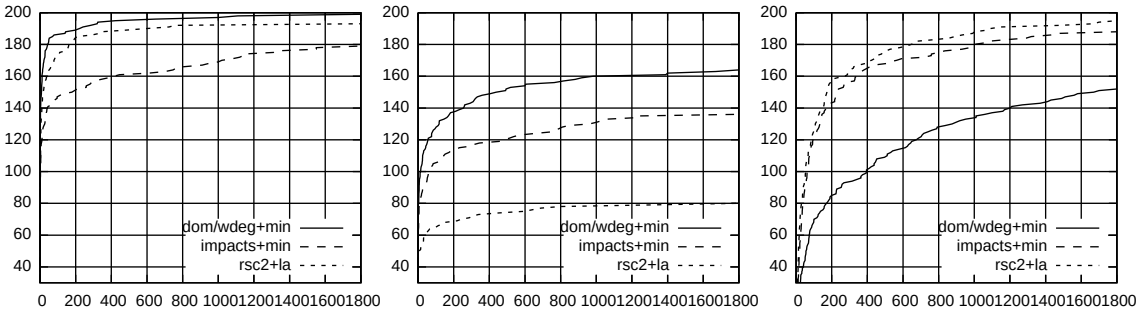


Figure 10.3.: Number of problems solved when using several strategies (yy axis) after a given time period (xx axis). The graphs show the results obtained for, from left to right, the graph coloring instances, the random instances, and latin square instances.

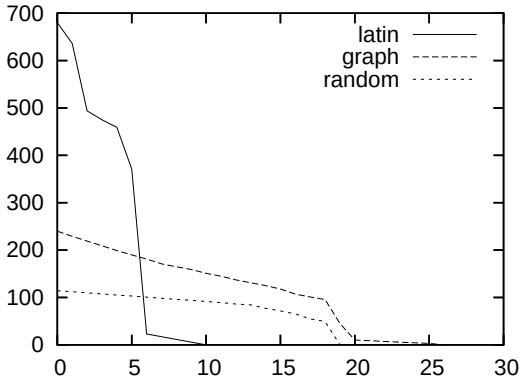


Figure 10.4.: Search space size during solving of a typical instance in each problem.

strategy	#timeouts	avgfails	stddevfails	avgtime	stddevtime
dom/wdeg	**2**	63549	422439	**13.7**	94.6
la	42	**8022**	30312	208.3	366.4

Table 10.1.: Results for finding the first solution to latin-15 with a selected strategy.

The networks for the graph coloring problem exhibit some locality, and propagation mostly affect variables in the same cluster of the variable being assigned, with limited propagation to variables far away in the network. Apparently, the dom/wdeg heuristics, by reasoning at constraint level, is able to "infer" such locality and take advantage of it for variable selection.

In contrast, in the latin square instances modeled by means of all different global constraints locality is not so marked (two variables often share the same constraint, if in the same row or column, or are separated by two constraints, one row and one column, and seldom by more than that, when both row and column pivot elements are already ground). Moreover, generalized arc consistency propagation virtually affects all variables after variable enumeration. The greater impact achieved in these problems, together with the lack of locality to be exploited by a constraint centered impact heuristic such as dom/wdeg, makes variable centered impact heuristics more adequate in this problem. We tried both heuristics in a set of smaller latin square instances modelled with pairwise distinct constraints while maintaining (singleton) node consistency and observed a different ranking, which confirms our thesis (see table 10.1).

10.5. Summary

This chapter presented an approach that incorporates look ahead information for directing backtracking search, and suggested that this could be largely done at no extra cost by taking advantage of the work already performed by singleton consistency enforcing algorithms. We described how such a framework could extend existing SC and RSC algorithms by requiring only minimal modifications. Additionally, a less expensive form of SC was revisited, and a new one proposed which involves revising only a subset of variables. Finally, empirical tests with two common benchmarks and with randomly generated CSPs showed promising results on instances near the phase transition.

Related work

- The impact heuristic introduced by Refalo [2004] suggests improving the variable selection heuristic based on the impact each variable assignment has on the future search space size. In their paper the use of a specific look-ahead procedure for measuring this impact is regarded as costly, and depreciated in favor of a method that accumulates this information

across distinct search branches and/or search iterations (restarts). Their results show that the method eventually converges to a good variable ordering (the value selection heuristic is not considered).

- In [Kask et al. 2004], belief updating techniques are used to estimate the likelihood of a value belonging to some solution. These likelihoods are then used to improve the value selection heuristic and as propagation: if it decreases to zero, the value is discarded from the domain. However, the integration of this kind of propagation with common local propagation algorithms is not explored in that paper.

Future work

- As discussed in the previous section, tests which use singleton consistency on a subset of variables defined by its cardinality were not consistently better or worse than the others, but may be very beneficial sometimes. We think this deserves more investigation, namely testing with more structured problems, and using a distinct selection criteria (other than domain cardinality).

- Improving the FF and BP measures. Look-ahead heuristics presented above use rather naive estimation of number of solutions for a given subproblem compared to, for example, the κ measure introduced in [Gent et al. 1996], or the probabilistic inference methods described in [Kask et al. 2004]. The κ measure, for example, takes into account the individual tightness of each constraint and the global density of the constraint graph. Their work shows strong evidence for best performance of this measure compared with standard FF heuristics, but also point out that the complexity of its computation may lead to suboptimal results in general CSP solving (the results reported are when using forward-checking). Given that we perform a stronger form of propagation and have look-ahead information available, the cost for computing κ may be worth while.

- Include the use of faster singleton consistency enforcing algorithms [Barták and Erben 2004; Bessière and Debruyne 2005], which should be orthogonal to the results presented here, and the use of constructive disjunction during the maintenance of SC, by pruning values from the domains of a variable that does not appear in the state of the problem for all values of another variable.

Chapter 11.

Overview of the CaSPER* Constraint Solvers

This chapter describes the *casperzito* and *casperzao* constraint solvers submitted to the third international CSP solver competition. These solvers are based on the CaSPER solver, the constraint solver integrating the techniques presented throughout this dissertation. Additionally, they implement automatic symmetry detection and symmetry breaking, the lookahead heuristics described in the previous chapter, and search strategy sampling. The structure of the chapter is as follows. We first describe the competition purpose, scope, and rules (§11.1), and then focus on the techniques integrated on solvers submitted to the competition, namely concerning propagation (§11.2), symmetry breaking (§11.3), and search (§11.4). Finally, we present and discuss the results obtained (§11.5).

11.1. The third international CSP solver competition

The goal of this competition is to help identifying successful techniques in constraint solving by comparing solvers in the same environment presented with a large set of benchmarks. The third edition of the competition happened in 2008 and the results were presented during the CP'2008 conference. The rules of the competition are as follows.

Instances are represented in XCSP 2.1, a XML based format of CSP instances, for which all solvers must provide a parser (see [Roussel and Lecoutre 2009]). Solvers may compete in several categories:

2-ARY-EXT only binary constraints defined in extension

2-ARY-INT only binary constraints (some of them being defined by a predicate)

N-ARY-EXT some n-ary constraints (all constraints defined in extension)

N-ARY-INT some n-ary constraints and some constraints defined by a predicate

GLOBAL some global constraints

The categories 2-ARY-EXT and N-ARY-EXT allows defining positive and negative table constraints, i.e. enforce that the values of a given sequence of variables either belong or do not

constraint	value	bounds	domain
positive table	no	no	[Bessière and Régin 1997; Gent et al. 2007] (a)
negative table	custom (b)	no	custom (b)
distinct	custom	[Lopez-Ortiz et al. 2003]	[Régin 1994]
element	no	custom	custom
linear	no	[Yuanlin and Yap 2000]	no
cumulative	no	[Beldiceanu and Carlsson 2002; Mercier and Hentenryck 2008]	no

Table 11.1.: Global constraint propagators used in solvers.

belong to a given table. The set of predicates in 2-ARY-INT and N-ARY-INT allows defining arithmetic and logical expressions, and the available set of global constraints in GLOBAL is restricted to CUMULATIVE, DISTINCT, WEIGHTEDSUM, and ELEMENT.

A solver is given an amount of time (30 minutes) and memory (900MB), and one CPU for each benchmark. The solver must decide if the problem is satisfiable or unsatisfiable. In the former case it must also present a satisfiable solution to the problem. A solver giving a wrong answer is disqualified from that category. The ranking of the solvers is based on the number of solved instances.

11.2. Propagation

11.2.1. Predicates

Except for global constraints (see below), arithmetic predicates used in the competition are enforced in CaSPER using bounds consistency. In many benchmarks used in the competition, predicates are conjoined together in larger predicates, and we found that decomposing them was sometimes penalizing performance. To solve this problem, we translated these conjunctions of predicates to positive or negative table constraints (whichever is smaller) by solving the corresponding subproblems before search, and enforced domain consistency on these constraints during search. We only did this in *zao*, since we were not sure this was a good idea.

11.2.2. Global constraints

Table 11.1 describes the propagators used for the global constraints in the competition.

For the domain consistency propagator for positive table constraints (a) we used the first algorithm [Bessière and Régin 1997] on *zito* and the new trie-based propagator of [Gent et al.

2007] in *zao*. While the latter exhibited better results in our tests, we still found the first more efficient for small arity constraints.

The most popular algorithm for propagating the negative table constraint (b) seems to be the two watched literal scheme introduced for SAT, which achieves value (node) consistency. In [Bessière and Régin 1997], a domain consistency algorithm for this constraint that makes heavy use of hashing for checking disallowed tuples was presented. We have extended the work of [Gent et al. 2007] which focus on positive table constraints to handle negative table constraints as well. While we also base our idea in the trie data structure, this is in fact a completely different propagator which has the advantage of performing much cheaper tests compared to the hashing proposal. For the competition we used the value consistency propagator for negative table constraints in *zito* and our new trie-based propagator in *zao*.

11.3. Symmetry breaking

We mostly followed the ideas of Puget [2005] for automatic symmetry detection using computational group theory and [Aloul et al. 2006; Puget 2005] for symmetry breaking. The basic idea of the detection process is to translate the given CSP to a graph which expresses the symmetries associated with each constraint. The automorphism group of this graph defines the set of symmetries in the original CSP. Puget shows how to translate some common global constraints, e.g. the alldifferent constraint. Extending the idea for the predicates and global constraints used in the competition is straightforward. Additionally, both solvers perform a small amount of symbolic computation in order to circumvent some situations where the symmetries in the CSP would be hidden by the formulation. Although the detection process is able to identify both variable and value symmetries, we just focused on the first kind[1].

Since the number of detected variable symmetries is quite large for most problems, we followed the method of Aloul et al. [2006] for breaking symmetries, that is we restrict to the variable symmetries present on the generators of the symmetry group (also referred as the GEN class in [Puget 2005]), and added a number of lexicographic ordering constraints [Carlsson and Beldiceanu 2002] before starting search. This is a popular technique known as static symmetry breaking (SSB) [Puget 1993]. Moreover, both solvers do some effort to identify symmetries in sets of variables known to be all different, in which case we break symmetries by enforcing a total ordering.

It is known that SSB may potentially make the task of solving a satisfiable problem harder, since it can prune the solutions that would be found first by the search heuristics. In our preliminary tests we also tried breaking symmetries using the dynamic lex method [Puget 2006] which does not have this drawback. Despite performing slightly better, this method requires a fixed value selection ordering, which we found too restrictive for our exploration strategies described in the next section.

[1] We adapted the code from saucy, a graph automorphism generator [Darga et al. 2004].

11.4. Search

11.4.1. Heuristics

It is well known that variable and value selection heuristics play a crucial goal for guiding search towards a solution, or for proving that no solution exists. Recently, the *dom/wdeg* [Boussemart et al. 2004] heuristic has been given a lot of attention, although we have found that impact based heuristics [Refalo 2004] perform better for some problems, as seen in the previous chapter. For the competition we considered a portfolio composed of the *dom/wdeg* and impact variable heuristics for the *zito* solver, and also the *lookahead* variable heuristic (as described in the previous chapter) for the *zao* solver.

While there has been recent work aiming at informed value selection heuristics [Hsu et al. 2007], the most popular is probably the *min-conflicts* which selects the value having less conflicts with the values of other variables. Other common value selection heuristics select the values in increasing ordering (*min*), or just randomly (*rand*). Unfortunately, for the competition we didn't have time to implement anything more sophisticated than the *min* and *rand* value selection ordering.

11.4.2. Sampling

In order to choose which variable-value heuristic combination is finally used for solving a given instance, we introduced a sampling phase in the solving process (alg. 20). We evaluate each strategy based on the criteria of first-failness and best-promise [Haralick and Elliott 1980; Geelen 1992]. Roughly, first-failness is the ability of the heuristic to easily find short refutations for large regions of the search tree that contain no solutions, while best-promise characterizes the potential to guide search quickly towards a solution. Typical search strategies combine these two components, usually by associating first-failness with the variable selection heuristic, and best-promise with the value selection heuristic.

Informally, our sampling phase works by performing several time bounded search runs (restarts) with each possible strategy while collecting information regarding its behavior. The time slice is increased geometrically from one run to the next in order to provide a basis for projecting the behavior of the strategy on a real (time unbounded) search run. After each run we compute an approximation of the ratio of the explored search space by analyzing the visited search tree, and store this information in F. After the sampling process, we compute from F an estimate of the first-failness and best-promise coefficients for each variable and value heuristic and select the best combination. Although the approximation makes a strong assumption that the search tree is uniformly balanced, our preliminary tests revealed that most of the times this method selects the best heuristics, specially when the choice of heuristic is crucial.

Algorithm 20: Search strategy sampling

Input: A set $\mathscr{S}$ of possible exploration strategies, initial and final time slice T_i, T_f, and geometric ratio r

Output: One of **SAT**,**UNSAT** or ⟨**UNKNOWN**, F⟩

```
1  F ← {}
2  foreach s ∈ 𝒮 do
3      t ← T_i
4      while t ≤ T_f do
5          f ← search(s,t) /* Search with strategy s, timeout at t.  */
6          if f=SAT or f=UNSAT then
7              return f
8          e ← ratioOfExploredSearchSpace()
9          F ← F ∪ ⟨s, t, e⟩
10         t ← t × r
11
12 return ⟨UNKNOWN, F⟩
```

11.4.3. SAC

Enforcing singleton arc consistency on a constraint network is a popular pruning technique [Debruyne and Bessière 1997], although its time complexity can be limiting. For the competition, both solvers enforce a time bounded SAC on the first propagation only. However, given that RSAC [Prosser et al. 2000], a restricted form of SAC, is achieved while evaluating the lookahead heuristic (only on *zao*), then it may happen that RSAC is always enforced on some instances if it is selected by the method described in the previous section.

11.4.4. Restarts

For exploring the search tree we employed depth first search with time bounded restarts. Completeness is guaranteed by increasing the time allowed for each restart. We used 2.5 as the geometric ratio.

11.5. Experimental evaluation

Currently CaSPER does not implement any learning techniques, smart backjumping methods, constraint network analysis and (de)composition, or specialized data structures for CSPs given in extension (apart from table constraints). We think that these are required to be competitive in all categories except the GLB category, and perhaps on the set of instances from the INT and NINT categories that are too large to convert to extensional form. Unfortunately, there was a

	zito	*zao*
static symmetry breaking	yes	yes
predicate tabling	no	yes
lookahead var heuristic	no	yes
heuristic sampling	yes	yes
GAC for negative table	no	yes

Table 11.2.: Summary of features in each solver.

bug in the propagator achieving bounds consistency for the constraint $\left[\text{mod}(x,y) = z\right]$ which caused both solvers to be disqualified from the INT category. We will therefore focus on the GLB and NINT categories exclusively.

Table 11.2 summarizes the distribution of the previously discussed features among both solvers.

The results of the CPAI'08 solver competition for the NINT and GLB categories are given in tables B.43 and B.44 in the appendix, and also in the website of the competition [CPAI08 2008]. In the n-ary intension constraints category, *zao* was the third best solver out of 18 contestants, ranked after the two versions of the portfolio solver *cpHydra*. In the global constraints category, *zito* and *zao* were ranked in the 6th an 7th positions respectively out of 17 contestants, loosing to *cpHydra*, *Sugar* (a SAT solver), and *Mistral*.

