Techniques for Efficient Constraint Propagation

MIKAEL Z. LAGERKVIST

Licentiate Thesis
Stockholm, Sweden, 2008

TRITA-ICT/ECS AVH 08:10
ISSN 1653-6363
ISRN KTH/ICT/ECS AVH-08/10–SE
ISBN 978-91-7415-154-1

KTH
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen fredagen den 21 November 2008 klockan 13.15 i N2, Electrum 3, Kista.

Tryck: Universitetsservice US AB

On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Passages from the Life of a Philosopher
Charles Babbage

Abstract

This thesis explores three new techniques for increasing the efficiency of constraint propagation: support for incremental propagation, improved representation of constraints, and abstractions to simplify propagation.

Support for incremental propagation is added to a propagator-centered propagation system by adding a new intermediate layer of abstraction, advisors, that capture the essential aspects of a variable-centered system. Advisors are used to give propagators a detailed view of the dynamic changes between propagator runs. Advisors enable the implementation of optimal algorithms for important constraints such as extensional constraints and Boolean linear in-equations, which is not possible in a propagator-centered system lacking advisors.

Using Multivalued Decision Diagrams (MDD) as the representation for extensional constraints is shown to be useful for several reasons. Classical operations on MDDs can be used to optimize the representation, and thus speeding up the propagation. In particular, the reduction operation is stronger than the use of DFA minimization for the regular constraint. The use of MDDs is contrasted and compared to a recent proposal where tables are compressed.

Abstractions for constraint programs try to capture small and essential features of a model. These features may be much cheaper to propagate than the unabstracted program. The potential for abstraction is explored using several examples.

These three techniques work on different levels. Support for incremental propagation is essential for the efficient implementation of some constraints, so that the algorithms have the right complexity. On a higher level, the question of representation looks at what a propagator should use for propagation. Finally, the question of abstraction can potentially look at several propagators, to find cases where abstractions might be fruitful.

An essential feature of this thesis is an novel model for general placement constraints that uses regular expressions. The model is very versatile and can be used for several different kinds of placement problems. The model applied to the classic pentominoes puzzle will be used through-out the thesis as an example and for experiments.

Sammanfattning

Den här avhandlingen utforskar tre nya tekniker för att öka effektiviteten av villkorspropagering: stöd för inkrementell propagering, val av representation för villkor, samt abstraktion för att förenkla propagering.

Ett propageringssystem organiserat efter propagerare utökas med stöd för inkrementell propagering genom att lägga till ett nytt abstraktionslager: rådgivare. Detta lager fångar de essentiella aspekterna hos system organiserade efter variabler. Rådgivare används för att ge propagerare detaljerad information om de dynamiska ändringarna i variabler mellan körningar av propageraren. Utökningen innebär att det går att implementera optimala algoritmer för vissa viktiga villkor såsom tabellvillkor och Boolska linjära olikheter, något som inte är möjligt i ett system propagator-organiserat system utan rådgivare.

Användandet av MDDer (*Multivalued Decision Diagram*) som representation för tabellvillkor visas vara användbart i flera avseenden. Klassiska MDD-operationer kan användas för att optimera representationen, vilket leder till snabbare propagering. Specifikt så är reduktionsoperationen kraftfullare än användandet av DFA-minimering för reguljära villkor. MDD-representationen jämförs också med ett nyligen framlagt förslag för komprimerade tabeller.

Abstraktioner för villkorsprogram försöker fånga små men viktiga egenskaper i modeller.Sådana egenskaper kan vara mycket enklare att propagera än den konkreta modellen. Potentialen för abstraktioner undersöks för några exempel.

Dessa tre tekniker fungerar på olika nivåer. Stöd för inkrementell propagering är nödvändigt för att kunna implementera vissa villkor effektivt med rätt komplexitet. Valet av representation för villkor är på en högre nivå, då det gäller att se vilka algoritmer som skall användas för ett villkor. Slutligen så måste flera villkor i en modell studeras för att finna rätt typ av abstraktioner.

Ett utmärkande drag för den här avhandlingen är en ny modell för generella placeringsvillkor som använder reguljära uttryck. Modellen är mångsidig och kan användas för flera olika typer av placeringsproblem. Modellen specialiserad för pentominopussel används genomgående som exempel och för experiment.

Acknowledgements

First of all I would like to thank my advisor Christian Schulte for all the help, support, and time he has given me on my way to this thesis. I feel lucky for the opportunity I have had to work with him in this area. From Christian I have learned a lot, from high level areas such as the scientific method and what it means to be an academic, to more concrete things such as how to implement a system and how to write a paper.

I would also like to thank Guido Tack, with whom Christian and I work on Gecode, for all the interesting discussions and all the help. I have thoroughly enjoyed working together on a project like this, and feel that I have learned a lot from the experience.

Naturally, my thanks also go to Seif Haridi, my main supervisor, for support and help.

My heartfelt thanks to my two external collaborators Peter Tiedemann and Gilles Pesant. Many new ideas and possibilities arise from discussing known things with people that have a different perspective.

Getting to know my way in academic life has been easier thanks to the discussions with Johan, Thomas, and Ingo during lunches, and for that I'm grateful. Getting to know my way in life as a PhD student has been helped by Mikael C, Mikael M, and Mladen, fellow PhD students at the institution.

Of course, all my friends from the basic education at KTH continue to be important. Life at KTH would not have been as fun without them.

Last, but not least, a big thank you to my family for the support over the years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Constraint programming is a method for modeling and solving combinatorial (optimization) problems such as scheduling, rostering, routing, and sequencing. Typical uses might be for scheduling the dispatch of vehicles for a transport firm, finding a roster for nurses working at a hospital, or finding appropriate test case vectors for testing an integrated circuit. Constraint programming is a key method for solving these types of hard, typically NP-complete, problems in real life applications.

Modeling problems with constraint programming is done by defining the variables of the problem and the relations, called constraints, that must hold between these variables for them to represent a solution. A key feature is that variables have finite domains of possible values they can take.

Given a model of a problem, a constraint programming system can be used to solve the problem. The two main parts of the solving process are inference and search. Inference in constraint programming is called *constraint propagation*, or *propagation* for short. Propagation removes values for the variables that do not occur in any solution to a constraint. The process of repeated propagation steps for the constraints is guaranteed to eventually reach a fix-point. Propagation is typically not enough to find the solution of a problem: search must be used. To search for a solution, a guess is made that decomposes the problem into two or more disjoint sub-problems. The method of repeated propagation and heuristic decomposition continues recursively for the newly created sub-problems. The order of exploration will typically be depth first to limit the memory requirements.

Constraint programming systems are readily available for specifying and solving combinatorial models. For implementing propagation, components called propagators are used as implementations of constraints. A propagator encapsulates some inference method and operates on some set of variables. An important feature of propagators is independence from any other propagator in the system; communication between propagators is done solely by shrinking the domains of the variables. Propagation is a cornerstone of constraint programming in two senses: without propagation the constraint programming process would reduce to generate-and-test; most of the time spent solving a problem will be spent in propagation.

This thesis presents three new techniques for improving the efficiency of propagation. The techniques are concerned with the implementation of efficient incremental propagators, what representation of a constraint a propagator should use, and using abstractions of problems for more efficient propagation.

In the following section, a small example is presented where a Sudoku puzzle is solved with constraint programming. In the final section of the chapter, further details of this thesis are presented, including the contributions of the thesis.

1.1 Solving Sudoku with Constraint Programming

As a simple example, consider the popular Sudoku puzzle. In this puzzle, the object is to find values between one and nine for the squares in a 9 by 9 grid. The values must be placed so that all the values appear in each row, column, and major 3 by 3 block. An instance of the puzzle is a grid with some of the values already filled in, as is shown in Figure 1.1. One way to model the Sudoku puzzle with constraint programming, is to define a variable for each square with the initial domain of $\{1..9\}$. For the squares that are pre-filled, the other values are removed from the initial domain. The rules of the puzzle directly correspond to the constraints that must be satisfied in a solution.

To solve a model of a Sudoku instance, a constraint programming system is used. The system will have a set of variable types, constraints, heuristics, and search methods. Almost all systems have finite domain integer variables, so the variables from the model can be used directly. A very common constraint called `all-different`, enforcing that a set of variables must be pair-wise distinct, can be used to implement the rules for the rows, columns,

					3		6	
							1	
	9	7	5				8	
				9		2		
		8		7		4		
		3		6				
	1				2	8	9	
	4							
	5		1					

Figure 1.1: A Sudoku instance

and blocks. If `all-different` is not available then the constraint can be replaced by a set of `not-equals` constraints, one for each pair of variables in the original constraint. A typical heuristic used for decomposing a problem is called first fail, and works by taking the variable with the least amount of values left, and trying whether the variable can or can not be assigned to one of its still possible values. The typical search method used is depth first exploration.

Implementing the model in a constraint programming system such as Gecode [20] (see also Section 2.3), is straight-forward. The complete program for solving the puzzle takes less than 70 lines of C++ code. Since the `all-different` constraint is available (in Gecode called `distinct`), a total of 27 constraints are needed. Solving a 9 by 9 Sudoku instance takes a fraction of a millisecond on a modern computer. The search tree explored can be seen in Figure 1.4.

If stronger propagation is used, then the search tree will be smaller, in this case just a single node (as seen in Figure 1.5). Stronger propagation is invoked by adding the parameter `ICL_DOM` to the calls to distinct. For example, the call `distinct(this, m.row(i))` becomes `distinct(this, m.row(i), ICL_DOM)` instead. The reduction on search-tree size will in this case be balanced out by the increase in propagation time. In general a trade-off must be made between the level of effort spent on propagation and the potential reduction in search.

```
 1 #include "gecode/int.hh"
 2 #include "gecode/search.hh"
 3 #include "gecode/minimodel.hh"
 4 #include <iostream>
 5 using namespace Gecode;
 6
 7 /// Sudoku model.
 8 class Sudoku : public Space {
 9 public:
10   /// Values for the fields
11   IntVarArray x;
12   /// Model set-up
13   Sudoku(const int instance[9][9])
14     : x(this, 9*9, 1, 9) {
15     /// Access the x-array as a 9 by 9 matrix
16     Matrix<IntVarArray> m(x, 9, 9);
17
18     // Constraints for rows and columns
19     for (int i=0; i<9; i++) {
20       distinct(this, m.row(i));
21       distinct(this, m.col(i));
22     }
23
24     // Constraints for squares
25     for (int i=0; i<9; i+=3)
26       for (int j=0; j<9; j+=3)
27         distinct(this, m.slice(i, i+3, j, j+3));
28
29     // Fill-in predefined fields
30     for (int i=0; i<9; i++)
31       for (int j=0; j<9; j++)
32         if (int v = instance[i][j])
33           rel(this, m(i,j), IRT_EQ, v);
```

Figure 1.2: A C++ program for solving a Sudoku puzzle.

```
34      // Decomposition heuristic
35      branch(this, x, INT_VAR_SIZE_MIN, INT_VAL_SPLIT_MIN);
36    }
37
38    /// Constructor for cloning
39    Sudoku(bool share, Sudoku& s) : Space(share, s) {
40      x.update(this, share, s.x);
41    }
42    /// Perform copying during cloning
43    virtual Space* copy(bool share) {
44      return new Sudoku(share,*this);
45    }
46  };
47
48  int main(int argc, char* argv[]) {
49    int instance[9][9] = {
50      {8, 0, 0, 0, 0, 1, 0, 4, 0},
51      {2 ,0, 6, 0, 9, 0, 0, 1, 0},
52      {0, 0, 9, 0, 0, 6, 0, 8, 0},
53      {1 ,2 ,4, 0, 0, 0, 0, 0, 9},
54      {0, 0, 0, 0, 0, 0, 0, 0, 0},
55      {9 ,0, 0, 0, 0, 0, 8, 2, 4},
56      {0, 5 ,0, 4, 0, 0, 1, 0, 0},
57      {0, 8 ,0, 0, 7, 0, 2, 0, 5},
58      {0, 9 ,0, 5, 0, 0, 0, 0, 7}};
59    Sudoku* root = new Sudoku(instance);
60    Sudoku* sol  = dfs(root);
61    std::cout << sol->x << std::endl;
62    delete root; delete sol;
63    return 0;
64  }
```

Figure 1.3: A C++ program for solving a Sudoku puzzle (continued).

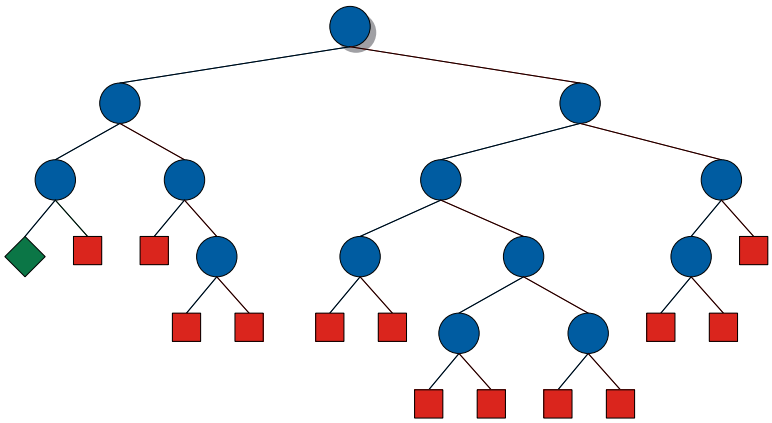


Figure 1.4: The search tree for the Sudoku instance from Figure 1.1 using the program from Figure 1.2. A circle represents a node where decomposition is needed, a square is a failure, and a rhombus form is a solution.

Figure 1.5: The search tree for the Sudoku instance from Figure 1.1 using the program from Figure 1.2 with stronger propagation. The solution is found through propagation only.

1.2 This thesis

This thesis explores three new techniques for increasing the efficiency of constraint propagation: support for incremental propagation in propagator-centered systems, representation of constraints, and using abstractions to simplify propagation. This section gives a short overview of each of these techniques.

Incremental propagation. A constraint programming system can organize the propagation process based on either modified variables (variable-centered propagation) or on the actual propagators that need to be executed (propagator-centered propagation). Variable-centered systems can give detailed modification information (needed for some incremental algorithms) relatively cheaply, while a propagator-centered system has good properties

for organizing propagators of vastly different kinds and complexity. Support for incremental propagation is added to a propagator-centered propagation system by adding a new intermediate layer of abstraction called advisors, capturing the essential aspects of a variable-centered system. Advisors are used to give propagators a detailed view of the dynamic changes between runs of the propagator. The addition allows the implementation of optimal algorithms for some important constraints such as extensional constraints and Boolean linear in-equations, which is not possible in a propagator-centered system lacking advisors.

