MONASH University

Semi-Automatic Learning-based Model Transformation

by

Kiana Zeighami

A thesis submitted for the degree of Master of Philosophy at

Monash University in 2019

Caulfield School of Information Technology

Supervisors:

Dr. Guido Tack

Prof. Maria Garcia de la Banda

Dr. Kevin Leo

Contents

List of Tables

List of Figures

Semi-Automatic Learning-based Model Transformation

Kiana Zeighami
Kiana.Zeighami@monash.edu
Monash University, 2019

Supervisor: Dr. Guido Tack
Guido.Tack@monash.au
Associate Supervisor: Prof. Maria Garcia de la Banda
Maria.GarciadelaBanda@monash.edu
Associate Supervisor: Dr. Kevin Leo
Kevin.Leo@monash.edu

Abstract

Combinatorial Optimisation/Satisfaction Problems (COPs/CSPs) are concerned with making decisions from a set of possible choices. These problems arise in many areas of our lives, and it is of great importance to solve them efficiently. Modern approaches to solve these problems first specify a model of the problem that formally describes it, and then compile this model for a particular solver which explores choices to find a solution. Unfortunately, real-world problems often have an exponential number of possible choice combinations, and solving them requires an extensive amount of time. There are also many possible ways to model a problem, with different models exhibiting vastly different solving time. It requires a great deal of expertise to develop a model that can be solved quickly. Thus, it is important to develop techniques that can help modellers improve their models. Recently, Shishmarev et al. [68] demonstrated how *learning solvers* can be used to improve the model. These solvers learn from their mistakes by generating an explanation for why a set of choices are incompatible, and Shishmarev et al. used these explanations to improve the model. However, their method was mostly manual and sufferred from some limitations. This thesis presents a framework that automates and improves the work of Shishmarev et al. [68]. In particular, it first simplifies the explanations, which makes them easier and more efficient to analyse later. Afterwards, it connects the explanations to the parts of the model that were involved in creating them. It then groups the explanations into more general *patterns*, making them easier to analyse. As a result, it provides information and suggestions to the modellers that can help them improve the model. Our experiments show that our method is practical, general for different types of problems, and can improve performance for certain models.

Semi-Automatic Learning-based Model Transformation

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Kiana Zeighami
November 7, 2019

Vita

Publications arising from this thesis include:

> Zeighami, Kiana & Leo, Kevin & Tack, Guido & Garcia de la Banda, Maria. (2018). Towards Semi-Automatic Learning-Based Model Transformation: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings. 403-419. 10.1007/978-3-319-98334-9_27.

Permanent Address:	Caulfield School of Information Technology
	Monash University
	Australia

This thesis was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

Acknowledgments

I owe many thanks to my supervisors, Maria, Guido and Kevin, for ceaselessly guiding me throughout my research. Particularly, I'd like to thank Maria, for giving the utmost attention to my research, and for providing me with precise feedback and advice, Guido, for all the technical and research skills that he patiently thought me, and Kevin, for being such a great source of motivation and encouragement, and for enthusiastically explaining various concepts to me.

I'd like to thank my officemates for creating such a great working environment. In particular, Maxim for all of our interesting discussions and his input and ideas in my research, as well as Jip, Alex and David for making this journey more enjoyable.

Furthermore, I'd like to express my gratitude to my parents Mahnaz and Shaya, and my brother Sepanta, for their overwhelming emotional support and encouragement.

Kiana Zeighami

Monash University
November 2019

Chapter 1

Introduction

Combinatorial Optimisation/Satisfaction Problems (COPs/CSPs) are concerned with making decisions from a *combination of choices* that satisfy a set of *constraints*, while *maximising* or *minimising* an objective function (if any) [40]. A classic example of a COP is the *Travelling Salesperson Problem* (TSP), where there are a number of cities that a salesperson must visit for trading products. The problem is to plan the *shortest* route to visit these cities, that satisfies the following constraints: each city must be visited once, and the first and last city in the journey must be the origin city.

COPs/CSPs arise in many important areas of our lives, such as public transportation, energy management and health care. Unfortunately, solving these problems is remarkably difficult due to the exponential number of possible choice combinations. Thus, it is of great importance to develop systems that can help solve these problems in a short period of time.

A common approach to solve COPs/CSPs is to first model them in a high-level modelling language, such as MiniZinc [44], OPL [76] or ESSENCE [32]. This model formally describes the problem's choices (in terms of decision variables), constraints and objective function (if any). Then, the model is combined with problem-specific data (e.g. the number of cities and the distances between them in TSP problems) to form an *instance* of the problem, which is a low-level (i.e. solver-level) program suitable for the particular solving technique (solver) selected by the modeller.

Many sophisticated techniques are available for solving combinatorial problems, such as Mixed Integer Programming (MIP) [1], Constraint Programming (CP) [65] and Boolean Satisfiability (SAT) [49]. While this work mainly concerns itself with CP-solvers, the goal is to improve solver-independent models which can, in turn, improve performance of other kinds of solvers on the same model.

CP-solvers combine *search* and *propagation*. Given a COP/CSP, the set of possible assignments of values to the decision variables from their domains is referred to as the *search space* of the problem [63]. Initially, CP-solvers apply propagation to reduce the search space, by removing values that are incompatible with respect to some constraints. Then, they explore the search space, by making a decision, which involves selecting a decision variable and assigning it a value from its domain.

Each time a decision is made, the CP-solver applies propagation to remove incompatible values from the variable domains. After propagation, if all variables are successfully assigned (having singleton domain) a *solution* has been found. However, if the domain becomes empty, the search reaches *failure.* When failure occurs, search reverts a previous decision and makes an alternative one [63].

There is a powerful class of CP-solvers called *learning solvers.* When the search fails, learning solvers aim at learning from this failure by generating an explanation for this failure, called a *nogood.* From this nogood, the learning solver generates and records a *clause*, that is the negated form of a nogood, to ensure that the solver cannot repeat the same mistake again. This can significantly reduce the amount of search space that needs to be explored by the solver.

The development of a model for COPs/CSPs involves multiple stages, such as identifying the problem constraints and objective function, modelling the variables of the problem and expressing the constraints and objective function in terms of these variables, preparing the data for the problem, compiling the resulting instance, and solving and testing the efficiency of the generated program. As a result, modellers will go through an iterative development loop, which involves modifying the model and repeating the other development stages iteratively. This can require a very large amount of time. In addition, there are many possible ways to model a problem, and while some may lead to very fast solving, others can take orders of magnitude longer. Testing each possible model and going through this iterative development loop again is not very practical. Thus, it is important to develop techniques that can help us automate the procedure of modelling.

One of the main techniques to improve a model is to add an effective *redundant constraint* (i.e. implied constraint) to the model. Redundant constraints are implied by the model constraints and, therefore, adding them to the model does not eliminate any solution [20]. However, adding redundant constraints can have a significant impact on solving time allowing the solver to better exploit the structure of the problem, and reduce the size of the search space.

While their addition help better exploit the structure of the problem and thus reduce the search space [19], unfortunately, they also increase the computational cost due to the need to handle the extra constraints. Hence, it is challenging to find an effective redundant constraint where the time saved by reducing the search space outweighs the increased computational burden.

Recently, Shishmarev et al. [68] demonstrated how the clauses learnt by a learning solver can be used to improve the model. To achieve this, they executed the model using the learning solver Chuffed [21], and then replayed its search decisions (in the same order) using the non-learning solver Gecode [67]. They then merged the two resulting search trees, counted the number of nodes explored by Gecode that were not explored by Chuffed, and assigned them to the learnt clause(s) that helped Chuffed fail before Gecode, and this formed a ranking of all clauses based on the total search space avoided by each clause, with the top-ranked clauses being the most effective for reducing the search space. Also, since clauses refer to the solver-level variables in the low-level program obtained after

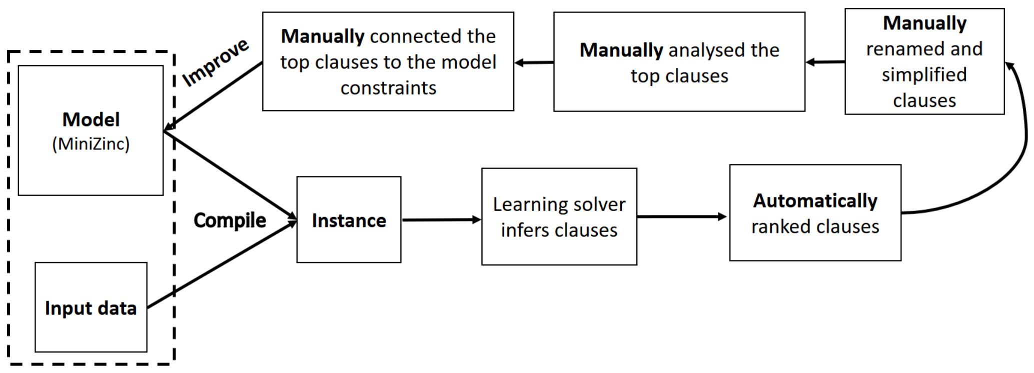
Manually connected the top clauses to the model constraints
Manually analysed the top clauses
Manually renamed and simplified clauses
Improve
Model (MiniZinc)
Compile
Instance
Learning solver infers clauses
Automatically ranked clauses
Input data

Figure 1.1: Overview of the framework developed by [68]

compilation, rather than to the model-level variables, they manually renamed variables in these clauses, simplified them and connected them to the model constraints, with the aim of identifying the constraints responsible for creating the clauses. Finally, they manually inspected the most effective clauses and focused on the model constraints that were involved in creating these clauses. The existence of clauses that contribute to a considerable reduction in the size of the search space can indicate that the constraints associated to the top clauses should be strengthened. To achieve this, they considered problem information that could be relevant to the occurrence of the top clauses, and realised that some problem information related to the clauses was not explicitly expressed in a model. Thus, the solver could not take that fact into consideration when solving. This helped them identify a redundant constraint that could be added to the model to explicitly express that information, and to better exploit the structure of the target problem. Figure 1.1 provides an overview of their method.

While the work of [68] is very innovative and has great potential, the proposed technique is mostly manual, requiring modellers to manually analyse each nogood and connect it to the model. In addition, the ranking of nogoods is done by considering a single instance of a problem, which is more restricted and inaccurate representation of the whole problem. This leads to the main question of this research:

Is it possible to analyse clauses automatically and connect them to the model constraints in such a way that they better support the modelling of combinatorial optimisation problems?

In answering this question I had to address the following sub-questions:

1. Can the learned clauses be automatically renamed, simplified and connected to the model constraints that were involved in creating the nogoods?

2. Can the ranking of the clauses be improved by combining the clauses learnt using different input data files and search strategies for a given model?

3. Can the ranked clauses be combined with model information that will help the modellers detect the reasons behind the occurrence of the clauses in a way that can help them modify the model?

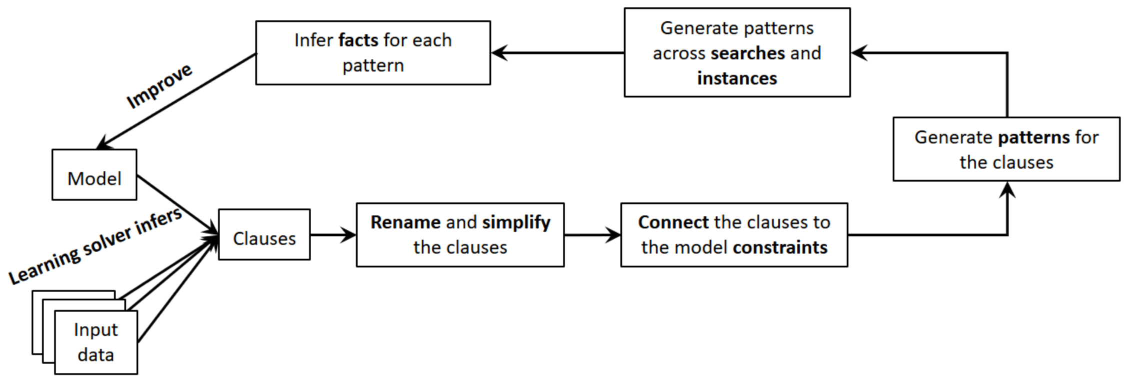
Infer **facts** for each pattern
Generate patterns across **searches** and **instances**
Generate **patterns** for the clauses
Improve
Model
Learning solver infers
Clauses
Rename and **simplify** the clauses
Connect the clauses to the model **constraints**
Input data

Figure 1.2: Overview of our framework

This thesis presents a framework (summarised in Figure 1.2) that first automatically renames the learned clauses to be expressed in terms of model variables rather than in terms of their compiled version. Then, it simplifies these clauses to reduce their length, and it identifies the model constraints that were involved in creating each of them. Afterwards, the framework groups the clauses with *similar structure* into more general *patterns*, which helps us obtain a more accurate ranking for each clause, and makes each clause easier to understand. It then combines these patterns with the model information to learn *facts* for each pattern. These facts serve as a condition for the occurrence of each pattern. The patterns, their associated facts and constraints are presented to the modellers, with the aim of guiding them to detect the potential constraints that should be reformulated or an effective redundant constraint that can be added to the model. The generated patterns and facts are not proven to be correct, and it is suggested to the modeller to prove them. Thus, this step of our framework is manual, and our method is semi-automatic.

There are two areas of research that are closely related to this work: *automated constraint acquisition* and *automated model transformation*. These areas are concerned with helping modellers better model their target problem so that it can be solved more efficiently. *Automated constraint acquisition* aims at automatically acquiring model constraints given a set of solutions, non-solutions, and a constraint library, while *automated model transformation* deals with automatically improving a model using common techniques, such as *detecting global constraints*, *redundant constraints* and *symmetry breaking constraints*, which we will discuss later. These lines of research have a similar goal to our work; however, their approach is different from ours and, although the developed systems are powerful, they still suffer from some limitations, such as restrictions on the type of the models that the system can process, and inability to take the clauses learnt by learning solvers into consideration. Since previous research has shown us the clauses inferred by learning solvers can be used to improve a problem model, our aim is to fill this research gap.

In conclusion, COPs/CSPs appear in many different areas of our lives, and it is important to improve the modelling of these problems to be able to solve them more efficiently. In this thesis, I present our framework that, given a model and data, automatically analyses the clauses learnt by learning solvers, and provides suggestions to the modeller regarding how to improve the model.

The main contribution of this thesis is the automation of Shishmarev et al.'s manual approach, and the improvement of some parts of their method. In achieving this, my work resulted in the following contributions:

- The framework automatically renames and simplifies the learnt clauses. This in turn reduces the time required by the framework to generate patterns and facts, and also helps us detect more patterns, and rank them more accurately.
- The framework automatically connects the learnt clauses to the model and detects the model constraints that were involved in creating them. The connection helps the modeller detect model constraints that can be strengthened, or identify the need to add a redundant constraint to the model that can help reduce the solving time.
- The framework automatically generalises groups of clauses into patterns. This helps us obtain more accurate ranking for a group of clauses by accumulating the estimated reduction in search space of the clauses associated to a pattern, and also it represents the clauses in a form of a generic constraint, that later along with its associated fact can be presented to the modeller.
- The framework takes the whole class of the problem into consideration rather than one instance. This helps us improve the model rather than one instance.
- The framework considers multiple search strategies to obtain clauses that may not be discovered by a certain search strategy. This improves the accuracy of the ranking technique proposed by Shishmarev et al., and also helps detect more patterns.
- The framework automatically learns facts for each pattern. This combines the background information with the information present in the patterns, which may serve as a condition for the occurrence of the patterns and, together with the patterns, can be suggested to the modeller as a potential constraint to be added to the model.
- Our experiments show that the framework is practical and general for a variety of models, and it improved four MiniZinc models. Importantly, the framework improved a MiniZinc standard decomposition of a global constraint, which is commonly used in a variety of models.

The structure of this thesis is as follows. Chapter 2 presents the background knowledge required to understand the rest of the thesis. It describes how COPs/CSPs are modelled and solved, and the basics of constraint programming and learning solvers. Also, the method of Shishmarev et al. is described in more detail, since this work builds upon it. Then, it discusses ConAcq which is the constraint acquisition algorithm we used in our framework for learning facts from the patterns; Chapter 3 discusses our method of contextualising and simplifying clauses; Chapter 4 presents our method for inferring patterns from a set of clauses, then it discusses our method for considering multiple instances and search strategies; Chapter 5 discusses our two methods for generating facts from the patterns by considering model information; Chapter 6 presents the experiments that were carried out to evaluate the efficiency and practicality of our framework. In addition, it discusses four case studies where using our framework improved the model or the decomposition of global constraints; Chapter 7 discusses the work related to our framework; Chapter 8 concludes the thesis and presents some directions for future work.

Chapter 2

Background

2.1 Introduction

This chapter provides the necessary background to understand the rest of the thesis. Section 2.2 describes what Constraint Optimisation/Satisfaction Problems (COPs/CSPs) are and how they are often modelled and solved. Section 2.3 explains the basics of Constraint Programming (CP), which is one of the paradigms used to solve a variety of COPs/CSPs, and it is the main focus of this research. More specifically, our focus is on a class of CP-solvers called Learning solvers, which are described in Section 2.4, and they were used by Shishmarev et al. [68] to propose a manual method to improve COP/CSPs models. This method, which inspired our work, is described in Section 2.5. Section 2.6 describes the toolchains developed by Leo et al. [42, 43] which are used by our method to connect the model to the program that executes it. Lastly, Section 2.7 describes the constraint acquisition systems, which are used by our method for learning facts for each pattern that is described in Chapter 5.

2.2 Constraint Optimisation/Satisfaction Problems

Constraint Optimisation/Satisfaction Problems (COPs/CSPs) are concerned with finding a *combination of choices* that satisfy a set of *constraints*, while *maximising* or *minimising* an objective function (if any) [40]. A CSP P, is formally defined as a tuple of $P{=}\langle X, D, C\rangle$, where X is a set of n *decision variables*, and D is a tuple of n *domains*, where $D(X_i)$ maps the decision variable X_i to its domain. C represents a set of constraints, where C_i describes a relation between a subset of variables (X). A solution to P is an n-tuple A, where $A_i \in D(X_i)$ and each $C_i \in C$ is satisfied [63]. COPs are a tuple of $P{=}\langle X, D, C, F\rangle$, where F is a function over the variables in X which is to be maximised or minimised, and an *optimal solution* is a solution that maximises or minimises the objective function.

Let us illustrate this by means of an example: the well-known `golomb ruler` optimisation problem [44]. A golomb ruler has a set of marks, where each mark has a unique value, and each pair of marks should have a unique distance, i.e. a distance different to any other pair of marks. The objective of the golomb ruler problem is, given a number `m` of marks, to minimize the final mark in the ruler. Figure 2.1 demonstrates an example of

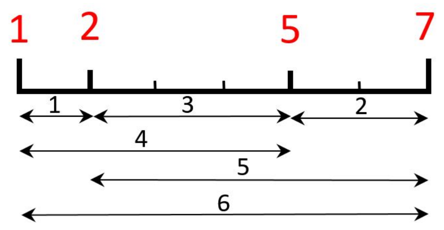

Figure 2.1: A `golomb ruler` of length 7

a `golomb ruler`. In this problem, the decision variables are `marks` and their `differences`. The constraint is that each pair of marks should have a unique distance, and the objective function is to minimise the longest mark in the ruler.

To model COPs/CSPs, modelling languages such as MiniZinc [44], OPL [76] and ESSENCE [32] are used. Let me illustrate the modelling of an optimisation problem in MiniZinc by means of the `golomb ruler` problem discussed before. A possible model is as follows:

Example 1. `Golomb ruler` model

```
1  int: m; % number of marks
2  int: n; % upper bound of marks
3  int: d = m*(m-1)/2; % number of differences
4
5  array[1..m] of var 0..n: marks;
6  array[1..d] of var 0..n: differences =
7  [marks[j]-marks[i]| i in 1..m, j in i+1..m];
8
9  include "globals.mzn";
10 constraint alldifferent(differences);
11
12 solve minimize mark[m];
```

Lines 1 and 2 define the model parameters, which will later get instantiated with input data. The parameter `m` indicates the number of marks, and the parameter `n` indicates the upper bound of marks i.e. the largest possible mark. Line 3 defines `d`, which represents the number of differences between marks.

Lines 5 to 7 define the decision variables. The decision variable `marks` is an array of integer variables, representing the value associated to each mark on the ruler. In addition, the decision variable `differences` contains the value associated to each difference between the value of each pair of `marks`.

In line 9, the library of global constraints is included. Global constraints are pre-defined structures that encapsulate a number of similar sub-problems that can be reused. Some solvers utilise specialised algorithms for them [40]. One commonly used global constraints is the `alldifferent` constraint, which is used in line 10. The `alldifferent` global constraint ensures that all the elements of the array that are passed to it (`differences`), take unique values, different from each other.

Lastly, in line 12, an objective function is defined, which states that the goal is to minimise the value assigned to the last element of the decision variable `marks` (the length of the ruler).

Models provide a generic representation that can be used for solving specific *instances* of a problem. Instance specific data is often separated from the model. For example, for the `golomb ruler` model described before, each instantiation of parameters `n` and `m`, represents an instance of the problem.

To solve COPs/CSPs, the model and its input data are compiled into a low-level program that is passed to an arbitrary solver algorithm. Many sophisticated solver algorithms are available to solve combinatorial problems, such as Mixed Integer Programming (MIP) [1], Constraint Programming (CP) [65] and Boolean Satisfiability (SAT) [49]. The focus of this research is mostly on CP-solvers which are described in Section 2.3.

2.3 Constraint Programming

Constraint Programming (CP) is a powerful and generic paradigm that can be used to solve COPs/CSPs. There are many different solvers that implement CP, which are known as CP-solvers such as Chuffed [21], Gecode [66] and Choco [58]. CP-solvers combine *search* and *propagation*, which are discussed in this section.

2.3.1 Search

For a given COP/CSP the set of possible assignments of values to the decision variables from their domains is referred to as the *search space* of the problem [63]. One of the most common techniques to explore the search space of a COP/CSP used by CP-solvers is to select a decision variable and assign it a value from its domain. We refer to a variable/value assignment as a *literal*. For example, for the `golomb ruler` problem, the literal `mark[4]=3` is a decision made by a solver. The solver repeats this procedure until all the variables are assigned, or a *failure* is encountered. If a failure occurs, CP-solvers use a *backtracking* search algorithm [24, 36] where search goes back to the previous decision, and makes an alternative decision [63]. If all the variables could be assigned the solver has a *solution*.