Finally, we would like to point out that *zito* and *zao* were the two best solvers for the subset of UNSAT problems in both categories. This supports our belief that the absence of a smart value ordering heuristic, which is mostly important for proving satisfiability, was a key factor for not obtaining even better overall results.

11.6. Conclusion

This chapter describes the *casperzito* and *casperzao* constraint solvers as submitted to the third international solver competition. These solvers are small instantiations of the CaSPER library which aims to provide a comprehensive environment for doing applied research in constraint programming.

After a brief description of the propagation model, we focused on the less standard features which were added specifically to the competition, such as automatic symmetry detection and symmetry breaking, or are product of our own research, such as search strategy sampling, and impact based search. Finally, we have made a global analysis of the results from the competition, which shows a good overall performance of both solvers.

Future work

- ▷ At the propagation level, a careful analysis on the best propagator to use for a given constraint when there is more than one choice is missing. A propagator for positive table constraints, for example, may be implemented based on a table [Bessière and Régin 1997], a trie [Gent et al. 2007], or a multivalued decision diagram [Cheng and Yap 2008], with distinct average runtime expectation. An online or offline sampling strategy could perhaps provide some insight on the best propagator for a specific constraint.
- ▷ The symmetry breaking framework could be made more dynamic and complete and extended to value symmetries as well. Search would certainly benefit from ideas such as learning from restarts [Lecoutre et al. 2007], conflict-based static value ordering [Mehta and van Dongen 2005] and better integration of our own work on lookahead heuristics.

Chapter 12.

Conclusions

This dissertation covered a number of techniques for improving the implementation of constraint solving. In general it shows that achieving a good compromise between flexibility and efficiency is feasible within a constraint solver, a large and relatively complex system. In particular, it presented the following contributions.

12.1. Summary of main contributions

We have described a constraint propagation kernel supporting different models of incremental propagation. We have seen that incremental propagation may be more efficient than non-incremental propagation for a number of constraints, in particular constraints over set domain variables. The support for incremental propagation adds an overhead to the propagation kernel which is a function of the model used for incremental propagation. Since this overhead is close to zero when no incremental propagators are used, this architecture actually implements a *pay-per-use* philosophy, allowing the designer of each propagator to perform an uncommitted choice of the best propagation model for each specific case. We believe this design boosts the flexibility of a constraint solver.

We have introduced views over multiple variables, which generalizes the model of Tack [2009]. The new model allows to derive efficient propagation algorithms for many more decomposable constraints, in particular arithmetic constraints. We have seen that the generalized model is sound and that its completeness may be approximated using a rule rewriting inference process. We have introduced an automatic compilation method of arbitrary decomposable constraints into box view models based on parametric polymorphic objects, yielding very efficient propagation algorithms. We remark that the top-down approach followed by the compilation algorithm does not bound subexpressions to specific propagation algorithms, as is usual in most constraint solvers. Instead it allows any propagation algorithm to be associated with an arbitrary decomposable constraint as a whole, which we consider a very important feature of a flexible constraint solver.

We performed a number of experiments with non-standard heuristics and propagation algorithms, which eventually allowed us to contribute with a new general constraint solving method that explores a fruitful combination of search heuristics integrating lookahead information and singleton consistency propagation algorithms. This work greatly benefited from the flexibility of our solver, which allowed us to easily obtain a sound and competitive implementation of the algorithms.

We contributed CaSPER, a highly efficient and flexible constraint solver, which is now a valuable resource for advanced research on constraint programming. The correctness, efficiency and flexibility of the solver are witnessed by the results obtained from the large number of experiments done with the platform, either when integrating all the techniques described above, or as a participant of the third international solver competition.

12.2. Future work

Future work which extends the material presented in this dissertation has been pointed out at the end of each corresponding chapter. Here we mention work which we intend to perform in the near future for improving our solver in general and unrelated ways.

- The recent trend consisting of introducing CPU's with multiple cores for increasing processing power has made exploring parallelism one major concern for any time critical computer program. Parallelizing a sequential program raises a number of important problems, much of them related to the concurrent access to shared resources such as memory, which are not trivially solved in general. However, the particular exploratory nature of constraint solving makes parallelization less hard to implement. A simple technique, which is being adopted by several constraint solvers such as Gecode [Chu et al. 2009], is to create several copies of the constraint program, and synchronize them in a way that they explore different parts of the search tree. Integrating this method in CaSPER should not be too hard and is orthogonal to any of the presented techniques.
- We intend to further develop our preliminary work on real valued reasoning. The propagators for constraints over real valued variables are based on box view models, and are very similar to the corresponding propagators for integer domain variables. Although propagation of real valued variables takes into account other concerns, such as safe rounding-off operations, in our solver we were able to use the same infrastructure for propagating box view models for real and integer valued variables. This gave us a comprehensible set of propagators for constraints over real-valued variables, almost *for free*. We intend to further develop this model by integrating algorithms which are specific to the propagation of constraints over real valued variables, such as the Newton method among others [Benhamou 1995].

Bibliography

ACP (2010). Standardization of Constraint Programming. The Internet.
`http://4c110.ucc.ie/cpstandards/`.
Cited on page 39.

Afred V. Aho, Michael R. Garey, and Jeffrey D. Ullman (1972). The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, volume 1(2):131–137, SIAM.
Cited on page 105.

Hassan Aït-Kaci (1991). *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press.
Cited on page 37.

Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov (2006). Efficient Symmetry Breaking for Boolean Satisfiability. *IEEE Transactions on Computers*, volume 55(5):549–558, IEEE Computer Society.
Cited on page 165.

Krzysztof R. Apt and Peter Zoeteweij (2003). A Comparative Study of Arithmetic Constraints on Integer Intervals. *ERCIM workshop on Constraint Solving and Constraint Logic Programming, CSCLP'03* (proceedings), volume 3010 of *Lecture Notes in Artificial Intelligence*, pp. 1–24. Springer.
Cited on pages 108 and 109.

Francisco Azevedo (2007). Cardinal: A Finite Sets Constraint Solver. *Constraints journal*, volume 12(1):93–129, Springer.
Cited on pages 59, 60, 61, and 75.

Thomas Ball, Andreas Podelski, and Sriram K. Rajamani (2003). Boolean and Cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer (STTT)*, volume 5(1):49–58, Springer.
Cited on page 14.

Roman Barták and Radek Erben (2004). A New Algorithm for Singleton Arc Consistency. *Florida Artificial Intelligence Research Society Conference, FLAIRS'04* (proceedings). AAAI Press.
Cited on pages 149, 150, 152, and 162.

Nicolas Beldiceanu and Mats Carlsson (2002). A New Multi-resource cumulatives Constraint with Negative Heights. *Principles and Practice of Constraint Programming, CP'02* (proceedings), volume 2470 of *Lecture Notes in Computer Science*, pp. 63–79. Springer.
Cited on page 164.

Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon (2010). Global Constraint Catalog. The Internet.
`http://www.emn.fr/x-info/sdemasse/gccat/index.html`.
Cited on pages 37 and 79.

Frédéric Benhamou (1995). Interval constraint logic programming. *Constraint programming: basics and trends* (proceedings), volume 910 of *Lecture Notes in Computer Science*, pp. 1–21. Springer.
Cited on pages 14, 38, and 172.

Frédéric Benhamou (1996). Heterogeneous Constraint Solving. *Algebraic and Logic Programming, ALP'96* (proceedings), volume 1139 of *Lecture Notes in Computer Science*, pp. 62–76. Springer-Verlag.
Cited on pages 14 and 17.

Christian Bessière and Romuald Debruyne (2004). Theoretical analysis of singleton arc consistency. *ECAI'04 workshop on Modelling and Solving Problems with Constraints* (proceedings), pp. 20–29.
Cited on page 150.

Christian Bessière and Romuald Debruyne (2005). Optimal and suboptimal singleton arc consistency algorithms. *International Joint Conference on Artificial Intelligence, IJCAI'05* (proceedings), pp. 54–59. Morgan Kaufmann Publishers, Inc.
Cited on pages 149, 150, 152, and 162.

Christian Bessière and Jean-Charles Régin (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. *Principles and Practice of Constraint Programming, CP'96* (proceedings), volume 1118 of *Lecture Notes in Computer Science*, pp. 61–75. Springer.
Cited on page 21.

Christian Bessière and Jean-Charles Régin (1997). Arc Consistency for General Constraint Networks: Preliminary Results. *Iternational Joint Conference on Artificial Intelligence, IJCAI'97* (proceedings), pp. 398–404. Morgan Kaufmann Publishers, Inc.
Cited on pages 157, 164, 165, and 169.

Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais (2004). Boosting Systematic Search by Weighting Constraints. *European Conference on Artificial Intelligence,*

ECAI'04 (proceedings), pp. 146–150. IOS Press.
Cited on pages 21, 155, and 166.

Daniel Brélaz (1979). New methods to color the vertices of a graph. *Communications of the ACM*, volume 22(4):251–256, ACM.
Cited on page 21.

Andrei A. Bulatov (2006). A dichotomy theorem for constraint satisfaction problems on a 3-element set. *Journal of the ACM*, volume 53(1):66–120, ACM.
Cited on page 19.

Björn Carlson (1995). *Compiling and executing finite domain constraints*. Ph.D. thesis, Uppsala University.
Cited on page 87.

Björn Carlson, Mats Carlsson, and Daniel Diaz (1994). Entailment of Finite Domain Constraints. *International Conference on Logic Programming, ICLP'94* (proceedings), pp. 339–353. MIT Press.
Cited on page 31.

Mats Carlsson and Nicolas Beldiceanu (2002). Revisiting the Lexicographic Ordering Constraint. Research Report T2002-17, Swedish Institute of Computer Science.
Cited on page 165.

Mats Carlsson, Greger Ottosson, and Björn Carlson (1997). An Open-Ended Finite Domain Constraint Solver. *Programming Languages: Implementations, Logics, and Programs, PLILP'97* (proceedings), volume 1292 of *Lecture Notes in Computer Science*, pp. 191–206. Springer.
Cited on page 31.

Peter Cheeseman, Bob Kanefsky, and William M. Taylor (1991). Where the Really Hard Problems Are. *International Joint Conference on Artificial Intelligence, IJCAI'91* (proceedings), pp. 331–340. Morgan Kaufmann Publishers, Inc.
Cited on page 156.

Kenil C. K. Cheng and Roland H. C. Yap (2008). Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints. *Principles and Practice of Constraint Programming, CP'08* (proceedings), volume 5202 of *Lecture Notes in Computer Science*, pp. 509–523. Springer.
Cited on pages 88, 127, and 169.

Choco (2010). Choco constraint programming system. The Internet.
`http://www.emn.fr/x-info/choco-solver/doku.php?id=.`
Cited on pages 75 and 144.

Geoffrey Chu, Christian Schulte, and Peter J. Stuckey (2009). Confidence-based Work Stealing in Parallel Constraint Programming. *Principles and Practice of Constraint Programming, CP'09* (proceedings), volume 5732 of *Lecture Notes in Computer Science*, pp. 226–241. Springer.
Cited on page 172.

Marco Correia, Pedro Barahona, and Francisco Azevedo (2005). CaSPER: A Programming Environment for Development and Integration of Constraint Solvers. *Workshop on Constraint Programming Beyond Finite Integer Domains, BeyondFD'05* (proceedings).
Cited on page 88.

CPAI08 (2008). Third International CSP Solver Competition. The Internet.
`http://cpai.ucc.ie/08/`.
Cited on page 168.

Joseph Culberson (2010). Graph Coloring Resources. The Internet.
`http://web.cs.ualberta.ca/~joe/Coloring/Generators/generate.html`.
Cited on page 156.

Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov (2004). Exploiting structure in symmetry detection for CNF. *Design Automation Conference, DAC'04* (proceedings), Annual ACM IEEE Design Automation Conference, pp. 530–534. ACM.
Cited on page 165.

Romuald Debruyne and Christian Bessière (1997). Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. *International Joint Conference on Artificial Intelligence, IJCAI'97* (proceedings), pp. 412–417. Morgan Kaufmann Publishers, Inc.
Cited on pages 149, 150, and 167.

Rina Dechter (2003). *Constraint Processing*. Morgan Kaufmann.
Cited on page 21.

Yves Deville and Pascal Van Hentenryck (1991). An efficient arc consistency algorithm for a class of CSP problems. *International Joint Conference on Artificial Intelligence, IJCAI'91* (proceedings), pp. 325–330. Morgan Kaufmann Publishers, Inc.
Cited on page 57.

Dotu, del Val, and Cebrian (2003). Redundant Modeling for the QuasiGroup Completion Problem. *Principles and Practice of Constraint Programming, CP'03* (proceedings), volume 2833 of *Lecture Notes in Computer Science*. Springer.
Cited on page 157.

Ivan Dotu, Manuel Cebrián, Pascal Van Hentenryck, and Peter Clote (2008). Protein Structure Prediction with Large Neighborhood Constraint Programming Search. *Principles and Practice of Constraint Programming, CP'08* (proceedings), volume 5202 of *Lecture Notes in Computer Science*, pp. 82–96. Springer.
Cited on page 3.

ECLiPSe (2010). ECLiPSe Prolog. The Internet.
`http://www.eclipse-clp.org/`.
Cited on pages 31, 75, and 144.

Eugene C. Freuder (1978). Synthesizing constraint expressions. *Communications of the ACM*, volume 21:958–966, ACM.
Cited on page 19.

Gecode (2010). Gecode: Generic Constraint Development Environment. The Internet.
`http://www.gecode.org`.
Cited on pages 30, 71, 134, and 145.

Pieter Andreas Geelen (1992). Dual viewpoint heuristics for binary constraint satisfaction problems. *European Conference on Artificial Intelligence, ECAI'92* (proceedings), pp. 31–35. John Wiley & Sons, Inc.
Cited on pages 22, 154, and 166.

I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh (1996). The Constrainedness of Search. *Conference on Artificial Intelligence, AAAI'96* (proceedings), volume 1, pp. 246–252. AAAI Press.
Cited on pages 152, 156, and 162.

Ian P. Gent, Christopher Jefferson, and Ian Miguel (2006a). Minion: A fast scalable constraint solver. *European Conference on Artificial Intelligence, ECAI'06* (proceedings), pp. 98–102. IOS Press.
Cited on pages 34 and 37.

Ian P. Gent, Christopher Jefferson, and Ian Miguel (2006b). Watched Literals for Constraint Propagation in Minion. *Principles and Practice of Constraint Programming, CP'06* (proceedings), volume 4204 of *Lecture Notes in Computer Science*, pp. 182–197. Springer.
Cited on pages 28 and 75.

Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale (2007). Data Structures for Generalised Arc Consistency for Extensional Constraints. *Conference on Artificial Intelligence, AAAI'07* (proceedings), pp. 191–197. AAAI Press.
Cited on pages 88, 127, 164, 165, and 169.

Ian P. Gent, E. MacIntyre, P. Prosser, Barbara M. Smith, and Toby Walsh (2001). Random Constraint Satisfaction: Flaws and Structure. *Constraints*, volume 6(4):345–372, Springer.
Cited on page 156.

Ian P. Gent and Toby Walsh (1999). CSPLib: a benchmark library for constraints. Technical report, APES-09-1999.
`http://www.csplib.org/`.
Cited on pages 71, 126, and 134.

Carmen Gervet (1994). Conjunto: Constraint Propagation over Set Constraints with Finite Set Domain Variables. *International Conference on Logic Programming, ICLP'94* (proceedings), p. 733. MIT Press.
Cited on page 59.

GNUProlog (2008). GNU Prolog. The Internet.
`http://www.gprolog.org/`.
Cited on page 144.

Robert M. Haralick and G. L. Elliott (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, volume 14:263–313, Elsevier.
Cited on pages 21 and 166.

Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan (2010). Constraint Programming and Combinatorial Optimisation in Numberjack. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pp. 181–185. Springer.
Cited on page 39.

Pascal Van Hentenryck and Yves Deville (1993). The cardinality operator: a new logical connective for constraint logic programming. *Constraint logic programming: selected research*, pp. 383–403, MIT Press.
Cited on page 83.

Pascal Van Hentenryck and Laurent Michel (2005). *Constraint-Based Local Search*. The MIT Press.
Cited on page 126.

Pascal Van Hentenryck, Perron, and Jean-François Puget (2000). Search and Strategies in OPL. *ACMTCL: ACM Transactions on Computational Logic*, volume 1, ACM.
Cited on page 126.

Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville (1992). Constraint Processing in cc(FD). Technical report, Brown University.
Cited on page 87.

Eric I. Hsu, Matthew Kitching, Fahiem Bacchus, and Sheila A. McIlraith (2007). Using Expectation Maximization to Find Likely Assignments for Solving CSP's. *Conference on Artificial Intelligence, AAAI'07* (proceedings), pp. 224–230. AAAI Press.
Cited on page 166.

ILOG (2003a). *ILOG Solver 6.0 Reference Manual*. ILOG s.a.
`http://www.ilog.com`.
Cited on pages 34, 88, and 144.

ILOG (2003b). *ILOG Solver 6.0 User's Manual*. ILOG s.a.
`http://www.ilog.com`.
Cited on page 75.

Richard Jones and Rafael D. Lins (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.
Cited on page 37.

Julien Vion (2007). Hybridation de prouveurs CSP et apprentissage. *Journées Francophones de la Programmation avec Contraintes, JFPC'07* (proceedings), pp. 159–168.
Cited on page 38.

Jean Christoph Jung (2008). *Value Orderings Based on Solution Counting*. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.
Cited on page 51.

Kalev Kask, Rina Dechter, and Vibhav Gogate (2004). New Look-Ahead Schemes for Constraint Satisfaction. *Artificial Intelligence and Mathematics, AMAI'04* (proceedings).
`http://rutcor.rutgers.edu/~amai/aimath04/AcceptedPapers/Kask-aimath04.pdf`.
Cited on pages 152 and 162.

Kautz, Ruan, Achlioptas, Gomes, Selman, and Stickel (2001). Balance and Filtering in Structured Satisfiable Problems. *International Joint Conference on Artificial Intelligence, IJCAI'01* (proceedings). Morgan Kaufmann Publishers, Inc.
Cited on page 157.

Ludwig Krippahl and Pedro Barahona (2002). PSICO: Solving Protein Structures with Constraint Programming and Optimization. *Constraints*, volume 7(3-4):317–331, Kluwer Academic Publishers.
Cited on page 38.

Thomas Kühne (1997). The Function Object Pattern. *C++ Report*, volume 9(9).
Cited on page 51.

François Laburthe and the OCRE project team (2008). CHOCO: implementing a CP kernel. *Proceedings of TRICS: Techniques for implementing constraint programming systems, a post-conference workshop of CP2000* (proceedings).
Cited on pages 31 and 57.

Mikael Z. Lagerkvist (2008). *Techniques for Efficient Constraint Propagation*. Licenciate's thesis, Royal Institute of Technology.
Cited on pages 57 and 75.

Mikael Z. Lagerkvist and Christian Schulte (2007). Advisors for Incremental Propagation. *Principles and Practice of Constraint Programming, CP'07* (proceedings), volume 4741 of *Lecture Notes in Computer Science*, pp. 409–422. Springer.
Cited on pages 58 and 75.

Vitaly Lagoon and Peter J. Stuckey (2004). Set Domain Propagation Using ROBDDs. *Principles and Practice of Constraint Programming, CP'04* (proceedings), volume 3258 of *Lecture Notes in Computer Science*, pp. 347–361. Springer.
Cited on page 75.

Charles F. Laywine and Gary L. Mullen (1998). *Discrete Mathematics Using Latin Squares*. Wiley-IEEE.
Cited on page 3.

Michel Leconte (1996). A bounds based reduction scheme for constraints of difference. *International Workshop on Constraint-Based reasoning* (proceedings), pp. 19–28.
Cited on page 19.

Christophe Lecoutre (2010). XCSP benchmarks. The Internet.
`http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`.
Cited on page 55.

Christophe Lecoutre and Stéphane Cardon (2005). A Greedy Approach to Establish Singleton Arc Consistency. *International Joint Conference on Artificial Intelligence, IJCAI'05* (proceedings), pp. 199–204. Morgan Kaufmann Publishers, Inc.
Cited on pages 149 and 150.

Christophe Lecoutre, Lakhdar Sais, Sebastien Tabary, and Vincent Vidal (2007). Recording and Minimizing Nogoods from Restarts. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, volume 1:147–167.
Cited on page 169.

Ro Lopez-Ortiz, Claude-Guy Quimper, John Tromp, and Peter Van Beek (2003). A fast and simple algorithm for bounds consistency of the alldifferent constraint. *International Joint*

Conference on Artificial Intelligence, IJCAI'03 (proceedings), pp. 245–250. Morgan Kaufmann Publishers, Inc.
Cited on pages 137 and 164.

Alan K. Mackworth (1977a). Consistency in Networks of Relations. *Artificial Intelligence*, volume 8(1):99–118, Elsevier.
Cited on pages 13 and 17.

Alan K. Mackworth (1977b). On Reading Sketch Maps. *International Joint Conference on Artificial Intelligence, IJCAI'77* (proceedings), pp. 598–606. Morgan Kaufmann Publishers, Inc.
Cited on page 17.

Michael J. Maher (2002). Propagation Completeness of Reactive Constraints. *International Conference on Logic Programming, ICLP'02* (proceedings), volume 2401 of *Lecture Notes in Computer Science*, pp. 148–162. Springer.
Cited on pages 14 and 17.

Deepak Mehta and Mark R. C. van Dongen (2005). Static Value Ordering Heuristics for Constraint Satisfaction Problems. *Second International Workshop on Constraint Propagation And Implementation* (proceedings), pp. 49–62.
Cited on page 169.

Sylvain Merchez, Christophe Lecoutre, and Frederic Boussemart (2001). AbsCon: A Prototype to Solve CSPs with Abstraction. *Principles and Practice of Constraint Programming, CP'01* (proceedings), volume 2239 of *Lecture Notes in Computer Science*, pp. 730–744. Springer.
Cited on page 38.

Luc Mercier and Pascal Van Hentenryck (2008). Edge Finding for Cumulative Scheduling. *Informs journal on computing*, volume 20.
Cited on page 164.

Ugo Montanari (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, volume 7:95–132.
Cited on pages 13 and 19.

Arnaldo V. Moura, Cid C. Souza, Andre A. Cire, and Tony M. Lopes (2008). Planning and Scheduling the Operation of a Very Large Oil Pipeline Network. *Principles and Practice of Constraint Programming, CP'08* (proceedings), volume 5202 of *Lecture Notes on Computer Science*, pp. 36–51. Springer.
Cited on page 3.

Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack (2007). MiniZinc: Towards a Standard CP Modelling Language. *Principles and Practice of Constraint Programming, CP'07* (proceedings), volume 4741 of *Lecture Notes in Computer Science*, pp. 529–543. Springer.
Cited on page 39.

Patrick Prosser, Kostas Stergiou, and Toby Walsh (2000). Singleton Consistencies. *Principles and Practice of Constraint Programming, CP'00* (proceedings), volume 1894 of *Lecture Notes in Computer Science*, pp. 353–368. Springer.
Cited on pages 149, 150, 151, 158, and 167.

Jean-François Puget (1992). PECOS: A High Level Constraint Programming Language. *Singapore International Conference on Intelligent Systems, SPICIS'92* (proceedings).
Cited on page 59.

Jean-François Puget (1993). On the Satisfiability of Symmetrical Constrained Satisfaction Problems. *ISMIS '93: Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems* (proceedings), pp. 350–361. Springer-Verlag, London, UK.
Cited on page 165.

Jean-François Puget (1998). A fast algorithm for the bound consistency of alldiff constraints. *Conference on Artificial Intelligence, AAAI'98* (proceedings), pp. 359–366. AAAI Press, Menlo Park, CA, USA.
Cited on page 19.

Jean-François Puget (2004). Constraint Programming Next Challenge: Simplicity of Use. *Principles and Practice of Constraint Programming, CP'04* (proceedings), volume 3258 of *Lecture Notes in Computer Science*, pp. 5–8. Springer.
Cited on page 39.

Jean-François Puget (2005). Breaking symmetries in all different problems. *International Joint Conference on Artificial Intelligence, IJCAI'05* (proceedings), pp. 272–277. Morgan Kaufmann Publishers, Inc.
Cited on page 165.

Jean-François Puget (2006). Dynamic Lex Constraints. *Principles and Practice of Constraint Programming, CP'06* (proceedings), volume 4204 of *Lecture Notes in Computer Science*, pp. 453–467. Springer.
Cited on page 165.

Jean-François Puget and Michel Leconte (1995). Beyond the Glass Box: Constraints as Objects. *International Logic Programming Symposium, ILPS'95* (proceedings), pp. 513–527.

MIT Press.
Cited on page 57.

Philippe Refalo (2004). Impact-Based Search Strategies for Constraint Programming. *Principles and Practice of Constraint Programming, CP'04* (proceedings), volume 3258 of *Lecture Notes in Computer Science*, pp. 557–571. Springer.
Cited on pages 155, 161, and 166.

Jean-Charles Régin (1994). A Filtering Algorithm for Constraints of Difference in CSPs. *Conference on Artificial Intelligence, AAAI'94* (proceedings), pp. 362–367. AAAI Press.
Cited on pages 19, 44, and 164.

Raphael M. Reischuk, Christian Schulte, Peter J. Stuckey, and Guido Tack (2009). Maintaining State in Propagation Solvers. *Principles and Practice of Constraint Programming, CP'09* (proceedings), volume 5732 of *Lecture Notes in Computer Science*, pp. 692–706. Springer.
Cited on page 37.

John Reynolds (1974). Towards a theory of type structure. *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer.
Cited on page 116.

Francesca Rossi, Peter Van Beek, and Toby Walsh (eds.) (2006). *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science.
Cited on pages 3, 17, 19, and 88.

Olivier Roussel and Christophe Lecoutre (2009). XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR*, volume abs/0902.2362.
Cited on pages 39 and 163.

Thomas J. Schaefer (1978). The complexity of satisfiability problems. *Symposium on Theory of Computing, STOC'78* (proceedings), pp. 216–226. ACM.
Cited on page 19.

Christian Schulte and Peter J. Stuckey (2004). Speeding Up Constraint Propagation. *Principles and Practice of Constraint Programming, CP'04* (proceedings), volume 3258 of *Lecture Notes on Computer Science*, pp. 619–633. Springer.
Cited on pages 30, 31, and 83.

Christian Schulte and Peter J. Stuckey (2005). When do bounds and domain propagation lead to the same search space? *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 27(3):388–425, ACM.
Cited on page 108.

Christian Schulte and Peter J. Stuckey (2008). Efficient Constraint Propagation Engines. *Transactions on Programming Languages and Systems*, volume 31(1):2:1–2:43, ACM.
Cited on pages 57 and 58.

Christian Schulte and Guido Tack (2005). Views and Iterators for Generic Constraint Implementations. *Recent Advances in Constraints* (proceedings), volume 3978 of *Lecture Notes in Computer Science*, pp. 118–132. Springer.
Cited on pages 88 and 145.

Christian Schulte and Guido Tack (2008). Perfect Derived Propagators. *Principles and Practice of Constraint Programming, CP'08* (proceedings), volume 5202 of *Lecture Notes on Computer Science*. Springer.
Cited on page 108.

Christian Schulte and Guido Tack (2010). Implementing Efficient Propagation Control. *TRICS: Techniques foR Implementing Constraint programming Systems, a conference workshop of CP 2010* (proceedings). St Andrews, UK.
Cited on pages 50 and 57.

SICStus (2006). *SICStus Prolog 3.12 User's Manual*, 3.12 edition.
`http://www.sics.se/sicstus/`.
Cited on page 144.

Sérgio Silva (2010). Interfacing the CaSPER Constraint Solver. Technical report, Faculdade de Ciências e Tecnologia / Universidade Nova de Lisboa.
Cited on page 39.

Helmut Simonis and Barry O'Sullivan (2008). Search Strategies for Rectangle Packing. *Principles and Practice of Constraint Programming, CP '08* (proceedings), volume 5202 of *Lecture Notes on Computer Science*, pp. 52–66. Springer.
Cited on page 3.

Carl Spitznagel (2010). Magic Squares. The Internet.
`http://www.jcu.edu/math/vignettes/magicsquares.htm`.
Cited on page 1.

Mark Swaney (2000). History of Magic Squares. The Internet.
`http://www.ismaili.net/mirrors/Ikhwan_08/magic_squares.html`.
Cited on page 1.

Guido Tack (2009). *Constraint Propagation – Models, Techniques, Implementation*. Doctoral dissertation, Saarland University.
Cited on pages 4, 6, 14, 16, 17, 30, 50, 57, 58, 88, 127, and 171.

Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara (2009). Compiling finite linear CSP into SAT. *Constraints*, volume 14:254–272, Springer.
Cited on page 38.

Ruben Viegas and Francisco Azevedo (2007). GRASPER: A Framework for Graph CSPs. *Workshop on Constraint Modelling and Reformulation, ModRef'07* (proceedings).
Cited on page 72.

Ruben Duarte Viegas (2008). *Constraint Solving over Finite Graphs.* Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.
Cited on pages 38, 72, and 73.

Ruben Duarte Viegas, Marco Correia, Pedro Barahona, and Francisco Azevedo (2008). Using Indexed Finite Set Variables for Set Bounds Propagation. *IBERAMIA 2008* (proceedings), volume 5290 of *Lecture Notes in Artificial Intelligence*, pp. 73–82. Springer.
Cited on page 75.

Toby Walsh (2003). Consistency and Propagation with Multiset constraints: A formal viewpoint. *Principles and Practice of Constraint Programming, CP'03* (proceedings), volume 2833 of *Lecture Notes in Computer Science*, pp. 724–738. Springer.
Cited on page 38.

Zhang Yuanlin and Roland H. C. Yap (2000). Arc Consistency on n -ary Monotonic and Linear Constraints. *Principles and Practice of Constraint Programming, CP'00* (proceedings), volume 1894 of *Lecture Notes in Computer Science*, pp. 470–483. Springer.
Cited on pages 108, 122, and 164.

Neng-Fa Zhou (2009). Encoding Table Constraints in CLP(FD) Based on Pair-Wise AC. *Logic Programming*, volume 5649 of *Lecture Notes in Computer Science*, pp. 402–416. Springer.
Cited on page 38.

Appendix A.