Constraint representation. The representation of a constraint can have a huge impact on the effectiveness of propagation. Using Multivalued Decision Diagrams (MDD) as the representation for extensional constraints is shown to be useful for several reasons. Classical operations on MDDs can be used to optimize the representation, and thus speeding up the propagation. In particular, the reduction operation is stronger than the use of DFA minimization for the `regular` constraint. The use of MDDs is contrasted and compared to a recent proposal where tables are compressed.

Abstraction. Abstractions for constraint programs try to capture small and essential features of a constraint program. These features may be much cheaper to propagate than the unabstracted program. The potential for abstraction is explored using several examples.

The three techniques described above work on different levels. Support for incremental propagation is essential for the efficient implementation of some constraints, so that the algorithms have the right complexity. On a higher level, the question of representation looks at what data structure a propagator should use for propagation. Finally, the question of abstraction can potentially look at several propagators, to find cases where the purposes of the model for capturing the problem, and the level of detail needed for efficient propagation differ.

An essential feature of this thesis is an novel model for general placement constraints that uses regular expressions. The model is very versatile and can be used for several different kinds of placement problems. The model as applied to the classic pentominoes puzzle will be used through-out the thesis as an example and for experiments.

1.2.1 Contributions

In the following list, the main contributions of this thesis are listed, with references to the chapters they are described in, as well as the papers that have been published.

Incremental propagation (Chapter 4) A model and an implementation of advisors, an addition to propagator centered systems that allow the same, incremental algorithms as in variable-centered systems. The addition is show to be worthwhile using both analytical and practical benchmarks. Advisors have been integrated into the Gecode system, and are available from version 2.0.0 [20].

Published in [38], joint work with Christian Schulte, KTH. The presentation here is extended to include a detailed description of variable-centered versus propagator-centered systems, some new possibilities for using advisors, and a full model for a realistic constraint programming systems using views.

Propagator representation (Chapter 5) A new representation for generic global constraint is proposed based on Multivalued Decision Diagrams. The properties of the constraint and of the representation is detailed, and compared with other representations. In particular, the DFA minimization used for the `regular` constraint is shown to be weaker than MDD reduction.

Manuscript under preparation, joint work with Peter Tiedemann, IT University, Copenhagen, Denmark.

Abstraction (Chapter 6) The potential for using abstractions of problems is shown, together with some potential uses of it.

On-going, joint work with Christian Schulte.

Placement problems (Chapter 3, Section 6.1) A general, versatile, efficient, and simple model for placement problems is introduced that uses `regular` constraints. The model has been included in Gecode from version 2.0.0 [20].

Published in [37], joint work with Gilles Pesant, École Polytechnique de Montréal, Montreal, Canada.

Chapter 2

Constraint Programming

Constraint programming is a method to specify and solve combinatorial problems. This chapter gives an overview of constraint programming from both a formal perspective and how a system is implemented. First, the formal model of constraint programming is presented. In Section 2.2 constraint programming systems are described, since we will need to discuss the execution properties of a constraint system. Finally, in Section 2.3 the Gecode system is described.

2.1 Formal model

In the following, a formal model is defined that captures the essential aspects of constraint programming.

Variables and domains. There is a finite set of variables Var and a finite set of values Val. A *domain* $d \in \mathit{Dom}$ is a set of values a variable can take, $\mathit{Dom} = \mathcal{P}(\mathit{Val})$. A *store* $s \in \mathit{Store}$ is a complete mapping from variables to domains, $\mathit{Store} = \mathit{Var} \to \mathit{Dom}$. An *assignment* is a store where the range of the function is restricted to singleton sets ($\{\{v\} \mid v \in \mathit{Val}\}$), instead of the full set of domains. A pair of a variable x and a value v is called a *literal*.

Set operations $\diamond$ are lifted to a pair of stores S_1 and S_2 in the natural, point-wise way ($S_1 \diamond S_2 = \lambda x \in \mathit{Var}.\, S_1(x) \diamond S_2(x)$). Similarly, set relations $\sim$ are also lifted to stores ($S_1 \sim S_2 = \forall x \in \mathit{Var}.\, S_1(x) \sim S_2(x)$). Both operations and relations can be restricted to a subset of the variables by sub-scripting the operation or relation with the set. For example, stores S_1

and S_2 are *equal* with respect to a set of variables $X \subseteq \mathit{Var}$, written as $S_1 =_X S_2$, iff $\forall x \in X.S_1(x) = S_2(x)$.

A store S_1 is *stronger* than a store S_2, written $S_1 \leq S_2$, if $S_1 \subseteq S_2$. A store S_1 is *strictly stronger* than a store s_2, written $S_1 < S_2$, if $S_1 \leq S_2$ and $S_1 \neq S_2$. The *disagreement set* $\mathrm{dis}\,(S_1, S_2)$ of stores S_1 and S_2 is defined as $\{x \in \mathit{Var} \mid S_1 \neq_{\{x\}} S_2\}$.

A tuple of values over variables $x_1, \ldots, x_n$ can be turned into a store in the following way.

$$\mathit{Store}\,(\langle v_1, \ldots, v_n\rangle) = \lambda x \in \mathit{Var}.\begin{cases} \{v_l\} & \text{if } x = x_l \text{ for some } l \in \{1, \ldots, n\} \\ \mathit{Val} & \text{otherwise} \end{cases}$$

Constraints. A *constraint* $c \in \mathit{Con}$ over the set of variables $\mathrm{var}(c) = \{x_1, \ldots, x_n\}$ is defined as the set of assignments that are solutions to the constraint, $\mathit{Con} = \mathcal{P}(\{\langle v_1, \ldots, v_n\rangle \mid v_i \in \mathit{Val}\})$. A tuple $\langle v_1, \ldots, v_n\rangle$ in a constraint, where $\mathrm{var}(c) = \{x_1, \ldots, x_n\}$, is called a *support* for any literal $\langle x_i, v_i\rangle$.

A constraint c can be turned into a store by taking the union of all the solutions to the constraint, $\mathit{Store}\,(c) = \bigcup_{t \in c} \mathit{Store}\,(t)$. In the opposite direction, a store can be turned into a constraint over variables $x = \{x_1, \ldots, x_n\}$ using $\mathit{Cons}\,(s, x) = \{\langle v_1, \ldots, v_n\rangle \mid \forall i.v_i \in s(x_i)\}$.

Constraint satisfaction problems. A *constraint satisfaction problem* (CSP) is a pair of a set of constraints C and a store S, $\langle C, S\rangle \in \mathcal{P}(\mathit{Con}) \times \mathit{Store}$. A tuple $\langle v_1, \ldots, v_n\rangle$ in a constraint over variables $x_1, \ldots, x_n$ is *valid* under the store S iff $\forall i.v_i \in S(x_i)$. An assignment a is a *solution* to the CSP $\langle C, S\rangle$, both over variables $\{x_1, \ldots, x_n\}$, iff the assignment is a solution for each constraint, $\forall c \in C.\langle a(x_1), \ldots, a(x_n)\rangle \in c$, and the assignment is valid for the store, $a \subseteq S$. The solutions to a CSP, $\mathit{sol}(\langle C, S\rangle)$, is the set of all assignments that are solutions.

A constraint c is *entailed* in the store S iff $\mathit{Cons}\,(S, \mathrm{var}(c)) \subseteq c$. Entailed constraints can safely be removed from the CSP since they no longer restrict the set of solutions.

2.2 Constraint programming systems

To solve a CSP in practice, a constraint programming system is used. A system will typically not use the CSP formulation directly: constraints defined as the set of tuples they accept are exponential in the arity of the constraint. Instead, when possible intensional components called propagators are used, where a propagator will implement a constraint. Typical examples are relations, linear equations, and `all-different`.

This section describes constraint programming systems. The main components are variables and propagators. Additionally, a constraint programming system needs to organize propagation and perform search.

Variables. Variables with finite domains of uninterpreted values are fine for a formal model, while in practice the set of values will have some meaning. The most typical set of values is some finite subset of integer. Another type of value is finite sets of integers, where the domain of a variable will be a set of sets of values.

Propagators. A *propagator* is a function p that takes a store s as input and returns a new store s'. A propagator p must be *contracting*: $p(s) \leq s$ for all stores s. A propagator p must also be *monotonic*: if $s_1 \leq s_2$ then $p(s_1) \leq p(s_2)$ for all stores s_1 and s_2. A store s is a *fix-point* of a propagator p, if $p(s) = s$. That a propagator is contracting means that it can only remove values from a domain. That a propagator is monotonic means that starting propagation from a larger domain never gives more propagation.

A propagator p that references variables $x_1, \ldots, x_n$ is said to *implement* its associated constraint c_p. The associated constraint is defined as the set of assignments that the propagator identifies as solutions:

$$c_p = \{\langle v_1, \ldots, v_n\rangle \mid p(\mathit{Store}\,(\langle v_1, \ldots, v_n\rangle)) = \mathit{Store}\,(\langle v_1, \ldots, v_n\rangle)\}$$

For a given constraint c, any propagator p such that $c_p = c$ can be used. Note that there usually exists many different propagators taht can be used. The difference is in how much propagation they can perform.

Constraint programs. Analogously to constraint satisfaction problems, we can combine a store and a set of propagators to form a *constraint program*

(CP) $\langle P, S\rangle$. The set of solutions to a CP is defined as:

$$sol(\langle P, S\rangle) = \{v = \langle v_1, \ldots, v_n\rangle \mid \forall p \in P.p(Store\,(v)) = Store\,(v)\}$$

Similar to constraints in a CSP, a propagator is *entailed* for a store s iff for all stores $S' \leq S$, it holds that $p(S') = S'$. An entailed propagator can be removed from the system, since it will no longer do any propagation.

Variable dependencies. To manage propagation efficiently, a constraint programming system needs to know which propagators may affect which variables, and for which variables a domain change might make a propagator not be at a fix-point.

A set of variables $X \subseteq Var$ is *sufficient* for a propagator p, if it satisfies the following properties. First, no output on other variables is computed, that is, $S =_{Var-X} p(S)$ for all stores S. Second, no other variables are considered as input: if $S_1 =_X S_2$, then $p(S_1) =_X p(S_2)$ for all stores S_1, S_2.

For each propagator p a sufficient set of variables, its *dependencies*, $\mathrm{var}(p) \subseteq Var$ is defined. Dependencies are used in propagation as follows: if a store S is a fix-point of a propagator p, then any store $S' \leq S$ with $\mathrm{var}(p) \cap \mathrm{dis}\,(S, S') = \emptyset$ is also a fix-point of p. To better characterize how propagators and variables are organized in an implementation, the set of propagators $\mathrm{prop}[x]$ depending on a variable x is defined as $p \in \mathrm{prop}[x]$ if and only if $x \in \mathrm{var}(p)$.

Propagation. Constraint propagation refers to the process of finding the greatest mutual fix-point (equivalently, the weakest mutual fix-point with respect to the strenght of stores) of the set of propagators from an initial store S that propagation starts from. Since propagators are defined to be monotonic contracting functions, it is guaranteed that there exists a unique greatest mutual fix-point. The cornerstone of a propagation algorithm is to maintain some representation of what propagators might not be at fix-point. There are two main possibilities of what to keep track of, propagators and variables. Variable-centered propagation is controlled by the set of modified variables with some additional information (for example, variable and constraint in AC3 [41], variable and value in AC4 [43]). Propagator-centered propagation is controlled by the set of propagators still to be propagated, see for example [6].

```
Propagate(P,s)
begin
    N ← P;
    while N ≠ ∅ do
        remove p from N;
        s' ← p(s);
        N ← N ∪ ⋃_{x∈dis(s,s')} prop[x];
        s ← s';
    return d;
end
```

Algorithm 1: Propagator-centered propagation

```
Propagate(P,s)
begin
    V ← Var;
    while V ≠ ∅ do
        remove v from V;
        foreach p ∈ prop[v] do
            s' ← p(s);
            V ← V ∪ {v | v ∈ dis(s, s')};
            s ← s';
    return d;
end
```

Algorithm 2: Variable-centered propagation

Propagator-centered propagation is shown in Algorithm 1. It is assumed that all propagators are contained in the set P. The set N contains propagators not known to be at fix-point. The remove operation is left unspecified, but a realistic implementation bases the decision on priority or cost, see for example [52].

Variable-centered propagation is shown in Algorithm 2. The difference from propagator-centered propagation is that instead of a set of propagators to run (N in Algorithm 1), a set of variables that have been modified is used (V in Algorithm 2). It is common that a propagator will in addition to the store receive the variable that is being propagated (for incremental propagation), and will not modify that variable.

Algorithms 1 and 2 do not spell out some details. Failure is captured by computing a failed store (a store s with $s(x) = \emptyset$ for some $x \in Var$) by

propagation. Letting propagators signal failure directly (and thereby aborting the propagation loop directly) is more efficient. A real system should also pay attention to entailment or idempotency of propagators. Propagation events describing how stores change are discussed in Sect. 4.4. For a complete discussion of constraint propagation algorithms see [7], and for the implementation of these algorithms in constraint programming systems see [51].

Search. Propagation alone is most often not enough to reduce the store to a single solution. The simplest example is when there might be more than one solution. In this case, the system must resort to searching for a solution.

When propagation has finished a constraint programming system must make a heuristic guess. This guess might be a generic procedure (split the domain of the variable with the fewest values left) or it might be something problem specific. To represent the choices, new propagators are added to the system. A set of propagators $\{p_1, \ldots, p_n\}$ are said to be *branching propagators* for a CP $\langle P, S\rangle$ if the propagators induce a partition of the solutions to the CP. Formally, both the following conditions must hold.

$$sol(\langle P, S\rangle) = \bigcup_{p\in\{p_1,\ldots,p_n\}} sol(\langle P \cup \{p\}, S\rangle)$$

$$sol(\langle P \cup \{p\}, S\rangle) \cap sol(\langle P \cup \{p'\}, S\rangle) = \emptyset, \quad \forall p, p' \in \{p_1, \ldots, p_n\}.p \neq p'$$

The first property means that the set must not remove any solutions. The second property ensures that solutions are not duplicated in the branches. The first property is neccessary for ensuring the soundness of the process. The second property is neccessary to not do any unneccessary work.

In Algorithm 3 a basic version of depth first search is shown. It uses propagation as a subroutine, and it will return the first solution found. For a real implementation of search, issues such as interruption, memory management, stack size, and restartability must be taken into account.

2.3 Gecode

Gecode [20] is an open source constraint programming system that implements propagator-centered propagation. It is implemented in C++, and features: a small, simple, and variable domain agnostic kernel; several variable domains (Booleans, finite domain integers, finite sets of integers, ...);

```
DFS(P,s)
begin
    s ← Propagate(P, s);
    choose branching propagators {p_1, ..., p_n};
    foreach p ∈ {p_1, ..., p_n} do
        s' ← DFS(P ∪ {p}, s);
        if s ≠ ⊥ then
            return s'
    return ⊥
end
```

Algorithm 3: Depth first search

many global propagators (`all-different`, `regular`, `circuit`, `cumulatives`, `global-cardinality`, ...); standard search engines (DFS, LDS, and BAB); a graphical interactive search tool (Gist); as well as many example problems. The system is used in this thesis for implementation and benchmarking.