The decisions that are made by a solver are often visualised in the form of a *search tree*. As an example, the following figure demonstrates the search tree explored by Gecode for the `golomb ruler` model provided in example 1 with instance data `m=5`.

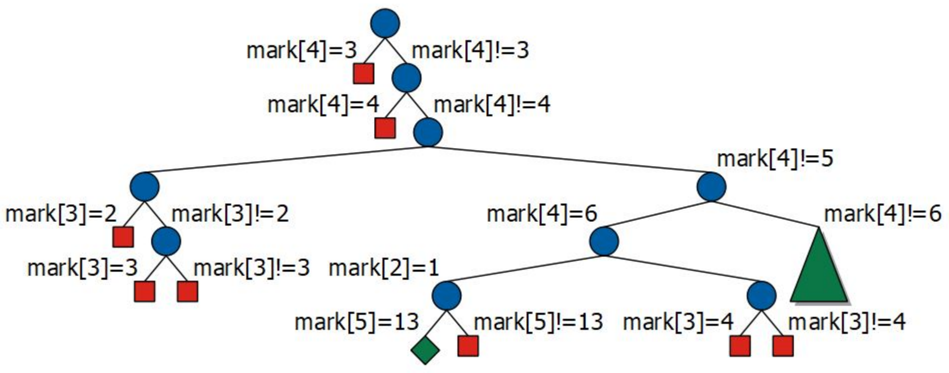

Figure 2.2: Search tree explored by Gecode for the `golomb ruler` problem

In the search tree above, each blue circle represents a *decision variable* picked by Gecode, e.g. `mark[4]`, and each branch represents a *decision* made by the solver (e.g. `mark[4]=3`). The red squares indicate *failures*, the green diamond represents a *solution*, and the green triangle represents a compact sub-tree for simplicity.

To improve the performance of the classical backtracking algorithm, there are a variety of techniques such as backjumping, restarts and nogood recording (which are discussed in Section 2.4). Also, to search more efficiently, there are various *search strategies* for selecting a decision variable and assigning a value to it. One of the most common ones is to select the variables that are more likely to be difficult to satisfy first, e.g. those with the smallest domain [61].

To solve optimisation problems, CP-solvers combine backtracking search with a *branch and bound algorithm*. When a solution is found by this algorithm, the value of objective function is calculated, and a constraint is added to force the search to find solutions with a better objective value. Branch and bound improves the solving time by reducing the search space [63].

2.3.2 Propagation

One of the strengths of CP-solvers is *constraint propagation*. Constraint propagation aims at reducing the search space by removing values from the variables' domains that cannot be part of any solution [40].

To achieve this, there are special algorithms designed for each constraint called *propagators*. Every time the search makes a decision, the propagators associated to constraints that contain the assigned variable are executed. These propagators may remove values from the domain of any of its variables. These removals, may in turn trigger the activation of more propagators. This continues until propagation reaches a fix point (when no more values can be removed from the variable domains).

As an example, consider a CSP with variables x_1 and x_2, domains $\{1, 2, 3, 4\}$ and $\{0, 1, 2, 3\}$ respectively, and constraints:

$$x_2 = x_1 \tag{2.1}$$

$$x_1 \neq 3 \tag{2.2}$$

Initially, the propagator for constraint 2.1 removes value 4 from the domain of x_1, and value 0 from the domain of x_2. Hence, the new domains of will be: {1, 2, 3} and {1, 2, 3}. Then, the propagator for constraint 2.2 further reduces the domains to {1, 2} and {1, 2, 3}. Since the value of x_1 has changed, the propagator for constraint 2.1 is activated, removing 3 from the domain of x_2, and leaving the domains as $x_1 \epsilon \{1, 2\}$ and $x_2 \epsilon \{1, 2\}$. At this stage, the propagation engine cannot remove any more values from the variable's domains and a fixed point is reached. Since the problem is not solved yet, the search engine will try different combinations of values (depending on the search strategy) in order to find one or more solutions.

2.4 Learning Solvers

Learning solvers [54, 30] are a powerful class of CP-solvers that analyse their failures to learn from their previous mistakes by recording the reasons for each failure. To achieve this, all propagators are instrumented to explain the domain changes they performed on the domain of the variables in terms of *equality* ($x = d$ for $d \in D(x)$), *disequality* ($x \neq d$) or *inequality* ($x > d$ or $x < d$) literals. An *explanation* for literal ℓ is an implication $S \rightarrow \ell$, where S is a set of literals (interpreted as a conjunction).

For example, the explanation for the propagator of the constraint $x \neq y$ inferring literal $y \neq 5$, given literal $x = 5$, is $\{x = 5\} \rightarrow y \neq 5$. Each new literal inferred by a propagator is recorded together with its explanation, forming an *implication graph.*

Example 2. Consider a Constraint Satisfaction Problem (CSP) with the following constraints:

$$\neg x_1 \vee x_2 \tag{2.3}$$

$$\neg x_1 \vee x_3 \vee x_9 \tag{2.4}$$

$$\neg x_2 \vee \neg x_3 \vee x_4 \tag{2.5}$$

$$\neg x_4 \vee x_5 \vee x_{10} \tag{2.6}$$

$$\neg x_4 \vee x_6 \vee x_{11} \tag{2.7}$$

$$\neg x_5 \vee \neg x_6 \tag{2.8}$$

$$x_1 \vee x_7 \vee \neg x_{12} \tag{2.9}$$

$$x_1 \vee x_8 \tag{2.10}$$

$$\neg x_7 \vee \neg x_8 \vee \neg x_{13} \tag{2.11}$$

Assume that the solver first makes the decision $\neg x_9$, that is literal $x_9 = 0$. Since x_9 only appears in Constraint 2.4 and, the decision does not cause domain changes to the variables of this constraint, the search continues. Assume the next selection of decisions is $\neg x_{10}$, $\neg x_{11}$, x_{12} and x_{13}. Sine none of these decisions cause the propagators of Constraints 2.6, 2.7, 2.9 and 2.11 to reduce the domains, the search continues, and we assume now that the next decision is x_1. This causes Constraints 2.9 and 2.10 to be satisfied, and causes x_2 to be true, as it is the only literal that is not assigned in Constraint 2.3. This also causes the propagator of Constraint 2.4 to infer x_3. This leads the propagator of Constraint 2.5 to infer x_4. Consequently, the propagator of Constraint 2.6 infers x_5, and that of Constraint 2.7 infers x_6. This leads to failure because Constraint 2.8 cannot be satisfied. Figure 2.3 demonstrates an implication graph for this example.

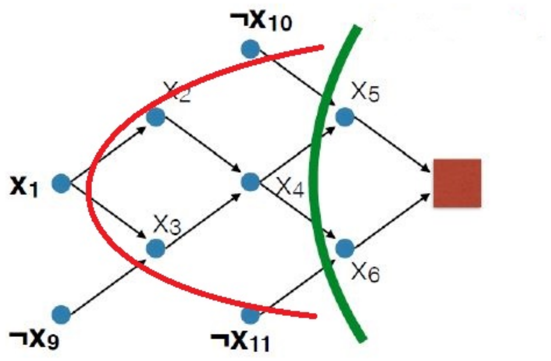

Figure 2.3: Implication graph

Each search decision and its consequences have an associated decision level [71]. The implication graph above shows all the nodes in the last decision level, $\boldsymbol{x_1}$, that contributed to the failure. The nodes $\boldsymbol{x_1}$, $\boldsymbol{\neg x_9}$, $\boldsymbol{\neg x_{10}}$ and $\boldsymbol{\neg x_{11}}$ are the decisions, while the other nodes represent the implications.

Whenever the search reaches a failure, learning solvers use the implication graph to compute a *nogood* N, that is, a set of literals (interpreted as a conjunction) that represents the reason for the failure and cannot be extended to a solution. Then, they add a *clause* that is, the negation of the nogood, $\neg N$, represented as a set of literals interpreted as their disjunction as a new constraint, to ensure that search cannot fail for the same reasons.

The 1-UIP scheme is a popular method to generate nogoods. As you can see in Figure 2.3, at the last decision level $\boldsymbol{x_1}$, any path that connects $\boldsymbol{x_1}$ to the failure passes also through $\boldsymbol{x_4}$, we call such a node a *Unique Implication Point* (UIP). UIPs are important because they are the single source of the failure [78], and the UIP that is closest to the failure is called a 1-UIP. In this method, a nogood is generated by partitioning the implication graph, in such a way that all the decision variables are separated from the failure. The part of the graph that contains the decision variables is called the *reason side* [78], and the other part is the *failure side*, such a partitioning is a *cut*. All the nodes that have at least one edge to the failure side are the reasons for the failure, and a nogood can be generated for any cut to describe the reasons. For example, for the red cut, the nogood $\neg x_{10} \wedge x_1 \wedge \neg x_9 \wedge \neg x_{11}$ can be generated, and for the green cut the nogood $\neg x_{10} \wedge x_4 \wedge \neg x_{11}$.

When failure occurs, instead of performing the traditional backtracking technique mentioned in Section 2.3.1, a *backjumping* technique is used. The aim of this technique is to go back to the source of the failure [27] and avoid repeating the same mistake, as this often prunes the search space. For example, in the implication graph above, the decisions that contributed to the red cut, were made in the following order: $\neg x_9$, $\neg x_{10}$, $\neg x_{11}$ and x_1. As you can see $\neg x_{11}$ is the last decision before x_1 which is the current decision. To avoid repeating the same mistake, search backjumps to $\neg x_{11}$, and tries x_{11}. Another technique that is usually combined with clause learning is *restart*: After a certain portion of search space is explored, the search will be interrupted and started from the beginning, while incorporating the learnt clauses [70]. This, together with backjumping, can result in a large portion of the search space being avoided.

Learning solvers are powerful because they strengthen the propagation by adding the learnt clauses, which helps them reduce the search space. Also, due to the backjumping technique, they change the search order guiding search towards a more promising part of the search space. However, adding propagation can increase the execution time, and it is challenging to achieve a good trade-off. Thus, there are several techniques focusing on filtering the learnt clauses, and keeping the ones with the highest impact.

2.5 Learning from Learning Solvers

Shishmarev et al. [68] demonstrated how the clauses learnt by a learning solver can be used to improve the associated model. Their method can be summarised as follows: First, they extracted the clauses that are potentially the most effective in improving the search. Then, they executed the model using the learning solver Chuffed [21], replayed its search decisions (in the same order) using the non-learning solver Gecode [67], and compared the two resulting search trees. This comparison allowed them to estimate the effectiveness of each clause in reducing the search space. To achieve this, for each clause, they calculated the number of the nodes explored by Gecode that were not explored by Chuffed, and assigned this number to each clause. This formed a ranking of all clauses based on the total number of search nodes avoided by each clause, with the top-ranked clauses being the most effective for reducing search. Afterwards, they manually inspected the 10 most effective clauses to understand the reason behind each failure. For this purpose, first they manually renamed the clauses, because when a MiniZinc model and data are compiled into a low-level program (FlatZinc) the compiler modifies the variable names, and introduces new variables. Thus, to understand the clauses, they manually renamed the clauses to be expressed in the model-level variable names. They then manually linked each clause to the model constraints and the model parameters that may be related to the clause. This helped them learn extra information about the model that was not explicit in the clauses learned by the solver. Because, clauses only contain information regarding instance variables, even though their correctness may also require particular values for the instance parameters which we call *facts*. They learnt such facts for each clause, and by linking each clause and its associated facts to the model constraint, they discovered constraints that could be modified to reveal the same information explicitly. This model transformation helped improve its solving time. In Example 3, we use the `free pizza` problem of [68] to illustrate their manual process.

Example 3. In this problem customers get pizzas either by paying for them or by using vouchers. Each voucher $\langle a, b\rangle$ allows customers to get b number of pizzas for free if they pay for a number of pizzas, and none of the b pizzas are more expensive than the a ones. A customer who has m vouchers and wants n pizzas aims to minimise the amount they pay for the n pizzas. Figure 2.4 demonstrates their MiniZinc model.

```
1  int: n;          % number of pizzas wanted
2  set of int: PIZZA = 1..n;
3  array[PIZZA]   of int: price;           % price of each pizza
4  int: m;       % number of vouchers
5  set of int: VOUCH = 1..m;
6  array[VOUCH] of int: buy;            % buy this many to use voucher
7  array[VOUCH] of int: free;           % get this many free
8
9  set of int: ASSIGN = -m .. m; % i -i 0 (free/paid with voucher i or not)
10 array[PIZZA] of var ASSIGN: how;
11 array[VOUCH] of var bool: used;
12
13 constraint forall(v in VOUCH)(used[v]<->sum(p in PIZZA)(how[p]=-v)>=buy[v]);
14 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
15 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
16 constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
17                                    -> price[p2] <= price[p1]);
18 int: total = sum(price);
19 var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);
```

Figure 2.4: The `free pizza` model from [68]

Lines 1-7 introduce the parameters of the problems: Line 1 introduces `n` which represents the number of pizzas. Line 2, introduces the `PIZZA` set where the members represent pizzas 1 to `n`. Line 2 defines the `price` array, which stores the prices of pizzas. In Line 4, `m` represents the number of the vouchers, and Line 5 introduces the `VOUCH` set, which represents the vouchers 1 to `m`. Lines 6 and 7 introduce `buy` and `free` arrays s.t. $\langle buy[i], free[i]\rangle$ represents the `i`th voucher $\langle a, b\rangle$.

The next three lines define two arrays of decision variables: `used[v]`, which is true iff voucher `v` was used; and `how[p]`, which is `v` if pizza `p` was free thanks to voucher `v`, is `0` if `p` was paid for and not used in any voucher, and is `-v` if `p` was paid for and used to get free pizzas with voucher `v`.

Constraints start in Line 13, which states that if voucher `v` was used, then the total number of pizzas bought and assigned to `v` must be greater than or equal to the number of pizzas required by it. Line 14 states similar information but in the opposite direction: the total number of pizzas bought and assigned to voucher `v` must be less than or equal to used[v]*buy[v]. Together, they constrain the total number of pizzas bought for v to be equal to v, if used. The constraint in Line 15 states that the total number of free pizzas obtained thanks to voucher v must be smaller than or equal to the number of free pizzas allowed by v if used (used[v]*free[v]). The last constraint states that if there are two pizzas p1 and p2 assigned to the same voucher, with p2 being free and p1 being paid for (given how[p1]<how[p2] and how[p1]=-how[p2]), then the price of p2 must be lower than or equal to that of p1. Finally, the objective function is defined as the sum of the prices of the pizzas that are bought.

Table 2.1: Taken from [68]: The most effective original learnt clauses in free pizza

Rank	Reduction	Clause
1	3425	X_INTRODUCED_0_=-1 X_INTRODUCED_1_=-1 X_INTRODUCED_2_=-1 X_INTRODUCED_3_=-1 X_INTRODUCED_4_=-1 X_INTRODUCED_0_=-2 X_INTRODUCED_1_=-2 X_INTRODUCED_2_=-2 X_INTRODUCED_3_=-2 X_INTRODUCED_4_=-2 X_INTRODUCED_5_≤0 X_INTRODUCED_5_≥3
2	2068	X_INTRODUCED_6_≤2 X_INTRODUCED_6_≥4 X_INTRODUCED_0_≠-3 X_INTRODUCED_0_≥-2
3	1712	X_INTRODUCED_3_≠3 X_INTRODUCED_0_=-3 X_INTRODUCED_1_=-3 X_INTRODUCED_2_=-3 X_INTRODUCED_3_=-3
4	1636	X_INTRODUCED_4_≠-3 X_INTRODUCED_2_≠3
5	1636	X_INTRODUCED_7_≠-3 X_INTRODUCED_2_≠3
6	1636	X_INTRODUCED_8_≠-3 X_INTRODUCED_2_≠3
7	1636	X_INTRODUCED_9_≠-3 X_INTRODUCED_2_≠3
8	1489	X_INTRODUCED_5_≤2 X_INTRODUCED_5_≥4 X_INTRODUCED_1_≠-3 X_INTRODUCED_1_≥-2
9	1404	X_INTRODUCED_4_≠-3 X_INTRODUCED_3_≠3 X_INTRODUCED_3_≤2
10	1403	X_INTRODUCED_9_≠-3 X_INTRODUCED_3_≠3

Table 2.2: Taken from [68]: The most effective learnt clauses in free pizza

Rank	Reduction	Clause
1	3425	how[1]=-1 how[2]=-1 how[3]=-1 how[4]=-1 how[5]=-1 how[1]=-2 how[2]=-2 how[3]=-2 how[4]=-2 how[5]=-2 how[6]≤0 how[6]≥3
2	2068	how[7]≤2 how[7]≥4 how[1]≠-3 how[1]≥-2
3	1712	how[4]≠3 how[1]=-3 how[2]=-3 how[3]=-3 how[4]=-3
4	1636	how[5]≠-3 how[3]≠3
5	1636	how[8]≠-3 how[3]≠3
6	1636	how[9]≠-3 how[3]≠3
7	1636	how[10]≠-3 how[3]≠3
8	1489	how[6]≤2 how[6]≥4 how[1]≠-3 how[1]≥-2
9	1404	how[5]≠-3 how[4]≠3 how[4]≤2
10	1403	how[10]≠-3 how[4]≠3

Table 2.1 shows the original top clauses obtained by Shishmarev et al. As you can see, it is difficult to understand them, as the clauses are expressed in terms of FlatZinc variables. Thus, they first manually renamed each clause to be expressed in terms of model-level variables, and as a result the clauses were translated into the ones in Table 2.2. In this experiment the following data was used:

```
n = 10;
m = 4;
price = [70, 10, 60, 65, 30, 100, 75, 40, 45, 20];
buy = [1, 2, 3, 3];
free = [1, 1, 2, 1];
```

Where each clause is interpreted as the disjunction of its literals. For example, the clause ranked 4th, {`how[5]`≠`-3`, `how[3]`≠`3`}, states that pizza 3 cannot be obtained for free using voucher 3 by paying for pizza 5 with that voucher. This might be easier to see in its equivalent form ¬(`how[5]=-3` ∧ `how[3]=3`).

After obtaining the clauses and their associated rankings, Shishmarev et al. manually analysed the top clauses. They focused on the shorter clauses such as clause 4 {`how[5]`≠`-3`, `how[3]`≠`3`}, because they are easier to understand. This clause can also be represented in implication form `how[5]=-3` → `how[3]`≠`3`, which means that if pizza 5 is bought by voucher 3, pizza 3 cannot be obtained for free using voucher 3. Shishmarev et al. realized that this clause occurs because of the fact that pizza 3 is more expensive than pizza 5 (`price[3]`>`price[5]`). More importantly, this clause shares a *similar structure* with the clauses 5, 6, 7 and 10, because they share the same sequence of variable names {`how[_]`, `how[_]`}, and the same sequence of operators {≠, ≠}.

Since the relationship between prices of pizzas `price[p2]<=price[p1]` appear in the constraint in line 12, Shishmarev et al. realized that some of the top clauses were direct consequences of this single constraint. Hence, they reformulated this constraint with the following constraint:

```
constraint forall(p1,p2 in PIZZA)((how[p2]>0/\how[p1]= -how[p2])->
price[p2] <= price[p1]);
```

This constraint reveals the same information but in a stronger way. In the original constraint, the condition `how[p1]<how[p2]/\how[p1]=-how[p2]` implies `0<how[p2]/\how[p1]=-how[p2]`, and to check if the condition `how[p1]<how[p2]/\how[p1]=-how[p2]` holds for the pizzas `p1` and `p2`, for any two pizzas that `how[p2]` is larger than `how[p1]`, it checks if `how[p1]=-how[p2]` also holds. For some pairs of pizzas where `how[p1]` and `how[p2]` are both positive, the second condition `how[p1]=-how[p2]` will not be valid. Whereas, in the reformulated constraint the second condition will only be checked if `how[p2]>0`, which makes it more efficient, and, as a result, the solving time became faster compared to the original model.

The top clause {`how[1]=-1, how[2]=-1, how[3]=-1, how[4]=-1, how[5]=-1, how[1]=-2, how[2]=-2, how[3]=-2, how[4]=-2, how[5]=-2, how[6]`≤`0, how[6]=3`}, indicates that pizza 6 cannot be obtained for free by voucher 1 or 2 ({`how[6]`≤`0, how[6]=3`}), by buying pizzas 1, 2, 3, 4 and 5 using voucher 1 or 2. This happens because pizza 6 in the most expensive pizza in the clause, therefore it can never be obtained for free by any voucher. The opposite of this fact holds for the cheapest pizza: by buying the cheapest pizza using any voucher, no other pizza can be obtained for free using the same voucher (pizza 2 in this instance). Then, they realised that these facts about the most expensive and the cheapest pizzas are not explicitly expressed in the model. Therefore, they added the following redundant constraints to the model:

```
% the most expensive pizza can never be bought with a voucher
constraint forall(p in PIZZA)
(if forall(o in PIZZA where o!=p)(price[p]>price[o])
then how[p]<=0 else true endif);
% the cheapest pizza can never be used with a voucher
constraint forall(p in PIZZA)
(if forall(o in PIZZA where o!=p)(price[p]<price[o])
then how[p]>=0 else true endif);
```

The aim of this thesis is to automate and improve Shishmarev et al.'s method. For this purpose, our main steps can be summarised as follows:

1. Rename the learnt clauses to be expressed in terms of model-level variables (e.g. transform the clauses in Table 2.2 into Table 2.1).
2. Simplify the learnt clauses to reduce their length, and detect more clauses with similar structures which can help us perform the next steps more efficiently.
3. Connect the clauses to the model constraints to detect the constraint involved in creating the clauses.
4. Detect the clauses with similar structures, and condensely generalise them into patterns, this helps us improve their ranking technique, and also represent the clauses in a form of a constraint that can be suggested to the user to be added to the model
5. Learn information for each pattern, that may serve as a condition to make each pattern correct(e.g. the relationship between prices that was manually learnt by Shishmarev et al.)

This thesis presents the steps mentioned above in more details. In particular, steps 1 to 3 are discussed in Chapter 3, and steps 4 and 5 correspond to Chapters 4 and 5.