Proofs

A.1. Proofs of chapter 6

Proposition 6.22. *A view-based constraint checker $\check{\chi}_c$ is a sound and complete constraint checker.*

Proof. We want to prove that

$$\begin{aligned} \operatorname{con}(c) \cap S^n \neq \emptyset \Leftrightarrow 1 \in \check{\chi}_c\left(S^n\right) \wedge \\ \operatorname{con}(c) \cap S^n \neq S^n \Leftrightarrow 0 \in \check{\chi}_c\left(S^n\right) \end{aligned}$$

Recall that, by definition, $\check{\chi}_c(S^n) = \varphi_c^+(S^n)$. For the first equivalence we note that:

$$\begin{aligned} \operatorname{con}(c) \cap S^n \neq \emptyset &\Leftrightarrow \exists \mathbf{s}^n \in S^n : \mathbf{s}^n \in \operatorname{con}(c) \\ &\Leftrightarrow \exists \mathbf{t} \in \varphi_c^+\left(S^n\right) : \mathbf{t} = c\left(\mathbf{s}^n\right), \mathbf{s}^n \in \operatorname{con}(c) && \text{(def. 6.14)} \\ &\Leftrightarrow 1 \in \varphi_c^+\left(S^n\right) \end{aligned}$$

The proof for the second equivalence is analogous:

$$\begin{aligned} \operatorname{con}(c) \cap S^n \neq S^n &\Leftrightarrow \exists \mathbf{s}^n \in S^n : \mathbf{s}^n \notin \operatorname{con}(c) \\ &\Leftrightarrow \exists \mathbf{t} \in \varphi_c^+\left(S^n\right) : \mathbf{t} = c\left(\mathbf{s}^n\right), \mathbf{s}^n \notin \operatorname{con}(c) && \text{(def. 6.14)} \\ &\Leftrightarrow 0 \in \varphi_c^+\left(S^n\right) \end{aligned}$$

□

Proposition 6.24. *A view-based complete propagator for a constraint c is a sound and complete propagator for c.*

Proof. By def 6.14 $\varphi_c^-(\{1\}) = \operatorname{con}(c)$, therefore

$$\check{\pi}_c^\star\left(S^n\right) = \varphi_c^-(\{1\}) \cap S^n = \operatorname{con}(c) \cap S^n = \pi_c^\star\left(S^n\right)$$

□

Property 6.26. *φ_f^+ and φ_f^- are monotonic for any arbitrary function f.*

Proof. This can be shown for φ_f^+ as follows (the proof for φ_f^- is analogous): Assuming $S_1^n \subseteq S_2^n$, and rewriting it as $S_2^n = S_1^n \cup S_3^n$ where $S_3^n \in \mathbb{Z}^n$ (possibly the empty set), we have

$$\begin{aligned}
&\varphi_f^+\left(S_1^n\right) \subseteq \varphi_f^+\left(S_2^n\right) \\
&\varphi_f^+\left(S_1^n\right) \subseteq \varphi_f^+\left(S_1^n \cup S_3^n\right) \\
&\varphi_f^+\left(S_1^n\right) \subseteq \varphi_f^+\left(S_1^n\right) \cup \varphi_f^+\left(S_3^n\right) && \text{(prop. 6.25)}
\end{aligned}$$

which is always true. □

Proposition 6.28. *Let $S^i \subseteq \mathbb{Z}^i$, $S^k \subseteq \mathbb{Z}^k$ denote arbitrary tuple sets, $f : \mathbb{Z}^j \to \mathbb{Z}^k$, $g : \mathbb{Z}^i \to \mathbb{Z}^j$ arbitrary functions (possibly composition of functions). Then,*

$$\begin{aligned}
\varphi_{f\circ g}^+\left(S^i\right) &= \varphi_f^+ \circ \varphi_g^+\left(S^i\right) \\
\varphi_{f\circ g}^-\left(S^k\right) &= \varphi_g^- \circ \varphi_f^-\left(S^k\right)
\end{aligned}$$

Proof.

$$\begin{aligned}
\varphi_{f\circ g}^+\left(S^i\right) &= \left\{f \circ g\left(\mathbf{s}^i\right) : \mathbf{s}^i \in S^i\right\} = \\
&= \left\{f\left(\mathbf{t}^j\right) : \mathbf{t}^j = g\left(\mathbf{s}^i\right), \mathbf{s}^i \in S^i\right\} = \\
&= \left\{f\left(\mathbf{t}^j\right) : \mathbf{t}^j \in \varphi_g^+\left(S^i\right)\right\} = \\
&= \varphi_f^+ \circ \varphi_g^+\left(S^i\right)
\end{aligned}$$

$$\begin{aligned}
\varphi_{f\circ g}^-\left(S^k\right) &= \left\{\mathbf{s}^i : f \circ g\left(\mathbf{s}^i\right) \in S^k\right\} = \\
&= \left\{\mathbf{s}^i : g\left(\mathbf{s}^i\right) = \mathbf{t}^j, f\left(\mathbf{t}^j\right) \in S^k\right\} = \\
&= \left\{\mathbf{s}^i : g\left(\mathbf{s}^i\right) \in \varphi_f^-\left(S^k\right)\right\} = \\
&= \varphi_g^- \circ \varphi_f^-\left(S^k\right)
\end{aligned}$$

□

Corollary 6.29. *Let $S^i \subseteq \mathbb{Z}^i$, $S^k \subseteq \mathbb{Z}^k$ denote arbitrary tuple sets, $f : \mathbb{Z}^j \to \mathbb{Z}^k$, $g : \mathbb{Z}^i \to \mathbb{Z}^j$ arbi-*

trary functions (possibly Cartesian product of functions). Then,

$$\widehat{\varphi}^{-}_{f\circ g}\left(S^k, S^i\right) = \widehat{\varphi}^{-}_{g}\left(\widehat{\varphi}^{-}_{f}\left(S^k, \varphi^{+}_{g}\left(S^i\right)\right), S^i\right)$$

Proof.

$$\begin{aligned}
\widehat{\varphi}^{-}_{f\circ g}\left(S^k, S^i\right) =& \varphi^{-}_{f\circ g}\left(S^k\right) \cap S^i = && \text{(def. 6.17)}\\
=& \varphi^{-}_{g} \circ \varphi^{-}_{f}\left(S^k\right) \cap S^i = && \text{(lem. 6.28)}
\end{aligned}$$

Since $S^i = \varphi^{-}_{g} \circ \varphi^{+}_{g}\left(S^i\right) \cap S^i$ by property 6.27,

$$\begin{aligned}
\varphi^{-}_{g} \circ \varphi^{-}_{f}\left(S^k\right) \cap S^i =& \varphi^{-}_{g} \circ \varphi^{-}_{f}\left(S^k\right) \cap \varphi^{-}_{g} \circ \varphi^{+}_{g}\left(S^i\right) \cap S^i = && \text{(prop. 6.27)}\\
=& \varphi^{-}_{g}\left(\varphi^{-}_{f}\left(S^k\right) \cap \varphi^{+}_{g}\left(S^i\right)\right) \cap S^i = && \text{(prop. 6.25)}\\
=& \widehat{\varphi}^{-}_{g}\left(\widehat{\varphi}^{-}_{f}\left(S^k, \varphi^{+}_{g}\left(S^i\right)\right), S^i\right) && \text{(def. 6.17)}
\end{aligned}$$

□

Proposition 6.32. *A view-based constraint checker for the constraint $c \circ f$ may be obtained by*

$$\check{\chi}_{c\circ f}\left(S^n\right) = \varphi^{+}_{c} \circ \varphi^{+}_{f}\left(S^n\right)$$

Proof. For any arbitrary $S^n \subseteq \mathbb{Z}^n$:

$$\begin{aligned}
\check{\chi}_{c\circ f}\left(S^n\right) &= \varphi^{+}_{c\circ f}\left(S^n\right) && \text{(def. 6.21)}\\
&= \varphi^{+}_{c} \circ \varphi^{+}_{f}\left(S^n\right) && \text{(def. 6.28)}
\end{aligned}$$

□

Proposition 6.33. *A complete view-based propagator for the constraint $c \circ f$ may be obtained by*

$$\check{\pi}_{c\circ f}\left(S^n\right) = \widehat{\varphi}_{f}\left(\pi^{\star}_{c}\left(\varphi^{+}_{f}\left(S^n\right)\right), S^n\right)$$

In this case $\check{\pi}_{c\circ f}(S^n)$ is also idempotent, i.e. $\check{\pi}_{c\circ f}(S^n) = \check{\pi}^{\star}_{c\circ f}(S^n)$.

Proof. We only need to prove that $\check{\pi}_{c\circ f}(S^n) = \pi^\star_{c\circ f}(S^n)$ for any arbitrary $S^n \subseteq \mathbb{Z}^n$:

$$\begin{aligned}
\pi^\star_{c\circ f}\left(S^n\right) &= \widehat{\varphi}_{c\circ f}\left(\{1\}, S^n\right) && \text{(def. 6.23)}\\
&= \widehat{\varphi}^-_f\left(\widehat{\varphi}^-_c\left(\{1\}, \varphi^+_f\left(S^n\right)\right), S^n\right) && \text{(def. 6.29)}\\
&= \widehat{\varphi}^-_f\left(\pi^\star_c\left(\varphi^+_f\left(S^n\right)\right), S^n\right) && \text{(def. 6.23)}\\
&= \check{\pi}_{c\circ f}\left(S^n\right)
\end{aligned}$$

□

Lemma A.1. *Let π_c and π_d be two arbitrary propagators for constraints c, d, and S an arbitrary tuple set. Then, $\pi^\star_d\left(\pi^\star_c(S)\right) = \pi^\star_c(S)$ if and only if $\pi^\star_c(S) \subseteq \pi^\star_d(S)$.*

Proof. By definition $\pi^\star_c(S) = \mathrm{con}(c) \cap S$, and therefore

$$\begin{aligned}
\pi^\star_d\left(\pi^\star_c(S)\right) &= \mathrm{con}(d) \cap \pi^\star_c(S)\\
&= \mathrm{con}(d) \cap \mathrm{con}(c) \cap S\\
&= \pi^\star_d(S) \cap \pi^\star_c(S)
\end{aligned}$$

The lemma then follows from the fact that $A \cap B = A \Leftrightarrow A \subseteq B$ for any sets A, B. □

Proposition 6.35. *A complete constraint propagator (not necessarily idempotent) for the constraint $c \circ f$ may be obtained by*

$$\check{\pi}_{c\circ f}\left(S^n\right) = \widehat{\varphi}_f\left(\pi_c\left(\varphi^+_f\left(S^n\right)\right), S^n\right)$$

Proof. Let us denote the above propagator by p, and the idempotent propagator $\check{\pi}_{c\circ f}$ given by proposition 6.33 by q (note that therefore $q = q^\star$) . Now we have to show that propagator q and propagator p have the same fixpoint for every S^n, i.e.

$$q^\star\left(S^n\right) = p^\star\left(S^n\right)$$

Case 1. We will prove $p^\star(S^n) \subseteq q^\star(S^n)$ by proving the equivalent relation $q^\star\left(p^\star(S^n)\right) = p^\star(S^n)$ given by lemma A.1. We know that $q^\star\left(p^\star(S^n)\right) \subseteq p^\star(S^n)$ since propagators are contracting. To prove the converse let S^n be an arbitrary tupe set $S^n \subseteq \mathbb{Z}^n$ and $F^n \subseteq S^n$ the fixpoint of $p(S^n)$, i.e.

$$F^n = p^\star\left(S^n\right) = p^\star\left(F^n\right) = p\left(F^n\right)$$

Assume that $p^\star(S^n) \not\subseteq q^\star\left(p^\star(S^n)\right)$. Then,

$$\begin{aligned}
F^n \not\subseteq q^\star\left(F^n\right) &\Leftrightarrow \exists t^n \in F^n : t^n \notin q^\star\left(F^n\right) \\
&\Leftrightarrow \exists t^n \in F^n : t^n \notin \widehat{\varphi}_f\left(\pi_c^\star\left(\varphi_f^+\left(F^n\right)\right), F^n\right) && \text{(cor. 6.33)} \\
&\Leftrightarrow \exists t^n \in F^n : t^n \notin \varphi_f^- \circ \pi_c^\star \circ \varphi_f^+\left(F^n\right) \\
&\Leftrightarrow \exists t^n \in F^n : t^k \notin \varphi_f^- \circ \pi_c^i \circ \ldots \circ \pi_c^2 \circ \pi_c^1 \circ \varphi_f^+\left(F^n\right) \\
&\Leftrightarrow \exists t^n \in F^n : t^k \notin \varphi_f^- \circ \pi_c^i \circ \ldots \circ \pi_c^2 \circ \varphi_f^+ \circ \varphi_f^- \circ \pi_c^1 \circ \varphi_f^+\left(F^n\right) \\
&\Leftrightarrow \exists t^n \in F^n : t^k \notin \varphi_f^- \circ \pi_c^i \circ \ldots \circ \pi_c^2 \circ \varphi_f^+ \circ \widehat{\varphi}_f\left(\pi_c^1 \circ \varphi_f^+\left(F^n\right), F^n\right) && \text{(pty. 6.27)} \\
&\Leftrightarrow \exists t^n \in F^n : t^k \notin \varphi_f^- \circ \pi_c^i \circ \ldots \circ \pi_c^2 \circ \varphi_f^+ \circ p\left(F^n\right) \\
&\Leftrightarrow \exists t^n \in F^n : t^k \notin \varphi_f^- \circ \pi_c^i \circ \ldots \circ \pi_c^2 \circ \varphi_f^+\left(F^n\right) \\
&\Leftrightarrow \exists t^n \in F^n : t^k \notin F^n \\
&\Leftrightarrow \bot
\end{aligned}$$

Case 2. This case proves $q^\star(S^n) \subseteq p^\star(S^n)$. The proof is analogous to case 1.
The proposition follows from the union of the above cases. □

A.2. Proofs of chapter 7

Proposition 7.5. *Let $\Phi_1, \ldots, \Phi_8 \in \{\varphi, \delta, \beta\}$ be a set of approximation operators. Any view model $\langle \Phi_1, \ldots, \Phi_8 \rangle^{c \circ f}$ is a sound propagator for $c \circ f$.*

Proof. Let us write the view model corresponding to a complete view-based propagator for $c \circ f$:

$$\widehat{\pi}_{c\circ f} = \langle \varphi, \ldots, \varphi \rangle^{c\circ f} = \underset{\varphi}{\rightarrow} \varphi_f^+ \underset{\varphi}{\rightarrow} \pi_c^{\varphi\varphi} \underset{\varphi}{\rightarrow} \widehat{\varphi}_f \underset{\varphi}{\rightarrow}$$

Since all operations involved in a view model are monotonic, replacing any φ operator in the above expression by a weaker operator results in a weaker expression, that is

$$\langle \varphi, \ldots, \varphi \rangle^{c\circ f} \quad \subseteq \quad \langle \Phi_1, \ldots, \Phi_8 \rangle^{c\circ f}$$

and since $\widehat{\pi}_{c\circ f}$ is sound (by proposition 6.35) so must be $\langle \Phi_1, \ldots, \Phi_8 \rangle^{c\circ f}$. □

Proposition 7.7. *A $\Phi\Psi$ view model for a constraint $c \circ f$ is a $\Phi\Psi$-complete propagator for $c \circ f$. Moreover, it is also an idempotent propagator.*

Proof. Let m be a $\Phi\Psi$ view model for the constraint $c \circ f$. We only need to show that $m(S^n) =$

$\pi^{\Phi\Psi\star}_{c\circ f}(S^n)$ for any arbitrary $S^n \subseteq \mathbb{Z}^n$:

$$\begin{aligned}
\pi^{\Phi\Psi\star}_{c\circ f}(S^n) &= \llbracket \mathrm{con}(c\circ f) \cap \llbracket S^n \rrbracket^{\Phi} \rrbracket^{\Psi} \cap S^n && \text{(def. 7.1)}\\
&= \llbracket \widehat{\varphi}_{c\circ f}(\{1\}, \llbracket S^n \rrbracket^{\Phi}) \rrbracket^{\Psi} \cap S^n && \text{(def. 6.23)}\\
&= \llbracket \widehat{\varphi}_f\left(\widehat{\varphi}_c\left(\{1\}, \varphi^+_f(\llbracket S^n \rrbracket^{\Phi})\right), \llbracket S^n \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S^n && \text{(def. 6.29)}\\
&= \llbracket \widehat{\varphi}_f\left(\pi^{\star}_c \circ \varphi^+_f(\llbracket S^n \rrbracket^{\Phi}), \llbracket S^n \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S^n && \text{(def. 6.23)}\\
&= m(S^n)
\end{aligned}$$

□

Proposition 7.10. *Let S be an arbitrary tuple set and m a $\Phi\Psi$ relaxed view model for $c\circ f$. Then m is $\Phi\Psi$-complete if and only if any prunning achieved by the propagator for c is preserved by all involved view functions and approximations, formally*

$$\pi_c\left(\varphi^+_f(\llbracket S \rrbracket^{\Phi})\right) \quad \supseteq \quad \varphi^+_f(\llbracket m^{\star}(S) \rrbracket^{\Phi})$$

Proof. By proposition 7.7, the above propagator m is $\Phi\Psi$-complete iff

$$m^{\star} \quad \subseteq \quad \llbracket \widehat{\varphi}_f\left(\pi^{\star}_c \circ \varphi^+_f(\llbracket S \rrbracket^{\Phi}), \llbracket S \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S$$

Therefore, we have to prove

$$\begin{aligned}
m^{\star}(S) \quad &\subseteq \quad \llbracket \widehat{\varphi}_f\left(\pi^{\star}_c \circ \varphi^+_f(\llbracket S \rrbracket^{\Phi}), \llbracket S \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S && \text{(A.1)}\\
&\Leftrightarrow\\
\pi_c\left(\varphi^+_f(\llbracket S \rrbracket^{\Phi})\right) \quad &\supseteq \quad \varphi^+_f(\llbracket m^{\star}(S) \rrbracket^{\Phi}) && \text{(A.2)}
\end{aligned}$$