The system aims to be open, free, portable, efficient, and accessible. Gecode is designed to be open for modifications and additions, facilitating experiments. It is free since it is distributed under the MIT license. The system is portable to most major modern platforms and modern C++-compilers. The efficiency of the system is competitive with leading commercial systems on equivalent models. The extensive reference documentation makes the system accessible.

All the above points are important for using the system in the thesis. An open and accessible system is needed for modifying the architecture as in Section 4. A free system is needed for reproducibility of the results, and portability is needed for the same reason. An efficient system makes sure that improvements shown are not due to built-in inefficiencies that would disappear in a full system. This last point is very important for showing that the results are also practically relevant: they are demonstrated in a real, optimized system.

Chapter 3

Modeling Placement Problems

Solving a problem using constraint programming is a design process. What variables are needed to represent the problem, what kind of constraints are needed, and how should these constraints be implemented are some of the typical questions that arise.

In this section a model for general placement problems is developed. As a specialization, the model is used to solve the Pentominoes problem. Pentominoes are the shapes constructed from five equal-sized squares where the squares form a connected component, giving twelve possible shapes. The pieces can be rotated and mirrored to obtain symmetrical variants.

In Section 3.1 the `regular` constraint is described. In the following section, a model for general placement constraints is introduced that uses the `regular` constraint. In Section 3.3 the model is extended to handle the Pentominoes problem. The final section concludes with an evaluation of the model.

3.1 The regular constraint

The `regular` constraint [46] can be used to enforce that a vector of variables forms a word in a regular language. The constraint was introduced to model certain common structures in rostering problems, generalizing the `pattern` and `stretch` constraints.

To describe the constraint, some basic knowledge of regular languages is needed. After that the propagation algorithm for the constraint can be presented.

Regular languages. A *regular expression* (RE) is an expression in the language $R := RR \mid R^* \mid R|R \mid (R) \mid \epsilon \mid X$ where X is any value in *Val*. The regular expression R_1R_2 matches anything that matches R_1 followed by R_1 (concatenation), the expression R^* matches zero or more Rs (Kleene star), $R_1|R_2$ matches an R_1 or an R_2 (disjunction), (R) matches R (grouping), ϵ matches the empty string, and values match themselves. The expression R^n is a short-hand for the expression R concatenated n times, and the expression R^+ is shorthand for RR^*.

A *finite automaton* (FA) is a tuple $\langle S, T, q, A\rangle$, where S is a set of states, $T \in S \times \mathit{Val} \to \mathcal{P}(S)$ is a transition function, $q \in S$ is the start state, and $A \subseteq S$ is a set of accepting states. An FA matches a string $x_1, \ldots, x_n$ if there is a sequence of states $q_1, \ldots, q_{n+1}$ such that $q_1 = q$, $q_{i+1} \in T(q_i, x_i)$ for $i = 1, \ldots, n$, and $q_{n+1} \in A$.

If the range of the transition function is S instead of $\mathcal{P}(S)$, the FA is a *deterministic finite automaton* (DFA), otherwise it is a *non-deterministic finite automaton* (NFA). DFAs and NFAs can recognize the same languages, although a DFA might be exponentially bigger [30]. A DFA can be minimized cheaply ($O(n \log n)$), while minimization of an NFA is PSPACE-complete [33] and thus intractable in general. A regular expression can be translated into either an NFA or a DFA, and both methods are common in applications where regular expression matching is used.

Propagating a regular constraint. The propagation algorithm for the `regular` constraint uses a data structure called a *layered graph* (LG). The LG for a `regular` constraint over variables $x_1, \ldots, x_n$ is constructed by unfolding the DFA $\langle\{q_1, \ldots, q_m\}, T, q, A\rangle$ into a graph $\left\langle \cup_{i=1}^{n+1} N^i, E\right\rangle$, where layer N^i contains states $q_1^i, \ldots, q_m^i$, and edges are between consecutive layers following the DFA transitions, $E = \{\left\langle q_a^i, v, q_b^{i+1}\right\rangle \mid q_b \in T(q_a, v), v \in \mathit{Val}\}$. Propagation proceeds by finding the union of paths from q^1 to any q_f^{i+1} where $q_f \in A$. The set of paths has an edge between layers i and $i+1$ iff the edge value is valid for x_i, and the set of values used are the ones supported for x_i.

The sketch in Algorithm 4 shows how the base implementation of the regular constraint can be done. To improve efficiency, the layered graph is kept around as state, and updated incrementally each time depending on the changes in the domains of the variables [46]. A basic upper bound for the time-complexity of the algorithm is $O(|E|)$.

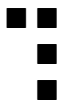
```
Regular(s)
begin
    Let LG be the edge-marked graph ⟨V, E⟩ = ⟨N_j^i, ∅⟩, where
    i ∈ {1, ..., n+1}, j ∈ {1, ..., m};
    foreach q_a, v, q_b such that q_b ∈ T(q_a, v) do
        foreach i ∈ {1, ..., n} do
            if v ∈ s(x_i) then
                E ← E ∪ ⟨q_a^i, v, q_b^{i+1}⟩;
    Mark all nodes and edges reachable from q^1;
    Remove edges not leading to a state in {q_k^{n+1} | q_k ∈ A};
    Remove values from s not support by marked edge in LG;
    return s
end
```

Algorithm 4: The propagation algorithm for regular

The propagation algorithm works for both DFAs and NFAs without any significant modifications. The reason to use DFAs is mainly that minimization is possible, which will remove any redundancies in the DFA and thus make E smaller. However, if the NFA is smaller than the minimized DFA it is of course preferable to use the NFA instead.

3.2 Shape Placement as a Regular Expression

Consider placing the shape in Figure 3.1(a) in the 4 by 4 grid shown in Figure 3.1(b). In order to encode its placement as a string, we cut the grid into horizontal strips corresponding to the rows and concatenate them starting with the top strip (Figures 3.1(c) and 3.1(d), we could equivalently have made vertical strips corresponding to columns). Each square of the resulting sequence ABC...P takes value 1 if the shape overlaps the square and value 0 otherwise. For example, placing the shape on squares B, C, G, and K (Figure 3.1(e)) results in the string 0110001000100000. This string and all other strings corresponding to placing that shape on the grid belong to the language described by regular expression $0^*110^310^310^*$: first comes some number of 0's, then two 1's in a row (covering squares B and C in our example placement), then come exactly three 0's (not covering places D, E, and F), and so on. The variable number of 0's at the beginning and at the end makes the expression match any placement of the shape in the grid.



(a) Shape to place

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

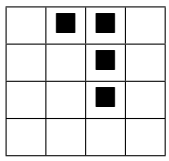
(b) Grid to place in

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

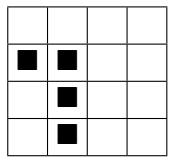
(c) Grid to place in cut into strips

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

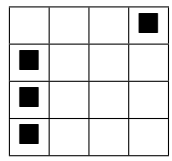
(d) Grid to place in as a sequence



(e) Possible placement of shape.



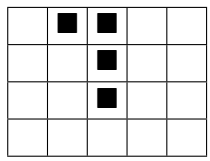
(f) Another possible placement of shape.



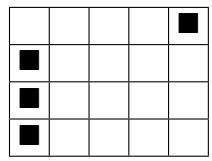
(g) Erroneous placement of shape.

A	B	C	D	**Q**
E	F	G	H	**R**
I	J	K	L	**S**
M	N	O	P	**T**

(h) Modified grid.



(i) Shape positioned on modified grid



(j) Erroneous placement of shape on modified grid (cannot place on last column)

A	B	C	D	**Q**	E	F	G	H	**R**	I	J	K	L	**S**	M	N	O	P	**T**

(k) Modified grid as a sequence.

Figure 3.1: Placing a shape in a grid.

There is one problem however, since the regular expression constructed allows "placing" the shape on squares D, E, I, and M (Figure 3.1(g)). That is, the shape may wrap around the grid. To prohibit this, we add a new dummy-column to the grid, so that it looks like the grid in Figure 3.1(h). Squares Q to T are fixed to zero, and the regular expression for the placement of the shape is modified to include the extra column, resulting in $0^*110^410^410^*$ matching, for example, the placement in Figure 3.1(i). The language of that regular expression is now precisely the strings corresponding to possible placements of the given shape. From now on, we will assume that grids have an extra column on the right.

3.3 Pentominoes

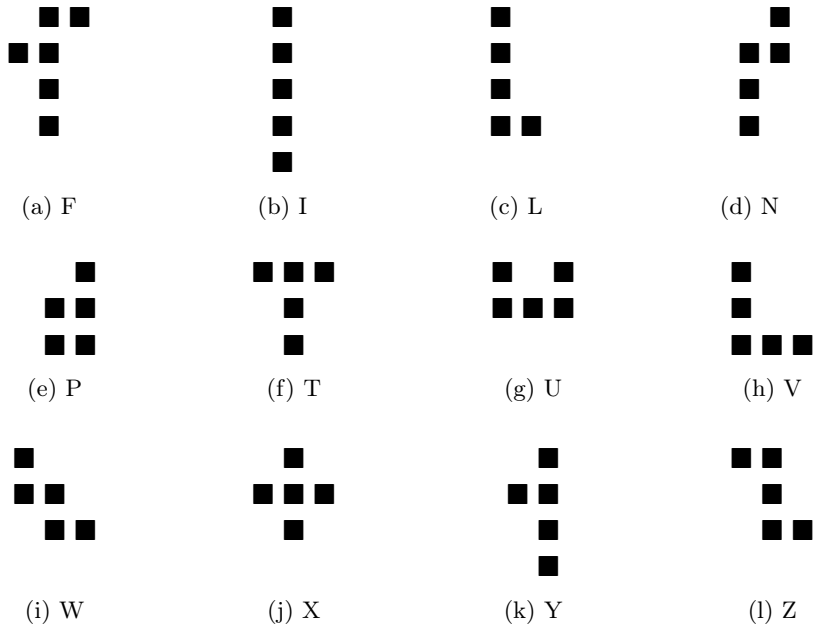
In this section a model for the classic Pentominoes problem is developed. Pentominoes are the shapes constructed from five equal-sized squares where the squares form a connected component. This leads to twelve possible shapes, as can be seen in Figure 3.2. The pieces are given standard names inspired by their shape. The pieces can be rotated and mirrored to obtain symmetrical variants.

There is a long history of using the pentomino pieces as a puzzle. For example, the pieces can be fitted together to form rectangles of sizes 6×10, 5×12, 4×15, and 3×20, as well as 8×8 with the middle 2×2 squares empty. The latter problem was solved in 1958 by Scott using a backtracking algorithm, finding all the 65 possible non-symmetrical solutions [55]. In 1965, Fletcher found the number of non-symmetrical solutions for the rectangles without empty squares, again using a backtracking algorithm [19]. These results were obtained using specialized algorithms implemented to solve these specific problems.

Other sizes of pieces can also be used, and are then called *polyominoes*. A very successful game based on polyominoes is Tetris, which is based on polyominoes composed of four squares (also called tetrominoes). In the following, we will consider placing a general set of polyominoes on a board of fixed size. The pieces to be placed can be rotated and mirrored.

To model pentominoes, the fact that more than one piece should be placed needs to be handled. Furthermore, symmetrical versions of pieces must be allowed.

We associate a distinct variable to each square on the board, whose value



(a) F (b) I (c) L (d) N

(e) P (f) T (g) U (h) V

(i) W (j) X (k) Y (l) Z

Figure 3.2: The twelve pentomino pieces

is the number of the piece which is placed over it. For each piece, we define one `regular` constraint for placing it on the board.

Rotating pieces. In difference to placing a shape as in the previous section, a pentomino piece may be rotated. To handle that, we can use disjunctions of regular expressions for all the relevant rotations. Consider the rotated versions of the piece from Figure 3.2(c) as shown in Figure 3.3. The regular expression for placing the piece shown in Figure 3.3(a) on a 5 by 5 grid is $0^*10^2 11110^*$, and for the piece in Figure 3.3(e) it is $0^*110^4 10^4 10^4 10^*$. To combine the regular expressions for the pieces we can simply use disjunction of regular expressions, arriving at the expression $(0^*110^4 10^4 10^4 10^*)|(0^*10^2 11110^*)$. There are 8 symmetries for the pieces in general. The 8 disjuncts for a particular piece might, however, contain less than 8 distinct expressions. This redundancy is removed when the automaton for the expression is computed, since it is minimized.

Using disjunctions is naturally not limited to placing symmetrical pieces, it can also be used for placing alternative shapes or alternative pieces.

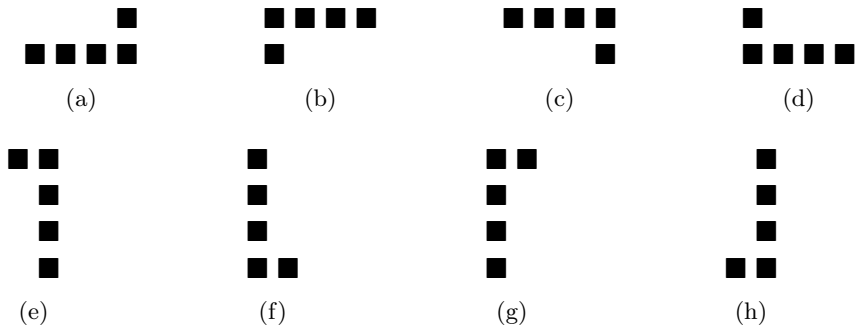


Figure 3.3: Rotations of the L-piece from Figure 3.2.

Placing several pieces. To generalize the above model to several pieces, we let the variables range from 0 to n, where n is the number of pieces to place. Given that we place three pieces, and that the above shown piece is number one, we will replace each 0-expression with the expression $\neg 1$, indicating all values other than 1. Thus, the original regular expression becomes $(\neg 1)^* 11 (\neg 1)^4 1 (\neg 1)^4 1 (\neg 1)^4 1 (\neg 1)^*$. Given that the expressions $\neg v$ can grow quite large, the number of edges in the DFAs may grow significantly. The effects of this growth, and a method for dealing with it is discussed in Section 6.1.

Additionally, the end of line marker gets its own value. This marker is used only in the places of an expression where an new-line should occur. Without a special value for the end marker we could not use the model to place non-contiguous pieces.

Removing board symmetries. Symmetry breaking is important to gain speed when searching for many (all) solutions. Puzzles of the pentomino form are often solved to find all possible non-symmetrical arrangements. The symmetries can be removed by adding a lexicographical constraint between the board and each one of its symmetrical rotations [23] and mirrorings. This adds 7 lexicographical constraints to the model for square boards, and 3 for non-square boards.

Symmetry breaking constraints can slow down searching for a single solution if the branching heuristic and the symmetry breaking is at odds. This

problem is helped by ensuring that the branching heuristic tries to find the lexicographically least solution first by ordering the values of the pieces in accordance with the heuristics choice.