2.6 Connecting Clauses to Model Constraints

As mentioned in Section 2.5, Shishmarev et al. used the learnt clauses to identify the need to strengthen the constraint in Line 16 of the model. This need was discovered by realising that some of the top clauses were direct consequences of this single constraint. To do this, Shishmarev et al. manually linked the learnt clauses (or more accurately, their literals) to the model constraints they were derived from. Automating this method requires us to automatically connect the clauses to the model constraints. This is more complex than it initially appears due to the complex process performed in MiniZinc.

A MiniZinc model and its input data are compiled into a low-level model program (FlatZinc). The FlatZinc file combines the information in the model with that contained in the input data file and expresses it in a way that can be given as input to different solvers. During compilation new variables are introduced, model constraints are decomposed and loops are unrolled. Learning solvers refer to the flattened variables to generate clauses. Thus, it is challenging to map the clauses learnt by the solver to the original model constraints and variables.

To achieve this, we will make use of the technique developed by Leo et al. [43], which is described in this section. Leo et al. developed a technique to trace the model-level variables and constraints through the compilation, and map them to their corresponding solver-level variables and constraints. For this purpose, they assign a unique identifier to each variable and constraint in the FlatZinc file, which they refer to as *variable paths* [42] and *constraint paths* [43], respectively. Each identifier describes the path that the compiler took when compiling a MiniZinc instance to FlatZinc, from the actual model to the point where new variables or constraints are introduced. For example, the following is a (simplified) constraint path:

```
(14:12-15:61 forall:p1=1,p2=6)(14:36-15:60 ->)(14:37-14:74 /\)(14:37-14:74 clause)
```

Each of its components has two parts: four numbers denoting the span of text in the MiniZinc model that the expression came from (with format from line:column-to line:column); and a textual description of what the expression represents. The above path represents a `clause` that was inserted into the final FlatZinc, as a result of encoding a negated (thus the `clause` part) conjunction `/\` appearing in the left hand side of the implication (`->`) that appears in the `forall` loop in line 17 of the `free pizza` provided by Figure 2.4, with index variables `p1=1` and `p2=6`. This corresponds to the expression `how[p1] < how[p2]`. The path provided above refers to the iteration of the for loop in which `p1=1` and `p2=6`. As we see in Chapter 3, our aim is to relate each clause to the model-level constraints not to the instances. Thus, we will need to generalise this path to the model. For this purpose, one simply needs to remove the identifying information that makes the path unique to an instance variable/constraint. In our example this can be achieved by removing the concrete values 1 and 6 for `p1` and `p2`, thus grouping all iterations of the loop.

2.7 Constraint Acquisition Systems

As mentioned before, one of our steps to automate the work of Shishmarev et al. is to combine model information with patterns to learn information that is not present in a clause, which may serve as a condition for occurrence of the patterns. To achieve this, we use two different methods, as you see in the Sections 5.3 and 5.4 of Chapter 5. In the second method we used a constraint acquisition algorithm called ConAcq that can help us learn facts for the patterns. In this Section, we briefly discuss the constraint acquisition systems and describe the ConAcq algorithm.

Modelling a constraint problem can be challenging for non-expert users as there are many possible ways to model a problem, and the different approaches can hugely impact its solving time [14]. Constraint acquisition systems help non-expert users model their target problems. Given a set of solutions, non-solutions and a constraint library, a constraint acquisition system learns a constraint network that is consistent with the solutions and non-solutions [12].

There are two main classes of constraint acquisition systems: passive and active. In passive systems [39, 12], the user chooses the set of examples in advance and independently of the acquisition process, whereas active systems [14] assist the user to choose more helpful examples to reduce the number of required examples.

As we see in Chapter 5, our method has a full set of examples and non-examples. Thus, passive constraint acquisition systems are more suitable for our application. More specifically, we chose ConAcq because it is simpler than most of the other algorithms, and thus, easier to implement.

2.7.1 The ConAcq Algorithm

ConAcq [12] is an approach to constraint acquisition based on the SAT-solving and version space learning [47] paradigms. Given a set of solutions (positive examples), non-solutions (negative examples) and a constraint library (a set of binary constraints) provided by the user. The system outputs a constraint network learnt from the library. The ConAcq algorithm is as follows (from [12]):

Algorithm 1 The ConAcq algorithm from [12]

```
   Input: examples($E^+$, $E^-$) and a constraint library $B$
   Output: a set of clauses $K$
   $K \leftarrow \emptyset$
1: foreach e in examples do
2:     $K_e \leftarrow \{b_{ij} \in B$ : e does not satisfy $b_{ij}\}$
3:     if e $\in E^-$ then $K := K \bigwedge(\bigvee_{b_{ij} \in K_e} b_{ij})$
4:     if e $\in E^+$ then $K := K \bigwedge(\bigwedge_{b_{ij} \in K_e} \neg b_{ij})$
5:     if UnitPropagation(K) detects $\perp$ then Return("collapsing")
```

The inputs of the algorithm are: a set of solutions (E^+) and non-solutions (E^-), and a constraint library (B) that contains a set of binary constraints. Each constraint b_{ij} in the library is a user-defined binary relation between two decision variables x_i and x_j, such as $x_i < x_j$.

The algorithm outputs K that is a conjunction of clauses that are consistent with solutions and non-solutions, where each literal in each clause is a binary constraint.

Initially K is empty. In Lines 1 to 5, the algorithm iterates through each example e, and Line 2 builds K_e, which consists of all the constraints b_{ij} in the constraint library B that reject e. Line 3 checks if e is a negative example, in that case, Line 3 builds a clause as the disjunction of the constraints b_{ij} that reject the negative example, represented by the expression: $K := K \bigwedge(\bigvee_{b_{ij} \in K_e} b_{ij})$. This ensures that at least one of the constraints in the learnt constraint network rejects the negative example. Line 4 checks if the example is positive, if that is the case, Line 4 ensures that none of the constraints that reject the positive example are valid constraints. The algorithm then builds a conjunction of the negated constraints, that reject the example, represented by $K := K \bigwedge(\bigwedge_{b_{ij} \in K_e} \neg b_{ij})$. Line 5 calls the *UnitPropagation*() method which performs unit propagation on constraint network K, which is a propagation technique commonly used by SAT-solvers (see [77]), that is used here to detect if the constraint network K is satisfiable, if the network is inconsistent the algorithm returns *collapsing*, otherwise, it returns K.

Example 4. Consider a CSP with two decision variables x_1 and x_2 both with domain $D = \{1, 2\}$, and the parameter $a = [70,\ 20]$, which is indexed by x_1 and x_2. For example, $x_1 = 2$ refers to $a[2]$. The solutions, non-solutions and constraint library are as follows:
$E^+ = \{(x_1 = 1,\ x_2 = 1),\ (x_1 = 2,\ x_2 = 1),\ (x_1 = 2,\ x_2 = 2)\}$
$E^- = \{(x_1 = 1,\ x_2 = 2)\}$
$B = \{a[x_1] \geq a[x_2],\ a[x_1] < a[x_2],\ a[x_1] = a[x_2],\ a[x_1] \neq a[x_2],\ x_1 \geq x_2,\ x_1 < x_2,\ x_1 = x_2,\ x_1 \neq x_2\}$

Initially K is empty. The algorithm iterates through the four examples. Assume that K^{+ij} represents the expression $\bigwedge_{b_{ij} \in K_e} \neg b_{ij}$ in line 4 of the algorithm for the positive example $(x_1 = i, x_2 = j)$, and K^{-ij} represents the expression $\bigvee_{b_{ij} \in K_e} b_{ij}$ for the negative example $(x_1 = i, x_2 = j)$ in line 4.

The following expressions are built for the examples:
K^{+11}=$\{\neg(a[x_1] < a[x_2]) \wedge \neg(a[x_1] \neq a[x_2]) \wedge \neg(x_1 < x_2) \wedge \neg(x_1 \neq x_2)\}$
K^{+21}=$\{\neg(a[x_1] \geq a[x_2]) \wedge \neg(a[x_1] = a[x_2]) \wedge \neg(x_1 < x_2) \wedge \neg(x_1 = x_2)\}$
K^{+22}=$\{\neg(a[x_1] < a[x_2]) \wedge \neg(a[x_1] \neq a[x_2]) \wedge \neg(x_1 < x_2) \wedge \neg(x_1 \neq x_2)\}$
K^{-12}=$\{a[x_1] < a[x_2] \vee a[x_1] = a[x_2] \vee x_1 \geq x_2 \vee x_1 = x_2\}$

As a result, the output will be: $K := K^{+11} \wedge K^{+21} \wedge K^{+22} \wedge K^{-12}$. In this example, the algorithm outputs $x_1 \geq x_2$, since this is the only constraint from the constraint library that satisfies all the expressions above. Note that $\neg(a[x_1] < a[x_2]) \wedge \neg(a[x_1] \geq a[x_2])$ does not represent false it simply represents the absence of those two constraints in the network (and thus in the model).

The ConAcq algorithm is an important part of this thesis, since it is used in one of our methods to generate facts that is described in Chapter 5.4.2. This algorithm helps us detect the facts that are valid for a given pattern among a network of candidate facts.

2.8 Summary

In this Chapter we discussed the background knowledge required for the remaining chapters. Section 2.2 described the basics of COPs/CSPs and how they are modelled and solved. Section 2.3 discussed CP-solvers that are a powerful and generic technique to solve COPs/CSPs, and this thesis focuses on them. Section 2.4 described learning solvers that are a class of CP-solvers, which are very powerful and used in our method. Section 2.5 summarised a technique developed by Shishmarev et al. that they show the clauses learnt by learning solvers can be used to improve the model, and it then described how our method automates and improves their method. Section 2.6 described a technique developed by Leo et al. that connects the learnt clauses to the model constraint that were involved in creating them. Their method is used in this thesis to connect the learnt clauses to the model constraints. Section 2.7, described constraint acquisition systems. In particular, the ConAcq algorithm which we used to learn facts for the patterns.

Chapter 3

Contextualising and Simplifying Clauses

3.1 Introduction

As mentioned in Chapter 2, a MiniZinc model and its associated data must be compiled into a FlatZinc program before it can be given to a solver to be solved. During compilation new variables are introduced, model constraints are decomposed and loops are unrolled. Since solvers can only reason about the program they have been given, learned clauses will be expressed in terms of the FlatZinc program. Shishmarev et al. manually renamed the clauses, to be able to understand, and further analyse them. For example, for the `free pizza` problem they transformed Table 2.1 into Table 2.2, provided in Chapter 2.

To obtain the clauses we used the same method as Shishmarev et al. as described in 2.5. Our first step towards automation is to rename the clauses. For example, consider the clause {`X_INTRODUCED_1_>1, X_INTRODUCED_2_≠5, X_INTRODUCED_3_>7`} which is inferred by Chuffed for the `golomb ruler` model of figure 1 with `m=5`. The variables `X_INTRODUCED_1_, X_INTRODUCED_2_ and X_INTRODUCED_3_` refer to the model-level variables `mark[2], mark[3]` and `mark[4]`. Thus, the solver-level clause can be translated to: {`mark[2]>1, mark[3]≠5, mark[4]>7`}. This helps us in different ways: After renaming, our method will be able to detect patterns, as you will see in Section 4.2. Also, it enables our method to connect variables across instances, as you will see in Section 4.3. Furthermore, It enables us to link the patterns to the model information to learn facts for them, which you will see in Section 5.

Once variables are renamed, we simplify the clauses based on their semantics. Let us have a look at one of the clauses for the `free pizza` problem (Table 2.2), {`how[6]≤2, how[6]≥4, how[1]≠-3, how[1]≥-2`}, this can be simplified to {`how[6]≠3, how[1]≠-3`}. This helps us reduce the length of the clauses, and also detect more clauses that share a similar structure, which later helps us detect more accurate patterns, and more precise ranking for each pattern.

After renaming and simplifying the clauses, our next step is to connect them to the model constraints. This is achieved by tracing the origin of the FlatZinc-level constraints that were responsible for introducing literals to the clause. This helps us improve the model, by either reformulating the model constraints or adding a redundant constraint to the model to express the same information in a stronger way.

The structure of this Chapter is as follows: Section 3.2 describes our method that renames the literals by mapping the solver-level literals to their corresponding model-level literals. Section 3.3 discusses our method that simplifies the clauses. Section 3.4 discusses the limitations of our methods for renaming and simplifying the clauses. Section 3.5 describes our method that detects the model constraints that were involved in creating the clauses.

3.2 Renaming Literals

As mentioned before, the clauses inferred by learning solvers are expressed in terms of solver-level variables rather than model-level variables. Consider the clauses for the `free pizza` problem provided in Table 2.2, these clauses are already the result of significant manual interpretation and analysis. For example, the 4th clause came from renaming (and simplifying) the solver-level clause {`X_INTRODUCED_4_≠-3`, `X_INTRODUCED_2_≠3`, `X_INTRODUCED_2_≤2`}, where the names `X_INTRODUCED_4_` and `X_INTRODUCED_2_` were introduced by the MiniZinc compiler for the model variables in array positions `how[5]` and `how[3]`, respectively.

Thus, our first step towards automation is to transform clauses to refer to model-level variables and expressions. For simple renamings as in the example above, the MiniZinc compiler already provides a mapping from solver-level to model-level names.

New variables may, however, also be introduced when flattening *expressions*. For example, the compiler may introduce an auxiliary variable for the result of an intermediate addition, or for variables introduced by a `let` expression connecting such variables back to model-level names is more complex. We propose to do this by using variable paths [42] (defined and explained in Chapter 2.6). Consider, for example, the literal `X_INTRODUCED_244_=true`, which appears in one of the clauses generated by Chuffed for `free pizza`. Since variable `X_INTRODUCED_244_` does not correspond directly to any model-level variable, we must use its path to help deduce its meaning. The path provided by the MiniZinc compiler for this variable is:

$$\boxed{\textit{14:12-15:61}\ \texttt{forall:p1=1,p2=6}}\boxed{\textit{14:36-15:60}\ \texttt{->}}\boxed{\textit{14:37-14:74}\ \texttt{/\textbackslash}}$$
$$\boxed{\textit{14:37-15:74}\ \texttt{clause}}\boxed{\textit{14:58-14:74}\ \texttt{=}}\boxed{\textit{14:58-14:74}\ \texttt{int\_lin\_eq}}$$

The final entry in the path shows the location in the model of the expression that corresponds to `X_INTRODUCED_244_`: Line 14, columns 58–74, which has expression `"how[p1]=-how[p2]"`. The path also shows that the expression is located within the `forall` that spans Lines 14:12-15:61, within the implication (`->`) that spans Lines 14:36-15:60, within the conjunction (`/\`) that spans Line 14:37-14:75, within the `clause` that resulted from encoding the negated conjunction, and within the = that spans Line 14:58-14:74. Since the path ends with an `int_lin_eq` constraint (integer linear =), we can deduce that the introduced variable is the Boolean control variable for the reified version of expression `"how[p1]= -how[p2]"` in the model. Since the path also records the values of loop variables `p1` and `p2`, these can be automatically substituted, yielding `"how[1]= -how[6]"=true`.

3.3 Simplifying Literals and Clauses

The length of the clauses can be reduced, and patterns can be better detected by simplifying their literals based on their semantics. We do this in two different ways. First, we simplify literals whose variables were introduced by the compiler, and which correspond to Boolean expressions in the model, as follows:

positive: if the right-hand side of a literal is true (e.g. `=true, =1, >=1`), and the left-hand side is a Boolean expression of the form e_1 *op* e_2, where *op* is a binary operator, then we can simply remove the right-hand side.

negative: if the right-hand side of a literal is false (e.g. `=false, =0, <=0`), and the left-hand side is a Boolean of the form `x` *op* `v`, where `x` is a variable, *op* is a binary operator and `v` is an integer value, then we can negate the left-hand side and remove the right-hand side.

For example, literal `"how[1]<how[6]"=true` becomes `how[1]<how[6]`, and literal `"how[1]=-1"<=0` (which comes from the sum of reified equality constraints in line 12 of the model) becomes `how[1]`$\neq$`-1`. Simplifying literals with more complex left-hand sides (i.e., arbitrary MiniZinc expressions), requires a deeper integration with the MiniZinc compiler and is left for future work.

Second, we eliminate from each clause any literal that entails (i.e., logically implies) other literals in the clause, since $A \vee B \vee C$ is equivalent to $A \vee B$, if C entails B. This makes the clause easier to understand and, as shown in Section 4, makes it easier to automatically detect clause patterns. We automatically simplify a clause by applying the following rules to literals that operate on the same variable x:

$\neq$: a literal of the form $x \neq v$ is entailed by any other literal $x = v'$ such that $v \neq v'$, and by any literal $x \leq v'$ ($x \geq v'$) such that $v' < v$ ($v' > v$). Thus, those other literals are eliminated. For example, clause $\{x = 1, x \leq 1, x \geq 4, x \neq 2, \dots\}$ is simplified to the equivalent clause $\{x \neq 2, \dots\}$.

$\geq$ ($\leq$): a literal of the form $x \geq v$ ($x \leq v$) is entailed by any other literal $x \geq v'$ ($x \leq v'$) s.t. $v < v'$ ($v > v'$). Thus, we only keep the literal with the lowest (highest) bound. For example, clause $\{x \leq 1, x \leq 2, x \geq 3, x \geq 4, \dots\}$ is simplified to the equivalent clause $\{x \leq 2, x \geq 3, \dots\}$.

$\leq\geq$: if a clause contains two literals $x \leq v_1$ and $x \geq v_2$, s.t. $v_2 - v_1 = 2$, then these two literals can be replaced by a single literal $x \neq v_2 - 1$. For example, clause $\{x \leq 1, x \geq 3, \dots\}$ is simplified to the equivalent clause $\{x \neq 2, \dots\}$.

Applying these rules to our clause {`how[5]`≠`-3`, `how[3]`≠`3`, `how[3]`≤`2`} yields the one ranked 4th in Table 2.2: {`how[5]`≠`-3`, `how[3]`≠`3`}, since `how[3]`≤`2` entails `how[3]`≠`3`, and is thus eliminated by the ≠ rule.

Note that all simplifications are done after renaming the introduced variables. This is useful, as literals that reference different introduced variables may later become a single literal. This happens often in `free pizza` as, for example, the `sum` function expects integer variables, but in the model receives Booleans. Each Boolean variable is coerced by the compiler to be integer by introducing a new integer variable and then posting a `bool2int` predicate which equates the Boolean to the integer one. For the expression `sum(p in PIZZA)(how[p]=-v)`, both the original Boolean variables and the introduced integer variables refer to the expression `"how[p]=-v"` in the model. However, before renaming, the literals may appear different (e.g., `X_INTRODUCED_45_=false` and `X_INTRODUCED_53_<=0`) even though they mean the same thing (`how[p]`≠`-v`). By performing the simplification after the renaming, both variables are known to be (or come from) a Boolean expression, and are thus simplified to `how[p]`≠`-v`. Interestingly, if these simplifications had been applied to the clauses in Table 2.2, Shishmarev et al. would have realised that clauses 2, 8 and 9, can be further simplified to {`how[7]`≠`3`, `how[1]`≠`-3`}, {`how[6]`≠`3`, `how[1]`≠`-3`} and {`how[5]`≠`-3`, `how[4]`≠`3`}.

3.4 Limitations of Renaming and Simplifying Clauses

One of the limitations of the approach described in the previous sections is that the reconstruction of expressions is purely syntactic. As a result, if a path points to a span of text that is in a user-defined function or predicate, the text will reference the parameters of this function or predicate, rather than to the top-level model variables which were passed as arguments. For example, a decomposition of the `alldifferent(array[int] of var int: x)` predicate present in the MiniZinc library, will contain the expression `x[i]`≠`x[j]`. If the model contains a call to this predicate, such as `alldifferent(y)`, our purely textual approach is not able to rename `x` to `y` in any clauses resulting from this constraint. Therefore, modellers will not be able to distinguish between different invocations of the same predicate. To avoid this problem, for our current experiments, we manually hoisted these variables into the model and replaced any calls to these functions or predicates with their definitions in terms of these model-level variables. For the example above, the decomposition of `alldifferent(array[int] of var int: x)` predicate for an array of integers is as follows:

```
predicate all_different_int(array[int] of var int: x) =
    forall(i,j in index_set(x) where i<j)(x[i]!=x[j]);
```

To be able to rename `x` to `y`, we can replace `constraint alldifferent(y)` in the model with its decomposition, and obtain the following:

```
constraint forall(i,j in index_set(y) where i<j)(y[i]!=y[j]);
```

A second limitation of our approach is that, variable paths approach do not always provide us with information about the expression types that are derived from the text in a function/predicate. For example, for the `radiation`[1] problem, after renaming and simplifying the clauses, the method obtains the following two literals: `'Q[2,6,2]'>=2`, `Q[2,6,2]>=5`. The quoted literal is directly obtained from the textual model and the type of variable `Q` is unknown to the simplifier, even though a manual inspection can easily show it refers to the decision variable `Q`, which is the same variable referred to by the second literal in the clause. Since the current implementation cannot be certain that `'Q[2,6,2]'` is the same as `Q[2,6,2]`, certain simplifications are unavailable and we cannot infer that these two literals could be simplified to just `Q[2,6,2]>=2`.

These limitations could be mitigated by instrumenting the MiniZinc compiler to collect and output more information about compilation. For instance, the mapping of the variable `y` to the local variable `x` is explicit to the compiler in the example above, but the compiler does not output it in the FlatZinc file. This relationship is not present in the variable path. Extending the compiler to record extra information (such as the variable type, and the mapping of a variable to a local variable in a function/predicate call) would make it possible for us to obtain more accurate variable names and types.

3.5 Connecting Clauses to the Constraints in the Model

One of the main achievements of Shishmarev et al. was to use the learnt clauses to identify the need to strengthen the constraint in line 12 of the model. This need was discovered by realising that some of the top clauses were direct consequences of a single constraint; the one in line 12. To do this, Shishmarev et al. manually linked the learnt clauses (or more accurately, their literals) to the model constraints they were derived from.

To automate this step, we instrumented Chuffed to record the solver-level constraint directly responsible for adding any literal to the clause database. That is, for each explanation $S \rightarrow \ell$ added by a constraint with identifier id_c, we record that ℓ was directly generated from id_c. Then, when a clause Cl is generated, we obtain the constraint identifier of each literal ℓ in Cl, ask that constraint to provide us with the explanation S that was used to generate ℓ, and recursively apply the same method for all literals in S. This allows us to trace back all constraints involved in generating the literals of Cl.