Case 1: Proof that eq. A.1 implies eq. A.2:

$$\begin{aligned}
& m^{\star}(S) \subseteq \llbracket \widehat{\varphi}_f\left(\pi^{\star}_c \circ \varphi^+_f(\llbracket S \rrbracket^{\Phi}), \llbracket S \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S\\
\Rightarrow & \llbracket m^{\star}(S) \rrbracket^{\Phi} \subseteq \llbracket \llbracket \widehat{\varphi}_f\left(\pi^{\star}_c \circ \varphi^+_f(\llbracket S \rrbracket^{\Phi}), \llbracket S \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S \rrbracket^{\Phi}\\
\Rightarrow & \varphi^+_f(\llbracket m^{\star}(S) \rrbracket^{\Phi}) \subseteq \varphi^+_f\left(\llbracket \llbracket \widehat{\varphi}_f\left(\pi^{\star}_c \circ \varphi^+_f(\llbracket S \rrbracket^{\Phi}), \llbracket S \rrbracket^{\Phi}\right) \rrbracket^{\Psi} \cap S \rrbracket^{\Phi}\right)\\
\Rightarrow & \varphi^+_f(\llbracket m^{\star}(S) \rrbracket^{\Phi}) \subseteq \varphi^+_f(\llbracket S \rrbracket^{\Phi})
\end{aligned}$$

Case 2: Proof that eq. A.2 implies eq. A.1:

$$
\begin{aligned}
&\varphi_f^+\left(\llbracket m^\star(S)\rrbracket^\Phi\right) \subseteq \pi_c\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right)\\
\Rightarrow &\pi_c\left(\varphi_f^+\left(\llbracket m^\star(S)\rrbracket^\Phi\right)\right) \subseteq \pi_c\left(\pi_c\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right)\right)\\
\Rightarrow &\pi_c\left(\varphi_f^+\left(\llbracket m^\star(S)\rrbracket^\Phi\right)\right) \subseteq \pi_c^\star\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right)\\
\Rightarrow &\widehat{\varphi}_f\left(\pi_c\left(\varphi_f^+\left(\llbracket m^\star(S)\rrbracket^\Phi\right)\right),S\right) \subseteq \widehat{\varphi}_f\left(\pi_c^\star\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right),S\right)\\
\Rightarrow &\llbracket\widehat{\varphi}_f\left(\pi_c\left(\varphi_f^+\left(\llbracket m^\star(S)\rrbracket^\Phi\right)\right),S\right)\rrbracket^\Psi \cap S \subseteq \llbracket\widehat{\varphi}_f\left(\pi_c^\star\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right),S\right)\rrbracket^\Psi \cap S\\
\Rightarrow &m\left(m^\star(S)\right) \subseteq \llbracket\widehat{\varphi}_f\left(\pi_c^\star\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right),S\right)\rrbracket^\Psi \cap S\\
\Rightarrow &m^\star(S) \subseteq \llbracket\widehat{\varphi}_f\left(\pi_c^\star\left(\varphi_f^+\left(\llbracket S\rrbracket^\Phi\right)\right),S\right)\rrbracket^\Psi \cap S
\end{aligned}
$$

□

Proposition 7.20. *The set of view models for a constraint ordered by the trivially stronger relation is a bounded lattice where* $\top = \langle\beta,\beta,\ldots,\beta\rangle$ *and* $\bot = \langle\varphi,\varphi,\ldots,\varphi\rangle$.

Proof. For any two approximation operators $\Phi,\Psi \in \{\varphi,\delta,\beta\}$, let

$$
\begin{aligned}
\min(\Phi,\Psi) &= \begin{cases}\Phi & \Leftarrow \Phi \preceq \Psi\\ \Psi & \Leftarrow \Psi \preceq \Phi\end{cases}\\
\max(\Phi,\Psi) &= \begin{cases}\Phi & \Leftarrow \Psi \preceq \Phi\\ \Psi & \Leftarrow \Phi \preceq \Psi\end{cases}
\end{aligned}
$$

Let $v_1 = \langle\Phi_1,\ldots,\Phi_n\rangle$, $v_2 = \langle\Psi_1,\ldots,\Psi_n\rangle$ be two arbitrary view models, and

$$
\begin{aligned}
v_1 \wedge v_2 &= \langle\min(\Phi_1,\Psi_1),\ldots,\min(\Phi_n,\Psi_n)\rangle\\
v_1 \vee v_2 &= \langle\max(\Phi_1,\Psi_1),\ldots,\max(\Phi_n,\Psi_n)\rangle
\end{aligned}
$$

The trivially stronger relation (def. 7.19) partially orders the set of view models for a constraint. The poset is a lattice because for any two view models v_1, v_2, we can always find two unique models $l = v_1 \wedge v_2$ and $u = v_1 \vee v_2$. Since the number of approximation operators is fixed the lattice has exactly 3^n nodes, and is bounded above and below by $\top = \langle\beta,\beta,\ldots,\beta\rangle$ and $\bot = \langle\varphi,\varphi,\ldots,\varphi\rangle$. □

Lemma 7.31. *Let f, g be arbitrary functions and c a constraint. Then,*

$$\begin{gathered}\underset{\Phi_1}{\rightarrow}\Phi^{+}_{2g}\underset{\Phi_3}{\rightarrow}\Phi^{+}_{4f}\underset{\varphi}{\rightarrow}\pi^{\varphi\varphi\star}_{c}\underset{\varphi}{\rightarrow}\Phi^{-}_{5f}\underset{\Phi_6}{\rightarrow}\Phi^{-}_{7g}\underset{\Phi_8}{\rightarrow}\\ =\\ \underset{\Phi_1}{\rightarrow}\Phi^{+}_{2g}\underset{\Phi_3}{\rightarrow}\pi^{\Phi_4\Phi_5\star}_{c\circ f}\underset{\Phi_6}{\rightarrow}\Phi^{-}_{7g}\underset{\Phi_8}{\rightarrow}\end{gathered}$$

Proof. We note that (see the proof of proposition 7.7),

$$\pi^{\Phi\Psi\star}_{c\circ f} \quad = \quad \Phi^{+}_{f}\underset{\varphi}{\rightarrow}\pi^{\varphi\varphi\star}_{c}\underset{\varphi}{\rightarrow}\Psi^{-}_{f}$$

and therefore

$$\begin{gathered}\pi^{\Phi_4\Phi_5\star}_{c\circ f} \;=\; \Phi^{+}_{4f}\underset{\varphi}{\rightarrow}\pi^{\varphi\varphi\star}_{c}\underset{\varphi}{\rightarrow}\Phi^{-}_{5f}\\ \underset{\Phi_1}{\rightarrow}\Phi^{+}_{2g}\underset{\Phi_3}{\rightarrow}\pi^{\Phi_4\Phi_5\star}_{c\circ f}\underset{\Phi_6}{\rightarrow}\Phi^{-}_{7g}\underset{\Phi_8}{\rightarrow} = \underset{\Phi_1}{\rightarrow}\Phi^{+}_{2g}\underset{\Phi_3}{\rightarrow}\Phi^{+}_{4f}\underset{\varphi}{\rightarrow}\pi^{\varphi\varphi\star}_{c}\underset{\varphi}{\rightarrow}\Phi^{-}_{5f}\underset{\Phi_6}{\rightarrow}\Phi^{-}_{7g}\underset{\Phi_8}{\rightarrow} \qquad \text{(cor. 7.26)}\end{gathered}$$

□

Lemma A.2. *Let $S^n \subseteq \mathbb{Z}^n$, $f : \mathbb{Z}^n \to \mathbb{Z}^i$, $g : \mathbb{Z}^n \to \mathbb{Z}^j$ two aribtrary functions, and $\Phi \in \{\varphi, \delta, \beta\}$. Then,*

$$\varphi^{+}_{f\times g}\left(\llbracket S^n \rrbracket^{\Phi}\right) \quad \subseteq \quad \varphi^{+}_{f}\left(\llbracket S^n \rrbracket^{\Phi}\right) \times \varphi^{+}_{g}\left(\llbracket S^n \rrbracket^{\Phi}\right)$$

with equality if f and g are functionally independent and $\Phi \in \{\delta, \beta\}$.

Proof.

$$\begin{gathered}\left\{\langle \mathbf{x}, \mathbf{x}\rangle : \mathbf{x} \in \llbracket S^n \rrbracket^{\Phi}\right\} \subseteq \left\{\mathbf{x} : \mathbf{x} \in \llbracket S^n \rrbracket^{\Phi}\right\} \times \left\{\mathbf{x} : \mathbf{x} \in \llbracket S^n \rrbracket^{\Phi}\right\}\\ \left\{\left\langle f(\mathbf{x}), g(\mathbf{x})\right\rangle : \mathbf{x} \in \llbracket S^n \rrbracket^{\Phi}\right\} \subseteq \left\{f(\mathbf{x}) : \mathbf{x} \in \llbracket S^n \rrbracket^{\Phi}\right\} \times \left\{g(\mathbf{x}) : \mathbf{x} \in \llbracket S^n \rrbracket^{\Phi}\right\}\\ \varphi^{+}_{f\times g}\left(\llbracket S^n \rrbracket^{\Phi}\right) \subseteq \varphi^{+}_{f}\left(\llbracket S^n \rrbracket^{\Phi}\right) \times \varphi^{+}_{g}\left(\llbracket S^n \rrbracket^{\Phi}\right)\end{gathered}$$

If f and g are functionally independent and $\Phi \in \{\delta, \beta\}$, then we can prove the converse case by proving the equivalent relation,

$$\mathbf{y} \in \varphi^{+}_{f}\left(\llbracket S^n \rrbracket^{\Phi}\right) \times \varphi^{+}_{g}\left(\llbracket S^n \rrbracket^{\Phi}\right) \Rightarrow \mathbf{y} \in \varphi^{+}_{f\times g}\left(\llbracket S^n \rrbracket^{\Phi}\right)$$

Deriving the expression on the left hand side,

$$
\begin{aligned}
&\mathbf{y} \in \varphi_f^+\left(\llbracket S^n \rrbracket^\Phi\right) \times \varphi_g^+\left(\llbracket S^n \rrbracket^\Phi\right) \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \mathbf{x}_1 \in \llbracket S^n \rrbracket^\Phi, \mathbf{x}_2 \in \llbracket S^n \rrbracket^\Phi \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \mathbf{x}_1 \in \mathrm{proj}_f\left(\llbracket S^n \rrbracket^\Phi\right), \mathbf{x}_2 \in \mathrm{proj}_g\left(\llbracket S^n \rrbracket^\Phi\right) \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \mathbf{x}_1 \in \llbracket \mathrm{proj}_f\left(S^n\right) \rrbracket^\Phi, \mathbf{x}_2 \in \llbracket \mathrm{proj}_g\left(S^n\right) \rrbracket^\Phi \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \langle \mathbf{x}_1, \mathbf{x}_2 \rangle \in \llbracket \mathrm{proj}_f\left(S^n\right) \rrbracket^\Phi \times \llbracket \mathrm{proj}_g\left(S^n\right) \rrbracket^\Phi \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \mathbf{x} \in \llbracket \mathrm{proj}_f\left(S^n\right) \rrbracket^\Phi \times \llbracket \mathrm{proj}_g\left(S^n\right) \rrbracket^\Phi \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \mathbf{x} \in \llbracket \mathrm{proj}_f\left(S^n\right) \times \mathrm{proj}_g\left(S^n\right) \rrbracket^\Phi \\
&\Rightarrow \mathbf{y} = \left(f(\mathbf{x}_1) \parallel g(\mathbf{x}_2)\right) : \mathbf{x} \in \llbracket S^n \rrbracket^\Phi \qquad \text{(since } f, g \text{ are f.i. and } \Phi \in \{\delta, \beta\}\text{)} \\
&\Rightarrow \mathbf{y} \in \varphi_{f \times g}^+\left(\llbracket S^n \rrbracket^\Phi\right)
\end{aligned}
$$

□

Lemma A.3. *Let* $\Phi \in \{\varphi, \delta, \beta\}$, $f : \mathbb{Z}^n \to \mathbb{Z}^i$, $g : \mathbb{Z}^n \to \mathbb{Z}^j$ *be two aribtrary functions,* $S^k \subseteq \mathbb{Z}^k$, $S^i = proj_{1\ldots i}\left(S^k\right)$, *and* $S^j = proj_{i+1\ldots i+j}\left(S^k\right)$. *Then,*

$$\varphi_{f \times g}^-\left(\llbracket S^k \rrbracket^\Phi\right) \quad \subseteq \quad \varphi_f^-\left(\llbracket S^i \rrbracket^\Phi\right) \cap \varphi_g^-\left(\llbracket S^j \rrbracket^\Phi\right)$$

with equality if $\Phi \in \{\delta, \beta\}$.

Proof.

$$
\begin{aligned}
\left\{\mathbf{x} : \left(f(\mathbf{x}) \parallel g(\mathbf{x})\right) \in \llbracket S^k \rrbracket^\Phi\right\} &\subseteq \left\{\mathbf{x} : f(\mathbf{x}) \in \mathrm{proj}_{1\ldots i}\left(\llbracket S^k \rrbracket^\Phi\right)\right\} \cap \left\{\mathbf{x} : g(\mathbf{x}) \in \mathrm{proj}_{i+1\ldots i+j}\left(\llbracket S^k \rrbracket^\Phi\right)\right\} \\
\left\{\mathbf{x} : \left(f(\mathbf{x}) \parallel g(\mathbf{x})\right) \in \llbracket S^k \rrbracket^\Phi\right\} &\subseteq \left\{\mathbf{x} : f(\mathbf{x}) \in \llbracket \mathrm{proj}_{1\ldots i}\left(S^k\right) \rrbracket^\Phi\right\} \cap \left\{\mathbf{x} : g(\mathbf{x}) \in \llbracket \mathrm{proj}_{i+1\ldots i+j}\left(S^k\right) \rrbracket^\Phi\right\} \\
\varphi_{f \times g}^-\left(\llbracket S^k \rrbracket^\Phi\right) &\subseteq \varphi_f^-\left(\llbracket S^i \rrbracket^\Phi\right) \cap \varphi_g^-\left(\llbracket S^j \rrbracket^\Phi\right)
\end{aligned}
$$

If $\Phi \in \{\delta, \beta\}$, then we can prove the converse case by proving the equivalent relation,

$$\mathbf{x} \in \varphi_f^-\left(\llbracket S^i \rrbracket^\Phi\right) \cap \varphi_g^-\left(\llbracket S^j \rrbracket^\Phi\right) \Rightarrow \mathbf{x} \in \varphi_{f \times g}^-\left(\llbracket S^k \rrbracket^\Phi\right)$$

Deriving the expression on the left hand side,

$$
\begin{aligned}
&\mathbf{x} \in \varphi_f^- \left([\![S^i]\!]^\Phi \right) \cap \varphi_g^- \left([\![S^j]\!]^\Phi \right) \\
&\Rightarrow \exists \mathbf{y}_1 \in [\![S^i]\!]^\Phi, \exists \mathbf{y}_2 \in [\![S^j]\!]^\Phi : f(\mathbf{x}) = \mathbf{y}_1, g(\mathbf{x}) = \mathbf{y}_2 \\
&\Rightarrow \exists \left(\mathbf{y}_1 \parallel \mathbf{y}_2 \right) \in [\![S^i]\!]^\Phi \times [\![S^j]\!]^\Phi : f(\mathbf{x}) = \mathbf{y}_1, g(\mathbf{x}) = \mathbf{y}_2 \\
&\Rightarrow \exists \left(\mathbf{y}_1 \parallel \mathbf{y}_2 \right) \in [\![S^i \times S^j]\!]^\Phi : f(\mathbf{x}) = \mathbf{y}_1, g(\mathbf{x}) = \mathbf{y}_2 \\
&\Rightarrow \exists \left(\mathbf{y}_1 \parallel \mathbf{y}_2 \right) \in [\![\mathrm{proj}_{1\ldots i} \left(S^k \right) \times \mathrm{proj}_{i+1\ldots i+j} \left(S^k \right)]\!]^\Phi : f(\mathbf{x}) = \mathbf{y}_1, g(\mathbf{x}) = \mathbf{y}_2 \\
&\Rightarrow \exists \left(\mathbf{y}_1 \parallel \mathbf{y}_2 \right) \in [\![S^k]\!]^\Phi : f(\mathbf{x}) = \mathbf{y}_1, g(\mathbf{x}) = \mathbf{y}_2 \qquad (\text{since } \Phi \in \{\delta, \beta\}) \\
&\Rightarrow \exists \left(\mathbf{y}_1 \parallel \mathbf{y}_2 \right) \in [\![S^k]\!]^\Phi : \left(f \times g \right)(\mathbf{x}) = \left(\mathbf{y}_1 \parallel \mathbf{y}_2 \right) \\
&\Rightarrow \exists \mathbf{y} \in [\![S^k]\!]^\Phi : \left(f \times g \right)(\mathbf{x}) = \mathbf{y} \\
&\Rightarrow \mathbf{x} \in \varphi_{f \times g}^- \left([\![S^k]\!]^\Phi \right)
\end{aligned}
$$

□

Proposition 7.32. *Let $S^n \subseteq \mathbb{Z}^n$, $S^i = proj_{1\ldots i}\left(S^k\right)$, and $S^j = proj_{i+1\ldots i+j}\left(S^k\right)$. Let $f : \mathbb{Z}^n \to \mathbb{Z}^i$, $g : \mathbb{Z}^n \to \mathbb{Z}^j$ be two arbitrary functions, and $\Phi, \Psi \in \{\varphi, \delta, \beta\}$. Then,*

$$
\begin{aligned}
[\![\varphi_{f\times g}^+ ([\![S^n]\!]^\Phi)]\!]^\Psi &\subseteq [\![\varphi_f^+ ([\![S^n]\!]^\Phi)]\!]^\Psi \times [\![\varphi_g^+ ([\![S^n]\!]^\Phi)]\!]^\Psi \\
[\![\varphi_{f\times g}^- \left([\![S^k]\!]^\Phi \right)]\!]^\Psi &\subseteq [\![\varphi_f^- \left([\![S^i]\!]^\Phi \right)]\!]^\Psi \cap [\![\varphi_g^- \left([\![S^j]\!]^\Phi \right)]\!]^\Psi \\
[\![\widehat{\varphi}_{f\times g} \left([\![S^k]\!]^\Phi, S^n \right)]\!]^\Psi &\subseteq [\![\widehat{\varphi}_f \left([\![S^i]\!]^\Phi, S^n \right)]\!]^\Psi \cap [\![\widehat{\varphi}_g \left([\![S^j]\!]^\Phi, S^n \right)]\!]^\Psi
\end{aligned}
$$

with equality if f and g are functionally independent and $\Phi \in \{\delta, \beta\}$.