3.4 Evaluation

To evaluate the model the four classic pentomino instances are tested: packing squares of sizes 20×3, 15×4, 12×5, and 10×6 with the twelve distinct pentominoes. The models are solved both for finding a first and all solutions, with and without symmetries. The branching strategy is to instantiate the variables from top to bottom, trying to place "harder" pieces first. The ordering of the pieces used is LFTWYIZNPUXV, which was chosen by hand. The experiments were run using Gecode 2.1.1 as the CP system on an Athlon 64 3500+ with 2GB of RAM. All results presented are based on 25 runs, with a deviation of less than 5%.

The results are shown in Tables 3.1 and 3.2. The results show that the problem can be solved readily using the simple model described here. The use of symmetry breaking helps a lot when searching for all solutions, and even in one case when searching for the first solution. If the ordering in the symmetry breaking is reversed, the time and number of failures for finding the first solution is increased substantially.

Table 3.1: Packing pentominoes, 1st solution. With and without symmetry breaking. Time is given in milliseconds.

	1st, `none`		1st, `sym`	
Size	failures	time	failures	time
20×3	25 129	25 484	8 697	9 408
15×4	4 700	4 478	4 700	4 498
12×5	541	553	541	542
10×6	893	1 103	893	1 114

Table 3.2: Packing pentominoes, all solutions. With and without symmetry breaking. Time is given in milliseconds.

	all, `none`			all, `sym`		
Size	solutions	failures	time	solutions	failures	time
20×3	8	35 680	34 989	2	8 740	9 471
15×4	1 472	649 068	587 144	368	238 743	229 520
12×5	4 040	2 478 035	2 390 850	1 010	788 310	810 120
10×6	9 356	5 998 165	6 760 211	2 339	1 837 711	2 206 540

Chapter 4

Propagator Implementation

While incremental propagation for global constraints is recognized to be important, little research has been devoted to how propagator-centered constraint programming systems should support incremental propagation. This chapter introduces advisors as a simple and efficient, yet widely applicable method for supporting incremental propagation in a propagator-centered setting. The chapter presents how advisors can be used for achieving different forms of incrementality and evaluates cost and benefit for several global constraints.

4.1 Introduction

Global constraints are essential in constraint programming as they are useful for modeling and crucial for efficient and powerful propagation. For many propagators implementing global constraints, incrementality is important for efficiency.

The key features to support incremental propagation are state for propagators (to store data structures for incremental propagation) and modification information (which variables have been modified and how have their domains changed). Without state, incrementality is impossible. Without modification information, the asymptotic complexity of a propagator is at least linear in the number of variables: a propagator must scan all its variables for modification.

Recall the division between variable-centered and propagator-centered propagation. Providing modification information to a propagator is straight-

forward with variable-centered propagation, and is used in systems such as Choco [36], ILOG Solver [31], and Minion [22]. This is not true for propagator-centered propagation which is for example used in CHIP [17], SICStus [14], and Gecode [20, 52]. While propagator-centered propagation typically lacks support for modification information, it is simple and has important advantages such as fix-point reasoning, priorities, and priority-based staging [52].

This chapter presents *advisors* as a simple, efficient, yet widely applicable method for supporting incremental propagation in a propagator-centered setting. The idea for advisors is not new; similar concepts called demons are used in CHIP [17] and SICStus [13]. This chapter, however, is the first attempt to define a model, to describe an implementation, and to analyze advisors.

Basic requirements and approach. For propagator-centered propagation, it is not too difficult to record for a propagator its modified variables. However, information about modified variables is often not what a propagator needs. For example, when a variable x is modified, a propagator might need to know the position of x in an array, or the node in a variable-value graph corresponding to x. That is, a propagator requires propagator-specific information.

Providing information on domain change is difficult in a propagator-centered setting: the information is specific to each propagator, in contrast to variable-centered propagation where the information is the same for all constraints. This relies on the convention that a propagator that is invoked for variable x in a variable-centered system should not change that variable. Thus, for all the propagators depending on x the information will be the same. Moreover, since most propagators do not use domain change information, the information should be computed on demand for the propagators that need it, if any. This is needed so that the overall efficiency of a system is not compromised.

Taking these issues into account, advisors are programmed for a particular propagator to support propagator-specific modification information. Like propagators, advisors are generic in that they can be used with arbitrary variable domains. Advisors are second-class citizens compared to propagators: advisors cannot propagate, they can only advise propagators in order to achieve incremental and more efficient propagation. The second-

class citizen status is a deliberate design choice: advisors are designed to be the simplest possible extension to support incremental propagation while introducing close to no overhead.

4.2 Detailed formal model

This section introduces a more detailed formal model for a propagation system than in Section 2.2. The detail is needed to reason about the way a propagator executes and how the domain of a variable changes. The setup is slightly uncommon: a propagator is a function that takes a store and a state as input, and returns a log as a sequence of tell operations and a new state. The log will describe the modifications to the variable domains, while the updated state is used for incremental propagation. These two concepts are essential for describing propagation with advisors.

Tells and logs. A *tell* $x \sim n$ describes how to update a domain, where $x \in \mathit{Var}$ and $n \in \mathbb{Z}$ and $\sim$ is one of the relation symbols $\leq, \geq, \neq$. The *store update* $s[x \sim n]$ of a store s by a tell $x \sim n$ is defined as follows: $s[x \sim n](x) = \{m \in s(x) \mid m \sim n\}$ and $s[x \sim n](y) = s(y)$ if $x \neq y$. Note that $s[t] \leq s$ for any tell t and store s. A tell t is *pruning* for a store s, if $s[t] < s$. The relation symbols in a tell captures most domain updates typically found in systems.

Propagators describe the result of propagation by a tuple of tells, called *log*. The *domain update* $s[l]$ of a store s by a log l successively applies the updates from l to s. The update $s[\langle\rangle]$ by the empty log is s itself. For a non-empty log $\langle t_1, \ldots, t_n\rangle$ with $n > 0$, the update $s[\langle t_1, \ldots, t_n\rangle]$ is defined as $(s[t_1])[\langle t_2, \ldots, t_n\rangle]$. Clearly, $s[l] \leq s$ for any log l and store s.

A log $\langle t_1, \ldots, t_n\rangle$ is *pruning* for a store s if $n = 0$, or if t_1 is pruning for s and $\langle t_2, \ldots, t_n\rangle$ is pruning for $s[t_1]$. Note that the empty log is pruning and that a pruning log can contain multiple tells for the same variable. An important property is that a pruning log will always be of finite size, since each tell will remove at least one value from one variable, and there are a finite number of values to remove.

Given a store s and a tell $t = x \sim n$, the set of values removed by performing the tell is $\Delta(s, x, t) = s(x) - s[t](x)$.

```
Propagate(P,s)
begin
    N ← P;
    while N ≠ ∅ do
        remove p from N;
        ⟨l, σ⟩ ← p(d, state[p]);
        s′ ← s; state[p] ← σ;
        foreach x ∼ n ∈ l do
            s′ ← s′[x ∼ n];
        N ← N ∪ ⋃_{x∈dis(s,s′)} prop[x];
        s ← s′;
    return s;
end
```

Algorithm 5: Propagator-centered propagation using logs

Propagators. A propagator can use state for incremental propagation where the exact details are left opaque. Here, a *propagator* is a function p that takes a store s and a state σ as input and returns a pair $\langle l, \sigma' \rangle$ of a log l and a new state σ'. The store obtained by propagation is the update $s[l]$ of s by l. It is required that l is pruning for s (capturing that a propagator only returns pruning and hence relevant tells but not necessarily a minimal log). While the set-up might look different from Section 2.2, the real difference is only in how detailed the model is. In essence, the log models the interaction between a propagator and the store.

As a simplifying assumption, the result of propagation is independent of state: for a propagator p and a store s for any two states σ_i with $p(s, \sigma_i) = \langle l_i, \sigma_i' \rangle$ $(i = 1, 2)$ it holds that $s[l_1] = s[l_2]$. Hence, the *result* of propagation $p[\![s]\!]$ is defined as $s[l]$ where $p(s, \sigma) = \langle l, \sigma' \rangle$ for an arbitrary state σ.

The properties of propagators in Section 2.2 are retained in the following way. A propagator p is *contracting*: by construction of a log, $p[\![s]\!] \leq s$ for all stores s. A propagator p must also be *monotonic*: if $s_1 \leq s_2$ then $p[\![s_1]\!] \leq p[\![s_2]\!]$ for all domains s_1 and s_2. A store s is a *fix-point* of a propagator p, if $p[\![s]\!] = s$ (that is, if $p(s, \sigma) = \langle l, \sigma' \rangle$ for states σ, σ', the log l is empty).

Propagation. Propagation is shown in Algorithm 5. It is assumed that all propagators are contained in the set P and that state$[p]$ stores a properly initialized state for each propagator $p \in P$. The difference from the propagator-centered Algorithm 1 is that the modifications to the store are

now lifted to the level of the propagation algorithm, instead of being left hidden in the propagators.

Computing the propagators to be added to N does not depend on the size of the log l. While the log can have multiple occurrences of a variable, each variable from $\mathrm{dis}\,(d, d')$ is considered only once.

4.3 Advised Propagation

Advised propagation adds advisors to the model to enable a broad and interesting range of techniques for incremental propagation while keeping the model simple. Simplicity entails in particular that capabilities of propagators are not duplicated, that the overhead for advisors is low, and that the essence of Algorithm 5 is kept. Ideally, a system with advisors should execute propagators not using advisors without any performance penalty.

The design of advisors takes two aspects into account: how an advisor gives advice to propagators (output) and what information is available to an advisor (input). Advisors are functions, like propagators are functions. From the discussion in the introduction it is clear that the input of an advisor must capture which variable has been changed by propagation and how it has been changed.

Based on the input to an advisor function, the only way an advisor can give advice is to modify propagator state and to decide whether a propagator must be propagated ("scheduled"). Modifying the state of a propagator enables the propagator to perform more efficient propagation. Deciding whether a propagator must be propagated enables the advisor to avoid useless propagation.

The model ties an advisor to a single propagator. This decision is natural: the state of a propagator should only be exposed to advisors that belong to that particular propagator. Additionally, maintaining a single propagator for an advisor simplifies implementation.

Advisors. An *advisor* a is a function that takes a store s, a tell t, and a state σ as input and returns a pair $a(s, t, \sigma) = \langle \sigma', Q \rangle$ where σ' is a state and Q a set of propagators. An advisor a gives advice to a single propagator p, written as $\mathrm{prop}[a] = p$ where p is referred to as a's propagator (not to be confused with the propagators $\mathrm{prop}[x]$ depending on a variable x). The set of propagators Q returned by a must be either empty or the singleton set

$\{p\}$. The intuition behind the set Q is that an advisor can suggest whether its propagator p requires propagation ($Q = \{p\}$) or not ($Q = \emptyset$). To ease presentation, adv$[p]$ refers to the set of advisors a such that prop$[a] = p$.

As for propagators, the model does not detail how advisors handle state: if $a(s, t, \sigma_i) = \langle \sigma_i', Q_i \rangle$ for an arbitrary store s and two states ($i = 1, 2$), then $Q_1 = Q_2$. In contrast to propagators, advisors have no own state but access to their propagators' state (an implementation most likely will decide otherwise).

Dependent advisors. Like propagators, advisors depend on variables. An advisor a, however, depends on a single variable var$(a) \in \mathit{Var}$. This restriction means that whenever an advisor a is executed, it is known that var(a) has been modified. Similar to propagators, the set of advisors adv$[x]$ depending on a variable x is: $a \in$ adv$[x]$ if and only if $x =$ var(a) (not to be confused with the advisors adv$[p]$ for a propagator p).

Variables of a propagator p and variables of its advisors are closely related. One goal with advised propagation is to make informed decisions in an advisor when a propagator must be re-executed. The idea is to trade variables on which the propagator depends for advisors that depend on these variables.

The set of *advised variables* avar$[p]$ of a propagator is defined as $\{x \in \mathit{Var} \mid \exists a \in \text{adv}[p] \,.\text{var}(a) = x\}$. For a propagator p, the set of dependent variables and advisors var$(p) \cup$ avar$[p]$ must be *sufficient* for p: if a store s is not a fix-point of p (that is, $p[\![s]\!] < s$), then for all pruning tells $x \sim n$ for s' such that $s'[x \sim n] = s$ holds, $x \in \text{var}(p)$ or $a(s, x \sim n, \sigma) = \langle \sigma', \{p\} \rangle$ for some advisor $a \in \text{adv}[x] \cap \text{adv}[p]$. This is needed so that no modifications that would lead to a propagator not being at a fix-point are lost, since that would lead to non-monotonic behavior of the propagator.

Propagation. Algorithm 6 performs advised propagation. The only difference to simple propagation is that the update by the log computed by a propagator executes advisors.

Advisors are executed for each tell t in the order of the log l. Each advisor can schedule its propagator by returning it in the set Q and potentially modify the state of its propagator. Note the difference between variables occurring in the log l and variables from dis(d, d'): if a variable x occurs multiply in l, also all advisors in adv$[x]$ are executed multiply. Variables

```
Propagate(P,s)
begin
    N ← P;
    while N ≠ ∅ do
        remove p from N;
        ⟨l, s⟩ ← p(s, state[p]);
        s′ ← s; state[p] ← s;
        foreach x ∼ n ∈ l do
            s′ ← s′[x ∼ n];
            foreach a ∈ adv[x] do
                ⟨s, Q⟩ ← a(s′, x ∼ n, state[prop[a]]);
                state[prop[a]] ← s; N ← N ∪ Q;
        N ← N ∪ ⋃_{x∈dis(s,s′)} prop[x];
        s ← s′;
    return s;
end
```

Algorithm 6: Advised propagation

in $\mathrm{dis}\,(d, d')$ are processed only once. The reason for processing the same variable multiply is to provide each tell $x \sim n$ as information to advisors.

Again, the propagation loop computes the weakest simultaneous fix-point for all propagators in P. This may be seen by considering the loop invariant: if $p \in P - N$, then d is a fix-point of p. Since the set of advised variables and dependencies of a propagator is sufficient for a propagator and an advisor always provides sufficient advice, the loop invariant holds. Hence, the result of advised propagation is as before.

The algorithm makes a rather arbitrary choice of how to provide tell information to an advisor: it first updates the store s' by $x \sim n$ and then passes the updated store $s'[x \sim n]$ together with $x \sim n$ to the advisor. It would also be possible to pass the not-yet updated store s' and $x \sim n$. This decision is discussed in more detail in Sect. 4.4.

An essential aspect of advised propagation is that it is *domain independent*: the only dependencies on the domain of the variables are the tells. All remaining aspects readily carry over to other variable domains.