Consider, for example, {`how[5]`≠`-3`, `how[3]`≠`3`}, for which our method identifies a single solver-level constraint as responsible for all literals. Using constraint paths, our method can trace it back to the expression `how[p1]=-how[p2]` on line 14 of `free pizza`, with loop variables `p1=5` and `p2=3`. Thus, our automatic method successfully identifies the line in which the constraint responsible for the clause appears.

3.6 Summary

This Chapter first discussed our method that renames the clauses to be expressed in terms of model-level variables rather than solver-level. For this purpose, we developed a method

[1]`https://github.com/MiniZinc/minizinc-benchmarks/tree/master/radiation`

that uses variable paths technique, developed by Leo et al. [42], which traces the model-level variables throughout compilation and keeps track of the original variable names. Renaming literals have several advantages: First, it enables us to detect general patterns from the clauses, because compiler introduces a new variable for each array position of a decision variable (e.g. for the array positions `how[5]` and `how[3]`, it introduces the variables `X_INTRODUCED_4` and `X_INTRODUCED_5`), thus without renaming we will not be able to detect the model-level decision variables that are involved in each clause, and that is required for our method to generate patterns, as you will see in Chapter 4.2. Also, renaming helps us connect the variable names across different instances. This happens because, the same model-level variable names may correspond to different solver-level variable names, and without renaming we will not be able to match the same decision variables. This becomes clearer in Chapter 4.3. Afterwards, we discussed our method for simplifying clauses. This is achieved by applying some simplification rules on the clauses. The main purpose of this step is to reduce the length of the clauses, which improves the efficiency of our framework to generate patterns and infer facts, as you will see in Chapters 4 and 5, respectively. Also, this helps us obtain more patterns, and a more accurate ranking for them. In addition, since simplifications are performed after renaming the introduced variables, some of the literals may later become a single literal. Lastly, we discussed our method for connecting clauses to the model constraints that add any literal to the clause. To achieve this, we instrumented Chuffed to record the solver-level constraint responsible for adding any literal to each clause. If a single clause is responsible for the occurrence of the clause, it may suggest that, this constraint is not propagating strongly, and we may reformulate it to improve its propagation. However, if multiple clauses are responsible for the occurrence of a clause, it may suggest that information is not explicitly expressed in the model, and by adding effective redundant constraints, we may be able to fix this issue.

Chapter 4

Finding Patterns among Clauses

4.1 Introduction

As described in Chapter 3, our automatic method can make a clause clearer, simpler and can visually connect it to the constraints it came from. However, a single clause may not be worth exploring, even if it led to a considerable search reduction. This is because clauses are in practice quite complex, even after renaming and simplification, and a single clause might only appear in a particular instance or search. It would therefore be preferable for the modeller to have stronger evidence before starting to explore the model constraints that are responsible for creating the top clauses. The evidence can be much stronger if several clauses with a similar *pattern* can be found to have significantly reduced the search. For example, Shishmarev et al. [68] focused on clause 4 ({`how[5]`≠`-3`, `how[3]`≠`3`}) in Table 2.2 not only because it was one of the top clauses and it was short (and thus easier to understand), but importantly, because it was part of several *similar clauses* in the top 10 (which they identified as clauses 5, 6, 7, and 10).

It is easy to show that these five clauses share the same *pattern*: {`how[A]`≠`-B`, `how[C]`≠`B`}, where `A` and `C` are pizzas, and `B` is a voucher. For example, clause 4 can be obtained by name/constant mapping {`A/5`, `C/3`, `B/3`}. In fact, with the simplification rules described in Chapter 3, all clauses in Table 2.2 except 1 and 3 can be shown to share this pattern and to come from the same constraint. Grouping clauses with the same pattern can help modellers in two ways: (a) while individual clauses may not seem to contribute much to the overall search reduction, a group of clauses with the same pattern may do so; and (b) a pattern may identify a general constraint, i.e., something that is true for a whole range of parameters, and may therefore suggest a redundant constraint that can be added to the model.

To obtain the patterns that are more likely to apply to the whole model, not just one instance, our method considers multiple instances of the problem. It also considers multiple search strategies, as there may exist some clauses that have a high rank but can only be discovered using a specific search strategy. After obtaining the patterns across different searches and instances, our method further clusters the clauses under each pattern based on the constraints responsible for creating them. This helps us detect whether all the clauses in the cluster are coming from a single constraint or multiple ones. If one constraint is responsible for the occurrence of a highly ranked cluster, it may suggest that the constraint does not strongly propagate, and we may be able to fix this by reformulating that constraint. However, if multiple constraints are associated to a cluster, it may be a sign that information is not explicitly expressed in the model, and to fix this we can add a redundant constraint to the model.

The structure of this chapter is as follows: Section 4.2 introduces our basic method for automatically generating patterns. Section 4.3 discusses how to find patterns across multiple search strategies and instances. Section 4.4 discusses the clustering of clauses under each pattern, and Section 4.5 discusses the limitations of our approach and, lastly, Section 6.5 provides a summary for this chapter.

4.2 Finding Patterns among Clauses

We say that clause c_1 is more general than the clause c_2 ($c_1 \preceq c_2$) if and only if all the clauses that are covered by c_1 are also covered by c_2 [25], and the most specific (or least) generalisation (MSG) of a set of clauses is the least general clause that is more general than all the clauses in the set [56]. We define a *pattern* as the MSG of a set of clauses. For example, suppose we have a set of clauses, containing {`X[2]`≠`X[3], X[2]=5`}, {`X[7]`≠`X[8], X[7]=9`} and {`X[1]`≠`X[4], X[1]=5`}. The clauses {`X[A]`≠`X[B], X[D]=C`} with mappings {`A/2`, `B/3`, `D/2`, `C/5`}, {`A/7`, `B/8`, `D/7`, `C/9`} and {`A/1`, `B/4`, `D/1`, `C/5`}, and the clause {`X[A]`≠`X[B], X[A]=C`} with mappings {`A/2`, `B/3`, `C/5`}, {`A/7`, `B/8`, `C/9`} and {`A/1`, `B/4`, `C/5`} are more general than all the clauses in the set. To automatically obtain patterns, our method first extracts the names of the model's decision variables that appear in each clause. This helps us detect the *objects* (constants and pattern variables) that appear in the clause. For instance, assuming that we have the clause {`how[8]=-3, how[3]=-2, used[3]=false`}, our method first extracts the decision variable names {`how, used`}. In this example, the clause refers to the pizzas 8 and 3, since the decision variable `how` is indexed by `pizzas`. Also, vouchers 2 and 3 appear in this clause, because the decision variable `how` is a `voucher` parameter (`how[8]` is assigned to voucher 3 in the first literal), and the decision variable `used` is indexed by `vouchers`. We refer to the type of objects such as pizza and vouchers as *type*.

To be able to automatically detect the decision variable names and the types that appear in each pattern, For this purpose, we first manually modified the MiniZinc model to express the actual type of decision variables (e.g. `pizza`, `voucher`) as enumerated types, and we also instrumented the MiniZinc compiler to output information about decision variables and parameters that appear in the model, we refer to this as *type information*. Specifically, it outputs the enumerated type/types of the variables and parameters, and also the types of their indices (e.g. it provides us the information that `how` is indexed by pizzas, and its enumerated type is voucher). Using the type information, our method first extracts the decision variable names that appear in the pattern, for our example it first extracts {`how, used`}. From the type information our method detects that the decision variable `how` is indexed by pizzas and its enumerated type is a voucher, and decision variable `used` is indexed by vouchers, and its type is Boolean. Then, it extracts all the pizza and voucher objects, for this example, the pizza objects are `A` and `C`, and the voucher objects are `B` and `D`. Also, using the type information, it extracts all the model parameters that are indexed by these objects (e.g. `price` is indexed by pizzas, `price[1]` represents the price of pizza 1). Algorithm 3 describes our method of detecting the objects of the same type, and the parameters that are indexed by these types, and how this information is used to learn facts.

To be able to detect the decision variable names and the object types associated to them, we need to first extract all the possible model decision variable names, the type of their indices and the type of their domains. For this purpose, we first modified the MiniZinc model to express the actual type of decision variables (e.g. `pizza`, `voucher`) as enumerated types (manually). We also instrumented the compiler to output information about the decision variables and parameters that appear in the model. Specifically, it outputs the type/types of the variables and parameters themselves and also the type of their indices. This type information can help us obtain correct patterns.

Example 5. Consider the clauses {`how[8]=-3, how[3]=-2, used[3]=false`} and {`how[6]=-4, how[4]=-2, used[4]=false`}. Without knowing the type information, we could obtain the pattern {`how[A]=-B, how[B]=-2, used[B]=false`}, with mappings {`A/8, B/3`} and {`A/6, B/2`}. However, using type information, we know that the second `B` is a pizza, whereas the first and third `B` are vouchers. They should therefore not be mapped to `B`; instead we need to assign different variables for pizza and voucher objects, obtaining the pattern {`how[A]=-B, how[C]=-2, used[B]=false`}, with mappings {`A/8, B/3, C/3`} and {`A/6, B/2, C/2`}, where `A` and `C` are pizzas and `B` is a voucher.

After extracting the object types that appear in each pattern, our next step is to find similar clauses that can form a pattern. To achieve this, our method sorts the literals in each clause by variable name, operator and constant value. Finally, for all clauses with the same length that have the same sequence of literal operators, and the same sequence of variable names, it computes the MSG for all its subsets, based on the algorithm designed by Plotkin et al. [57].

Example 6. Given the following two (renamed, simplified and sorted) clauses for our `free pizza` instance:
{`how[1]=-1, how[2]=-1, how[3]=-1, how[4]=-1, how[6]`$\neq$`1`} and
{`how[1]=-2, how[2]=-2, how[5]=-2, how[7]=-2, how[6]`$\neq$`2`}, have the same sequence of literal operators ({`=, =, =, =,` $\neq$ }) and the same sequence of decision variable names ({`how[_], how[_], how[_], how[_], how[_]`}). Thus, our method would compute the pattern {`how[1]=-A, how[2]=-A, how[B]=-A, how[C]=-A, how[6]`$\neq$`A`}, which can be mapped back to the clauses by applying the mappings {`A/1, B/3, C/4`} and {`A/2, B/5, C/7`}, respectively, where `B, C` are known to be pizzas and `A` a voucher. However, neither of these clauses would form a pattern with {`how[1]=-2, how[2]=-2, how[6]`$\neq$`2`}, as it has a different number of literals, or with clause {`how[1]=-1, how[2]=-1, how[3]=-1, used[1]=true, how[6]`$\neq$`1`}, as it has a different sequence of variable names.

Formally, each clause `cl` is stored as a tuple ⟨`id`, `seq_lits`, `red`, `cons`⟩ containing the clause identifier `id`, its renamed, simplified and sorted sequence of literals `seq_lits`, its associated search reduction `red`, and the constraints `cons` that generated it. A pattern `pt` is stored as a tuple ⟨`p_id`, `p_seq_lits`, `maps`, `p_red`, `p_cons`⟩ containing the pattern identifier `p_id`, its sequence of literals `p_seq_lits`, a set `maps` of tuples of the form ⟨`map`, `id`⟩, where applying `map` to `p_seq_lits` yields the sequence of literals in the clause identified by `id`, the total search reduction `p_red` achieved by its clauses (the sum of the `red` of each clause identified by `maps`), and the set of constraints `cons` that generated any of its clauses (the union of the `cons` of each of the clauses identified by `maps`).

Note that a clause may appear in many different patterns. In fact, this will often be the case, as our algorithm can be seen as computing a pattern for each subset of the set of (renamed, simplified and sorted) clauses that have the same sequence of (a) literal operators and (b) variables types. The algorithm starts with the set of renamed, simplified and sorted `clauses` computed for a given instance as described in previous sections, and an empty set of `patterns`. Then, for every clause `cl` = ⟨`id`, `seq_lits`, `red`, `cons`⟩ in `clauses` it performs the following steps:

1. Transform `cl` into pattern `pt` = ⟨`pid`, `seq_lits`, {⟨`[]`, `id`⟩}⟩, where `p_id` is a new identifier.
2. Set `new_patterns` to $\emptyset$
3. For each pattern `pt`′ = ⟨`p_id`′, `p_seq_lits`, `maps`, `p_reds`, `p_cons`⟩ in `patterns` with the same sequence of literal operators and variable types as `seq_lits`:
   (a) Find a most specific generalisation `p_seq_lits`′ for `seq_lit` and `p_seq_lits`, with associated mapping `maps`. Algorithm 2 shows our steps to compute the MSG.
   (b) Add ⟨`pid`′, `p_seq_lits`′, `maps` ∪ ⟨`map`, `id`⟩, `p_red`+`red`, `p_cons`∪`cons`⟩ to `new_patterns`
4. Merge `patterns` and `new_patterns`. While the pattern `pt` is known to be different from all others in `patterns` (since `pt` is essentially a clause and no two clauses are identical), it is possible to have a pattern `p_1` in `new_patterns` with a sequence of literals identical (up to variable renaming `ren`) to pattern `p_2` in `patterns`. We can detect this when merging `patterns` and `new_patterns` in step 4, and simply merge `p_1` and `p_2` (by first applying `ren` to them, and then computing the unions of their mapping and constraint sets, and summing up their reductions).
5. Add `pt` to `patterns`.

Once all patterns are computed, our method ranks the results by the amount of reduction associated with each pattern and presents it to the modeller. This allows modellers to notice clauses that may not result in significant search reductions when considered individually, but do when considered together as a group in a pattern.

Algorithm 2 shows our method to compute the MSG, given a pattern and a clause. The inputs of the algorithm are a sequence of literals `seq_lit` corresponding to the clause

Algorithm 2 Finding MSG of a clause and a pattern

```
    Input: seq_lit, p_seq_lits
    Output: p_seq_lits′

 1: var ← "A"
 2: var_old ← $\emptyset$
 3: p_seq_lits′ ← p_seq_lits
 4: types ← get_types(p_seq_lits)
 5: rep_maps ← $\emptyset$
 6: objects1, objects2 ← $\emptyset$
 7: foreach type $\in$ types do
 8:     objects1←extract_objects(seq_lit, type)
 9:     objects2←extract_objects(p_seq_lits, type)
10:     length ← |objects1|
11:     foreach index $\in$ {0.. length} do
12:         if objects1[index] $\neq$ objects2[index] then
13:             if ⟨objects1[index], objects2[index]⟩ $\notin$ rep_maps then
14:                 rep_maps ← rep_maps $\cup$ {⟨objects1[index], objects2[index]⟩: var}
15:                 update(p_seq_lits′, index, var)
16:                 increment(var)
17:             else
18:                 var_old ← rep_maps[⟨objects1[index], objects2[index]⟩]
19:                 update(p_seq_lits′, index, var_old)
20: return(p_seq_lits′)
```

and p_seq_lits corresponding to the pattern, and the algorithm outputs p_seq_lits′ corresponding to the generated pattern. Line 1 introduces var which represents a pattern variable and initialises it to "A". Line 2 introduces and updates var_old, which represents an existing pattern variable. Line 3 introduces and initialises p_seq_lits′. Line 4 calls get_types, which given a sequence of literals, extracts the types that appear in the sequence. For example, assume p_seq_lits= {how[A]=B, how[C]=D, how[E]≠F}, get_types returns {pizza, voucher}, because the decision variable how is indexed by pizzas and it represents vouchers itself. Line 5 introduces rep_maps which maps a tuple to a pattern variable. The first element in the tuple represents a constant number in seq_lit that is replaced by a variable in p_seq_lits′, and the second element represents a constant number or a pattern variable in p_seq_lits that is replaced by the same variable in p_seq_lits′. For example, assume seq_lit={how[1]=5, how[1]=6, how[2]≠1} and p_seq_lits = {how[A]=B, how[A]=7, how[2]≠C}, and the generated pattern is p_seq_lits′ = {how[A]=B, how[A]=C, how[2]≠D}, then rep_maps will contain {⟨1, "A"⟩:"A", ⟨5, "B"⟩:"B", ⟨6, 7⟩:"C", ⟨1, "C"⟩:"D"}. This helps us replace the same tuple with the same variable, for the example above, the tuples ⟨1, "A"⟩ are both replaced with the pattern variable "A". Line 6 introduces objects1 and objects2, which contain the objects that appear in seq_lit or p_seq_lits, respectively.

Line 7 goes through each type in types. First, Line 8 calls extract_objects that, given a sequence of literals and a type, returns all the constants or pattern variables of that type, and it updates objects1 to contain all the objects of the type type that appear in the `seq_lits`. Similarly, Line 9 updates `objects2`. Then, Line 10 updates `length` to the length of `objects1`. Note that the length of the `objects1` and `objects2` are the same, because we assumed that `seq_lit` and `p_seq_lits` have a similar structure. Line 11 goes through the indices of `objects1` and `objects2`. Line 12 checks if `objects1[index]` and `objects2[index]` are different, in that case they should be replaced by a variable in the `p_seq_lits`′. Line 13 checks if the tuple ⟨`objects1[index]`, `objects2[index]`⟩ does not exist in `rep_maps`. In that case, Line 14 maps the tuple to `var` and updates the `rep_maps`, and then Line 15 updates `p_seq_lits`′ by replacing the variable or constant at the index `index` by `var`. Then, Line 16 calls the increment function that updates `var` to the next character alphabetically. However, if the tuple exists in `rep_maps`, Line 2 reads the value that the tuple is mapped to, and updates `var_old`. Line 19 updates the object at the index `index` in `p_seq_lits`′ to `var_old`.

4.3 Finding Patterns across Searches and Instances

Finding a pattern whose cumulative search reduction is significant for a given instance, provides the modeller with some confidence regarding the importance of the pattern and the possibility of generalising it to a model constraint. However, confidence will be much stronger if clauses with the same pattern appear in different instances of the same model, as this suggests the pattern and its associated information holds across different input data and is, thus, more likely to be a general property of the model. Confidence would be even stronger if the pattern significantly reduced the search across different searches, as this would indicate the associated model modification may lead to speed-ups for all those searches.

To achieve this we have implemented a simple algorithm demonstrated by Figure 4.1. For each instance, the algorithm takes the union of all clauses obtained by the given set of searches, and then computes the patterns associated to them. After finding the patterns for multiple search strategies (search strategies 1 and 2 in the figure), our method computes the intersection of patterns from different instances of problem. This helps us to rule out the instance-specific clauses, and focus on the clauses that appear across instances, which can be generalised to the model.

4.4 Clustering the Clauses under each Pattern

As explained in Section 2.5, Shishmarev et al. realised that some of the top clauses are direct consequences of a single constraint in the model. This helped them manually reformulate that constraint to express the learnt information in a stronger way. In other case, they realised that the top clause results from multiple constraints in the model, and, therefore the information associated with the clause is not explicitly expressed in the model. This helped them to model a new redundant constraint that they then added to the model to capture that information. To achieve this automatically, our method further clusters the clauses under each pattern by the constraints they come from, that is, by

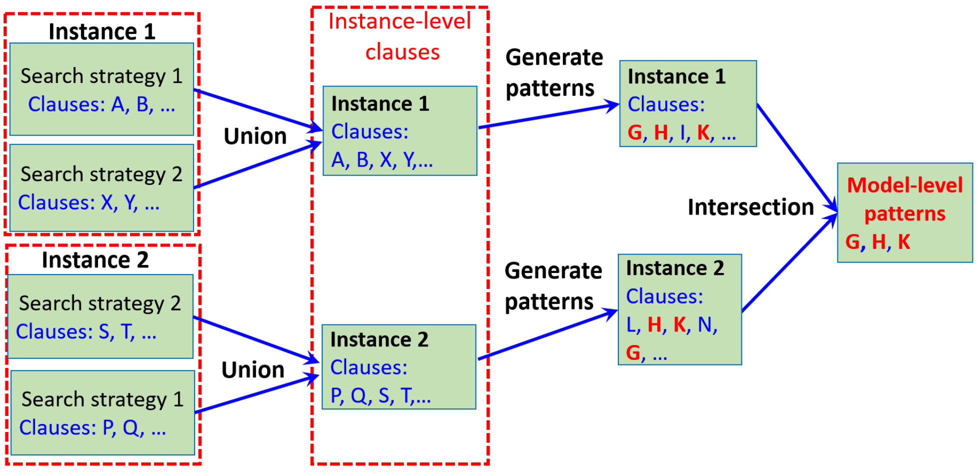

Figure 4.1: Overview of our method for finding patterns across searches and instances

placing the clauses that result from the same constraints into the same cluster. Afterwards, it ranks the clusters based on the number of associated constraints, in ascending order.

Whenever a cluster of clauses is associated to a single constraint, the modeller might be able to reformulate that constraint to improve its propagation and, consequently, improve the solving time. If multiple constraints are responsible, the modeller might need to add information that is not explicitly expressed in the model in the form of a redundant constraint.

For example, Table 4.1, shows the top pattern and its clusters for the `free pizza` problem with the following data:

```
n = 10;
price = [70, 10, 60, 65, 30, 100, 75, 40, 45, 20];
m = 4;
buy = [1, 2, 3, 3];
free = [1, 1, 2, 1];
```

The `Reduction` column represents the estimated search space saved by the pattern, and the next column (`Pattern`) represents the pattern itself. Rows 2 to 4 represent 3 clusters under this pattern. The mappings associated to the clauses under each cluster are provided in the column `Maps`, where each map corresponds to a single clause under the pattern. Lastly the part of the constraint paths responsible for a cluster of clauses is provided by the last column (`Constraints`). For example, 14.58-14.74 points at a location in the model from line 14, column 58 to line 14, column 74, which is the expression `how[p1]=-how[p2]`.

Note that the first cluster, which has the largest contribution in reducing the search space, is associated with a single constraint in line 14. This suggests that the constraint in line 14 should be reformulated to improve the search.