Proof. For the first equation we have,

$$
\begin{aligned}
&\varphi_{f\times g}^+ ([\![S^n]\!]^\Phi) \subseteq \varphi_f^+ ([\![S^n]\!]^\Phi) \times \varphi_g^+ ([\![S^n]\!]^\Phi) && (\text{lem. A.2}) \\
\Rightarrow &[\![\varphi_{f\times g}^+ ([\![S^n]\!]^\Phi)]\!]^\Psi \subseteq [\![\varphi_f^+ ([\![S^n]\!]^\Phi) \times \varphi_g^+ ([\![S^n]\!]^\Phi)]\!]^\Psi \\
\Leftrightarrow &[\![\varphi_{f\times g}^+ ([\![S^n]\!]^\Phi)]\!]^\Psi \subseteq [\![\varphi_f^+ ([\![S^n]\!]^\Phi)]\!]^\Psi \times [\![\varphi_g^+ ([\![S^n]\!]^\Phi)]\!]^\Psi
\end{aligned}
$$

When f and g are functionally independent and $\Phi \in \{\delta, \beta\}$ then all subset relations above are

replaced by equalities. The proof for the second equation is similar,

$$\begin{aligned}
&\varphi^-_{f\times g}\left(\llbracket S^k\rrbracket^\Phi\right) \subseteq \varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right) \cap \varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right) && \text{(lem. A.3)}\\
\Rightarrow &\llbracket\varphi^-_{f\times g}\left(\llbracket S^k\rrbracket^\Phi\right)\rrbracket^\Psi \subseteq \llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right) \cap \varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi\\
\Rightarrow &\llbracket\varphi^-_{f\times g}\left(\llbracket S^k\rrbracket^\Phi\right)\rrbracket^\Psi \subseteq \llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\rrbracket^\Psi \cap \llbracket\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi
\end{aligned}$$

The first two subset relations above are replaced by equalities when $\Phi \in \{\delta, \beta\}$. The last subset relation becomes an equality when f and g are functionally independent, which can be proved as follows. Firstly, note that set intersection commutes with their Cartesian product, i.e. for sets $S^i_1, S^i_2 \subseteq \mathbb{Z}^i$ and $S^j_1, S^j_2 \subseteq \mathbb{Z}^j$, it is always true that

$$\left(S^i_1 \times S^j_1\right) \cap \left(S^i_2 \times S^j_2\right) = \left(S^i_1 \cap S^i_2\right) \times \left(S^j_1 \cap S^j_2\right) \tag{A.3}$$

Deriving the right hand side of the second subset relation above gives us,

$$\begin{aligned}
&\llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right) \cap \varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi\\
&= \llbracket\left(\text{proj}_f\left(\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\right) \times \text{proj}_g\left(\mathbb{Z}^n\right)\right) \cap \left(\text{proj}_f\left(\mathbb{Z}^n\right) \times \text{proj}_g\left(\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\right)\right)\rrbracket^\Psi && \text{(since } f,g \text{ are f.i.)}\\
&= \llbracket\left(\text{proj}_f\left(\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\right) \cap \text{proj}_f\left(\mathbb{Z}^n\right)\right) \times \left(\text{proj}_g\left(\mathbb{Z}^n\right) \cap \text{proj}_g\left(\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\right)\right)\rrbracket^\Psi && \text{(eq. } A.3)\\
&= \llbracket\text{proj}_f\left(\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\right) \cap \text{proj}_f\left(\mathbb{Z}^n\right)\rrbracket^\Psi \times \llbracket\text{proj}_g\left(\mathbb{Z}^n\right) \cap \text{proj}_g\left(\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\right)\rrbracket^\Psi\\
&= \llbracket\text{proj}_f\left(\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\right)\rrbracket^\Psi \times \llbracket\text{proj}_g\left(\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\right)\rrbracket^\Psi\\
&= \text{proj}_f\left(\llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\rrbracket^\Psi\right) \times \text{proj}_g\left(\llbracket\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi\right)\\
&= \left(\text{proj}_f\left(\llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\rrbracket^\Psi\right) \cap \text{proj}_f\left(\mathbb{Z}^n\right)\right) \times \left(\text{proj}_g\left(\mathbb{Z}^n\right) \cap \text{proj}_g\left(\llbracket\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi\right)\right)\\
&= \left(\text{proj}_f\left(\llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\rrbracket^\Psi\right) \times \text{proj}_g\left(\mathbb{Z}^n\right)\right) \cap \left(\text{proj}_f\left(\mathbb{Z}^n\right) \times \text{proj}_g\left(\llbracket\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi\right)\right) && \text{(eq. } A.3)\\
&= \llbracket\varphi^-_f\left(\llbracket S^i\rrbracket^\Phi\right)\rrbracket^\Psi \cap \llbracket\varphi^-_g\left(\llbracket S^j\rrbracket^\Phi\right)\rrbracket^\Psi && \text{(since } f,g \text{ are f.i.)}
\end{aligned}$$

The proof for the third equation is very similar to the one just presented and therefore is ommited. □

Proposition 7.34. *Let $\Phi_1, \Phi_2 \in \{\varphi, \delta, \beta\}$ and c be an arbitrary constraint. An incomplete constraint checker $\langle\Phi_1, \Phi_2\rangle^c$ is a sound constraint checker for c.*

Proof. Since by definition $\widehat{\chi}_c(S) = \varphi^+_c(S)$ is a sound constraint checker and we have $\varphi^+_c(S) \subseteq \llbracket\varphi^+_c\left(\llbracket S\rrbracket^{\Phi_1}\right)\rrbracket^{\Phi_2}$ then $\langle\Phi_1, \Phi_2\rangle^c$ must be sound. □

A.3. Proofs of chapter 8

Lemma A.4. *Let S^n, S^k be two arbitrary δ-domains, $f : \mathbb{Z}^n \to \mathbb{Z}^k$ a Cartesian product of k n-ary functions, and $\Phi, \Psi \in \{\delta, \beta\}$. Let $N = 1 \ldots n$, $K = n+1 \ldots n+k$, and $c = [f(x_N) = x_K]$. Then,*

$$\begin{aligned} \llbracket \varphi_f^+ (\llbracket S^n \rrbracket^\Phi) \cap \llbracket S^k \rrbracket^\Phi \rrbracket^\Psi \cap S^k &= proj_K \circ \pi_c^{\Phi\Psi\star} \left(S^n \times S^k \right) \\ \llbracket \widehat{\varphi}_f \left(\llbracket S^k \rrbracket^\Phi, \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n &= proj_N \circ \pi_c^{\Phi\Psi\star} \left(S^n \times S^k \right) \end{aligned}$$

Proof. Let $t = n+k$ and $S^t = S^n \times S^k$. Additionally, we will explore the decomposition $c = e \circ g$, where $g(x^t) = \langle f(x_N), x_K \rangle$ and $e = [x_{1\ldots k} = x_{k+1\ldots 2k}]$. For the first equation we have,

$$\begin{aligned} \mathrm{proj}_K \circ \pi_c^{\Phi\Psi\star}(S^t) &= \mathrm{proj}_K \left(\llbracket \widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^t \right) \\ &= \mathrm{proj}_K \left(\llbracket \widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \rrbracket^\Psi \right) \cap \mathrm{proj}_K (S^t) \\ &= \llbracket \mathrm{proj}_K \left(\widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \right) \rrbracket^\Psi \cap S^k \\ &= \llbracket \mathrm{proj}_K \left(\varphi_g^- \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right) \right) \cap \mathrm{proj}_K (\llbracket S^t \rrbracket^\Phi) \rrbracket^\Psi \cap S^k \\ &= \llbracket \mathrm{proj}_{k+1\ldots 2k} \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right) \cap \llbracket S^k \rrbracket^\Phi \rrbracket^\Psi \cap S^k \\ &= \llbracket \mathrm{proj}_{1\ldots k} \left(\varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right) \cap \mathrm{proj}_{k+1\ldots 2k} \left(\varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right) \cap \llbracket S^k \rrbracket^\Phi \rrbracket^\Psi \cap S^k \\ &= \llbracket \varphi_f^+ (\llbracket S^n \rrbracket^\Phi) \cap \llbracket S^k \rrbracket^\Phi \rrbracket^\Psi \cap S^k \end{aligned}$$

The proof for the second equation is as follows.

$$\begin{aligned} \mathrm{proj}_N \circ \pi_c^{\Phi\Psi\star}(S^t) &= \mathrm{proj}_N \left(\llbracket \widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^t \right) \\ &= \mathrm{proj}_N \left(\llbracket \widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \rrbracket^\Psi \right) \cap \mathrm{proj}_N (S^t) \\ &= \mathrm{proj}_N \left(\llbracket \widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \rrbracket^\Psi \right) \cap S^n \\ &= \llbracket \mathrm{proj}_N \left(\widehat{\varphi}_g \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi), \llbracket S^t \rrbracket^\Phi \right) \right) \rrbracket^\Psi \cap S^n \\ &= \llbracket \widehat{\varphi}_f \left(\mathrm{proj}_{1\ldots k} \left(\pi_e^\star \circ \varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right), \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n \\ &= \llbracket \widehat{\varphi}_f \left(\mathrm{proj}_{1\ldots k} \left(\varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right) \cap \mathrm{proj}_{k+1\ldots 2k} \left(\varphi_g^+ (\llbracket S^t \rrbracket^\Phi) \right), \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n \\ &= \llbracket \widehat{\varphi}_f \left(\varphi_f^+ (\llbracket S^t \rrbracket^\Phi) \cap \llbracket S^k \rrbracket^\Phi, \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n \\ &= \llbracket \widehat{\varphi}_f \left(\llbracket S^k \rrbracket^\Phi, \llbracket S^n \rrbracket^\Phi \right) \rrbracket^\Psi \cap S^n \end{aligned}$$

□

Proposition 8.12. *Any box view model may be enforced by a set of propagators and a set of auxiliary domain variables.*

Proof. A box view model for constraint $c \circ f$ and a δ-domain S^n is an instance of the following expression:

$$m \quad = \quad \llbracket \widehat{\varphi}_f \left(\llbracket \pi_c^{\beta\beta\star} \left(\llbracket \varphi_f^+ \left(\llbracket S^n \rrbracket^\beta \right) \rrbracket^\beta \right) \rrbracket^\beta, \llbracket S^n \rrbracket^\beta \right) \rrbracket^\beta \cap S^n \tag{A.4}$$

Let us assume a function $f : \mathbb{Z}^n \to \mathbb{Z}^k$ and consequently fix the arity of c to k. Propagating m may be achieved by considering a CSP with a domain $S^n \times S^k$, where S^k is the Cartesian product of the domains of k new domain variables. The auxiliary variable's domain must be sufficiently large, specifically such that $\llbracket \varphi_f^+ \left(\llbracket S^n \rrbracket^\beta \right) \rrbracket^\beta \subseteq S^k$, and consequently

$$\llbracket \varphi_f^+ \left(\llbracket S^n \rrbracket^\beta \right) \rrbracket^\beta = \llbracket \varphi_f^+ \left(\llbracket S^n \rrbracket^\beta \right) \cap \llbracket S^k \rrbracket^\beta \rrbracket^\beta \cap S^k$$

Then, using the above lemma one may rewrite equation A.4 using propagators $\pi_c^{\beta\beta\star}$, and $\pi_d^{\beta\beta\star}$ where $d = \left[f\left(x_{1\ldots n}\right) = x_{n+1\ldots n+k} \right]$. □

Appendix B.

Tables

B.1. Tables of chapter 4

benchmark	#wsum	#distinct	#element	#cumulative	benchmark	#wsum	#distinct	#element	#cumulative
quasigroup3-7.xml		14	49		bibd-31-31-6-6-1_glb.xml	527			
quasigroup4-7.xml		14	49		bibd-25-30-6-5-1_glb.xml	355			
quasigroup6-7.xml		14	97		bibd-7-210-60-2-10_glb.xml	238			
quasigroup7-7.xml		14	97		bibd-14-26-13-7-6_glb.xml	131			
squares-10-10.xml	45			2	bibd-19-57-9-3-1_glb.xml	247			
squares-11-11.xml	55			2	bibd-15-35-7-3-1_glb.xml	155			
squares-9-9.xml	36			2	bibd-8-56-21-3-6_glb.xml	92			
bibd-6-60-30-3-12_glb.xml	81				patat-02-comp-17.xml		300		
bibd-6-50-25-3-10_glb.xml	71				patat-08-comp-10.xml		904		
bibd-7-42-18-3-6_glb.xml	70				patat-08-comp-15.xml		486		
bibd-7-49-21-3-7_glb.xml	77				patat-08-comp-8.xml		490		
bibd-8-42-21-4-9_glb.xml	78				latinSquare-dg-6_all.xml		24		
bibd-8-28-14-4-6_glb.xml	64				queens2.xml		3		
bibd-8-56-28-4-12_glb.xml	92				cabinet-5561.xml	7		98	
bibd-9-48-16-3-4_glb.xml	93				cabinet-5562.xml	7		98	
rcpsp20-06.xml				3	cabinet-5563.xml	7		98	
rcpsp20-10.xml				3	cabinet-5564.xml	7		98	
rcpsp20-18.xml				3	cabinet-5565.xml	7		98	
rcpsp20-19.xml				3	cabinet-5566.xml	7		98	
bibd-11-55-15-3-3_glb.xml	121				cabinet-5567.xml	7		98	
bibd-10-60-18-3-4_glb.xml	115				cabinet-5568.xml	7		98	
bibd-13-78-18-3-3_glb.xml	169				cabinet-5569.xml	7		98	
bibd-12-44-11-3-2_glb.xml	122				cabinet-5570.xml	7		98	
bibd-13-52-12-3-2_glb.xml	143				cabinet-5571.xml	7		98	
magicSquare-8_glb.xml	18	1			cabinet-5572.xml	7		98	
magicSquare-7_glb.xml	16	1			cabinet-5573.xml	7		98	
magicSquare-9_glb.xml	20	1			cabinet-5574.xml	7		98	
magicSquare-6_glb.xml	14	1			cabinet-5575.xml	7		98	
bibd-13-26-8-4-2_glb.xml	117				cabinet-5576.xml	7		98	
bibd-11-22-10-5-4_glb.xml	88				cabinet-5577.xml	7		98	
bibd-12-22-11-6-5_glb.xml	100				cabinet-5578.xml	7		98	
bibd-16-24-9-6-3_glb.xml	160				cabinet-5579.xml	7		98	
bibd-19-19-9-9-4_glb.xml	209				cabinet-5580.xml	7		98	
bibd-21-21-5-5-1_glb.xml	252				costasArray-13.xml		11		
bibd-15-70-14-3-2_glb.xml	190				costasArray-14.xml		12		
bibd-16-80-15-3-2_glb.xml	216				costasArray-15.xml		13		