The algorithm reveals the benefit of making advisors second-class citizens without propagation rights. Assume that an advisor could also perform propagation (by computing a log). Then, after propagation by an advisor, all advisors would need to be reconsidered for execution. That would leave

two options. One option is to execute advisors immediately, resulting in a recursive propagation process for advisors. The other is to organize advisors that require execution into a separate data structure. This would clearly violate our requirement of the extension to be small and to not duplicate functionality. Moreover, both approaches would have in common that it would become very difficult to provide accurate information about domain changes of modified variables.

Dynamic dependencies. One simplifying assumption in this thesis is that propagator dependencies and advised variables are static: both sets must be sufficient for all possible variable domains. Some techniques require dynamically changing dependencies, such as watched literals in constraint propagation [21]. The extension for dynamic dependencies is orthogonal to advisors, for a treatment of dynamic dependency sets see [53].

Views. Views is a technique to simplify the implementation of constraint programming systems [54]. The basic difference is that propagators operate on views of variables instead of directly on variables. A view is able to make simple transformations of its variable, for example adding a constant. Typical views used are identity views, offset views (adding a constant), scaling views (multiplying with a constant), and negation views. Given views, a propagator for $\sum_i x_i = y$ can also be used for a linear constraint with non-unit coefficients, $\sum_i a_i x_i = y$, through the use of scaling views. Views are compiled statically in the code of a system, giving the same performance as if the transformations were added directly in the code.

The formal model for a view φ for a variable x is of an injective function that transform the domain of the variable, $\varphi_x \in \mathit{Val} \to \mathit{Val}$. A family of views φ_x for all variables $x \in \mathit{Var}$ is taken as a view for a propagator. A propagator p will access the store through the view, and will produce tells that use the inverse view. In particular, the inverse view not only has to map the value the propagator removes to the corresponding value that should be removed from the store, but also the relation used needs to be mapped. Consider a tell on x that goes through a negation view φ_x adjusting the upper bound: $x < n$. The real tell to perform is actually $x > \varphi_x^-(n)$, since the lower bound for x should be adjusted. Thus views are extended to operate on the relation symbols also.

Given the above discussion, the integration of views and advisors should

be clear. Propagators need not be modified, they read variables through views, and produce tells that are translated into the real underlying domain. For advisors to match their associated propagator, they must also use the same views as the propagator. For example, consider a propagator that potentially uses a negation view (otherwise it uses an identity view) and is interested in changes to the upper bound of the variable. The propagator can use advisors with the same views as the propagator to ensure that the advisors work. For an advisor, the relation and value in the tell is translated using the view (not the inverse).

Multiple variable-advisors. While advisors as presented are connected to a single variable, it is also straightforward to extend advisors to handle more than one variable. If the advisor handles a constant number of variables, it can find out the particular variable that has changed in constant time. A typical use-case might be a view that handles two variables at the same time. The upside of using just one variable is that only one advisor needs to be created and maintained. While this does not change the actual complexity, it lowers the overhead. If the view handles more than a constant amount of variables then it can still do useful things, as discussed below.

Another typical use case is a propagator for lexicographic ordering between two vectors x and y of variables. An advisor can be connected to a pair of variables $\langle x_i, y_i \rangle$ to see if the ordering of the variables has changed. Another possibility is for a propagator $z = \max(y_1, \ldots, y_n)$, where a single advisor over all the y-variables can be used to check if anything potentially can be propagated to z. A single advisor on all the y variables can check the *possibility* of a decrease in the total upper bound of y since it suffices to inspect the delta information (regardless of which variable has been modified) and compare it to a cached upper bound. The gain in these cases is that fewer advisors are needed, thus keeping the memory overhead down. The drawback is that the constants for processing time might increase.

4.4 Implementation

This section discusses how advisors can be efficiently implemented: it details the model and assesses the basic cost and the potential benefit of advisors. Advisors is included in Gecode from version 2.0.0 [20].

Advisors. Advisors are implemented as objects. Apart from support for construction, deletion, and memory management, an advisor object maintains a pointer to its propagator object. The actual code for an advisor is implemented by a runtime-polymorphic method `advise` in the advisor's propagator. The call of `advise` corresponds to the application of an advisor in the model. Both advisor and modification information are passed as arguments to `advise`. As an advisor's propagator implements `advise`, the advisor does not require support for runtime polymorphism and hence uses less memory. The choice also gives more natural code, since an advisor needs to modify the state of the propagator.

Advisors are attached to variables in the same way as propagators are. Systems typically provide one entry per propagation event where dependent propagators are stored (corresponding to prop$[x]$ for a variable x). Typically, the propagators are organized in a suspension list, whereas in Gecode they are stored in a partitioned array. To accommodate for advisors, a variable x provides an additional entry where dependent advisors adv$[x]$ are stored. This design in particulars entails that advisors do not honor events (to be discussed below).

Logs. The log in the model describes how propagation by a propagator should modify the domain of its variables. Most systems do not implement a log but perform the update by tells immediately. This is also the approach taken in Gecode. A notable exception is SICStus Prolog, which uses a data structure similar to logs for implementing global constraints [32].

Performing updates immediately also executes advisors immediately. This differs from the model where execution of propagators and advisors is separated. In an implementation with immediate updates, the advisors of a propagator will be run while the propagator is running itself. When designing advisors and propagators this needs to be taken into account, in particular to guarantee consistent management of the propagator's state. This is often natural, since many propagators have data structures that have monotonic behavior in the changes. If the data structures of the propagator needs to be protected, then one can add a flag in the state that the advisor checks before modifying the state, thus creating a sort of critical section.

Modification information. During propagation, the domain and the tell provide information to an advisor which variable has changed and how it has

changed. This information, provided as a suitable data structure, is passed as an argument to the advise function of an advisor object.

As discussed in Sect. 4.3, there are two options: either first modify the domain and then call the advisor, or the other way round. We chose to first modify the domain as in Algorithm 6: many advisors are only interested in the domain after update and not in how the domain changed.

There is an obvious trade-off between information accuracy and its cost. The most accurate information is $\Delta(s, x, t)$, since it is the precise set of values removed. Accuracy can be costly: whenever a variable x is modified by a tell, $\Delta(x)$ must be computed regardless of whether advisors actually use the information.

As a compromise between accuracy and cost, our implementation uses the smallest interval $I(x) = \{\min \Delta(x), \ldots, \max \Delta(x)\}$ as approximation. Hence, for a store s' the interval for the pruning tell $x \leq n$ is given by $\{n+1, \ldots, \max s'(x)\}$, for $x \geq n$ it is $\{\min s'(x), \ldots, n-1\}$, and for $x \neq n$ it is $\{n\}$. For other more complicated domain operations, such as the removal of a set of arbitrary values, $\emptyset$ can be passed to signal that anything might have changed.

Propagation events. Systems typically use propagation events to characterize changes to domains by tells. For finite domain systems, common propagation events are: the domain becomes a singleton, the minimum or maximum changes, or the domain changes. Sets of dependent variables for propagators are then replaced by event sets: only when an event from a propagator's event set occurs, the propagator is considered for re-execution.

The same approach can be taken for advisors, using sets of advised events rather than sets of advised variables. In our implementation, advisors do not use propagation events for the following reasons: Events are not essential for a system where propagator execution has little overhead [52, 53]. Per event type additional memory is required for each variable. Events for advisors would increase the memory overhead for advisors even in cases where no advisors are being used. The domain change information available to an advisor subsumes events, albeit not with the same level of efficiency.

Performance assessment. Advisors come at a cost. For memory, each variable x requires an additional entry for adv$[x]$ regardless of whether advisors are used or not. If an advisor for a variable x and a propagator p is used

Table 4.1: Performance assessment: runtime

Example	`base`	`a-none`	`a-run`	`a-avoid`
`stress-exec-1`	45.38	+0.2%	+55.1%	+63.5%
`stress-exec-10`	114.93	+0.9%	+88.7%	+98.7%
`queens-n-400`	519.14	±0.0%	+1316.7%	+634.5%
`queens-s-400`	14.57	+0.7%	+28.6%	+12.2%

rather than using x as a dependency of p (that is, $x \in \mathrm{var}(p)$), additional memory for an advisor is required (this depends on the additional information an advisor stores, in our implementation the minimal overhead is 8 bytes on a 32-bit machine). For runtime, each time a variable x is modified by a tell, the tell information must be constructed and the advise function of all advisors in adv[x] must be called.

Table 4.1 shows the runtime for systems using advisors compared to a system without advisors (`base`, runtime given in milliseconds). The system `a-none` provides advisors without using them, `a-run` uses advisors that always schedule their propagators (fully replacing propagator dependencies by advised variables), whereas advisors for `a-avoid` decide whether the execution of a propagator can be avoided. All runtime are relative to `base`.

All examples have been run on a Laptop with a 2 GHz Pentium M CPU and 1024 MB main memory running Windows XP. Runtimes are the average of 25 runs, the coefficient of deviation is less than 5% for all benchmarks.

The example `stress-exec-1` posts two propagators for $x < y$ and $y > x$ with $d(x) = d(y) = \{0, \ldots, 1000000\}$, whereas `stress-exec-10` posts the same propagators ten times. The advisor for avoiding propagation (system `a-avoid`) checks by $\max d(x) < \max d(y)$ and $\min d(x) > \min d(y)$ whether its propagator is already at fix-point. `queens-n-400` uses $O(n^2)$ binary disequality propagators, whereas `queens-s-400` uses 3 `all-different` propagators to solve the 400-Queens problem.

These analytical examples clarify that the overhead of a system with advisors without using them is negligible and does not exceed 1%. Advisors for small and inexpensive propagators as in `stress-exec-*` and `queens-n-400` are too expensive, regardless of whether propagation can be avoided. Only

Table 4.2: Performance assessment: memory

Example	base	a-none	a-run	a-avoid
queens-n-400	24 656.0	±0.0%	+67.6%	+67.6%
queens-s-400	977.0	±0.0%	+5.6%	+5.6%

Table 4.3: Performance assessment: break-even

Example	base	a-none	a-avoid
bool-10	0.01	+0.2%	−16.6%
bool-100	0.09	+7.6%	−22.3%
bool-1000	1.43	+33.0%	−30.2%
bool-10000	238.23	+20.6%	−94.7%

for sufficiently large propagators (such as in queens-s-400), the overhead suggests that advisors can be beneficial. Exactly the same conclusions can be drawn from the memory overhead shown in Table 4.2, where memory is given as peak allocated memory in KB.

4.5 Using Advisors

This section demonstrates advisors for implementing incremental propagation. Central issues are to *avoid* useless propagation, to *improve* propagation efficiency, and to *simplify* propagator construction.

Boolean linear in-equations. A Boolean linear in-equation $\Sigma_i x_i \geq n$ indicates that at least n variables from x should be set to true. Whenever a traditional propagator is invoked, it has to check all remaining variables to see what and how they have changed, leading to linear run-time. Given advisors, this scan is no longer needed since the advisor knows both the variable and how it has changed. The advisor will only need to schedule

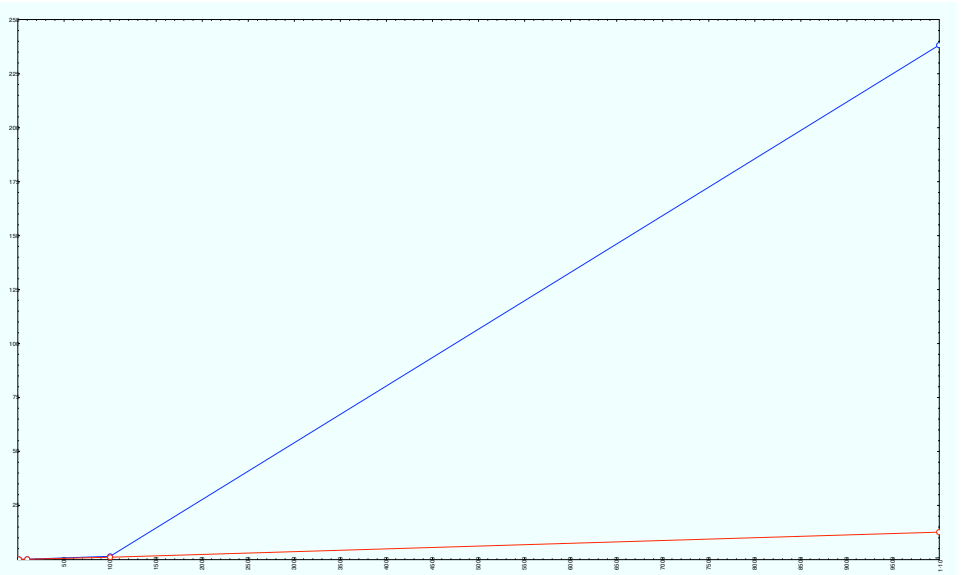


Figure 4.1: Boolean linear in-equations: Advised vs. non-advised propagation. The upper line shows propagation without advisors, while the lower shows propagation with advisors.

the propagator if there is propagation to do. This is similar to how Boolean linear in-equations are implemented using watched literals in [22].

Table 4.3 gives a first impression that advisors can actually be useful. The example `bool-`n has a single propagator propagating that the sum of $4n + 1$ Boolean variables is at least $2n$ where $2n$ variables are successively assigned to 0 and then propagated. System `a-avoid` uses $2n + 1$ advisors (constant runtime) where the other systems use a single propagator (linear runtime) with $2n + 1$ dependencies. As the number of variables increases, the benefit of advisors truly outweigh their overhead. That there is a real asymptotic difference can be seen in the graph in Figure 4.1, where the data from Table 4.3 is shown graphically.

Extensional constraints. We consider two algorithms for implementing n-ary extensional constraints, GAC-2001 [10] and GAC-Schema [9]. GAC-2001 uses a data-structure (called `last`) where the last valid support for a variable value pair is recorded. When the propagator is invoked, all variable

value pairs are checked, starting from the last valid support. In many cases, this is still a valid support. For binary constraints, GAC-2001 has optimal time-complexity. GAC-Schema is a more involved algorithm, where also the inverse mapping to `last` is maintained, i.e., what tuples involving a variable value pair is used as a support in `last`. This data structure is called a support list. Support lists need to be maintained, so that whenever a value is removed from a variable, then all the tuples including that literal used as support should be removed. If any literal looses all its supports in this way, a new support must be found.

Implementing GAC-Schema with advisors is straightforward. If a variable is modified, support for the deleted values is removed from the support lists. If a value loses a support, a new support is found. If no support can be found, the value is deleted. Advisors remove supports, while the propagator deletes values. Advisors are necessary since removing supports efficiently needs both the variable (with respect to the propagator data-structures) and the deleted values. However, advisors as well as the propagator can potentially be used to find new supports leading to eager and lazy search for supports.

Tables 4.4 and 4.5 compares runtime and number of propagator executions for different extensional propagators. The column `base` is the GAC-2001 propagator, `cheap` is a GAC-Schema propagator where the propagator searches for new supports, and `expensive` is a GAC-Schema propagator where advisors search for new supports.