4.5 Limitations

The current implementation of the most specific generalisation of a pattern is very limited, as it (a) only matches sequences of literals that have the same sequence of operators and

Table 4.1: The top pattern and its clusters for an instance of `free pizza`

Reduction	Pattern	Maps	Constraints
98053	how[A]≠B how[C]≠-B		

89041	”	1851:{A=3 C=5 B=3}
		562:{A=10 C=4 B=2}
		...
			14.58-14.74

7850	”	5632:{A=2 C=8 B=3}
		2125:{A=1 C=6 B=3}
		...
			13.30-13.76
			11.43-11.80
			13.30-13.54
			14.58-14.74

1162	”	7941:{A=1 C=6 B=2}	13.33-13.76 11.43-11.80 17.27-17.65 13.33-13.57

variable types, and (b) only matches literals appearing in the same position in each pair of clauses. Due to (a) we might miss a relationship between clauses that have different numbers of literals but share a *parametric* pattern. For example, clauses `{w[1]<3,w[2]<3}`, `{w[1]<3,w[2]<3,w[3]<3}` and `{w[1]<3,w[2]<3,w[3]<3,[w4]<3}`, share the pattern `{w[x]<3| x ∈ 1..n}`.

Due to (b) we might miss patterns that were obscured by the sorting or renaming of literals. For example, for clauses `{w[10]=1,w[20]=2}` and `{w[40]=2,w[50]=1}` we currently only find a pattern with sequence of literals `{w[A]=B,w[C]=D}`, and maps `{A/10,B/1,C/20, D/2}` and `{A/40,B/2,C/50,D/1}`, while matching the first and second literal of each clause would yield a more specific pattern with sequence `{w[A]=1,w[B]=2}` and maps `{A/10,C/20}` and `{A/50,C/40}`. More sophisticated algorithms (such as the anti-unification of [38]) can handle different numbers of literals or matching any literals with the same operator. However, given the computational cost, it might only be worth it for top-ranking clauses.

4.6 Summary

This chapter introduced our method for automatically computing patterns from the learnt clauses, which helps us represent a large group of clauses concisely and rank the clauses more accurately. Then, it discussed our method for finding patterns across searches and instances, which can help us obtain more general patterns that are valid for the whole model, rather than just an instance of the problem. Afterwards, this chapter discussed our method for clustering the clauses under each pattern. Using our method, if a single constraint is responsible for creating a cluster, this may suggest that the constraint should be reformulated to improve the search. If multiple constraints are responsible for a cluster, this may suggest that adding a redundant constraint to the model may improve its execution. Then, it identified the limitations of our method, which cause it to miss some patterns. To mitigate this problem and obtain more accurate patterns, more accurate algorithms such as anti-unification, can be used in future.

Chapter 5

Learning Facts for Patterns

5.1 Introduction

Clauses only contain information regarding instance variables, even though their correctness may also require particular values for the instance parameters which we call *facts*. This information was manually obtained by Shishmarev et al. [69]. For example, for clause {`how[5]`$\neq$`-3`, `how[3]`$\neq$`3`} they considered the fact that pizza 3 is more expensive than pizza 5, and for {`how[1]=-1, how[2]=-1, how[3]=-1, how[4]=-1, how[5]=-1, how[1]=-2, how[2]=-2, how[3]=-2, how[4]=-2, how[5]=-2, how[6]`$\leq$`0, how[6]`$\geq$`3`} they obtained that pizza 6 is more expensive than any other pizza in the clause.

When generalising a set of clauses to a pattern, these additional facts can serve as *conditions* under which the pattern is indeed a valid (implied) constraint. For example, let us again consider the pattern {`how[`*A*`]`$\neq$`-`*B*, `how[`*C*`]`$\neq$*B*} or, in implication form, `how[`*A*`]=-`*B* $\rightarrow$ `how[`*C*`]`$\neq$*B*. This pattern is clearly not valid for arbitrary values of *pattern variables* *A*, *B*, and *C*. However, for all clauses that contributed to the pattern (including clauses 2, 4-10), it is true that pizza *A* is cheaper than pizza *C*. This yields a valid constraint: if pizza *A* is cheaper than pizza *C*, then for any voucher *B*, if we use *A* with voucher *B*, then we cannot get pizza *C* for free with voucher *B*. If we could automatically infer this constraint, then we could present it to the user as a candidate to be added to the model (expressed as the contrapositive of the statement above to make it more similar to the original formulation):

```
constraint forall(A,C in PIZZA, B in VOUCH) ((how[A]=-B /\ how[C]=B)
                                              -> price[C]<=price[A]);
```

In the `free pizza` model, the parameter `price` is indexed by pizzas, and the objects `A` and `C` are pizzas. This suggests that by comparing the prices of the pizzas that appear in the pattern, we may find a fact that is relevant to the pattern.

Thus, our first step towards automatically obtaining such facts is to detect, in each pattern, the pattern variables of the same type. We then extract the model parameters that are indexed by these objects whose relations may be relevant to the pattern. For example, for pattern `how[`*A*`]=-`*B* $\rightarrow$ `how[`*C*`]`$\neq$*D*, where variables `A` and `C` are both pizzas, and `B` and `D` are vouchers, our algorithm extracts `A` and `C` as objects of the same type (since they are both pizzas), and similarly, it extracts `B, D` as vouchers. Then, it detects the model parameters that are indexed by pizzas and vouchers, which are {`price`}, and {`buy, free`}, respectively.

The second step is to develop a method to infer facts for each pattern. This method, given the extracted objects and parameters, and a predefined set of relations such as {$\geq$, =, <}, computes the percentage of clauses under the pattern whose parameters or objects satisfy that learned relation. For our example, it may infer facts such as `price[C]`>`price[A]` for 100% of the learned clauses, and `A>C` for 50% of the clauses. Note that since this method only uses the clauses learnt from the solver, it might miss clauses (and thus facts) that are valid but not learned. To improve the accuracy of our method we implemented a second method to learn facts based on the ConAcq algorithm [12] described in Chapter 2.7. This method first extracts a full set of positive and negative examples that satisfy or violate the pattern. Then, the full set of examples are passed to our implementation of ConAcq, which generates facts.

The structure of this chapter is as follows: Section 5.2 provides an overview of both of our methods and then discusses our method that, for each pattern, extracts the relevant objects and model parameters that are indexed by these objects. Section 5.3 discusses our percentage-based method to compute facts. Section 5.4 discusses our ConAcq-based method to compute facts. Section 5.5 describes the limitations of our framework to compute facts. Section 5.6 provides a summary of this chapter.

5.2 Finding Relevant Facts for each Pattern

To relate the background information to each pattern, our first step is to find objects of same type. For example, assume we have a pattern {`how[A]`$\neq$`-B`, `how[C]`>`B`, `used[D]=false`} for the `free pizza` example 3, the objects `A` and `C` are pizzas, and the objects `B` and `D` are vouchers. Then, our next step is to detect the parameters that are indexed by the objects that appear in each pattern. For our example, pizzas and vouchers are the objects that appear in the pattern, and pizzas are indexed by the parameter `price`, and vouchers are indexed by the parameters `buy` and `free`. This suggests that the relationships between {`price[A]` and `price[B]`}, {`buy[B]` and `buy[D]`} and {`free[B]` and `free[D]`}, may be relevant facts for the pattern that explain its occurrence. In this section, we describe our method that performs these two steps.

To automate this, as described in Chapter 4.2, we instrumented the MiniZinc compiler to output type information that gives us information about decision variables and parameters that appear in the model. Specifically, it outputs the enumerated type/types of the variables and parameters, and also the types of their indices. Using the type information, our method first extracts the decision variable names that appear in the pattern, and the associated types to each decision variable. Then, it extracts all the objects of that type. Also, using the type information, it extracts all the model parameters that are indexed by these objects. Algorithm 3 describes our method of detecting the objects of the same type, and the parameters that are indexed by these types, and how this information is used to learn facts.

Algorithm 3 Algorithm to find the relevant facts for each pattern

```
Input: pattern=⟨p_id, p_seq_lits, maps, p_reds, p_cons⟩, model, data

1: variable_names←extract_variable_names(p_seq_lits)
2: foreach variable_name ∈ variable_names do
3:     types←get_type_info(variable_name)
4:     foreach type ∈ types do
5:         objects←extract_objects(p_seq_lits, type)
6:         parameters←extract_parameters(type)
7:         facts%←infer_facts%(objects, parameters, maps)
8:         facts_ConAcq←infer_facts_ConAcq(objects, parameters, pattern, model, data)
9: return(facts%, facts_ConAcq)
```

The input of the algorithm is a MiniZinc model and a pattern as defined in Chapter 4, that is, a tuple containing a pattern ID (`p_id`), a sequence of pattern literals (`p_seq_lits`), the maps under the pattern (`maps`), where each map represents a clause that matches the pattern, the estimation of the reduction in search space associated with the pattern (`p_reds`) and the model constraints associated to the pattern `p_cons`. Line 1 extracts the set of variable names that appear in the pattern. For example, for the pattern {`how[A]`≠`-B`, `how[C]`≠`B`}, the `extract_variable_names` function returns the set {`"how"`}. Line 2 goes through each `variable_name`, and Line 3 calls the `get_type_info` function that returns all the `types` associated to `variable_name` and stores them in `types`. For our example, the decision variable `"how"` is indexed by pizza objects, and it represents a voucher object (e.g. `how[1]=2`, the index 1 is a pizza and 2 is a voucher). Line 4 iterates through the `types`. Then, Line 5 calls the `extract_objects` function, which extracts all the objects of `type` type that appear in `p_seq_lits` (i.e. pattern), and stores them in `objects`. For example, for the pattern above in the first iteration the `type` is `pizza`, and the `objects` will be `A` and `C`. Line 6 calls the `extract_parameters` function that returns the parameters that are indexed by `type`, and stores them in `parameters`. For example, for the type `pizza`, the `parameters` will be set to {`price`}, and for the type `voucher` it will be set to {`buy, free`}. Then, Line 7 calls the `infer_facts%` function, which takes `objects`, `parameters` and `maps` as input arguments, and returns the learned facts, which is stored in `facts%`. This method corresponds to our percentage-based approach, which is described in Section 5.5. Line 8 calls the `infer_facts_ConAcq` function that takes `objects`, `parameters`, `pattern` and `model` as input arguments, and returns the learned facts which are stored as `facts_ConAcq`. This corresponds to our ConAcq-based approach, which is described in Section 5.6.

After extracting the objects and parameters related to each pattern, we define the relations that we are looking for, that may hold between these objects and parameters by Definition 1. These relations are embedded into both of our methods.

Definition 1. *The type of relation between objects and parameters of the same type that our method considers to find relevant facts for each pattern:*

- *Unary relations `X=0, X<0 or X=max(p), p[X]=0, p[X]<0 or p[X]=max(p)`: when there is object `X` in the pattern, and model parameter `P` is indexed by `X`*

- *Binary relations `X=Y`, `X≠Y`, `p[X]=p[Y]`, `p[X]≠p[Y]`, `p[X]≤p[Y]`, `p[X]=-p[Y]`: when there are two objects `X` and `Y` of the same type in a pattern, and model parameter `P` is indexed by `X` and `Y`*

5.3 Inferring Facts: A Percentage-based Approach

As mentioned before, our aim is to discover facts that hold for all the clauses under each pattern. However, since patterns may contain many clauses, the facts are unlikely to always be true for all clauses. Consider the pattern {`how[A]≠-B`, `how[C]≠B`}, from our `free pizza` example. Our method infers the fact `price[C]>price[A]` for 100% of the clauses. We therefore go through each pre-defined relation defined by Definition 1 and compute the *percentage* of clauses in the pattern for which the fact holds, and only report facts for which that percentage is above a certain threshold. Also, if two facts are complementary, such as `X≠Y` and `X=Y`, we report the fact with the higher percentage.

Algorithm 4 infer_facts%

```
    Input: objects, parameters, maps
    Output: all_facts

 1: relations ← {"=", ">", "≥"}
 2: fact, fact_param, all_facts ← ∅
 3: fact_p, fact_param_p ← ∅
 4: foreach relation ∈ relations do
 5:     foreach ⟨O_1, O_2⟩ ∈ objects do
 6:         ⟨fact, fact_p⟩ ← get_fact(O_1, O_2, relation, maps)
 7:         all_facts ← ⟨fact, fact_p⟩ ∪ all_facts
 8:         foreach param ∈ parameters do
 9:             ⟨fact_param, fact_param_p⟩ ← get_fact_param(O_1, O_1, param, relation, maps)
10:             all_facts ← ⟨fact_param, fact_param_p⟩ ∪ all_facts
11: return(all_facts)
```

Algorithm 4 provides the `infer_facts%` function, called from Algorithm 3. For simplicity, in this algorithm we only describe our method of finding the binary relationships, which can be generalised to other types of relations in a straight forward way. A similar approach is used to find unary relationships, as briefly discussed below.

The inputs of the algorithm are `objects`, `parameters` and `maps`, described in Algorithm 3. The algorithm returns a set of tuples (`all_facts`), where each tuple is of the form of ⟨`fact`, `fact_p`⟩, `fact` is a relation between two objects or parameters (such as `price[C]>price[A]` or `A<C`), and `fact_p` is the percentage of the maps (clauses under the pattern) for which that `fact` holds.

Line 1 defines the `relations` considered for the facts. In our implementation, we consider the relations $\{=, >, \geq\}$. The complement of these relations $\{\neq, \leq, <\}$ can also be learnt, which we will discuss below. Line 4 defines a nested loop that considers for every relation in each iteration of the loop, every pair of object $\langle O_1, O_2\rangle$ and does the following.

In Line 6, the `get_fact` function checks the percentage of the clauses (corresponding to `maps`) for which O_1 `rel` O_2 is valid. If the percentage is above 50% it returns a tuple consisting of the `fact` O_1 `rel` O_2 and its percentage (`fact_p`). For example, $\langle A > B$, 60%$\rangle$. If the percentage is less than 50% it returns the complement of that relation and its percentage. The get_fact function can be also used to infer unary relations with predefined values such as $\emptyset$, `max(p)` or `min(p)` for a certain pattern p. Line 7 takes the union of the new fact with `all_facts`, and updates `all_facts`.

Lines 8- 10 define another loop, that calls the `get_fact_parameter` function for each parameter, but computes relations in the form of `param[`O_1`] rel param[`O_2`]`, and their associated percentage, and again updates `all_facts` to include the new facts.

This is a very simple but incomplete method, and it will be interesting to pursue further research into how more general facts can be extracted automatically in a reasonable amount of time. To improve the completeness of our method, we have defined a second method that is described in the next section.

5.4 Inferring Facts: A ConAcq-based Approach

The percentage-based approach defined above checks if a certain fact is valid for the clauses inferred by the solver. This method is incomplete, because depending on the search, Chuffed may infer different sets of clauses. Therefore, we do not have a complete set of mappings that makes a pattern a valid constraint. To improve the completeness of our technique, we defined a second method based on the ConAcq algorithm described in Chapter 2.7. This method obtains a full set of mappings that are valid for a given pattern given a set of data files. To clarify, in pattern {`how[A]`≠`-B`, `how[C]`≠`B`}, any combination of `A, B, C` values that makes the pattern a valid constraint, i.e. a constraint that can be added to the original problem without removing any solution, is considered as a *positive example* or a *valid mapping*, while any possible combination of `A, B, C` that is not part of the positive examples is a *negative example*. We refer to the combination of positive and negative examples as *training examples*. Also, since the positive and negative examples are complementary, we only obtain positive examples, and then any example that is not covered by the set of positive examples is considered as a negative example. After generating the training examples, the next step is to infer facts for the objects or their associated parameters (e.g. `price`).

To infer facts for each pattern, our algorithm is as follows:

Algorithm 5 provides the `infer_fact_ConAcq` function, called from line 8 of Algorithm 3 to infer a more complete set of facts. The inputs of the algorithm are `objects`, `parameters`, `MiniZinc model` and `pattern`. The algorithm returns `all_facts`, the set of facts (such as `A>B`) learnt by the system.

Line 2 goes through each pair of objects $\langle O_1, O_2\rangle$, and Line 3 generates all training examples for $\langle O_1, O_2\rangle$ with the input pattern, as described in Section 5.4.1. Line 4 calls the `run_ConAcq` function. The input argument of this method is `training_examples`, and it outputs a set of facts, which is stored in `facts`, this method is described in Section 5.4.2. Line 5 updates `all_facts`. Then, Lines 6-9, repeat the same procedure for each `parameter`

Algorithm 5 infer_fact_ConAcq

```
    Input: objects, parameters, pattern, model
    Output: all_facts
 1: all_facts, facts $\leftarrow \emptyset$
 2: foreach $\langle O_1, O_2\rangle \in$ objects do
 3:     training_examples $\leftarrow$ generate_training_examples($\langle O_1, O_2\rangle$, pattern, model)
 4:     facts $\leftarrow$ run_ConAcq(training_examples)
 5:     all_facts $\leftarrow$ facts $\cup$ all_facts
 6:     foreach parameter $\in$ parameters do
 7:         training_examples_param $\leftarrow$ generate_training_examples_param(training_examples,
   parameter)
 8:         facts $\leftarrow$ run_ConAcq(training_examples_param)
 9:         all_facts $\leftarrow$ facts $\cup$ all_facts
10: return(all_facts)
```

to infer facts related to parameters associated to each pattern, and similarly, update `all_facts`. Line 10 returns `all_facts`.

5.4.1 Generating Training Examples

To automatically obtain the training examples for each pattern, our method first obtains all the positive examples for each pattern, then each example that is not covered by this set is considered as a negative example. The positive examples are computed using all the possible values to which the pattern's variables can be mapped. For example, for the pattern `how[A]=B`$\vee$`how[C]=-B`, the positive examples of pattern variables `A`, `B` and `C` are the mappings of `A`, `B` and `C` that are also partial solutions of the original model. To compute this, we could add each pattern to the MiniZinc model and output all the solutions (i.e. positive examples). However, to ensure that the substitution of each positive example forms a valid constraint, we need to try each constraint individually and check if by adding that constraint to the model we remove any solution. For example, let us consider that we add the pattern `how[A]`$\neq$`B`$\vee$`how[C]`$\neq$`-B` to the `free pizza` model as a constraint and one of the solutions is {`A=4, C=1, B=3`}. To check if this forms a valid constraint `how[4]=3`$\vee$`how[1]=-3`, we first count the number of solutions of the original model, and then add the pattern `how[A]`$\neq$`B`$\vee$`how[C]`$\neq$`-B` to the model and again count the number of solutions to see if the added constraint removes any. If no solution was removed we consider the mapping as a positive example. This approach has the disadvantage that it requires counting all solutions.

An easier alternative is to add the negation of a pattern to the model (a *nogood pattern*) and output all the mappings that are valid for (i.e. solutions to) the nogood pattern (which are thus non-solutions of the original pattern). By doing this we can eliminate the solution counting step. However, by adding the nogood pattern to the model, we obtain the positive examples for the nogood pattern rather than the clause pattern. Therefore, the facts learned by ConAcq should be later negated to be suggested to the modeller.

Example 7. To obtain all the positive examples of the pattern clause `how[A]`$\neq$`B`$\vee$`how[C]`$\neq$`-B` for two variables `A` and `C` (because they have the same type), we added its negation `how[A]=B∧how[C]=-B` to the `free pizza` model provided in Chapter 2.5. Figure 7 provides the modified model, and the changes are highlighted and can be seen in lines 15 to 20.

```
1  int: n;     set of int: PIZZA = 1..n;     % number of pizzas wanted
2  array[PIZZA]   of int: price;             % price of each pizza
3  int: m;     set of int: VOUCH = 1..m;   % number of vouchers
4  array[VOUCH] of int: buy;               % buy this many to use voucher
5  array[VOUCH] of int: free;              % get this many free
6  set of int: ASSIGN = -m .. m; % i -i 0 (free/paid with voucher i or not)
7  array[PIZZA] of var ASSIGN: how;
8  array[VOUCH] of var bool: used;
9  constraint forall(v in VOUCH)(used[v]<->sum(p in PIZZA)(how[p]=-v)>=buy[v]);
10 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
11 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
12 constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
13                                  -> price[p2] <= price[p1]);
14
15 var 1..n: A; % represents pizza 1
16 var 1..n: C; % represents pizza 2
17 var 1..m: B; % represents a voucher
18 constraint (how[A]=B /\ how[C]=-B);  % pattern
19 solve ::int_search([A, C], input_order, indomain_min, complete) satisfy;
20 output [show(A)++","++show(C)++"\n"];
```

Figure 5.1: The `free pizza` model with the added pattern

Lines 15 to 17 introduce the variables in the nogood pattern. The pattern is added in line 18 as a constraint. Line 19 adds the solve item, where the first argument `[A, C]` describes the variables affected by this search strategy, `input_order` states that the variables should be chosen based on the order specified by the first argument, `indomain_min` indicates that the smallest value in the domain should be chosen first, and lastly `complete` indicates the search strategy to be performed.

Executing the model with every data file will output all the combinations of variables `A` and `C`. For example, let us consider the following data file:

```
n = 5;
price = [17, 98, 76, 36, 69];
m = 8;
buy = [4, 4, 1, 4, 2, 1, 1, 3];
free = [2, 4, 1, 1, 4, 2, 3, 3];
```

Executing the model with this data will output all the combinations of `A` and `C` that satisfy the pattern and model constraints, that is, all the combinations of pizza 1 (`A`) and pizza 2 (`C`) that cannot be bought using the same voucher (`B`). Table 5.1 provides the output.