Table B.1.: Number of global constraints of each kind present in each benchmark

algorithm	propagator-centered				event-centered			
event policy	cost		FIFO		cost		FIFO	
filter policy	cost	FIFO	cost	FIFO	cost	FIFO	cost	FIFO
quasigroup3-7.xml	30.55	43.83	31.34	47.02	472.59	455.16	2322.28	2275.57
quasigroup4-7.xml	28.63	41.91	30.18	45.69	470.21	469.29	2194.11	2217.49
quasigroup6-7.xml	1.11	1.54	1.15	1.67	28.67	29.07	100.09	101.77
quasigroup7-7.xml	8.66	12.09	8.96	13.16	202.88	209.24	737.07	756.63
squares-10-10.xml	14.47	47.3	14.24	47.39	114.34	128.24	114.51	126.78
squares-11-11.xml	0.63	0.8	0.61	0.79	1.68	1.91	1.68	1.91
squares-9-9.xml	0.99	0.98	0.98	0.97	1.94	2.2	1.96	2.17
bibd-6-60-30-3-12_glb.xml	1.86	1.77	1.84	1.75	46.32	47.27	44.44	47.04
bibd-6-50-25-3-10_glb.xml	0.41	0.41	0.41	0.4	9.34	9.55	9.36	9.63
bibd-7-42-18-3-6_glb.xml	3.47	3.34	3.45	3.28	47.65	46.4	48.11	47.53
bibd-7-49-21-3-7_glb.xml	25.13	25.2	24.93	24.95	516.81	549.29	509.87	547.62
bibd-8-42-21-4-9_glb.xml	0.52	0.51	0.52	0.5	9.02	8.72	8.91	8.77
bibd-8-28-14-4-6_glb.xml	0.4	0.38	0.4	0.37	4.65	4.63	4.62	4.61
bibd-8-56-28-4-12_glb.xml	0.5	0.48	0.5	0.48	12.73	13.08	12.8	13.19
bibd-9-48-16-3-4_glb.xml	19.18	18.06	19.3	17.88	361.73	367.75	361.59	365.39
rcpsp20-06.xml	21.1	42.08	21.05	41.98	86.18	101.04	85.44	102.69
rcpsp20-10.xml	1.13	1.95	1.13	1.95	4.29	4.96	4.28	4.91
rcpsp20-18.xml	30.51	58.9	30.55	58.78	124.26	146.21	124.29	145.59
rcpsp20-19.xml	0.78	1.42	0.79	1.41	3.71	4.15	3.65	4.11
bibd-11-55-15-3-3_glb.xml	0.28	0.28	0.28	0.28	8.26	8.27	8.17	8.26
bibd-10-60-18-3-4_glb.xml	0.67	0.65	0.68	0.66	17.99	18.37	18.02	18.36
bibd-13-78-18-3-3_glb.xml	0.92	0.88	0.92	0.9	38.17	38.5	38.38	39.05
bibd-12-44-11-3-2_glb.xml	0.51	0.48	0.52	0.48	10.34	10.7	10.32	10.68
bibd-13-52-12-3-2_glb.xml	0.16	0.16	0.16	0.16	4.65	4.78	4.64	4.78
magicSquare-8_glb.xml	1.63	2.38	1.58	2.52	2.35	2.81	3.2	3.17
magicSquare-7_glb.xml	43.86	62.9	43.86	65.6	81.41	81.94	94.94	94.98
magicSquare-9_glb.xml	11.63	18.08	11.42	19	22.46	22.07	25.89	25.72
magicSquare-6_glb.xml	1.08	1.53	1.08	1.61	1.92	1.92	2.29	2.29
bibd-13-26-8-4-2_glb.xml	0.16	0.16	0.16	0.15	1.83	1.87	1.83	1.44
bibd-11-22-10-5-4_glb.xml	2.52	2.39	2.51	2.36	22.18	22.63	22.03	22.34
bibd-12-22-11-6-5_glb.xml	0.32	0.3	0.32	0.29	2.95	2.94	2.96	2.92
bibd-16-24-9-6-3_glb.xml	0.73	0.68	0.72	0.69	6.93	7.07	6.93	6.92
bibd-19-19-9-9-4_glb.xml	0.46	0.39	0.46	0.39	3.42	3.32	3.4	3.3
bibd-21-21-5-5-1_glb.xml	0.04	0.05	0.04	0.05	0.35	0.3	0.39	0.37
bibd-15-70-14-3-2_glb.xml	3.3	3.23	3.39	3.26	103.82	105.89	103.85	105.9
bibd-16-80-15-3-2_glb.xml	0.93	0.93	0.93	0.93	40.1	40.4	39.62	40.62

Table B.2.: Propagation time (seconds) for solving each benchmark using each model (table 1/2)

algorithm	propagator-centered				event-centered			
event policy	cost		FIFO		cost		FIFO	
filter policy	cost	FIFO	cost	FIFO	cost	FIFO	cost	FIFO
bibd-31-31-6-6-1_glb.xml	0.14	0.15	0.14	0.14	1.8	1.79	1.81	1.8
bibd-25-30-6-5-1_glb.xml	0.41	0.41	0.41	0.41	5.47	5.36	5.46	5.37
bibd-7-210-60-2-10_glb.xml	0.19	0.19	0.19	0.2	32.43	31.28	32.47	32.35
bibd-14-26-13-7-6_glb.xml	43.05	40.24	42.14	40.25	443.59	446.59	444.1	449.1
bibd-19-57-9-3-1_glb.xml	0.24	0.25	0.23	0.25	7.3	7.59	7.32	7.5
bibd-15-35-7-3-1_glb.xml	0.07	0.07	0.07	0.08	1.22	1.25	1.23	1.24
bibd-8-56-21-3-6_glb.xml	8.98	8.1	9.05	8.09	184.62	185.28	186.44	185.6
patat-02-comp-17.xml	3.17	3.52	3.21	3.72	4.25	4.45	7.77	7.55
patat-08-comp-10.xml	1.98	2.12	1.91	2.22	2.52	2.52	4.97	4.92
patat-08-comp-15.xml	19.62	26.43	19.38	26.01	23.87	23.11	49.99	49.24
patat-08-comp-8.xml	0.91	1.02	0.9	1.04	1.23	1.23	1.96	1.94
latinSquare-dg-6_all.xml	0.96	1.1	0.96	1.13	1.11	1.09	1.8	1.79
queens2.xml	1.6	3.99	1.61	4.41	2.41	2.37	4.72	4.6
cabinet-5561.xml	14.7	15.76	14.37	15.45	55.69	53.59	69.2	67.25
cabinet-5562.xml	14.68	15.64	14.43	15.46	55.7	53.78	68.63	68.43
cabinet-5563.xml	14.55	15.81	14.42	15.49	55.94	53.97	68.99	66.92
cabinet-5564.xml	14.56	15.77	14.47	15.45	54.58	53.12	69.17	67.82
cabinet-5565.xml	14.6	15.73	14.49	15.45	55.79	53.84	69.55	67.29
cabinet-5566.xml	14.6	15.72	14.47	15.52	55.1	53.89	69.15	66.96
cabinet-5567.xml	14.61	15.76	14.49	15.42	55.27	53	69.03	67.11
cabinet-5568.xml	14.59	15.72	14.43	15.35	55.3	53.45	69.38	67.21
cabinet-5569.xml	14.62	15.7	14.46	15.46	55.64	53.54	69.56	67.22
cabinet-5570.xml	14.61	15.68	14.44	15.43	55.25	53.42	69.21	68.16
cabinet-5571.xml	14.62	15.85	14.47	15.68	55.44	53.5	69.53	67.3
cabinet-5572.xml	14.67	15.78	14.53	15.47	55.57	53.58	69.17	67.19
cabinet-5573.xml	14.75	15.84	14.54	15.5	55.13	53.68	69.23	67.16
cabinet-5574.xml	14.68	15.9	14.58	15.56	55.14	53.13	69.73	67.5
cabinet-5575.xml	14.72	15.97	14.54	15.52	56.18	53.63	69.24	67.08
cabinet-5576.xml	14.98	16.11	14.76	15.78	56.67	54.6	71.3	68.46
cabinet-5577.xml	14.98	16.11	14.73	15.78	56.95	54.67	70.51	68.41
cabinet-5578.xml	15.06	16.1	14.71	15.75	56.93	54.73	70.46	68.68
cabinet-5579.xml	14.91	16.14	14.79	15.91	57.01	54.16	70.42	68.64
cabinet-5580.xml	14.94	16.21	14.71	15.76	57.03	54.59	71.06	68.4
costasArray-13.xml	1.96	2.91	1.95	3	2.53	2.51	3.61	3.59
costasArray-14.xml	1.86	2.88	1.86	2.99	2.37	2.35	3.49	3.44
costasArray-15.xml	10.28	16.52	10.14	17.15	13.25	13.18	20.09	19.75

Table B.3.: Propagation time (seconds) for solving each benchmark using each model (table 2/2)

B.2. Tables of chapter 5

B.2.1. Social golfers

	#updates	avg-update	time	#props
NON-INCREMENTAL	8.65E+6	6.37E-1	2.72	8,970,454
INCREMENTAL-PROPAGATOR	6.01E+6	6.06E-1	2.64	8,970,454
INCREMENTAL-VARIABLE	6.01E+6	6.07E-1	2.44	8,970,454
INCREMENTAL-HYBRID	5.42E+6	6.05E-1	2.94	8,970,454
GECODE	N/A	N/A	2.92	3,734,251

Table B.4.: Social golfers: 5w-5g-4s (v=25,c/v=11.2,f=25421)

	#updates	avg-update	time	#props
NON-INCREMENTAL	6.85E+8	5.64E-1	307	740,977,445
INCREMENTAL-PROPAGATOR	4.70E+8	6.49E-1	298	740,977,445
INCREMENTAL-VARIABLE	4.70E+8	6.49E-1	282.65	740,977,445
INCREMENTAL-HYBRID	4.24E+8	6.42E-1	303.62	740,977,445
GECODE	N/A	N/A	304	319,631,533

Table B.5.: Social golfers: 6w-5g-3s (v=30,c/v=13.6,f=1582670)

	#updates	avg-update	time	#props
NON-INCREMENTAL	1.25E+7	5.36E-1	5.36	13,597,902
INCREMENTAL-PROPAGATOR	8.17E+6	5.64E-1	5.75	13,597,902
INCREMENTAL-VARIABLE	8.17E+6	5.64E-1	5.13	13,597,902
INCREMENTAL-HYBRID	8.00E+6	5.61E-1	6.14	13,597,902
GECODE	N/A	N/A	6.65	6,265,389

Table B.6.: Social golfers: 11w-11g-2s (v=121,c/v=55.4,f=10803)

B.2.2. Hamming codes

	#updates	avg-update	time	#props
NON-INCREMENTAL	2.14E+6	1.48E+0	1.04	4,130,735
INCREMENTAL-PROPAGATOR	1.52E+6	6.20E-1	0.88	4,138,480
INCREMENTAL-VARIABLE	1.52E+6	6.20E-1	0.83	4,138,480
INCREMENTAL-HYBRID	1.37E+6	8.93E-1	0.9	4,138,480
GECODE	N/A	N/A	1.38	1,918,498

Table B.7.: Hamming codes: 20s-15l-8d (v=42,c/v=10,f=7774)

	#updates	avg-update	time	#props
NON-INCREMENTAL	8.34E+6	3.19E+0	5.66	16,769,450
INCREMENTAL-PROPAGATOR	5.61E+6	7.33E-1	3.78	16,775,413
INCREMENTAL-VARIABLE	5.61E+6	7.33E-1	3.73	16,775,413
INCREMENTAL-HYBRID	5.37E+6	1.11E+0	4.12	16,775,413
GECODE	N/A	N/A	5.16	6,420,351

Table B.8.: Hamming codes: 10s-20l-9d (v=22,c/v=5,f=59137)

	#updates	avg-update	time	#props
NON-INCREMENTAL	1.81E+7	1.88E+0	12.39	37,788,547
INCREMENTAL-PROPAGATOR	1.40E+7	5.84E-1	10.12	37,857,135
INCREMENTAL-VARIABLE	1.40E+7	5.84E-1	9.46	37,857,135
INCREMENTAL-HYBRID	1.23E+7	1.08E+0	10.97	37,857,135
GECODE	N/A	N/A	15.02	14,005,989

Table B.9.: Hamming codes: 40s-15l-6d (v=82,c/v=15.1,f=27002)

B.2.3. Balanced partition

	#updates	avg-update	time	#props
NON-INCREMENTAL	1.74E+6	5.34E+0	3.06	11,810,152
INCREMENTAL-PROPAGATOR	2.06E+6	5.28E-1	2.77	11,810,152
INCREMENTAL-VARIABLE	2.06E+6	5.27E-1	2.68	11,810,192
INCREMENTAL-HYBRID	1.02E+6	1.61E+0	2.86	11,810,152
GECODE	N/A	N/A	3.57	5,361,471

Table B.10.: Balanced partition: 150v-70s-162m (v=212,c/v=11.7,f=48525)

	#updates	avg-update	time	#props
NON-INCREMENTAL	2.01E+6	6.06E+0	3.86	14,504,618
INCREMENTAL-PROPAGATOR	2.50E+6	5.41E-1	3.41	14,504,618
INCREMENTAL-VARIABLE	2.50E+6	5.40E-1	3.29	14,504,658
INCREMENTAL-HYBRID	1.17E+6	1.75E+0	3.29	14,504,618
GECODE	N/A	N/A	4.64	6,598,932

Table B.11.: Balanced partition: 170v-80s-182m (v=242,c/v=13.4,f=54573)

	#updates	avg-update	time	#props
NON-INCREMENTAL	2.17E+6	6.77E+0	4.66	16,572,703
INCREMENTAL-PROPAGATOR	2.82E+6	5.52E-1	3.94	16,572,703
INCREMENTAL-VARIABLE	2.82E+6	5.51E-1	3.81	16,572,743
INCREMENTAL-HYBRID	1.26E+6	1.88E+0	4.4	16,572,703
GECODE	N/A	N/A	5.6	7,481,239

Table B.12.: Balanced partition: 190v-90s-202m (v=272,c/v=15,f=57424)

B.2.4. Metabolic pathways

	time	#props
INCREMENTAL-PROPAGATOR	0.9	1,030,438
INCREMENTAL-VARIABLE	0.74	1,268,920

Table B.13.: Metabolic pathways: g250_croes_ecoli_glyco (f=916)

	time	#props
INCREMENTAL-PROPAGATOR	10.3	7,163,926
INCREMENTAL-VARIABLE	9.38	20,681,066

Table B.14.: Metabolic pathways: g500_croes_ecoli_glyco (f=2865)

	time	#props
INCREMENTAL-PROPAGATOR	52.6	4,517,381
INCREMENTAL-VARIABLE	24.1	4,584,786

Table B.15.: Metabolic pathways: g1000_croes_scerev_heme (f=2466)

	time	#props
INCREMENTAL-PROPAGATOR	171	15,657,341
INCREMENTAL-VARIABLE	88.1	60,006,142

Table B.16.: Metabolic pathways: g1500_croes_ecoli_glyco (f=4056)

B.2.5. Winner determination problem

	time	#props
INCREMENTAL-PROPAGATOR	2.82	37,014
INCREMENTAL-VARIABLE	2.17	35,693

Table B.17.: Winner determination problem: 200 (f=1550)

	time	#props
INCREMENTAL-PROPAGATOR	13.4	69,030
INCREMENTAL-VARIABLE	9.92	66,536

Table B.18.: Winner determination problem: 300 (f=2924)

	time	#props
INCREMENTAL-PROPAGATOR	39	160,878
INCREMENTAL-VARIABLE	29.5	155,775

Table B.19.: Winner determination problem: 400 (f=6693)

	time	#props
INCREMENTAL-PROPAGATOR	80.4	393,580
INCREMENTAL-VARIABLE	61.7	381,869

Table B.20.: Winner determination problem: 500 (f=16213)