Examples `rand-10-20-10-`n are random instances from the Second International CSP Solver Competition, and are originally from [40]. The `crowded-chess-`n example is a structured problem where several different chess pieces are placed on an $n \times n$ chess board. The placement of bishops and knights is modeled by two n^2-ary extensional constraints on 0/1 variables.

Tables 4.4 and 4.5 clarifies that using an incremental approach to propagate extensional constraints reduces the number of propagator executions. Using advisors to remove supports also reduces runtime. Finding new supports by advisors reduces the number of propagations the most, but is also consistently slowest. The reason is that many more supports are entered into the support-lists as new supports are searched for eagerly. In contrast, searching for a new support in the propagator is done on demand. There is also a problem with priority inversion, where expensive (exponential time) advisors are run before cheap propagators.

Table 4.4: Runtime (in ms) for extensional propagation

Example	`base`	`cheap`	`expensive`
`rand-10-20-10-0`	4 010.33	−11.4%	+164.0%
`rand-10-20-10-1`	64 103.00	−23.1%	+163.7%
`rand-10-20-10-2`	68 971.00	−16.0%	+239.5%
`rand-10-20-10-3`	7 436.80	−20.8%	+165.5%
`rand-10-20-10-4`	4 362.33	−1.6%	+168.6%
`rand-10-20-10-5`	28 009.20	−16.3%	+224.5%
`crowded-chess-5`	1.44	−1.1%	+7.4%
`crowded-chess-6`	468.29	−17.1%	+273.7%

Table 4.5: Propagation steps for extensional propagation

Example	`base`	`cheap`	`expensive`
`rand-10-20-10-0`	16 103	−24.3%	−57.9%
`rand-10-20-10-1`	290 163	−37.1%	−63.0%
`rand-10-20-10-2`	257 792	−18.3%	−56.6%
`rand-10-20-10-3`	34 046	−36.5%	−63.2%
`rand-10-20-10-4`	16 988	−29.7%	−65.4%
`rand-10-20-10-5`	84 805	−7.4%	−53.8%
`crowded-chess-5`	586	+0.7%	+0.5%
`crowded-chess-6`	2 720	−2.7%	−3.1%

As for memory, GAC-Schema will naturally use more memory than GAC-2001 since it uses an additional large data structure. Given a maximum domain size of d and n variables, the memory usage will be in $O(nd)$ for GAC-2001 vs. in $O(n^2d)$ for GAC-Schema. For the random problems, the memory overhead is around 5 to 6 times on the whole problem.

Regular. The algorithm used in our experiments deviates slightly from both variants presented in [46]: it is less incremental in that it rescans all support information for an entire variable, if one of the predecessor or successor states for a variable is not any longer reachable.

Advisors for regular store the index of the variable in the variable sequence. When an advisor is executed, it updates the supported values taking the information on removed values into account. If a predecessor or a successor state changes reachability after values have been updated, the advisor can avoid scheduling the propagator. This can potentially reduce the number of propagator invocations. Besides improving propagator execution, advisors lead to a considerably simpler architecture of the propagator: advisors are concerned with how supported values are updated, while the propagator is concerned with analyzing reachability of states and potentially telling which variables have lost support.

Tables 4.6 and 4.7 compares runtime and number of propagator executions not using advisors (`base`), using advisors but ignoring domain change information (`advise`), and using advisors and domain change information (`domain`). The memory requirements are the same for all examples. The `nonogram` example uses regular over 0/1 variables to solve a 25×25 nonogram puzzle, and the `pentominoes-*` example uses regular to place irregularly shaped tiles into a rectangle (8×8 with 10 tiles, 10×6 with 12 tiles) in the same manner as described in Section 3.

The advisor-based propagators reduce the number of propagation steps by half in case there is little propagation (propagation for `nonogram` is rather strong). But the reduction in propagation steps does not translate directly into a reduction in runtime: executing the regular propagator in vain is cheap. With larger examples a bigger improvement in runtime can be expected, suggested by the improvement for `placement-2` compared to `placement-1`.

Table 4.6: Runtime (in ms) for regular

Example	base	advise	domain
nonogram	803.13	+11.6%	+11.9%
placement-1	214.35	±0.0%	−0.5%
placement-2	7 487.81	−4.4%	−4.8%

Table 4.7: Propagation steps for regular

Example	base	advise	domain
nonogram	122 778	+3.1%	+3.1%
placement-1	2 616	−44.8%	−44.8%
placement-2	91 495	−50.4%	−50.4%

All-different. The propagator used for domain-consistent `all-different` follows [50]. The key to making it incremental is how to compute a maximal matching in the variable-value graph: only if a matching edge (corresponding to a value) for a variable x is removed, a new matching edge must be computed for x. An observation by Quimper and Walsh [47] can be used to avoid propagation: if a variable domain changes, but the number of values left still exceeds the number of variables of the propagator, no propagation is possible.

Tables 4.8 and 4.9 shows the runtime and number of propagator executions for examples using the domain-consistent `all-different` constraint. `base` uses no advisors, for `avoid` advisors use the above observation to check whether the propagator can propagate, for `advise` advisors maintain the matching and use the observation, and for `domain` advisors maintain the matching by relying entirely on domain change information. For `domain`, the observation is not used to simplify matching maintenance by advisors. `golomb-10` finds an optimal Golomb ruler of size 10, `graph-color` colors a graph with 200 nodes based on its cliques, and `queens-s-400` is as above.

While the number of propagator invocations decreases, runtime never de-

Table 4.8: Runtime (in ms) for all-different

Example	base	avoid	advise	domain
golomb-10	1 301.80	+5.6%	+12.9%	+11.2%
graph-color	191.90	+1.7%	+3.1%	+4.9%
queens-s-400	3 112.13	−0.1%	+27.4%	+23.3%

Table 4.9: Propagation steps for all-different

Example	base	avoid	advise	domain
golomb-10	3 359 720	±0.0%	−18.6%	−18.6%
graph-color	150 433	−3.4%	−8.1%	−7.3%
queens-s-400	2 397	−0.1%	−0.3%	−0.3%

creases. Using the observation alone is not beneficial as it does not outweigh the overhead of advisors. The considerable reduction in propagator executions for `advise` and `domain` is due to early detection of failure: advisors fail to find a matching without executing their propagator. The increase in runtime is not surprising: edges are matched eagerly on each advisor invocation. This is wasteful as further propagation can remove the newly computed matching edge again before the propagator runs. Hence, it is beneficial to wait until the propagator actually runs before reconstructing a matching. Another problem with eager matching is similar to the observations for extensional constraints: prioritizing matching by advisors over cheaper propagators leads to priority inversion between cheap propagators and expensive advisors.

Advisors again lead to an appealing separation of concerns, as matching becomes an orthogonal issue. However, the examples clarify another essential aspect of incremental propagation: even if a propagator does not use advisors, it can perform incremental propagation (such as matching incrementally). And for some propagators, it can be important to defer computation until perfect information about all variables is available when the

propagators is actually run. Being too eager by using advisors can be wasteful.

Summary. The above experiments can be summarized as follows. Advisors are essential to improve asymptotic complexity for some propagators (in particular for propagators with sub-linear complexity, such as Boolean or general linear equations [27]). The optimal complexity can not be achieved in a purely propagator-centered set-up. Advisors help achieving a good factorization of concerns for implementing propagators. However, the effort spent by an advisor must comply with priorities and must not be too eager. Unfortunately, substantial efficiency improvements might only be possible for propagators with many variables.

4.6 Further possibilities

While advisors have been developed here for collecting and processing information for propagators, they can also be used to give information interactively to a user. Consider the case of using a constraint system for interactive configuration, where fast response times and incremental updates of variable domains is very important. Typically, the propagation process will be relatively time-consuming, since strong propagation is desired. By connecting an advisor to each displayed variable, it is possible to observe the removal of values in real-time. There are many possible application areas for this, including configuration, puzzle design, and implementing debugging support for constraint programs.

4.7 Conclusions

This chapter has shown how to add advisors to a propagator-centered propagation system for supporting more efficient propagation. Advisors are simple and do not duplicate functionality from propagators: no propagation is done and execution is immediate. In particular, advisors satisfy the key requirement of not slowing down propagation when they are not used. That makes advisors a viable approach also for other propagator-centered constraint programming systems.

Advisors can be used to increase the efficiency, in particular improving asymptotic complexity, and achieving a better factorization of concerns in

the implementation of propagators (relying on the fact that advisors are programmable). Furthermore, two important issues have been clarified First, advisors must comply with priorities in a propagator-centered approach with priorities. Second, for some propagators it is more important that an incremental algorithm is used rather than running the algorithm eagerly on variable change.

Advisors, like propagators, are generic. It can be expected that for variable domains with expensive domain operations (such as finite sets or graph variables[18]), the domain change information provided to an advisor can be more useful than for finite domain propagators. Adapting advisors for a particular variable domain only needs to define which domain change information is passed to an advisor by a tell. In Gecode from version 2.0.0, advisors are also implemented for set variables.

Chapter 5

Constraint Representation

A *generic* global constraint is a global constraint that is capable of expressing any other constraint on a finite number of variables over finite domains. While global constraints with dedicated propagators such as `all-different` are to be preferred, these are not always able to express all constraints of a model. Depending on the model the usual fall-back is to use a decomposition into a set of simple constraints or one or more tables of tuples to express the remaining constraints. The first of these options usually results in less propagation. The latter option involving a table of tuples is the most basic of generic global constraints. This chapter explores the use of Multivalued Decision Diagrams (MDD) as a data-structure for representing generic global constraints.

In Section 5.1, the MDD representation is described. In Section 5.2 approaches that relate to the approach taken in this paper are summarized. The following Section presents the MDD constraint showing how to construct and propagate it. Section 5.4 discusses entailment detection. In Section 5.5 Cartesian Product Tables are discussed. It is shown how these can be constructed by using MDD construction as an intermediate step. This is followed by Section 5.6 which reports empirical results for all the techniques discuss in the paper. The final Section 5.7 give conclusions and discuss future work.

5.1 Decision diagrams

A *decision diagram* (DD) is a tuple $DD = \langle V, v, E \rangle$, where V is a set of vertices including the terminal vertices $\bot$ and $\top$, $v \in V \rightarrow Var$ is a function from vertices to variables, $E \in \mathcal{P}(V \times Val \times V)$ is a set of marked edges, and $\langle V, E \rangle$ forms a directed, rooted, acyclic graph where all paths from the root node reaches either $\bot$ or $\top$, and where the path does not contain two nodes u_1, u_2 for which $v(u_1) = v(u_2)$. A DD is *ordered* if there exists an ordering $<_v$ on the variables such that for all edges (u_1, u_2) it is the case that $v(u_1) <_v v(u_2)$. Any edge (u_1, u_2) for which there exists a variable $x \in Var$ such that $v(u_1) <_v x <_u v(u_2)$ is said to *skip* x and is called a *long edge*. A DD is said to be *deterministic* if the out-going edges of each node have distinct markings. As is common, DDs will be assumed to be deterministic. A DD is *reduced* if there exists no two nodes with identically labelled outgoing edges leading to the same nodes, and further that no node exists with outgoing edges marked with all possible values and leading to the same child node. Reduced ordered deterministic decision diagrams are canonical (given a fixed ordering), which is important in many applications. A path from the root to $\top$ gives an assignment of values to variables for which the DD is true. If a variable is not assigned a value on a path, it is a "don't care" variable, and can be assigned any value. Finally, if the set of values is restricted to 0 and 1 the DD is a binary decision diagram (BDD), otherwise it is a multi-valued decision diagram (MDD). The most common kind of decision diagrams are Reduced Ordered Binary Decision Diagrams (ROBDDs) [12]. There exist many variations of the basic DDs discussed above as well as several other related graph based representations (see for example [16] and [42]).

5.2 Related work

The concept of graph-based representations of constraints is not new. For example [2] derives automata from constraint signatures and obtains propagation algorithm based on these automata and [39, 28] uses BDDs for implementing complete set variable domains and propagators. Some previous representations in particular relate closely to the MDD proposed in this paper. The ad-hoc Boolean constraint from [15] makes use of a Binary Decision Diagram and describes a partially incremental propagation algorithm. The

regular constraint introduced in [46], makes use of an unfolded Deterministic Finite Automata (DFA), resulting in a representation that closely resembles an MDD but is not fully reduced. We will in our empirical comparison verify that this additional reduction can provide a significant reduction in the propagation time. Finally, the Case DAG available in SICStus Prolog [32] also resembles an MDD, but details of the construction and propagation algorithm are not available.

5.3 The Multivalued Decision Diagram constraint

In this section we will introduce the MDD representation of a constraint and discuss how to construct and propagate it.

5.3.1 Construction

Binary Decision Diagrams are usually constructed from logical expression. Each atomic logical expression is represented as a small BDD and the APPLY operator [12] is then used to conjoin these BDDs into a single BDD. For implementing MDDs, it is common to use a group of Boolean variables to represent a single domain variable and then perform compilation as mentioned above using a variable order that keeps the Boolean variables corresponding to the same domain variable together. Given such a BDD, the corresponding MDD can be built in linear time in the size of the resulting MDD.

A special case occurs when the input is a table of tuples, in this case the MDD can be built directly by first constructing the decision tree corresponding to the tuples under some variable ordering and then reducing it. This takes linear time in the size of table. Another special case is input in the form of a regular language. In this case the technique from [46] can be used to obtain a layered graph that can then be converted into a more succinct MDD in time proportional to the unfolded DFA.

5.3.2 MDD reduction subsumes DFA minimization

Performing MDD reduction results in a more succinct data structure than minimizing a DFA and then unfolding it as described in [46]. It is easy to see that DFA minimization will not yield a more succinct representation since a reduced MDD is unique and minimal with respect to the fixed variable order. That MDD reduction is superior is due to the fact that DFA minimization

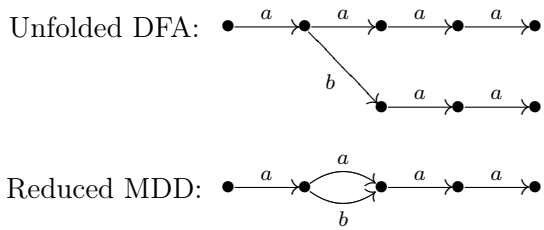


Figure 5.1: Graph for recognizing strings from $a^*|a(baa^*)^*$ of length 4 using the unfolded minimized DFA (top) and the reduced MDD (bottom).

minimizes prior to the unfolding of the DFA and therefore with respect to the whole language represented by the DFA (and not just strings of length n), and that the state structure is maintained over all layers. A small toy example showing the MDD to be more succinct is shown in Figure 5.1. In practice the difference between unfolded DFAs and reduced MDDs can be very large, as we will see in Section 5.6.