Table 5.1: Positive examples for `pizzas`

A	1	1	1	1	3	4	4	4	5	5
C	2	3	4	5	2	2	3	5	2	3

All combinations not present in Table 5.1, are considered as negative examples. Note that Table 5.1 is used to infer facts for two objects of the same type (e.g. $A > C$). We are also interested in finding relations between the parameters that are indexed by these objects, as described in Algorithm 5, line 7 (e.g. $price[A] > price[C]$). To achieve this, we retrieve the parameter value of each of the table entries, and generate a new set of training examples for each parameter. For instance, for the parameter `price`, for the objects in Table 5.2 we generate the following table:

Table 5.2: Positive examples for the parameter `price`

A	price[1]	price[1]	price[1]	price[1]	price[3]	price[4]	price[4]	price[4]	price[5]	price[5]
C	price[2]	price[3]	price[4]	price[5]	price[2]	price[2]	price[3]	price[5]	price[2]	price[3]

5.4.2 Inferring Facts using ConAcq

To obtain facts for each pattern, we implemented `ConAcq` in MiniZinc based on the ConAcq algorithm described in Chapter 2.7.1, and also we developed a MiniZinc model that passes each set of `pos_examples`, their corresponding domain and a set of constraint library to the `ConAcq` model. This model is represented by Figure 5.2. Our framework first generates all the tables of positive examples for each pattern, as described in Section 5.4.1. It then iteratively runs the model provided in Figure 5.2 for each table. Assume that we are interested in finding valid facts for two integer variables `X` and `Y`, which can represent two objects of the same type that appear in the pattern (e.g. Table 5.1), or valid values of a parameter corresponding to two objects of the same type (e.g. Table 5.2)

```
1  array[int] of int: domain; % domain of X and Y
2  array[int,int] of int: pos_examples;
3  enum BIAS = {X_LESS_Y, X_GREATEREQ_Y, X_EQ_Y, X_NEQ_Y}; % the constraint library
4  array[BIAS] of var bool: bias; % learnt constraint network
5
6  constraint ConAcq(pos_examples, domain, bias);
7
8  solve minimize sum(bias);
```

Figure 5.2: MiniZinc model for inferring facts using ConAcq

Parameter `domain` in Line 1 represents the domain of `X` and `Y`. Line 2 introduces the `pos_examples` parameter, which represents all the positive examples (all the valid combination of `X` and `Y` values). Line 3 introduces `enum BIAS` which represents the relations defined by Definition 1: all the binary relations between `X` and `Y` ($=, \neq, <, \geq$). Note that the relations $\leq$ and $>$ are also covered by the bias, since $\leq$ can be represented as $\{< \vee =\}$, and $>$ as $\{\neq \wedge \geq\}$.

The decision variable `bias` in line 4, indicates the learnt constraint network (output of the ConAcq algorithm), which is indexed by the `BIAS` enum. If any of the elements of `bias` is set to true, it indicates that the constraint is valid. The constraint in line 6, calls the `ConAcq` predicate which we discuss below. Lastly, the solve item in line 8, minimizes the number of learnt constraints. This helps us report the shortest learnt constraint network which indicates the strongest fact. This is required because the ConAcq algorithm outputs a conjunction of facts, and this conjunction is a condition that makes the pattern a valid constraint, thus when there are fewer facts in the conjunction, it is easier to make the pattern a valid constraint.

The `ConAcq` predicate is as follows:

```
1  predicate CONACQ(array[int,int] of int: pos_examples, int: domain, array[BIAS] of
       var bool: bias)  =
2      forall(i,j in {domain[n]|n in index_set(domain)}) (
3          let
4          {
5          bool: pos_example = exists (k in index_set_1of2(pos_examples))(
               pos_examples[k,..] = [i,j]); % if (i, j) is a positive example,
               pos_example is set
6                  set of BIAS: k = {b | b in BIAS where rejects(i,j,b)};   % stores
                       all the constraints in the bias that reject (i, j)
7          }
8          in if pos_example then
9                  not exists (b in k) (bias[b]) % none of the constraints in the
                       bias (learnt constraint network) should reject a positive
                       example
10         else
11                 exists (b in k) (bias[b]) endif % at least one of the constraints
                       in the bias (learnt constraint network) should reject the
                       negative  example
12         );
```

The input arguments of `ConAcq` are: positive examples (`pos_examples`), domain of the variables corresponding to the `pos_examples` (`domain`), and the decision variable `bias`. This algorithm ensures that none of the constraints in the `bias` reject any of the positive examples, and that there is at least one constraint in the `bias` that rejects all the negative examples.

Line 2 goes through each training example $\langle i, j\rangle$. If $\langle i, j\rangle$ is a positive example, Line 5 sets the `pos_example`. Otherwise, it sets it to `false`. Line 6 goes through each constraint `b` in `BIAS`, and calls the `rejects` function, which returns True if bias `b` rejects the example $\langle i, j\rangle$, otherwise it returns false. The `reject` function is provided by Figure 5.4.2, and it is discussed later. Then, if the example $\langle i, j\rangle$ is positive (line 8), line 9 ensures that none of the constraints in the `bias` (learnt constraint network), rejects the example $\langle i, j\rangle$. Otherwise, if the example is negative, line 11 ensures that at least one of the constraints in the bias (learnt constraint network), rejects the negative example.

```
test rejects(int: X, int: Y, BIAS: b) =
  if b=X_LESS_Y then X >= Y
  elseif b=X_GREATEREQ_Y then X < Y
  elseif b=X_EQ_Y then X != Y
  elseif b=X_NEQ_Y then X = Y
  else abort("wrong␣bias") endif;
```

Figure 5.3: The `rejects` function

The input arguments are two integers `X` and `Y`, and a constraint `b`. The `rejects` function returns a learnt constraint from the bias that rejects `b`.

An example of the input data for the `ConAcq` MiniZinc model is as follows, this is specific for the `free pizza` problem, with the input data provided in Section 5.4.1, to learn facts regarding to parameter `price` for two pizzas:

```
n = 5;
domain = [17, 98, 76, 36, 69];
pos_examples=[|
17,98|
17,76|
17,36|
17,69|
3,98|
4,98|
4,76|
4,69|
5,98|
5,76|
|];
```

As a result, the `ConAcq` instance outputs: `X` $\leq$ `Y`. By keeping a map from the name of the model decision variables to the `ConAcq` parameters, say (`X: price[A], Y:price[C]`), our method translates this to `price[A]` $\leq$ `price[C]`.

5.5 Limitations

Incorporating background information into the learned clauses is challenging, and both of the methods that were described in this chapter suffer from some limitations. Both of the methods only consider a limited type of relations (unary and binary) to compute facts. Hence, we may miss some important facts that can be obtained by considering other forms of relations, such as linear constraints. Also, each of the methods have their benefits and limitations, which we discuss in this section.

Percentage-based approach: This method only considers the clauses inferred by Chuffed, and simply checking if a fact is valid for all the clauses under the pattern. Thus, there may exist valid clauses that were not discovered by Chuffed, which reduces the accuracy of the result. However, the main benefit of this approach over the ConAcq-based approach is that it provides more facts to the modellers, because, it outputs any fact that holds for a majority of the clauses. This may help modellers better understand the pattern.

ConAcq-based approach: This method considers a complete set of examples to generate facts. ConAcq helps us obtain more accurate facts compared to the percentage-based method, because it considers the negative examples also, and the output network of facts (learnt constraint network) must reject all the negative examples and accept all the positive ones. However, our experiments show that, for most of the patterns we are unable to infer facts using this method. Further, as described in Section 5.4.1, the generating training examples step is performed by adding each pattern to the model as a new constraint automatically. This can be challenging, especially if a pattern contains literals that are not derived from the model-level variables. As described in Chapter 3, during compilation the compiler may introduce new variables, and there may not be an equivalent model-level variable corresponding to those variables. For example, in one of our experiments Chuffed inferred the clause {`"XI:38|23|38|35:(i=8):builtins.mzn:365|39|365|53"` $\leq$ `201`, `x[1]` $\leq$ `-24`, `x[3]` $\leq$ `45`}. The literal {`XI:38|23|38|35:(i=8):builtins.mzn:365|39|365|53`} indicates that a variable is introduced in line 38 of the model from column 23 to 35, which is the expression `abs(x1 - x2)`. This expression is located inside the function `approx_distance`, and `(i=8)` indicates that the function `approx_distance`, is called within a loop and the variable is introduced in the 8th iteration. `builtins.mzn:365|39|365|53` points to the variable `is_defined_var` located inside `builtins.mzn` in line 365 from column 39 to 53, which is in the `abs` function definition. Since the first literal of this pattern is not a model-level variable, our method is unable to add this pattern to the model. Thus, the ConAcq-based approach does not process any pattern that contains such literals that cannot be renamed.

5.6 Summary

As patterns may not be valid constraints our aim is to infer facts that might be necessary for each pattern. These facts can thus serve as a condition to make the pattern a valid constraint. In this chapter we discussed two methods of inferring facts for each pattern. Section 5.2 provided an overview of our methods to infer facts. Our method first extracts all the objects of the same type that appear in the pattern, and also all the parameters that are indexed by these objects, and then considers the binary and unary relations between the objects and their associated parameters, as potential facts. Then, for each pattern it passes these potential facts to both of our methods to infer facts. Section 5.3, described our first method: a percentage-based approach. This approach goes through each of the potential facts given as input and computes the percentage of the clauses under the pattern that follow that relation. It then returns the facts with the highest associated percentages. This method suffers from some limitations, such as only considering the clauses that are inferred by Chuffed, which do not cover the full set of positive examples for the pattern. Therefore, to mitigate this we developed a second method: based on ConAcq. Section 5.4.2 described our ConAcq-based approach, which first generates a full set of positive and negative example for each pattern for a given set of data files, and it then passes these examples to ConAcq which infers the facts that accept the valid examples and reject the invalid ones.

Section 5.5 discussed the limitations of both of our methods. Both of the methods only consider a limited type of relations (unary and binary) to compute facts. Hence, we may miss some important facts that can be obtained by considering other forms of relations, such as linear constraints. Also, each of the methods has its benefits and limitations. The percentage-based approach is less accurate than the ConAcq-based approach, but it is also less strict. Thus, it can infer more facts that the ConAcq-based approach.

Chapter 6

Experiments and Case Studies

6.1 Experimental Set-up

The overall performance, efficiency and effectiveness of the framework is demonstrated using models from the annual MiniZinc challenge [2]. The chapter further demonstrates the usefulness of our approach using three case studies. Our experimental set-up is as follows. Our fully-automated framework is used to first infer clauses using Chuffed, with a *fixed search* strategy, where the user-specified search in the original model is followed. Then, it replays the same search using Gecode (ensuring it makes the same decisions as Chuffed). Then it repeats the same steps using a *free search* strategy. To explain this search strategy, first I will recapitulate the *restart technique* that is commonly used by learning solvers, including Chuffed. In this technique, after a certain portion of the search space is explored, search is interrupted and started from the beginning, while incorporating the learnt clauses [70]. In Chuffed's free search strategy, whenever the search restarts it switches the search strategy between *fixed search* and an *activity-based search*, where the decision variables with the highest activity are chosen first. At this point, the framework has the clauses inferred by Chuffed for two search strategies it then computes their associated ranking, by estimating the additional search space explored by Gecode that was avoided by Chuffed thanks to learning that clause. Our method then computes the union of the clauses inferred, simplifies and renames the clauses and connects them to their corresponding model constraint, as described in Chapter 3. The renamed and simplified clauses along with their associated constraints are then passed to the method described in Chapter 4, for generating patterns. Then, the resulting patterns, along with their associated attributes, are passed to the method described in Chapter 5, for generating facts. Afterwards, the same procedure is repeated using a different instance of the problem, which helps identify patterns that appear in multiple instances, and thus be more generalisable to the model.

The structure of this chapter is as follows. Section 6.2 discusses the benchmarks used in the experiments. Section 6.3 provides the results of our experiments which were carried out to evaluate the performance and efficiency of our framework. In particular, we first provide a few examples of the output of the framework. Then, we provide the execution time of each component of the framework for various models. Afterwards, we discuss our experiments that evaluate the practicality and efficiency of our method for simplifying clauses. Section 6.4 describes in detail the result of applying our framework in three case studies that our method improved. Lastly, Section 6.5 provides the conclusions of this chapter.

6.2 Benchmarks

This section discusses the benchmark models used in our experiments. We conducted experiments on models from the MiniZinc challenge [2, 72, 73] years 2015 to 2018. These models were selected because they are realistic examples of expert models. The models summarised below are those in the `minizinc-benchmarks`[1] as:

community-detection: A constrained community detection problem, where the aim is to detect communities with maximum modularity value, that is communities where the connection between the nodes in the community is dense, but the connections between the nodes in different communities are sparse. Some of the vertices must be assigned to the same community, and some cannot.

mario: A routing problem based on the world of Nintendo's Super Mario, where Mario plans to visit **n** houses, starting from his house and ending at his brother Luigi's house. There is a specific amount of gold in each house, and Mario has a certain amount of fuel to consume for the entire trip. The aim is to find a route that will earn Mario maximum amount of gold, without running out of fuel.

neighbours: In this problem, there is $n \times m$ number of people living in an $n \times m$ grid. A natural number N ($N \geq 1$) is assigned to each resident, in such a way that all the numbers $1, 2, ..., N-1$ must be assigned to its neighbours. The aim is to maximise the sum of the assigned numbers.

tpp: A grid-based asymmetric travelling purchaser problem, where there are `n` cities on a grid and `m` products to purchase. The traveller must plan a route to visit cities to purchase the products, as the products are offered at different prices in different cities. Travel is only allowed between adjacent cities vertically or horizontally, and travel costs are asymmetric. The objective is to minimise the total travel and purchase costs.

grid-colouring: A grid colouring is a colouring of an $n \times m$ grid `x` where no sub-rectangle of the grid has all its corner cells assigned the same colour. This problem is problem is the first of the three case studies, and is explained in more detail in Section 6.4.1.

tcgc2: A time changing graph colouring problem, where the given initial colouring of a graph must be transitioned to a given final colouring. The transition requires a certain number s of steps, each performing at most k modifications to the colours, while maintaining a valid graph colouring. The aim is to first minimise the number of steps required, and then the number of modifications. This problem is the second problem of the three case studies, and is explained in more detail in Section 6.4.2.

concert-hall-cap: In the capacitated concert hall problem, there is a number of concert halls, each with a certain capacity, and a number of event offers to be assigned

[1]`https://github.com/MiniZinc/minizinc-benchmarks/tree/master/<model>`

to the concert halls, each requiring a hall larger than a certain capacity. Each event has a specific start and end time and a price that will be gained by accepting its offer. The organiser needs to assign events to the concert halls, in a way that concurrent events are not assigned to the same hall, and the capacity of a hall assigned to an event is greater than the capacity required by that event. The aim is to maximise the profit. This problem is the last of the three case studies, and is explained in more detail in Section 6.4.3.

oc-roster: The on-call rostering problem is a scheduling problem, where there are `N` staff members, and the problem is to assign a staff member to each day during a rostering period. The staff members may not be available on some particular days, may require to be on-call on some days, should not be on-call for more that two consecutive days, and have certain preferences, for example, they prefer to work the same number of days during the rostering period. The aim is to generate a rostering period that satisfies staff members preferences, as much as possible.

opt-cryptanalysis: A cryptanalytic problem, a key differential attack against standard block cipher AES, which is further described in [35].

seat-moving: In the seat-moving problem there is a number of seats some of which have a person has a goal seat. The problem is to move the persons to empty seats or swap people until every goal seat is reached, and always ensuring certain restrictions on swapping the people are satisfied. The aim is to reach the goal set positions in a minimum number steps.

steiner-tree: Given a weighted graph, the aim of this problem is to detect a sub-graph and is a tree with minimum weights that contains all the terminal nodes.

team-assignment: This problem tries to find an assignment of players to teams, where each player has a rating. The aim is to distribute the players so that the teams are balanced.

train: In this problem there are `n` trains moving along a track with `m` stations. There is a constant flow of passengers arriving at all the stations, except the first and last one. Trains cannot overtake preceding trains, but they can skip or wait longer in a station to collect more passengers. The aim is to reschedule the trains when there is a delay in a way that the average travel time of the passengers in minimised.

vrplc: In the vehicle routing problem with location congestion, vehicles depart a depot to pickup and deliver products, and then they return to the depot. Delivery requests have an associated weight, and each vehicle has a weight capacity. For each request, the product must be delivered within that time frame. In addition, the number of requests that can be served at any time is restricted. The aim is to traverse the route in a minimum amount of time.

6.3 Experiments and Results

This section presents the results of our experiments using the set-up described in Section 6.1. First, we present the output of the framework for a few models. Then, we provide a table that shows the execution time of each component of the framework. Lastly, we present a table that shows the effectiveness of our approach for simplifying the clauses.

In Table 6.1 each row provides the top pattern and its associated facts inferred by our framework for several MiniZinc models. For simplicity and readability, we only display patterns with a small number of literals. The **Model** column shows the model names; the **Instance** column shows the name of the instances for which the patterns were inferred; the **pattern** column shows the top patterns inferred for each model; the **F(%)** column shows the inferred facts and the percentage of clauses for which the fact is valid, using the **percentage-based approach** described in Chapter 5.3, the **F(ConAcq)** column shows the inferred facts using the ConAcq-based approach described in Chapter 5.4; the **Cl** column shows the percentage of all inferred clauses that are captured by the pattern, and **Consts** column shows the parts of the model constraints that are responsible for generating the clauses. Each part is of the form `StartLine.StartColumn-EndLine.EndColumn`. For example, `7.9-7.32` points to the part of the constraint that starts in line 7, column 9 and ends in line 7, column 32.

The results in Table 6.1 demonstrate that the system can successfully discover and expose interesting patterns and their corresponding facts. For example, the pattern for `grid-colouring` covers 40% of the inferred clauses, while each clause under this pattern has a relatively small contribution in reducing the search space, when considering all together as a pattern, the importance of this clause becomes apparent. Note that this is not always the case. For example, for the `neighbours` problem, the top pattern only covers 0.08% of the clauses, which indicates that these few clauses have a high reduction themselves. Another important observation is that our percentage-based approach to generate facts, can often infer facts between the variables in the pattern. However, for all the models except `freepizza`, the learned facts did not provide much insight for the occurrence of the pattern. This motivates the search for a more powerful method. Unfortunately, the `ConAcq`-based approach is not able to detect facts for most cases.

Table 6.2 shows the execution time of the framework for the MiniZinc 2018 benchmark models together with different statistics regarding the execution that aim to provide insight into the solving times. Note that for the MiniZinc 2018 benchmark models that are not present in this table, the learned clauses inferred by Chuffed did not have a contribution in reducing the search space in Gecode, and our framework associates zero reduction in search space for these clauses, and the framework does not further analyse the clauses that did not reduce the search space. In addition, the benchmark displayed in Table 6.2 is different from that of Table 6.1. This is because, in Table 6.1 for simplicity and readability we chose the benchmarks that the patterns are shorter, as their length can be very large. But, in Table 6.2 we provided the results for all the MiniZinc 2018 benchmarks except those where the clauses do not appear to reduce the size of the search space. The `Model` and `Instance` columns show the name of the MiniZinc models and their corresponding instance. The next four columns show Chuffed's solving statistics, `N` is the number (in Thousands) of the nodes explored by Chuffed, `F` the number (in Thousands) of failures, `S` the number of solutions, and `T` the solving time in minutes. These figures are computed by first obtaining the number of nodes, failures, solutions and solving time with fixed search strategy, and then free search strategy, then taking the average of the average of

Model	Instance	Pattern	F(%)	F(ConAcq)	Cl	Consts
community-detection	dolphin-.s62.k3 mexican-.s26.k4	x[A]≠x[B] x[C]≠2	100%:A>B	-	34%	49.33-49.57
mario	mario-_medium_1 mario-_medium_3	succ[A]≠B succ[B]=C succ[B]≠D succ[B]=E succ[B]≥F succ[B]≤G	100%:D<F 100%:D>G 100%:E<F, ...	-	7%	6.47-6.58 31.3-31.73 45.23-45.37
neighbours	neighbours1	x[A,B]≤2 x[C,D]≤2 x[neigh[E,1],neigh[E,2]]=F x[neigh[E,1],neigh[E,2]]=G	100%:B>F 100%:D>G 75%:F<G, ...	-	0.08%	170.13-170.25
tpp	tpp-_4_5_20_1 tpp-_5_3_30_1	purchaseLoc[A]≠B succ[B]≠B	100%:A>B	-	22%	3.47-6.58 31.3-31.73
freepizza	pizza6	how[A]≠-B how[C]≠B	100%: pp[C]>pp[A]	pp[C]>pp[A]	12%	14.37-14.74
concert-hall	concert-cap02 concert-cap03	assign[A]=0 assign[B]≠C assign[D]≠C	100%: D<C 66.66%: pp[D]>pp[C] ...	-	12%	7.9-7.32
tcgc2	k2_42 k5_05	a[A,B]≠a[C,B] a[A,B]=D a[C,B]≠D	100%:A>B ...	-	6%	51.66-51.83
grid-colouring	4_8 4_11	x[A,B]≠x[C,D] x[E,F]≠x[G,H] x[A,I]≠J x[K,L]≠J x[M,N]≠J	100%:A<M 94.73%:G=M, ...	-	40%	51.66-51.83

Table 6.1: The top patterns and their associated facts

the results. The next four columns provide information regarding the input size of the data processed by our framework: The `Cl` column shows (up to a maximum of 50 top clauses for each instance); the `Lits` column shows the average number of literals in the clauses; the `Vars` column shows the average number of variables in the pattern that have the same type; the `Pars` column shows the average number of parameters that are indexed by the same type. The next three columns represent show the execution time of different components of the framework in seconds: `P` shows how long it takes to generate patterns; `F1` shows how long it takes to generate facts using the percentage-based approach; `F2` shows how long it takes to generate facts using the ConAcq-based approach. Also, we did not show the execution time required to rename and simplify the clauses, because it is negligible in all benchmarks.

The results in Table 6.2 showed us that the framework is often very fast, typically performing its functions in less than a second. However, for `oc-roster` with data `2s-200d`, and `team-assignment` with data `data1_6_24`, the execution time for generating patterns are 14.2s and 11.2s, respectively. This is because the number of clauses that share the same patterns are quite high for these cases.