B.3. Tables of chapter 9

B.3.1. Systems of linear equations

	#vars	#vars/const	time	#fails	#props
VARS	41	7	19.3	11,983,390	1.14E+7
VARS+GLOBAL	20	4	18.2	11,983,390	5.72E+6
PVIEWS	20	4	18.1	11,983,390	5.72E+6
SVIEWS	20	4	18.9	11,983,390	5.72E+6
GECODE-VARS	41	7	19.9	11,983,390	8.75E+6
GECODE-VARS+GLOBAL	20	4	18.4	11,983,390	2.90E+6

Table B.21.: Linear 20var-7vals-7cons-4arity-6s (SAT) (S=56.15)

	#vars	#vars/const	time	#fails	#props
VARS	62	15	142	6,444,629	5.97E+8
VARS+GLOBAL	20	8	46.7	6,444,629	7.01E+7
PVIEWS	20	8	42.3	6,444,629	7.41E+7
SVIEWS	20	8	57	6,444,629	7.41E+7
GECODE-VARS	62	15	155	6,444,629	9.56E+8
GECODE-VARS+GLOBAL	20	8	30.36	6,444,629	6.62E+7

Table B.22.: Linear 20var-30val-6cons-8arity-2s (UNSAT) (S=98.14)

	#vars	#vars/const	time	#fails	#props
VARS	508	39	134	1,515,353	5.25E+8
VARS+GLOBAL	40	20	21.4	1,515,353	1.81E+7
PVIEWS	40	20	25.8	1,515,353	1.81E+7
SVIEWS	40	20	58.4	1,515,353	1.81E+7
GECODE-VARS	508	39	163	1,515,353	9.21E+8
GECODE-VARS+GLOBAL	40	20	15.6	1,515,353	2.07E+7

Table B.23.: Linear 40var-7val-12cons-20arity-3s (UNSAT) (S=112.29)

	#vars	#vars/const	time	#fails	#props
VARS	988	79	144	700,752	6.17E+8
VARS+GLOBAL	40	40	16.8	700,752	1.00E+7
PVIEWS	40	40	20.3	700,752	1.00E+7
SVIEWS	40	40	62	700,752	1.00E+7
GECODE-VARS	988	79	168	700,752	1.07E+9
GECODE-VARS+GLOBAL	40	40	11.4	700,752	1.54E+7

Table B.24.: Linear 40var-7val-10cons-40arity-3s (UNSAT) (S=112.29)

B.3.2. Systems of nonlinear equations

	#vars	#vars/const	time	#fails	#props
VARS	90	15	53.1	651,189	1.89E+8
VARS+GLOBAL	60	12	46.9	651,189	1.24E+8
PVIEWS	20	8	19.6	651,189	2.40E+7
SVIEWS	20	8	21	651,189	2.40E+7
CPVIEWS	20	8	31.5	651,189	6.79E+7
PVIEWS+GLOBAL	20	8	25	651,189	2.40E+7
GECODE-VARS	90	15	40.3	651,189	2.67E+8
GECODE-VARS+GLOBAL	60	12	28.7	651,189	1.72E+8

Table B.25.: NonLinear 20var-20val-10cons-4term-2fact-2s (SAT) (S=86.44)

	#vars	#vars/const	time	#fails	#props
VARS	183	15	136	1,451,881	4.79E+8
VARS+GLOBAL	126	12	119	1,451,881	3.10E+8
PVIEWS	50	8	48	1,451,881	5.70E+7
SVIEWS	50	8	56.4	1,451,881	5.70E+7
CPVIEWS	50	8	73.7	1,451,881	1.58E+8
PVIEWS+GLOBAL	50	8	60.5	1,451,881	5.70E+7
GECODE-VARS	183	15	105.8	1,451,881	6.34E+8
GECODE-VARS+GLOBAL	126	12	74.15	1,451,881	4.10E+8

Table B.26.: NonLinear 50var-10val-19cons-4term-2fact-1s (SAT) (S=166.1)

	#vars	#vars/const	time	#fails	#props
VARS	358	23	93	514,680	2.90E+8
VARS+GLOBAL	134	15	86.7	514,680	2.23E+8
PVIEWS	50	12	38	596,669	2.79E+7
SVIEWS	50	12	47.4	596,669	2.79E+7
CPVIEWS	50	12	63.4	514,680	1.18E+8
PVIEWS+GLOBAL	50	12	43.9	596,669	2.79E+7
GECODE-VARS	358	23	71.3	514,680	3.95E+8
GECODE-VARS+GLOBAL	134	15	64.4	514,680	3.67E+8

Table B.27.: NonLinear 50var-10val-28cons-4term-3fact-1s (UNSAT) (S=166.1)

	#vars	#vars/const	time	#fails	#props
VARS	270	23	51.2	342,311	1.60E+8
VARS+GLOBAL	110	15	46.8	342,311	1.20E+8
PVIEWS	50	12	16.9	346,762	1.18E+7
SVIEWS	50	12	20.4	346,762	1.18E+7
CPVIEWS	50	12	29.4	342,311	5.37E+7
PVIEWS+GLOBAL	50	12	19	346,762	1.18E+7
GECODE-VARS	270	23	39.1	342,311	2.20E+8
GECODE-VARS+GLOBAL	110	15	33.5	342,311	1.88E+8

Table B.28.: NonLinear 50var-5val-20cons-4term-3fact-1s (UNSAT) (S=116.1)

	#vars	#vars/const	time	#fails	#props
VARS	440	31	68.6	282,476	2.12E+8
VARS+GLOBAL	362	28	65.3	282,476	1.75E+8
PVIEWS	50	16	27.6	287,138	1.30E+7
SVIEWS	50	16	35.8	287,138	1.30E+7
CPVIEWS	50	16	59.9	282,476	8.67E+7
PVIEWS+GLOBAL	50	16	28	287,138	1.30E+7
GECODE-VARS	440	31	51.7	282,476	2.92E+8
GECODE-VARS+GLOBAL	362	28	47.3	282,476	2.72E+8

Table B.29.: NonLinear 50var-6val-26cons-4term-4fact-1s (UNSAT) (S=129.25)

	#vars	#vars/const	time	#fails	#props
VARS	420	31	1.34	7,742	4.18E+6
VARS+GLOBAL	348	28	1.27	7,742	3.39E+6
PVIEWS	60	16	0.58	7,742	2.68E+5
SVIEWS	60	16	0.75	7,742	2.68E+5
CPVIEWS	60	16	1.1	7,742	1.59E+6
PVIEWS+GLOBAL	60	16	0.59	7,742	2.68E+5
GECODE-VARS	420	31	1.13	7,743	5.68E+6
GECODE-VARS+GLOBAL	348	28	1.06	7,742	5.47E+6

Table B.30.: NonLinear 60var-4val-24cons-4term-4fact-5s (UNSAT) (S=120)

B.3.3. Social golfers

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	4100	48	0.89	5,035	3.47E+6
PVIEWS+GLOBAL	100	32	0.6	5,035	5.06E+5
SVIEWS+GLOBAL	100	32	0.82	5,035	5.06E+5
CPVIEWS+GLOBAL	100	32	1.07	5,035	4.23E+6
GECODE-VARS+GLOBAL	4100	48	0.89	5,035	3.47E+6

Table B.31.: Social golfers: 5week-5group-4size (S=432.19)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	3465	27	9.88	33,969	4.25E+7
PVIEWS+GLOBAL	90	18	5.84	33,969	6.24E+6
SVIEWS+GLOBAL	90	18	7.66	33,969	6.24E+6
CPVIEWS+GLOBAL	90	18	9.74	33,969	3.72E+7
GECODE-VARS+GLOBAL	3465	27	5.31	33,969	2.81E+7

Table B.32.: Social golfers: 6week,5group,3size (S=351.62)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	14602	147	38.2	120,110	1.50E+8
PVIEWS+GLOBAL	196	98	26.2	120,110	1.74E+7
SVIEWS+GLOBAL	196	98	32.4	120,110	1.74E+7
CPVIEWS+GLOBAL	196	147	41.4	120,110	1.40E+8
GECODE-VARS+GLOBAL	14602	147	27.9	120,110	7.05E+7

Table B.33.: Social golfers: 4week,7group,7size (S=1100.48)

B.3.4. Golomb ruler

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	55	55	0.11	1,703	3.77E+5
PVIEWS+GLOBAL	10	10	0.08	1,707	5.07E+4
SVIEWS+GLOBAL	10	10	0.09	1,707	5.07E+4
CPVIEWS+GLOBAL	10	10	0.1	1,703	2.24E+5
GECODE-VARS+GLOBAL	55	55	0.08	1,707	3.36E+5

Table B.34.: Golomb ruler: 10 (S=58.07)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	66	66	0.63	7,007	2.12E+6
PVIEWS+GLOBAL	11	11	0.47	7,063	2.41E+5
SVIEWS+GLOBAL	11	11	0.52	7,063	2.41E+5
CPVIEWS+GLOBAL	11	11	0.56	7,007	1.23E+6
GECODE-VARS+GLOBAL	66	66	0.37	7,012	1.59E+6

Table B.35.: Golomb ruler: 11 (S=68.09)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	78	78	36.58	283,156	1.16E+8
PVIEWS+GLOBAL	12	12	27.6	284,301	1.19E+7
SVIEWS+GLOBAL	12	12	30.7	284,301	1.19E+7
CPVIEWS+GLOBAL	12	12	32.4	283,156	6.73E+7
GECODE-VARS+GLOBAL	78	78	21.4	283,162	1.00E+8

Table B.36.: Golomb ruler: 12 (S=76.91)

B.3.5. Low autocorrelation binary sequences

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	296	296	17.6	214,151	5.15E+7
PVIEWS-PARTIAL-1	65	65	13.8	214,151	2.61E+7
PVIEWS-PARTIAL-2	44	44	10.6	214,151	1.34E+7
PVIEWS+GLOBAL	23	23	6.13	214,151	2.18E+6
SVIEWS+GLOBAL	23	23	14.6	214,151	2.18E+6
CPVIEWS+GLOBAL	23	23	7.51	214,151	8.97E+6
GECODE-VARS+GLOBAL	296	296	12.22	214,151	6.63E+7

Table B.37.: Low autocorrelation binary sequences: 22 (S=22)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	347	347	65.5	724,517	1.88E+8
PVIEWS-PARTIAL-1	71	71	49.9	724,517	9.11E+7
PVIEWS-PARTIAL-1	48	48	39.1	724,517	4.83E+7
PVIEWS+GLOBAL	25	25	23.4	724,517	7.38E+6
SVIEWS+GLOBAL	25	25	57.2	724,517	7.38E+6
CPVIEWS+GLOBAL	25	25	28.5	724,517	3.24E+7
GECODE-VARS+GLOBAL	347	347	44.73	724,517	2.46E+8

Table B.38.: Low autocorrelation binary sequences: 24 (S=25)

B.3.6. Fixed-length error correcting codes

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	6721	52	17.8	240,475	4.93E+7
PVIEWS+GLOBAL	641	20	9.25	240,475	1.25E+7
SVIEWS+GLOBAL	641	20	12.7	240,475	1.25E+7
CPVIEWS+GLOBAL	641	20	12.9	240,475	4.10E+7
GECODE-VARS+GLOBAL	6721	52	53.5	240,475	2.41E+7

Table B.39.: Fixed-length error correcting codes: 2-20-32-10-hamming (S=640)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	4201	50	7.96	153,207	2.46E+7
PVIEWS+GLOBAL	526	15	4.56	153,207	7.24E+6
SVIEWS+GLOBAL	526	15	5.89	153,207	7.24E+6
CPVIEWS+GLOBAL	526	15	6.51	153,207	2.09E+7
GECODE-VARS+GLOBAL	4201	50	24.57	153,207	1.84E+7

Table B.40.: Fixed-length error correcting codes: 3-15-35-11-hamming (S=525)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	24961	148	35.1	240,475	1.71E+8
PVIEWS+GLOBAL	641	20	9.74	240,475	6.46E+6
SVIEWS+GLOBAL	641	20	23.8	240,475	6.46E+6
CPVIEWS+GLOBAL	641	20	29.4	240,475	2.09E+8
GECODE-VARS+GLOBAL	24961	148	292	240,475	1.48E+8

Table B.41.: Fixed-length error correcting codes: 2-20-32-10-lee (S=640)

	#vars	#vars/const	time	#fails	#props
VARS+GLOBAL	15226	155	76.6	1,066,081	3.82E+8
PVIEWS+GLOBAL	526	15	34.5	1,283,612	2.86E+7
SVIEWS+GLOBAL	526	15	86.5	1,283,612	2.86E+7
CPVIEWS+GLOBAL	526	15	55.7	1,066,081	3.65E+8
GECODE-VARS+GLOBAL	15226	155	807	1,066,081	3.29E+8

Table B.42.: Fixed-length error correcting codes: 3-15-35-10-lee (S=525)

B.4. Tables of chapter 11

Rank	Solver	Version	Number of solved instances	% of VBS	Cumulated CPU time on solved instances	Average CPU time per solved instance
Virtual Best Solver (VBS)			660	100%	20689.09	31.35
1	cpHydra	k_10	569	86%	31124.32	54.70
2	cpHydra	k_40	569	86%	49899.25	87.70
3	**casper**	**zao**	**562**	**85%**	**31205.61**	**55.53**
4	Mistral-prime	1.313	560	85%	17929.82	32.02
5	MDG-probe	2008-06-27	560	85%	20146.70	35.98
6	**casper**	**zito**	**555**	**84%**	**31629.81**	**56.99**
7	Mistral-option	1.314	554	84%	43364.56	78.28
8	MDG-noprobe	2008-06-27	530	80%	18115.71	34.18
9	choco2_impwdeg	2008-06-26	518	78%	25745.10	49.70
10	Abscon 112v4	ESAC	488	74%	36101.57	73.98
11	Abscon 112v4	AC	486	74%	25860.43	53.21
12	Sugar	v1.13+minisat	486	74%	29306.62	60.30
13	Sugar	v1.13+picosat	481	73%	26580.81	55.26
14	bpsolver	2008-06-27	459	70%	77397.38	168.62
15	Concrete + CPS4J	2008-05-30	456	69%	91862.96	201.45
16	Concrete + CSP4J - Tabu Engine	2008-05-30	199	30%	36056.08	181.19
17	SAT4J CSP	2008-06-13	174	26%	19628.31	112.81
18	Concrete + CSP4J - WMC Engine	2008-05-30	174	26%	23628.78	135.80

Table B.43.: CPAI'08 competition results (n-ary intension constraints category)

Rank	Solver	Version	Number of solved instances	% of VBS	Cumulated CPU time on solved instances	Average CPU time per solved instance
Virtual Best Solver (VBS)			501	100%	12233.86	24.42
1	Sugar	v1.13+picosat	424	85%	23575.77	55.60
2	cpHydra	k_40	420	84%	80832.22	192.46
3	cpHydra	k_10	419	84%	33879.89	80.86
4	Sugar	v1.13+minisat	405	81%	30617.59	75.60
5	Mistral-prime	1.313	403	80%	22849.50	56.70
6	**casper**	**zito**	**397**	**79%**	**32937.25**	**82.97**
7	**casper**	**zao**	**390**	**78%**	**24804.15**	**63.60**
8	Mistral-option	1.314	383	76%	37615.36	98.21
9	choco2_dwdeg	2008-06-26	358	71%	26103.18	72.91
10	MDG-noprobe	2008-06-27	353	70%	16548.16	46.88
11	choco2_impwdeg	2008-06-26	347	69%	28393.77	81.83
12	bpsolver	2008-06-27	347	69%	48728.37	140.43
13	MDG-probe	2008-06-27	337	67%	17668.90	52.43
14	Minion/Tailor	2008-07-04	216	43%	14889.77	68.93
15	Abscon 112v4	AC	184	37%	31394.49	170.62
16	Abscon 112v4	ESAC	173	35%	33623.73	194.36
17	SAT4J CSP	2008-06-13	63	13%	8092.34	128.45

Table B.44.: CPAI'08 competition results (global constraints category)