5.3.3 Propagation

As noted previously the layered graph used as representation in the propagation algorithm for regular [46] is closely related to an MDD, albeit less succinct. For this reason it is relatively straightforward to adapt the incremental propagation algorithm for the regular constraint to work on the MDD. The existence of long edges in the MDD is however a problem for that algorithm. To avoid changing the algorithm, all long edges can be replaced by paths accepting all values for the missing layers. Unfortunately, this introduces very many new edges into the graph, and thus slowing down propagation. By allowing *wild cards* as edge markers (matching any value), the number of additional edges required is kept down. Note that such a wild-card edge gives support to all values in the domain, and therefore there would exist a number of supports super-linear in the number of edges in the MDD. To keep the complexity down, we keep separate track of the count of wild-card edges in each layer. Any layer in which a wild-card edge still exists allows all values in the corresponding domain. The resulting propagation algorithm guarantees an amortized complexity of $O(|E|)$ over any path in

the search tree.

As an alternative to adapting the above incremental algorithm, the partially incremental propagation algorithm of [15] can also be used with only small adjustments. This algorithm is basically a standard DFS computing the supported values. When it examines a node u, then if it has already determined that at least one outgoing edge is valid (and thus the node), and all possible support has already been found for $v(u)$ and later variables, then it is not necessary to examine the remaining outgoing edges and their substructures for supports. However, this technique is likely to translate poorly into larger domains, as the scenario of all possible supports being found for a set of variables is much less likely. Therefore we will in our empirical results be making use of the previously described incremental algorithm.

5.3.4 Propagation complexity

As indicated above, regardless of the choice of algorithm, the critical parameter that determines the complexity of the propagation algorithm is the size of the MDD that must be propagated. This is also why we can expect the more succinct MDD representation to outperform the regular constraint. We will verify this behavior empirically in Section 5.6. Since an ROMDD is canonical with respect to the ordering used, the potential for optimizations lies in finding a good ordering. The difference in size can be exponential, and finding the best ordering is NP-complete. In practice simple greedy techniques are used to obtain good orderings. These techniques involve either interchanging nearby layers within a certain window size or moving a single layer through all possible positions (known as sifting) [49]. We will not go into more detail, but instead view the available software packages as black boxes that have automated heuristics for optimizing ROMDDs.

5.4 Entailment

When a propagator is entailed, it can safely be removed from the system. This is done to avoid executing a propagator in vain, since we know that the entailed propagator will not do any propagation.

Unfortunately, entailment detection for general constraints is a co-NP complete problem, which makes it intractable in practice. In this section we introduce a simple entailment detection technique that can be supported for any constraint that allows the number of solutions of the constraint to be

counted with respect to a given store. We then introduce two techniques unique to the MDD representation which rely on structural properties to detect entailment.

5.4.1 Solution counting

Counting the number of solutions to a constraint can be used for detecting entailment. Given a constraint c over variables $\langle x_1, \ldots, x_n \rangle$ and a store S, the number of solutions to the constraint valid in the store is given by $\#sol(c, S) = |Cons\,(S, \mathrm{var}(c)) \cap c|$. The key observation is that if the store restricted to the variables of for the constraint (call it $S_{\mathrm{var}(c)}$) admits the same number of solutions as are valid for c in S, the constraint is entailed:

$$\#sol(c, S) = |sol(S_{\mathrm{var}(c)})| \Rightarrow Cons\,(S, \mathrm{var}(c)) \subseteq c$$

That MDDs support solution counting in linear time is well-known, and the same algorithm has also been used on the closely related representation used in the regular constraints [59]. The solutions to an MDD can be counted recursively: for each node the solution count is determined by multiplying the number of solutions obtained by following each outgoing edge, remembering to adjust for long edges. Hence we can do entailment detection on an MDD in linear time by just computing the number of solutions allowed under the current domains. The drawback of this technique is that it will always use linear time in the MDD size.

5.4.2 Structural detection

A decision diagram that forms a path is describes a store. Using this observation, a simple check can be performed to detect entailment. If the MDD is reduced but with all layers kept, this is a strict condition since that is the minimal representation. Propagation during search might violate the reduced property, but it can be restored at any time in linear time. While this may yield additional benefits besides just entailment detection, it requires linear time in the best case just like the approach based on solution counting. However, even if the decision diagram is not reduced during search, the check can still be used as a heuristic.

Full entailment detection can be done efficiently even without requiring that the MDD is reduced. Iff the MDD is entailed, every node in a layer will have the same set of values on its outgoing edges (ignoring those edges

marked with values inconsistent with the store). This can be seen by noting that if all nodes in each layer allow the same values to be chosen, then the set of solutions to the MDD is expressible as a single Cartesian product, exactly the object that describes a domain stores solution set.

The two above techniques share the advantage that the analysis can be aborted as soon as the structural property used is violated. This gives a sub-linear average complexity. Furthermore, since any layer satisfying the above two properties at some node in the search tree will continue to do so in all descendant search nodes, we can easily ensure linear amortized time over any path in the search tree by always resuming the entailment check from the node which caused the entailment check to abort the last time. In practice it is beneficial to do the structural analysis bottom up, since it is common that the variable order in the MDD corresponds to the order in which variables are branched on. Early termination of the analysis is thus more likely.

5.5 Cartesian product tables

A *Cartesian Product Table* (CPT) is a generalization of a set of tuples where a set of stores on the constraint variables is used for specification instead. To ease presentation, CPTs will be written using tuples as the stores, and identifying singleton sets with values. As an example, the constraint $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,1\rangle\}$ represents implication. The minimal CPT representation for this constraint is $\{\langle 0,\{0,1\}\rangle, \langle 1,1\rangle\}$. Cartesian product tables have been introduced as a data-structure for array based logic [44] for specifying configuration problems. Using CPTs in constraint programming has been done as a compression schema [35], where GAC-Schema was generalized to use a CPT as specification.

5.5.1 MDDs vs. CPTs

Since both MDDs and CPTs are more succinct constraint representations than a set of tuples, it is interesting to see which is the more succinct one. If we assume that each tuple is only represented once in a CPT, then it follows that the corresponding MDD is smaller. Consider a decision tree with fixed variable ordering and wild-card edges obtained from a CPT. Reducing this into an MDD will always maintain or decrease the size. However if a tuple is allowed to be stored multiple times, then the corresponding MDD might no

longer be the more succinct. In Example 5.1, the minimal CPT is smaller than the MDD for the same constraint (seen in Figure 5.2). The reason is that a CPT can exploit don't care values on a per store basis, as opposed to the deterministic MDD where a choice for a value at layer i affect the choices at all subsequent layers.

Example 5.1. *Consider the constraint that is specified by the following tuple set:*

$$\{\langle 00\rangle, \langle 01\rangle, \langle 02\rangle, \langle 10\rangle, \langle 11\rangle, \langle 13\rangle, \langle 20\rangle, \langle 22\rangle, \langle 31\rangle, \langle 33\rangle\}$$

This constraint has the following minimal CPT:

$$\{\langle\{01\}\{01\}\rangle, \langle\{02\}\{02\}\rangle, \langle\{13\}\{13\}\rangle\}$$

This CPT contains the tuples $\langle 00\rangle$ *and* $\langle 11\rangle$ *twice each. If no sharing is allowed, a CPT with four stores is needed.* □

Even in cases where the MDD is smaller, CPTs can sometimes still be of interest. This is because the overhead of propagating a CPT is somewhat lower than for an MDD. Hence in cases where the CPTs are not too much bigger than the MDD they are still a valid alternative.

5.5.2 Constructing CPTs using MDDs

There are a number of published algorithms for constructing CPTs. All of these requires that the input is a set of tuples. The earliest ones are in the context of array-based logic, where a column elimination method is used to compress normal tuple sets [44, 34]. The idea is to remove variables one by one, and to collapse all the tuple that can be collapsed for each variable

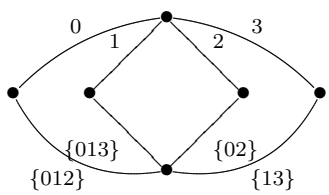


Figure 5.2: Decision diagram of constraint from Example 5.1

removed. This can be efficiently implemented using universal hashing [34]. Katsirelos and Walsh use decision trees to find small CPTs [35] and show the resulting propagation of the CPT to be faster than propagating the uncompressed tables. For all the above techniques the resulting CPTs only stores each tuple once and thus cannot be more succinct than MDDs.

However, a CPT can also easily be derived from an MDD: each node path from the root to the terminal in the MDD correspond to a Cartesian product formed by including all the edge values occurring on the node path. An important observation is that this allows the CPT to benefit from variable reordering performed on the MDD. For example the MDD in Figure 5.2 would result in the CPT $\{\langle\{0\}\{012\}\rangle, \langle\{1\}\{013\}\rangle, \langle\{2\}\{02\}\rangle, \langle\{3\}\{13\}\rangle\}$. The practical evaluation in Section 5.6 shows that this approach is competitive with the technique from [35]. An additional advantage of constructing CPTs from MDDs is that the full set of tuples is no longer required as input, in contrast to all previously suggested approaches. Given an MDD for the constraint in question, the equivalent CPT can be produced in time liner in the size of the output. One such example are the Pentomino problems evaluated in the next section: the constraints from the instances often admit up to 10^{50} solutions.

5.6 Evaluation

In this section we present an empirical evaluation of some of the introduced techniques. All experiments were run using BuDDy 2.4 as the BDD-package and Gecode 2.0.1 as the CP system on an Athlon 64 3500+ with 2GB of RAM.

CPT construction using MDDs. The instances investigated are the ones used in [35] for evaluation from the 2005 Solver Competition. The results are presented in Table 5.1. The random example is the `rand-3-20-20-1` instance, other random instances showed the same kind of behavior. For each row, the number of tables, the average size, the average size of the CPT using DD construction, with re-ordering also, and the best size from [35], is given. From the results, we can see that using decision diagrams to find CPTs is a viable method. While the DD method does not dominate, in the instances for which it is outperformed it is so by a factor of less than two, while it on the better instances improves upon [35] with a factor of more than 8. Vari-

Table 5.1: CPT construction using MDDs

Instance	Tables	Avg size	CPT from DD	Re-ordered	K&W
`cril-0`	27	1482.11	464.82	459.90	291.23
`cril-6`	9	281.33	28.77	27.06	142.65
`cril-7`	9	292.67	34.77	31.66	180.31
`rand`	60	2944.00	399.95	376.70	211.37
`renault`	89	2182.76	278.23	134.88	1133.28
`wordlist`	6	8584.33	5662.20	5448.08	-

able reordering only gives small improvements on most instances, most likely because the variable ordering is already good or there is no especially good variable ordering. In the `renault` instance variable reordering is important, since it halfs the size of the resulting CPT.

Reduction. The effect of MDD reduction is tested using the Nonogram problem (problem 12 in CSPLib). The line constraints use extensional propagation over the set of solutions. The instances are tested using tuple sets (`gac2001` and `gac-schema`), minimized DFAs from the disjunction of the solutions (`dfa`), and reduced version of the same (`mdd`). Instances are from the Gecode nonogram example. For each version, the time (in milliseconds, including construction time), and size of the specification (number of tuples/nodes) are given in Tables 5.2 and 5.3. We can see that using the reduced decision diagram results in a significantly smaller specification than for the DFA. Looking at the time, we can see that this smaller specification also translates into faster propagation. Overall we can also see that MDDs gives the most stable solution time.

Entailment detection. To test the effect of entailment detection, Pentomino placement problems specified using regular expressions are run using the incremental propagation algorithm described earlier. Versions using no entailment detection (`base`), entailment detection using solution counting (`count`), using path detection (`path`), and using structural entailment detection (`struct`) are compared. For each version, the total number of propagator invocations and the total time (in milliseconds) is presented in Tables 5.4 and 5.5. Instances are from the Gecode Pentomino example.

Table 5.2: MDD Reduction for Nonogram constraints, size.

Instance	Failures	gac2001 size	gac-schema size	dfa size	mdd size
nonogram-0	0	12.17	12.17	242	53
nonogram-3	22	22.6	22.6	334	59
nonogram-5	3983	41.5	41.5	753	59
nonogram-7	0	374.5	374.5	2239	242

Table 5.3: MDD Reduction for Nonogram constraints, time

Instance	Failures	gac2001 time	gac-schema time	dfa time	mdd time
nonogram-0	0	2.5	3.5	5.0	4.5
nonogram-3	22	25.0	24.0	32.5	18.5
nonogram-5	3983	4837.5	5225.5	4480.0	2480.0
nonogram-7	0	595.0	592.5	2147.5	745.5

Suprisingly, the results show that the solution counting method does not detect all entailments. This is due to overflow problems of counting the number of solutions. While using an arbitrary-precision package would solve the problem, it would also be much more expensive. Another reason why solution counting is so slow, is that each time the whole decision diagram must be visited, while the structural methods can abort early very often. There is no difference in the amount of entailments detected by the `path` method and the `struct` method. This is due to the particular problem investigated, where all entailed constraints will have the path structure, even though no reduction is performed. While the `path` method is better on these examples than the `struct` method, the latters greater potential for detecting entailments combined with its relatively low overhead makes it an interesting choice.

Table 5.4: Entailment detection, steps

Instance	base	count	path	struct
pentomino-1	1509	1483	1400	1400
pentomino-2	111	111	111	111
pentomino-4	45477	45070	40940	40940
pentomino-5	2167	2163	2084	2084
pentomino-6	289841	288897	275229	275229

Table 5.5: Entailment detection, time

Instance	base	count	path	struct
pentomino-1	223.6	271.6	214.4	221.2
pentomino-2	18.8	20.0	18.4	18.4
pentomino-4	5830	6886	5053	5250
pentomino-5	353.3	403.3	333.3	343.3
pentomino-6	50742	62680	47176	48985

5.7 Conclusions and future work

This chapter has introduced the MDD global constraint and shown how to propagate it by adapting the algorithm from [46]. Empirical results have shown that the additional reduction offered by this representation has a significant impact on the propagation complexity compared to using the regular constraint. It was also shown how to efficiently perform entailment detection on the MDD constraint and the empirical results indicated that the heuristic structural detection was most efficient in practice. Finally, a technique for constructing CPTs using MDDs as an intermediate step was developed. An empirical comparison with previous techniques showed it to be a competitive technique that in some cases outperformed the other techniques by a large margin.

In future work it would be interesting to examine more succinct variations of the MDD, for example using AOMDDs or non-deterministic decision diagrams as the representation. If heuristics could be developed for constructing non-deterministic diagrams this would also benefit the construction of

CPTs using the technique described in this paper. In general, the use of non-deterministic data-structures is very interesting for finding succinct representations, but also usually very hard. As an example, even for a finite language finding the minimal NFA is at least NP-hard, and the best known upper bound is Σ_2^P (the second level of the Polynomial Hierarchy) [26]. This means that an exact algorithm is most probably out of reach for any practical examples. Given that we need to use heuristics and approximations, the approximation complexity is of interest. Unfortunately, a general NFA can not be minimized within an approximation factor of $O(n)$ efficiently, unless P=PSPACE [25]. This implies that efficient heuristics should be the main focus.