Table 6.3 displays the efficiency of our method for simplifying and renaming the literals and clauses. We present the results in two categories of fixed and free search. The `Model` and `Instance` columns are as before. The next four columns show, for `Fixed` search, the following information: `Avg.`, `Min` and `Max` show the average, minimum and maximum number of literals, respectively, reduced for each clause due to the simplification; and `Cl` shows the percentage of clauses for which all literals were successfully renamed. For some

Model	Instance	Solving details				Input size				Time(s)		
		N	F	S	T	Cl	Lits	Vars	Pars	P	F1	F2
concert-hall-cap	**02**	452K	437K	34	3	50	20.1	0.4	8.0	0.3	0	0.2
concert-hall-cap	**03**	390K	364K	20	3	50	17.4	0.3	6.7	0.2	0	0
oc-roster	**2s-200d**	197K	3K	0.5	1.5	50	99.6	1.9	4.3	14.2	0.1	0
oc-roster	**4s-100d**	822K	68K	0	3	50	64.3	1.4	5.1	5.1	0.1	0
opt-cryptanalysis	**r6**	26K	18K	4	3	10	190.3	0	1	0.1	0	0
opt-cryptanalysis	**r8**	15K	11K	5	3	49	581.2	2.9	1.8	2.9	0	0
seat-moving	**sm-10-12-00**	232K	154K	3.5	3	50	13.6	0.2	1.1	0	0	0
seat-moving	**sm-10-20-05**	117K	141K	14.5	1.5	50	17.4	0.5	1.3	0.2	0	0
steiner-tree	**10**	3K	3K	2.5	0.1	50	16.3	0.1	1.8	0.5	0	0.4
steiner-tree	**es10fst03.stp**	2K	2K	5.5	0.1	50	14.9	0.1	1.9	0.4	0	0.4
team-assignment	**data1_6_24**	9K	3K	22	3	50	363.1	0.4	2	11.2	0	11.6
team-assignment	**data2_4_15**	32K	261K	52	3	50	38.9	0.1	2.2	2	0	0.8
train	**instance.4**	103K	78K	9	3	36	18.2	0.1	3.1	0.1	0	0.1
train	**instance.5**	229K	63K	1,050	3	50	11.5	0.1	3.5	0.1	0	0.3
vrplc	**vrplc9_5_10_s1**	78K	33K	2.5	3	50	42.4	0.1	6.0	0.1	0	0
vrplc	**vrplc9_5_10_s3**	80K	74K	5	3	50	11.5	0.1	3.5	0.1	0	0.3

Table 6.2: The execution time of different components of the framework

of our models the entry for column `Cl` is 0, because there was no clause contributed in reducing the search space. Thus, our system did not process them, and the next four columns show the same information in free search.

The results from Table 6.3 show that our method for simplifying clauses is practical, and is able to reduce the length of the clauses in most cases. Interestingly, in the fixed search category it reduced 63 literals of a clause for `seat-moving` with data `sm-10-12-00`, and in the free search category 71 literals. As the length of these clauses were significantly reduced, this further reduced the execution time of our framework to process them.

6.4 Case Studies

This section discuss in more detail these case studies where the framework helped improve the models.

6.4.1 Grid Colouring Problem

The method allowed us to discover a redundant constraint that can be added to the model of the `grid-colouring` problem. A simplified version of the MiniZinc model for this problem is presented below. The loop on lines 4 − 8 enumerates all sub-rectangles of the grid, and states that at least one pair of orthogonally adjacent corners must be assigned distinct colours. Looking at the clauses produced for instances of this problem, we found a frequent, high-ranking pattern with literals `{x[A,B]≠x[A,C],x[A,B]≠x[D,B], x[A,C]≠E,x[D,B]≠E,x[D,C]≠E}`. Note that this pattern is different from the one provided

Model	Instance	Fixed				Free			
		Avg.	Min	Max	Cl	Avg.	Min	Max	Cl
concert-hall	**02**	0	0	1	100	0	0	10	47.3
concert-hall	**03**	6.3	0	9	67	6	0	10	67.7
oc-roster	**2s-200d**	0.1	0	3	0.6	0.1	0	3	0.0
oc-roster	**4s-100d**	0.5	0	5	8.8	0.6	0	5	9.0
opt-cryptanalysis	**r6**	0	0	0	0	0	0	0	0
opt-cryptanalysis	**r8**	1.3	0	5	0	1.3	0	5	0
seat-moving	**sm-10-12-00**	12.4	0	63	62.8	12.4	0	63	62.8
seat-moving	**sm-10-20-05**	0	0	0	0	12.7	0	71	100
steiner-tree	**10**	0	0	0	0	0	0	16	5.3
steiner-tree	**es10fst03.stp**	0	0	18	2	0	0	18	2
team-assignment	**data1_6_24**	3.4	0	14	0	3.4	0	14	0
team-assignment	**data2_4_15**	3.8	0	11	39.8	3.7	0	11	37
train	**instance.4**	0.8	0	2	100	0.7	0	3	100
train	**instance.5**	0	0	0	0	0	0	3	100
vrplc	**vrplc9_5_10_s1**	1.5	0	19	1.75	1.5	0	19	1.7
vrplc	**vrplc9_5_10_s3**	0.9	0	7	0	0.9	0	7	0

Table 6.3: Efficiency of the renaming and simplifying method

```
int: n; int: m; % Width and height of grid
array[1..n,1..m] of var 1..min(n,m): x;       % Colour in each cell

constraint forall(i in 1..n, j in i+1..n, k in 1..m, l in k+1..m)(
                  (   x[i,k]!=x[i,l] \/ x[i,l]!=x[j,l]
                   \/ x[j,k]!=x[j,l] \/ x[i,k]!=x[j,k]) );

solve minimize max(x); % Number of colours used
```

in Table 6.1, because this is not the pattern with the highest rank, but we could not improve the model based on the pattern provided in Table 6.1. One interpretation of the pattern states that if corner `x[A,B]` has the same colour as `x[A,C]` and `x[D,B]`, then one of `x[A,C]`,`x[D,B]` or `x[D,C]` must be assigned a different colour. Upon manual examination of the model, it became clear that the constraints in the model only indirectly compared diagonally adjacent corners. To address this weakness, we added the following redundant constraints:

```
constraint forall(i in 1..n, j in i+1..n, k in 1..m, l in k+1..m)
                ((x[i,l]=x[j,k] /\ x[i,l]=x[j,l]) -> (x[i,k]!=x[i,l]))
```

These constraints did not improve Chuffed's performance but did improve Gecode's significantly. Table 6.4 shows Gecode's solve time (in seconds) and the node count for several instances of the original model and the modified one. While the number of extra constraints result in Gecode spending more time at each node , the added propagation leads to faster solve times. Note that the experiments were performed within the time limit of 5 minutes, and ∞indicates that solving was not completed with the time limit.

6.4.2 Time Changing Graph Colouring

Our framework helped us discover redundant constraints that helped improve the solving time. The following shows an extract from the MiniZinc model used. The model takes as input, among others, the maximum number `k` of transformations per step, and sets the maximum number of steps `max_s` to 10. It has an array of decision variables, where `a[i,n]=j` represents the colour `j` of node `n` in step `i`, and a decision variable `s` representing the final number of steps. The constraint displayed states that in every non-final step `i`, the sum of all transformations must be less or equal to `k`.

```
   int: k;                              % maximum colour changes per step
   int: max_s = 10;                     % maximum number of steps
   array [STEPS,NODES] of var COLORS: a; % Colours of nodes at each time step
   var 2..max_s: s;                     % final number of steps
   ...

51 constraint forall(i in 1..max_s-1)
52            (i < s -> sum (n in NODES) ( a[i,n] != a[i+1,n] ) <= k);
53 ...
```

Our method identified pattern {`a[A,B]`$\neq$`a[A+1,B], a[A,B]=C, a[A+1,B]`$\neq$`C`} as interesting, since its clauses are highly ranked across different searches of different instances and, when combined, are responsible for a large search reduction. Note that this pattern is different from the one provided in Table 6.1, because this is not the pattern with the highest rank, however, we could not improve the model based on the highest ranked pattern provided in Table 6.1. Further, the pattern is short (and thus easy to understand) and all its clauses come from a single model constraint (which often indicates lack of expected propagation by the constraint). Upon manual examination, we felt the pattern stated something so simple it should have already been captured by the propagation of the model constraints. The first literal, derived from the expression highlighted on line 52, represents the result of a reified not-equals constraint (a reified constraint introduces a boolean variable to represent the truth value of a constraint [63]). The literal is true when the variables `a[i,n]` and `a[i+1,n]` take different values. If this is false, the remaining literals state that either the variables both take the value `C` or they take some other value. This becomes clear when presented in implication form:

{`a[A,B]=a[A+1,B]` $\wedge$ `a[A,B]`$\neq$`C` $\rightarrow$ `a[A+1,B]`$\neq$`C`} and {`a[A,B]=a[A+1,B]` $\wedge$ `a[A+1,B]=C` $\rightarrow$ `a[A,B]=C`}.

To improve the propagation, we manually modified the model to add the following information:

```
1 array[1..max_s-1,NODES] of var bool: aa;
2 constraint forall (i in 1..max_s-1, n in NODES)
3            (aa[i,n] <-> forall (c in COLORS) (a[i,n]=c <-> a[i+1,n]=c));
4 constraint forall (i in 1..max_s-1)
5            (i < s -> sum (n in NODES) ( (not aa[i,n] ) ) <= k);
6 . . .
```

Line 1 introduces the decision variable `aa` is a two dimensional boolean array, where the first dimension `1..max_s-1` represents the steps and the second one `NODES`, represents

		Original		Modified	
n	m	time (s)	nodes	time (s)	nodes
5	5	0.10	34,987	0.07	5,452
5	6	0.13	35,223	0.09	5,468
6	6	0.55	131,661	0.29	16,773
6	7	1.53	9,484,042	0.77	37,727
7	7	65.47	11,565,900	23.15	904,148

Table 6.4: Gecode's solving times for different instances of the `grid-colouring` problem

	Fixed		Free		Alternate	
Instance	Original	Modified	Original	Modified	Original	Modified
k5_5	48.48	**40.54**	**83.68**	86.20	**66.65**	86.62
k9_39	∞	∞	∞	∞	**262.28**	295.16
k10_31	∞	∞	∞	∞	78.35	**63.81**
k10_34	23.66	**21.18**	**80.65**	91.60	77.57	**69.59**
k10_41	**1.67**	1.99	**17.80**	19.60	**3.15**	3.25

Table 6.5: Solving times for different instances of `tcgc2` using different search strategies.

the nodes. For example, `a[i,n]=True` represents that node `n` in step `i` is assigned to True. This indicates that the colour of node `n` did not change from step `i` to `i+1`. Line 3 ensures that for each node `n` at each step `i`, if `a[i,n]=a[i+1,n]` for all the colours, `aa[i,n]` is set to True. Line 5 ensures that for each step `i`, if `i` is not the final step, the number of the nodes that their colour did not change from step `i` to `i+1` (`not aa[i,n]`), should be less than the maximum number of colour changes allowed in each step (`k`).

Table 6.5 shows the solve times for instances of the original and modified model. Three different search strategies with a 5 minute time limit were executed. The strategies were the fixed strategy defined in the model, Chuffed's free search strategy, and, one that alternated between the two. The results show the constraint can improve Chuffed's solving performance on some (but not all) instances of the problem. For a traditional CP solver with a domain consistent reified not-equals constraint, these extra constraints will slow down propagation, and an annotation indicating the need for domain propagation would be preferred. This shows the strong need for the modeller's input.

6.4.3 Capacitated Concert Hall Problem

Our framework helped us strengthen a model constraint in the `concert-hall-cap` model and, more importantly, helped us improve a MiniZinc global constraint. In this case study, our focus is on the first constraint which ensures concurrent events can not be assigned to the same hall. The MiniZinc implementation of this constraint is provided below:

```
include "globals.mzn";
int: num_offers;
int: num_halls;
set of int: Offer = 1..num_offers;
set of int: Hall = 0..num_halls;
array [Offer] of int: start;
array [Offer] of int: end;
array [Offer] of int: price;
set of int: Time = min(start)..max(end);
array [Offer] of var Hall: assign;
function bool: overlaps(Offer: o, Time: t) = start[o] <= t /\ t < end[o];
array [Offer] of set of Offer: cliques = [{p|p in Offer where overlaps(p, start[o])
    }|o in Offer];
function bool: clique_is_maximal(set of Offer: c) = forall (d in cliques where card
    (d) > card(c)) (not (c subset d));
%% Overlapping events cannot share a hall.
constraint forall (clique in cliques where clique_is_maximal(clique)) (
    alldifferent_except_0([assign[o] | o in clique]));
```

Line 1 includes the MiniZinc library of global constraints. Lines 2 and 3 introduce the `num_offers` and `num_halls` parameters, respectively. Line 4 defines a set that represents the offers, with domain 1..`num_halls`. Similarly, to represent the halls line 5 defines the `Hall` set with domain 0..`num_halls`, where 0 indicates no hall. Lines 6, 7 and 8, introduce the `start`, `end` and `price` parameters, which represent the starting time, ending time and price of each event, respectively. Line 9 introduces the `Time` set, which represents the time from when the earliest event starts to the time the last event ends. Line 10 introduces the decision variable `Hall`. Line 11 defines the `overlaps` function, which takes as input the arguments `Offer O` and `Time t`, and checks if the offer `o` overlaps with `Time t`. This happens if `Offer O` starts before `Time t` (`start[o]<=t`) and it ends after `Time t` (`t<end[o]`). Line 12 introduces `cliques` which maps each offer to a set of offers that overlap that offer. To compute this, it iterates through `Offer` (`o in Offer`) and for each offer `o`, it again iterates through the offers (`p in Offer`), and checks if offer `p` overlaps offer `o`. This forms a `clique` that maps each offer to its overlapping offers. Line 13 defines the `clique_is_maximal` function. The input argument of this function is a `clique c`, which is a set of `Offer`. It goes through all the `cliques` that have more members than `clique c` (`card (d)>card(c)`), and if there is a `clique d` that `clique c` is subset of, the function returns false. Otherwise, if there is no `clique` that is a subset of `clique c` it will return `true`, indicating that `clique c` is maximal. Line 15 goes through each `clique` that is maximal, and imposes the `alldifferent_except_0` global constraint. This constraint takes an array as an input argument and ensures that all the elements in the array are different from one another, except the ones that are zero. `[assign[o] | o in clique]` creates an array of the `assign` decision variables that are indexed by `Offer o` in a `clique`, which are basically the overlapping offers. The `alldifferent_except_0` ensures that these overlapping offers are either unassigned (assigned to 0), or they are assigned to different halls.

One of the top patterns that our method identifies is: $\{$`assign[A]`$\geq$`B`, `assign[A]`$\leq$`C`, `assign[D]`$\geq$`B`, `assign[D]`$\leq$`C`, `assign[E]`$\geq$`B`, `assign[E]`$\leq$`C`$\}$, and it comes from the `alldifferent_except_0` constraint in line 15. By inspection, we discovered the fact: `B-D =3`. Our automated method was not able to discover this fact because, as mentioned in Chapter 5, our method for inferring facts is limited, and it does not consider summation and subtraction relations. Hence, this pattern states that if we have a range of two values between `B` and `C`, and 3 variables `assign[A]`, `assign[D]` and `assign[E]`, at least one of them should be assigned to a value outside of this range, which is trivial since we only have two values.

From this experiment, we realised that the relation described by the pattern above is not explicitly expressed in the model. Therefore, we added the following redundant constraint:

```
constraint forall(c in index_set(cliques) where clique_is_maximal(cliques[c]))(

     forall(h in Hall) (
          sum(o in Offer where o in cliques[c])(assign[o]=h)<=1 )
  );
```

Line 1 goes through each `cliques[c]` if its maximal. Line 2 goes through each hall `h`. Line 3 ensures that the number of the offers in `cliques[c]` that are assigned to the hall `h` must be at most one.

However, adding this constraint did not have a significant effect in the execution time of either Gecode or Chuffed. The trade off of adding redundant constraints to the model is that it increases the computational costs, and even though our method might indicate that a pattern contributes to a large reduction in search space, it does not guarantee that the time would be saved by adding that pattern to the model. Therefore, we decided to try modifying the constraint in line 15. For this purpose, we referred to the MiniZinc implementation of the `alldifferent_except_0` global constraint, which is provided below:

```
predicate alldifferent_except_0(array[int] of var int: vs) =
    forall(i, j in index_set(vs) where i < j) (
       (vs[i] != 0 /\ vs[j] != 0) -> vs[i] != vs[j]
    );
```

The `alldifferent_except_0` predicate accepts an array as an input argument. It then goes through each pair of the elements in the array, represented by `i` and `j`, and if both of the elements are not 0, it ensures that they have different values. To obtain faster propagation, we modified the implementation of `alldifferent_except_0` is as follows:

```
predicate modified_alldifferent_except_0(array[int] of var int: vs) =
    forall(i, j in index_set(vs) where i < j) (
            (vs[i] != 0) -> (vs[i] != vs[j])
      /\ (vs[j] != 0) -> (vs[i] != vs[j])
    );
```

Both implementations are logically equivalent, but in the original implementation both of the conditions `vs[i]!=0` and `vs[j]!=0` must hold to activate the propagation of constraint `vs[i]!=vs[j]`, while, in our modified version, if any of the conditions `vs[i]!=0` and `vs[j]!=0` holds, the constraint propagates, which helps reduce the search space. This improved the propagation of `alldifferent_except_0`. Further, improving `alldifferent_except_0` can help us improve solving time for a variety of models. Afterwards, to further improve the performance, we replaced the `alldifferent_except_0` in line 15 with the following constraint:

```
constraint forall (c in index_set(cliques) where clique_is_maximal(cliques[c])) (
    forall(h in Hall) (
         sum(o in Offer where o in cliques[c])(assign[o]=h) <= 1
     )
     );
```

This constraint goes through each maximal clique `cliques[c]`, and then goes through each hall `h`, and ensures that at most one of the overlapping offers can be assigned to the hall `h`. This constraint is logically equivalent to the `alldifferent_except_0` global constraint. However, instead of imposing a constraint on each pair of elements in each maximal clique, it imposes a single constraint on all the elements in the clique. Hence, it has a global view of all the elements in each maximal clique. This did not have a significant effect on Gecode solving time, but improved the solving time of Chuffed, comparing to the original model. However, the solving time is still slower than our modified version of `alldifferent_except_0`.

	Fixed(s)			Free(s)			Alternate(s)		
Instance	Org	Mod1	Mod2	Org	Mod1	Mod2	Org	Mod1	Mod2
concert-cap-2	**120.2**	**0.4**	**15**	240.3	**0.5**	25.10	**22.72**	**0.4**	12.3
concert-cap-3	∞	**0.4**	240	∞	**0.4**	440.1	∞	**15**	240.4
concert-cap-6	∞	**0.7**	∞	∞	**0.6**	∞	∞	**0.9**	∞
concert-cap-148	∞	**0.4**	∞	37	**0.3**	24	∞	**0.4**	∞

Table 6.6: Chuffed solving times for different instances of `concert-hall-cap` using different search strategies.

	Fixed(s)			Free(s)		
Instance	Org	Mod1	Mod2	Org	Mod1	Mod2
concert-cap-2	**49.8**	**0.4**	60.1	**41**	**0.5**	49.8
concert-cap-3	∞	**0.5**	∞	∞	**0.3**	∞
concert-cap-6	∞	**0.5**	∞	∞	**0.5**	∞
concert-cap-148	∞	**0.4**	∞	∞	**0.3**	∞

Table 6.7: Gecode solving times for different instances of `concert-hall-cap` using different search strategies.

Tables 6.6 and 6.7 indicate the solving time of Chuffed and Gecode for a variety of instances of `concert-hall-cap`, respectively. For Chuffed we considered fixed, free and alternate search strategies, but for Gecode only fixed and free search strategies, because Gecode is not able to perform alternate search strategy. For each search strategy there are three columns: `Org` shows the solving time of the original model; `Mod1` shows the solving time of the model with the replacement of the `alldifferent_except_0` with the `modified_alldifferent_except_0`; `Mod2` shows the solving time of the model with the replacement of the `alldifferent_except_0` with the summation constraint. From the tables, it is clear that `Mod1` gives a significant improvement for both of the solvers, while, `Mod2` only improves Chuffed.

6.5 Summary

This chapter discussed the results of the experiments carried out to determine the efficiency and effectiveness of the framework. In these experiments given a model and instance, our method automatically infers the clauses, renames and simplifies them, detects the patterns in the clauses, and learns facts for each pattern. Our experiments showed that the framework is capable of helping users improve their models and is not prohibitively time-consuming to run. The case studies that arose from these experiments demonstrate how the framework can not only improve constraints in a user's model, but can also be used to improve the MiniZinc's standard decomposition of global constraints.

Chapter 7

Related Work

7.1 Introduction

This chapter discusses the literature relevant to the framework we have presented in this thesis. Section 7.2 discusses the area of anti-unification, which is concerned with automatically obtaining the Most Specific Generalisation (MSG) of a group of clauses. This is related to the approach we use for generating patterns from a set of clauses, as discussed in Chapter 4. Section 7.3 discusses Inductive Logic Programming (ILP), which is focused on learning first-order clausal theories from a set of examples and background knowledge by inductive inference [52]. This is related to the approach we used for inferring facts from a set of clauses, presented in Chapter 5. Section 7.4 describes and compares the existing techniques of automated constraint acquisition, while Section 7.5 describes the existing techniques in the area of automated model transformation. Since our goal is to automatically learn redundant constraints or reformulate model constraints, the two mentioned areas of automated constraint acquisition and model transformation have similar goals, but different approaches. Lastly, Section 7.6 provides a conclusion for this chapter.

7.2 Anti-Unification

As mentioned in Chapter 4, our approach for generating patterns from a set of clauses is concerned with finding the Most Specific Generalisation (MSG) of a group of clauses. This lies within the *anti-unification* line of research [57]. As descussed in Chapter 4.2, the clause c_1 is said to be more general than the clause c_2 ($c_1 \preceq c_2$) if and only if all the examples that are covered by c_1 are also covered by c_2 [25], and the most specific (or least) generalisation (MSG) of a set of clauses is the least general clause that is more general than all the clauses in the set [56]. The anti-unification problem is concerned with finding the MSG of a set of clauses.

Plotkin et al. [57] and Reynolds et al. [62] independently introduced the concept of MSG and provided algorithms to compute it. In these algorithms, the MSG is unique up to renaming and always exists for two given clauses [18], this is the algorithm that we implemented to compute patterns, as discussed in Chapter 4.2. One of the limitations of their algorithms is that they could only obtain MSGs for clauses with a fixed number of literals. To mitigate these limitations and expand anti-unification to different threories and applications, several algorithms have been developed, such as these in [3, 4, 7, 18, 6, 28]. As in-depth discussion of many methods and applications of anti-unification is out scope for this thesis. Instead, we focus on one of the recent methods of anti-unification developed by Kutsia et al. [38], which can be incorporated in our method for generating patterns.

Kutsia et al. developed a technique which performs anti-unification for clauses with different numbers of literals. For example, for clauses `{w[1]<3,w[2]<3}`, `{w[1]<3,w[2]<3, w[3]<3}` and `{w[1]<3,w[2]<3,w[3]<3,[w4]<3}`, the MSG `{w[x]<3| x ∈ 1..n}` can be obtained. Also, in their system they can match a literal in a certain position in the first clause to a literal in a different position in the second clause. For example, for clauses `{w[10]=1,w[20]=2}` and `{w[40]=2, w[50]=1}`, their system can yield `{w[A]=1,w[B]=2}`. Our method for generating patterns is not able to match clauses with different numbers of literals, and cannot match the literals that are in different positions in the clauses either. However, this technique has a high computation cost. It would be interesting to explore how to incorporate anti-unification to generate patterns, while keeping the cost low, perhaps by only considering the top ranking ranking clauses.