Chapter 6

Abstraction

In previous chapters the efficiency of a single propagator has been discussed. However, looking at a complete problem, there might be interesting possibilities for increasing the efficiency of propagation. We will start with a motivating example first, improving the efficiency of the Pentomino problem, and then give some idea about how it can be formalized. This chapter will not give a full solution, it is intended as an idea about where to find further possibilities for increasing the performance of propagation.

6.1 Example: Revisiting Pentominoes

Consider again the problem of placing pentominoes on a grid, as explained in Chapter 3. While the model is clear and follows the original statement closely, it is not as efficient as it can be. The main problem is that each constraint talks about every value in every variable (through the use of complemented values), even though it constrains only one of the values. This leads to repeated uses of subexpressions of the form $n_1|\cdots|n_{i-1}|n_{i+1}|\ldots|n_k$ for denoting all values except n_i.

To improve the performance, it is possible to abstract away the dependence on the placement of the other values, and only focus on a specific value n_i for a constraint. Assume that the variables x from Chapter 3 describe a board of size mn, with k pieces to place. In order to abstract the values, we define additional 0-1 variables y_{ij}:

$$y_{ij} = 1 \Leftrightarrow x_i = j, \quad 0 \leq i < nm,\ 1 \leq j \leq k$$

Table 6.1: Packing pentominoes, standard versus 0-1 model, 1st solution

Size	failures	`original` time	`0-1` time
20×3	25 129	25 484	6 695
15×4	4 700	4 478	1 015
12×5	541	553	126
10×6	893	1 103	237

Table 6.2: Packing pentominoes, standard versus 0-1 model, all solutions

Size	failures	solutions	`original` time	`0-1` time
20×3	35 680	8	34 989	9 320
15×4	649 068	1 472	587 144	147 210
12×5	2 478 035	4 040	2 390 850	576 270
10×6	5 998 165	9 356	6 706 211	1 517 150

To connect the x and y variables, a channeling propagator can be used for each x_i and corresponding $\langle y_{i1}, \ldots, y_{ik} \rangle$. The `regular` constraints can now be defined using 0-1 expressions on the y variables for the right piece instead. This extended model has many more propagators than the original model, $mn + k$ compared to just k, but the extra propagators are very cheap to propagate. We gain in efficiency by reducing the number of times the `regular` constraints have to propagate, as well as by reducing the size of the layered graphs used.

As can be seen in Tables 6.1 and 6.2, the abstracted 0-1 model improves the solutions times significantly.

6.2 Abstraction variants

There are several potential ways that problems can be abstracted. In the previous Section, the original propagators were removed, since the new propagators captured the full semantics of the original problem. Using the idea of staging [52, 53], cheap but incomplete propagators can be added to a prob-

lem to speed up the process of finding a fix-point by executing the expensive propagators fewer times.

The technique used can also vary. In the previous Section, sets of values for variables were merged to represent higher level concepts. Other transformations are also possible. For example, the set of solutions that a propagator allows could be extended (e.g., dropping some side constraints from a global constraint). Another possibility is to choose a subset of the variables to propagate on.

6.3 Knapsack

As an example of an abstraction that adds new cheap propagators, merges values, and extends the set of solutions, consider knapsack problems. There is a complete pseudo-polynomial propagation algorithm for knapsack constraints [56] that uses the MDD unfolding of the problem. This algorithm is based on the standard pseudo-polynomial dynamic programming algorithm for knapsack. Unfortunately, this can take quite some time if the items have large sizes, as can be quite common. When solving knapsack problems, it is quite common to use a heuristic that divides the cost of all elements with a constant. While the costs are still ordered with respect to each other, the complexity of the dynamic programming algorithm goes down since the range is compressed. The drawback is that the algorithm is no longer exact, since dividing the costs may merge nearby values so that they become indistinguishable.

To use the heuristic in a constraint programming setting, it is necessary to keep some propagator that ensure the final solution is in fact a solution to the original problem. This can be the original, pseudo polynomial propagator. If the size of the costs in the original knapsack constraint are too large, then a normal bounds propagator can be used as the checking propagator. The choice of the checking propagator determines if the problem will have the same amount of propagation (and thus the same search tree), or if the search tree might be larger.

6.4 Abstraction in a system

Finding and applying abstractions can quite possibly be automated, since we are interested in finding instances of pre-defined patterns. For example, given

that a `regular` constraint is in the problem, the sets of values used in state-transitions can be analyzed to find potential abstractions. It is preferable to do this on the whole problem at once, since for an example such as the one in Section 6.1, using global propagators for the channeling of information is useful to keep the number of added propagators down.

There are several challenges in implementing a system for abstraction. In the following list, a few are given.

- The problem must be inspected. Should this be done in a modeling layer, or should it be done directly in the Gecode system? While doing the inspection directly in Gecode is harder, the potential might be higher since it could even be used dynamically. On the other hand, a model in a modelling layer might have much more structure than the concrete representation in Gecode.

- For each potential abstraction, the analysis must be implemented. Should these analysises operate on individual constraints, on groups of constraints, or even on the whole problem?

- Deciding when an abstraction is beneficial can potentially be quite hard. For example, when are the costs for a knapsack constraint large enough for an abstraction to be worthwhile.

- If several propagators of the same kind are used in the system, but some are obviously cheaper than others, how should the priorities of the propagators be changed? For example, given an abstracted and an unabstracted propagator for knapsack, we want to ensure that the abstracted propagator has a higher priority than the unabstracted, even though they have the same type.

Chapter 7

Conclusions and Further Work

This thesis has presented new ways to improve the efficiency of constraint propagation. Three main avenues has been explored: propagator implementation, constraint representation, and model abstraction.

To close the complexity gap between propagator-centered and variable centered constraint programming systems, advisors have been added (Chapter 4). Using advisors, it is possible to program the change information that a propagator receives directly, and to specialize it for each propagator. For propagators that do not need detailed change information (comprising most of the propagators in a system) the addition does not cost any significant amount of time or memory. While this addition has existed before this thesis, often called Demons, this is the first model and evaluation of it.

Given that the propagators in a system is implemented in the most efficient way, potentially using advisors, we must choose the right representation to use for a particular problem. While the choice of propagators is sometimes clear, the choice of representation for generic constraints is less clear. In Chapter 5, the use of Multivalued Decision Diagrams (MDDs) is proposed for generic constraints. MDDs have many interesting properties as representations of constraints, and are easy to propagate using an adaptation of the algorithm for `regular` constraints.

Given that the propagators are implemented in the most efficient way, and that the best propagators have been chosen, it is natural to look at a higher level on the problem to solve to find new possibilities for improving the efficiency. As has been demonstrated with the improved model for placing several pieces (Section 6.1), finding common features in and between

propagators can be a fruitful way to improve the efficiency of a model.

Finally, the model presented for general placement problems is a very versatile and general model. The pieces can be quite general: non-connected pieces, alternative pieces, and pieces with several colors. The latter technique is used in [37] to solve the Solitaire Battleship problem by encoding water-boundaries into the pieces (representing ships) to ensure separation. As shown, the model can solve the traditional pentominoes problem in reasonable time with a very low modeling effort.

Future work. On the implementation side, a full, non-prototype integration of MDD handling into the current `regular` constraint in Gecode could speed up many examples, as well as providing for users that want to specify constraints using MDDs.

On the conceptual side, the potential of abstractions should be explored further, and a few abstractions implemented.

For the placement model, a direct comparison with the `geost` constraint [3] would be very interesting. The comparison in [37] is unfortunately lacking in that different branching schemes and potentially slightly different models are used.

Bibliography

[1] Nicolas Beldiceanu, Pascal Brisset, Mats Carlsson, François Laburthe, Martin Henz, Eric Monfroy, Laurent Perron, and Christian Schulte. Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a workshop of CP 2002. Technical Report TRA9/02, School of Computing, National University of Singapore, 55 Science Drive 2, Singapore 117599, September 2002. Referenced on page 72.

[2] Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005. Referenced on page 50.

[3] Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, Rida Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In Bessiere [8], pages 180–194. Referenced on page 68.

[4] Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte. Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000. Technical Report TRA9/00, School of Computing, National University of Singapore, 55 Science Drive 2, Singapore 117599, September 2000. Referenced on page 73.

[5] Frédéric Benhamou, editor. *Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, Nantes, France, September 2006. Springer-Verlag. Referenced on pages 71 and 73.

[6] Frédéric Benhamou. Heterogeneous constraint solving. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag. Referenced on page 12.

[7] Christian Bessiere. Constraint propagation. In Rossi et al. [48], chapter 3, pages 29–84. Referenced on page 14.

[8] Christian Bessiere, editor. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007. Referenced on pages 69, 73, and 75.

[9] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997. Referenced on page 40.

[10] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005. Referenced on page 40.

[11] Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors. *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*. IOS Press, 2006. Referenced on pages 70 and 71.

[12] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. Referenced on pages 50 and 51.

[13] Mats Carlsson. Personal communication, 2006. Referenced on page 28.

[14] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Glaser et al. [24], pages 191–206. Referenced on page 28.

[15] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In Brewka et al. [11], pages 78–82. Referenced on pages 50 and 53.

[16] A. Darwiche and P. Marquis. A knowledge compilation map, 2002. Referenced on page 50.

[17] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahamane Aggoun, Thomas Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988. Referenced on page 28.

[18] Grégoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In van Beek [57], pages 211–225. Referenced on page 47.

[19] John G. Fletcher. A program to solve the pentomino problem by the recursive use of macros. *Commun. ACM*, 8(10):621–623, 1965. Referenced on page 21.

[20] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`. Referenced on pages 3, 8, 14, 28, and 35.

[21] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In Benhamou [5], pages 284–298. Referenced on page 34.

[22] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Brewka et al. [11], pages 98–102. Referenced on pages 28 and 40.

[23] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Handbook of constraint programming. In Rossi et al. [48], chapter Symmetry in Constraint Programming, pages 329–376. Referenced on page 23.

[24] Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors. *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, volume 1292 of *Lecture Notes in Computer Science*. Springer, 1997. Referenced on page 70.

[25] Gregor Gramlich and Georg Schnitger. Minimizing NFA's and regular expressions. *J. Comput. Syst. Sci.*, 73(6):908–923, 2007. Referenced on page 61.

[26] Hermann Gruber and Markus Holzer. Computational complexity of NFA minimization for finite and unary languages. In *1st International Conference on Language and Automata Theory and Applications (LATA '07), Tarragona, Spain*, number 35/07 in Technical Report, pages 261–272. Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, 2007. Referenced on page 61.

[27] Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. In Beldiceanu et al. [1], pages 39–46. Referenced on page 46.

[28] Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using robdds. *J. Artif. Intell. Res. (JAIR)*, 24:109–156, 2005. Referenced on page 50.

[29] Brahim Hnich, Mats Carlsson, François Fages, and Francesca Rossi, editors. *Recent Advances in Constraints*, volume 3978 of *Lecture Notes in Computer Science*. Springer, 2006. Referenced on page 74.

[30] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages and Computation*. Low Price Edition. Addison Wesley Longman, Inc, Reading, Mass., USA, 2 edition, 2001. Referenced on page 18.

[31] ILOG Inc., Mountain View, CA, USA. *ILOG Solver 6.3 reference Manual*, 2006. Referenced on page 28.

[32] Intelligent Systems Laboratory. SICStus Prolog user's manual, 4.0.0. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, 2007. Referenced on pages 36 and 51.

[33] Tao Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993. Referenced on page 18.

[34] Jyrki Katajainen and Jeppe Nejsum Madsen. Performance tuning an algorithm for compressing relational tables. In Penttonen and Schmidt [45], pages 398–407. Referenced on pages 56 and 57.

[35] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In Bessiere [8], pages 379–393. Referenced on pages 55 and 57.

[36] François Laburthe. CHOCO: implementing a CP kernel. In Beldiceanu et al. [4], pages 71–85. Referenced on page 28.

[37] Mikael Z. Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First Workshop on Bin Packing and Placement Constraints (BPPC'08), associated to CPAIOR'08*, 2008. http://contraintes.inria.fr/CPAIOR08/BPPC.html. Referenced on pages 8 and 68.

[38] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Bessiere [8], pages 409–422. Referenced on page 8.

[39] Vitaly Lagoon and Peter J. Stuckey. Set domain propagation using robdds. In Wallace [58], pages 347–361. Referenced on page 50.

[40] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In Benhamou [5], pages 284–298. Referenced on page 41.

[41] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. Referenced on page 12.

[42] Robert Mateescu and Rina Dechter. Compiling constraint networks into and/or multi-valued decision diagrams (aomdds). In Benhamou [5], pages 329–343. Referenced on page 50.

[43] Roger Mohr and Gérald Masini. Good old discrete relaxation. In Yves Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988. Pitmann Publishing. Referenced on page 12.

[44] Gert Møller. *On the Technology of Array Based Logic*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1995. Referenced on pages 55 and 56.

[45] Martti Penttonen and Erik Meineche Schmidt, editors. *Algorithm Theory - SWAT 2002, 8th Scandinavian Workshop on Algorithm Theory,*

Turku, Finland, July 3-5, 2002 Proceedings, volume 2368 of *Lecture Notes in Computer Science*. Springer, 2002. Referenced on page 72.

[46] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Wallace [58], pages 482–495. Referenced on pages 17, 18, 43, 51, 52, and 60.

[47] Claude-Guy Quimper and Toby Walsh. The all different and global cardinality constraints on set, multiset and tuple variables. In Hnich et al. [29], pages 1–13. Referenced on page 44.

[48] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006. Referenced on pages 70, 71, and 74.

[49] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. Referenced on page 53.

[50] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle, WA, USA, 1994. AAAI Press. Referenced on page 44.

[51] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Rossi et al. [48], chapter 14, pages 495–526. Referenced on page 14.

[52] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Wallace [58], pages 619–633. An extended version is available as [53]. Referenced on pages 13, 28, 37, and 64.

[53] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines, 2006. Available from `http://arxiv.org/abs/cs.AI/0611009`. Referenced on pages 34, 37, 64, and 74.

[54] Christian Schulte and Guido Tack. Perfect derived propagators. In Peter J. Stuckey, editor, *Fourteenth International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *Lecture*

Notes in Computer Science, pages 571–575, Sydney, Australia, September 2008. Springer-Verlag. Referenced on page 34.

[55] Dana S. Scott. Programming a combinatorial puzzle. Technical Report Technical Report No. 1, Department of Electrical Engineering, Princeton University, 1958. Referenced on page 21.

[56] Michael A. trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118(1):73–84, June 2003. Referenced on page 65.

[57] Peter van Beek, editor. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*. Springer, 2005. Referenced on page 71.

[58] Mark Wallace, editor. *Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2004. Referenced on pages 73 and 74.

[59] Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. In Bessiere [8], pages 743–757. Referenced on page 54.