7.3 Inductive Logic Programming

In Chapter 5 we discussed our method for learning relevant facts for a set of clauses. This is related to research performed in the area of Inductive Logic Programming (ILP). ILP bridges a gap between machine learning and logic programming [50]. It is a discipline concerned with learning first-order clausal theories from a set of examples and background knowledge by inductive inference [52]. ILP is powerful compared to the classical machine learning systems because it mitigates the difficulties that some of the machine learning systems face in incorporating background knowledge in the learning process [50].

The ILP algorithms often explore a search space consisting of a set of hypotheses using a divide and conquer approach [39]. They search for a hypothesis that covers part of the examples, and separates positive and negative examples. Then, they repeatedly search for the examples that are not covered and assign them to a hypothesis, until all the examples are covered by at least one hypothesis [39].

There are two main ILP approaches: top-down and bottom-up. In the top-down approach used for example by the Foil [60], Claudien [26] and Mobal [37] systems, the search starts with the most general hypothesis and tries to specialise it with respect to the background knowledge. Search continues until all positive examples are covered by the hypothesis, and none of the negative ones. The bottom-up approach, implemented by the Golem [53], Cigol [51] and Marvin [64] systems, starts with the most specific hypothesis, and further generalises it with respect to the background knowledge [74].

Both approaches suffer from some drawbacks: bottom-up approaches can be inefficient and time-consuming at the beginning of the search when the background knowledge is large [74], while top-down approaches might spend time exploring areas that cover no positive example. If the search space is large and unstructured both of these strategies fail. Hence, hybrid approaches have emerged to address these issues, such as those of [74, 39].

One of the most recent ILP techniques was developed by Lallouet et al. [39]. Their work is at the intersection of CP and ILP. Their goal is to automatically acquire a constraint model, given solutions and non-solutions of related problems. To achieve this, they developed a constraint acquisition framework based on ILP, which has two main components: First, given solutions, non-solutions and background information, their framework learns theories that cover the solutions and reject the non-solutions. Second, it reformulates the learned theories into a CSP. For the ILP part they developed a hybrid ILP algorithm by combining bottom-up and top-down approaches to achieve a good trade-off, and also their system is tailored for CSP problems. This was required because, most of the ILP techniques handle Disjunctive Normal Form (DNF), however, CSP problems are expressed in Conjunctive Normal Form (CNF), and the second component of their framework is out the scope of this thesis.

As mentioned before, our current method for learning relevant facts misses some of these facts. Thus, it would be interesting to explore how ILP can be blended into our method to help us learn more relevant facts. However, one of the challenges is that the research in ILP has been slowed down, and to the best of our knowledge the existing frameworks are no longer maintained.

7.4 Automated Constraint Acquisition

Correctly articulating the constraints of a problem can be challenging [55]. Thus, many techniques have been implemented to automate the acquisition of the constraints for CSPs/COPs. Given a set of solutions, non-solutions and a constraint library, a constraint acquisition system learns a constraint network that is consistent with the solutions and non-solutions [12]. Although this area is related to this research, it tackles the problem of automated constraint acquisition from a very different perspective to ours: while our aim is to improve an already existing model. However, we can use their technology to infer conditions for our clauses, as we did in Chapter 5.4.

In particular, constraint acquisition systems acquire the target model from scratch, however, in our case an initial model is given to the system, and we aim at improving this model.

There are two main classes of constraint acquisition systems: passive and active. In passive systems [13, 12, 17, 10, 39], the modeller chooses the set of examples in advance and independently of the acquisition process, whereas active systems [31, 17, 14] assist the user to choose more helpful examples, with the aim of reducing the number of examples required.

One of the earliest passive constraint acquisition systems is ConAcq [12], which we discussed in Chapter 2.7.1. ConAcq is implemented based on version space learning [48] and SAT-solving [29]. While powerful, it is not efficient when the size of the constraint library is large, and it also focuses on an instance of the problem, rather than on the model. Another limitation is that the user needs to provide a full set of solutions. To mitigate these limitations, Bessiere et al. [14] improved ConAcq by developing an active query-driven constraint acquisition system was reduced. Thus, a user does not need to have all the solutions in advance, and they reduced the number of queries required during the acquisition process. The method of Bessiere et al. also detects the model parameters from the solutions. Although their work has great potential, the main limitation is that it requires an exponential number of queries to acquire constraints [16].

As we described in Section 7.2, Lallouet et al. [39] developed a passive constraint acquisition system with the aim of mitigating some of the limitations of ConAcq. Their system can acquire the constraints of a CSP model from the solutions and non-solutions of a different instance of the problem. Their method learns rules from the solutions and non-solutions of a previously solved problem with different parameters, and then the rules are rewritten as CSP specifications. Thus, the modeller does not need to have solutions and non-solutions of the problem in advance. As described in Section 7.2, it might be interesting to incorporate their technique into our method for learning facts to obtain more facts.

One of the most recent constraint acquisition systems is QuAck [11], an active constraint acquisition system that aims at reducing the number of queries required to converge to the target model. Their method asks partial queries that the user should identify as a solution or non-solution. If the user classifies a partial query as a solution, the constraints that are violated by this solution are eliminated. If the user classifies it as a non-solution, QuAck asks partial queries to identify the scope of the violated constraint to learn that constraint. The advantage of their method is that the problem does not need to be previously solved, and the number of queries required for convergence is significantly reduced. However, the number of queries can still be exponential. A more efficient system is MultiAcq [5], which learns multiple constraints from negative examples, and only require a number of queries linear to the size of the problem. However, the required time for generation of queries is large. To compromise between QuAck and MultiAcq, MAcq-Co combines these systems to learn multiple constraints from a negative example, while requires less time to generate queries than MultiAcq. In addition, it generates shorter queries, which are easier to answer [5]. Another state of the art system is QuAck [75] which improved QuAck and MultiAcq by generating fewer queries (logarithmic in the size of input) in a shorter amount of time.

Our method that we described in Chapter 5.4 uses ConAcq to infer facts. The reason is that passive constraint acquisition systems are more suitable for our technique, because our method is automatic, and thus we aim at minimising the need for user supervision, and also the ConAcq algorithm is easier to implement. However, in future it would be interesting to implement more recent passive constraint acquisition systems.

7.5 Automated Model Transformation

As mentioned before, our aim is to automatically improve the models by adding effective redundant constraints or reformulating the model constraints by learning from the clauses learnt by a learning solver. There are several techniques that could help modellers improve their model. This subsection focuses on the most popular techniques and their associated systems. One of the most popular techniques is to add effective redundant constraints.

Redundant constraints are implied by the model constraints and adding them to the model does not eliminate any of the solutions. Adding redundant constraints can help better exploit the structure of the problem and reduce the search space [19]. However, it can also increase the computational cost due to the propagation of the added constraint. Hence, the challenge is to find an effective redundant constraint, that is, one where the time saved by reducing the search space, outweighs the increased computational burden.

Another popular technique used to improve models is to detect symmetries and add symmetry breaking constraints to the model. Symmetries, in the context of CSP, are solutions that are equivalent to each other [34]. Thus, breaking symmetries in the model can improve solving and prevent similar search space being explored over and over. A popular method for breaking symmetries is to add constraints (known as static symmetry breaking constraints) to the model.

Replacing some of the model constraints with an equivalent global constraint can often help improve the model by providing a better view of the structure of the problem. Global constraints are pre-defined structures that encapsulate common sub-problems that can be reused, and many solvers utilise specialised algorithms and efficient propagators for them [40]. Hence, substituting a group of constraints with a global constraint can reduce the search space and improve the model [15]. There are different works to automatically transform models by adding redundant constraints [33], symmetry breaking constraints [46] and global constraints. The following discusses a few example of some of these methods.

Miguel et al. developed CGRASS [33], a rule-based system that automatically generates redundant and symmetry breaking constraints for CSP models. The focus of this system is on improving the instances rather that the model itself. Further, the method can only detect `alldifferent` global constraints and it is thus not general. Colton et al. [23] developed a system that first solves small instances of the model, and then passes the solutions to an automated theorem formation system(HR [22]), to learn and output redundant constraints, while their method is powerful, however it is semi-automatic and the modeller has to analyse and prove the correctness of the output generated by HR and translate it for the modelling language [23]. To tackle these limitations Charnley et al. [19] developed a fully automated method based on the system of Colton et al., which uses the Otter theorem prover [45] to prove the correctness of the constraints learned by HR and then translates them to the Essence constraint language. Although their technique is fully automated, it has significant restrictions: their model can only be parametrised by a single integer and must be expressed in first-order logic. The reason for these restrictions is that proving the correctness of the redundant constraints imposes restrictions on the generality and efficiency of their approach. This is one of the reasons we do not prove the correctness of the constraints learnt by our method; As a result, our method is very generic and can be applied to any COP/CSP model.

The automated detection of both symmetry and global constraints are in a different line of research, but share some similarity with our work: They use instances to infer model information, and require the user to prove that the generated information is correct.

However, it is worth mentioning some of the work in this area. The system developed by Mears et al. [46] is an example of model transformation by automatically adding symmetry breaking constraints. Their method detects symmetries in instances of a problem, finds the possible symmetries and then generalizes them to the model. While they achieved great results in their evaluation, one of the issues with the proposed method is that it only discovers known symmetries. Another contribution in this area is the work of Puget et al. [59], which constructs a graph for a given CSP problem. Afterwards, the symmetries are found using a graph automorphism algorithm. The proposed method is powerful, but fails to detect some of the existing symmetries in the graph, and is only focused on, instances rather the whole problem class. This method provides automatic proving for a certain class of symmetries, thus it would be interesting to explore incorporating their technique into our framework, to automatically prove the correctness of some of our the constraints (facts combined with patterns) that our system generates.

An example of constraint acquisition systems that detects global constraints present in the model are Constraint Seeker and Model Seeker, both developed by Beldiceanu et al. [8, 9]. Given positive examples to the Constraint Seeker system, the systems arrange them as a matrix and detect the global constraints, and ranks these constraints. Model Seeker learns an entire model, given positive examples. This system uses Constraint Seeker to acquire problem constraints, it then simplifies the generated candidate constraints and transform them into a model. Another work in this area is the system developed by Leo et al. [41], given a constraint model as input, suggests global constraints to replace parts of the model. For this purpose, the original model is divided into sub-models consisting of different structures of the model. Then, the sub-models are compared with the library of global constraints. Global constraints with a similar solution space to the sub-models are considered as candidate global constraints. Afterwards, they present a ranking technique that filters and ranks the candidate global constraints to find the most accurate ones. These are then suggested to the user. However, there are some global constraints that the system is unable to detect. In addition, some of the complex structures in the model are not processed. This system considers an entire model rather than a problem instance, but it does not prove the correctness of the learned global constraints.

Although several techniques exist to automate the modelling of CP problems, none of them is taking the learnt clauses into consideration. As mentioned in Section 2.4, there is a possible relationship between model constraints and clauses that can help with the process of automated model reformulation, and in this thesis our aim is to explore this research gap. In addition, most of the above mentioned techniques improve the instances rather than the model of the problem, but our method aims at improving the entire model.

7.6 Summary

This Chapter discussed and compared works related to our method. Firstly, Section 7.2 discussed an existing anti-unification technique that can be incorporated into our method to learn more accurate patterns. Then, Section 7.3 presented some available ILP techniques, which can be blended into our method for generating facts from a set of clauses, as our system currently suffers from some limitations that prevent it from learning useful facts for some of the patterns. Afterwards, Section 7.4 discussed the area of automated constraint acquisition, and compared the existing works in that area. While automated constraint acquisition shares some goals with our method, their focus is on acquiring the whole model from a set of solutions and non-solutions, rather than on inferring effective redundant constraints. Lastly, Section 7.5 discussed the related works in the area of model transformation, which aims to improve the model by automatically detecting information, such as global constraints, symmetry breaking constraints and redundant constraints, that can be used to transform the model. While there exist powerful techniques in the area of model transformation, none of them take learnt clauses into account, which is the main novelty in the approach presented in this thesis.

Chapter 8

Conclusions

COPs/CSPs appear in many different areas of our lives, such as public transportation, health care and management. Unfortunately, modelling and solving these problems is quite difficult and consumes significant amount of time. Further, there are many possible ways in which a problem can be modelled and the associated solving time can dramatically vary from one model to the next. Therefore, a great deal of expertise is required to develop a model that can be solved quickly. Recently, Shishmarev et al. [68] discovered that the clauses learnt by learning solvers can be used to improve the models. However, their method was mostly manual and suffered from certain limitations, such as they only considered one instance of the problem which restricted the applicability of their results.

This thesis presented a framework that automates and improves the work of Shishmarev et al. [68]. In particular, we have shown how the literals in the clauses learned by the solver for a set of instances can be automatically renamed, simplified and tied back to the parts of the model they originated from. Then, we presented a method that groups the renamed and simplified clauses into generic patterns. We then described a method that combines the background information with patterns to derive relevant facts for these patterns. Then, we presented our experiments and results that were carried out to evaluate the practicality and efficiency of the framework. In this section, we summarise the contributions of this thesis, and discuss the potential future work.

Our main contribution is the definition and implementation of a framework that automates Shishmarev et al.'s manual approach, and improves some parts of their method. The main contributions of this framework can be divided into the following contributions:

- The framework automatically renames and simplifies the learnt clauses. Renaming the clauses is achieved by using the *variable paths* technique developed by Leo et al. [42], which traces the model variables during compilation and connects them to their original name in the model. This helps us increase the accuracy of later steps (such as simplification and pattern generation) Then, our method simplifies the learnt clauses by applying simplification rules that reduce the length of the clauses. This in turn reduces the time required by the framework to generate patterns and facts, and also helps us detect more patterns, and ranking them more accurately.

- The framework automatically connects the learnt clauses to the model and detects the model constraints that were involved in creating them. This is achieved by instrumenting a learning solver (Chuffed) to record the solver-level constraint directly responsible for adding any literal to the clause database. The connection helps the modeller detect model constraints that can be strengthened, or identify the need to add a redundant constraint to the model that can help reduce the solving time.
- The framework automatically generalises groups of clauses into patterns. This helps us obtain more accurate ranking for a group of clauses by accumulating the estimated reduction in search space of the clauses associated to a pattern, and also it represents the clauses in a form of a generic constraint, that later along with its associated fact can be presented to the modeller.
- The framework takes the whole class of the problem into consideration rather than one instance. This is achieved by generating patterns across instances and intersecting them. This helps us improve the model rather than one instance.
- The framework considers multiple search strategies to obtain clauses that may not be discovered by a certain search strategy. This improves the accuracy of the ranking technique proposed by Shishmarev et al..
- The framework automatically learns facts for each pattern. This combines the background information with the information present in the patterns, which may serve as a condition for the occurrence of the patterns and, together with the patterns, can be suggested to the modeller as a potential constraint to be added to the model.
- Our experiments show that the framework is practical and general for a variety of models, and it improved four MiniZinc models. Importantly, the framework improved a MiniZinc standard decomposition of a global constraint, which is commonly used in a variety of models.

Future work There are still several areas left to be explored. First, in our method for renaming the literals in the clauses, presented in Chapter 3, the reconstruction of path expressions is purely syntactic. If a path points to a span of text that is in a user-defined function or predicate, the text will reference the parameters of this function or predicate, rather than the top-level model variables which were passed as arguments. Secondly, the variable path approach does not always provide us information about the expression types that are derived from the text in a function/predicate. These two limitations could be mitigated by instrumenting the MiniZinc compiler to collect and output more information during compilation. Also, our method for generating patterns, presented in Chapter 4, can be further improved by incorporating anti-unification techniques, such as the one proposed by Kutsia et al. [38], to obtain more accurate patterns and detect some of the patterns that are currently missed. While this might be time consuming, we might be able to achieve a good trade-off by applying anti-unification only to the top clauses. Our method to infer facts from the patterns can also be improved. Our current implementation is not able to infer facts for some of the patterns, as demonstrated in Chapter 6.3. While this area is very challenging, one possible solution is to embed Inductive Logic Programming (ILP) techniques, such as the one proposed by Lallouet et al. [39], into our framework to potentially learn more facts.

References

[1] Gurobi optimizer reference manual. `http://www.gurobi.com/documentation/`, 2013. accessed: 1-03-2019.

[2] The MiniZinc challenge. `https://www.minizinc.org/challenge.html`, February 2014. accessed: 28-05-2019.

[3] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. A modular equational generalization algorithm. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 24–39. Springer, 2008.

[4] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. Order-sorted generalization. *Electronic Notes in Theoretical Computer Science*, 246:27–38, 2009.

[5] Robin Arcangioli, Christian Bessiere, and Nadjib Lazaar. Multiple constraint aquisition. In *IJCAI: International Joint Conference on Artificial Intelligence*, pages 698–704, 2016.

[6] Eva Armengol and Enric Plaza. Bottom-up induction of feature terms. *Machine Learning*, 41(3):259–294, 2000.

[7] Franz Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In *International Conference on Rewriting Techniques and Applications*, pages 86–97. Springer, 1991.

[8] Nicolas Beldiceanu and Helmut Simonis. A constraint seeker: Finding and ranking global constraints from examples. In *International Conference on Principles and Practice of Constraint Programming*, pages 12–26. Springer, 2011.

[9] Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, pages 141–157. Springer, 2012.

[10] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 123–137. Springer, 2004.

[11] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[12] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, pages 23–34, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[13] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *European Conference on Machine Learning*, pages 23–34. Springer, 2005.

[14] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In *IJCAI*, volume 7, pages 50–55, 2007.

[15] Christian Bessiere, Remi Coletta, and Thierry Petit. Learning implied global constraints. In *IJCAI*, pages 44–49, 2007.

[16] Christian Bessiere and Frédéric Koriche. Non learnability of constraint networks with membership queries. Technical report, Technical report, Coconut, Montpellier, France, 2012.

[17] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

[18] Jochen Burghardt. E-generalization using grammars. *Artificial intelligence*, 165(1):1–35, 2005.

[19] John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In *ECAI*, volume 141, pages 73–77, 2006.

[20] BMW Cheng, Jimmy Ho-Man Lee, and JCK Wu. Speeding up constraint propagation by redundant modeling. In *International Conference on Principles and Practice of Constraint Programming*, pages 91–103. Springer, 1996.

[21] Geoffrey G. Chu. *Improving combinatorial optimization.* PhD thesis, The University of Melbourne, 2011.

[22] Simon Colton. The HR program for theorem generation. In Andrei Voronkov, editor, *Automated Deduction—CADE-18*, pages 285–289, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[23] Simon Colton and Ian Miguel. Constraint generation via automated theory formation. In *Principles and Practice of Constraint Programming—CP 2001*, pages 575–579. Springer, 2001.

[24] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[25] Luc De Raedt. *Logical and relational learning.* Springer Science & Business Media, 2008.

[26] Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In *IJCAI*, volume 10581063, page 1994, 1993.

[27] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[28] Arthur L. Delcher and Simon Kasif. Efficient parallel term matching and anti-unification. *Journal of Automated Reasoning*, 9(3):391–406, 1992.

[29] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[30] Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In Ian P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.

[31] Eugene C. Freuder and Richard J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *International Conference on Principles and Practice of Constraint Programming*, pages 192–204. Springer, 1998.

[32] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *IJCAI*, volume 7, pages 80–87, 2007.

[33] Alan M. Frisch, Ian Miguel, and Toby Walsh. Cgrass: A system for transforming constraint satisfaction problems. In *International ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, pages 15–30. Springer, 2002.

[34] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In *Proceedings of the 14th European conference on artificial intelligence*, pages 599–603. IOS press, 2000.

[35] David Gerault, Marine Minier, and Christine Solnon. Using constraint programming to solve a cryptanalytic problem. In *IJCAI 2017-International Joint Conference on Artificial Intelligence-Sister Conference Best Paper Track*, page 5, 2017.

[36] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[37] Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In *Inductive logic programming.* Citeseer, 1992.

[38] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *Journal of Automated Reasoning*, 52(2):155–190, 2014.

[39] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 45–52. IEEE, 2010.

[40] Kevin Leo. *Making the Most of Structure in Constraint Models.* PhD thesis, Monash University, 2018.

[41] Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Globalizing constraint models. In *International Conference on Principles and Practice of Constraint Programming*, pages 432–447. Springer, 2013.

[42] Kevin Leo and Guido Tack. Multi-pass high-level presolving. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 346–352. AAAI Press, 2015.

[43] Kevin Leo and Guido Tack. Debugging unsatisfiable constraint models. In Domenico Salvagnin and Michele Lombardi, editors, *CPAIOR 2017*, volume 10335 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2017.

[44] Kim Marriott, Peter J. Stuckey, Leslie De Koninck, and Horst Samulowitz. A minizinc tutorial, 2014.

[45] William McCune. Otter 2.0. In *International Conference on Automated Deduction*, pages 663–664. Springer, 1990.

[46] Christopher Mears, Maria Garcia de la Banda, Mark Wallace, and Bart Demoen. A method for detecting symmetries in constraint models and its generalisation. *Constraints*, 20(2):235–273, 2015.

[47] Tom M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[48] Tom M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[49] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[50] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

[51] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Machine Learning Proceedings 1988*, pages 339–352. Elsevier, 1988.

[52] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[53] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

[54] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = Lazy Clause Generation. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.

[55] Barry O'Sullivan. Automated modelling and solving in constraint programming. In *AAAI*, pages 1493–1497, 2010.

[56] Gordon D. Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.

[57] Gordon D. Plotkin. A further note on inductive generalization. *Machine intelligence*, 6(101-124):248, 1971.

[58] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco documentation. *TASC, INRIA Rennes, LINA CNRS UMR*, 6241, 2014.

[59] Jean-François Puget. Automatic detection of variable and value symmetries. In *International Conference on Principles and Practice of Constraint Programming*, pages 475–489. Springer, 2005.

[60] J. Ross Quinlan and R. Mike Cameron-Jones. Foil: A midterm report. In *European conference on machine learning*, pages 1–20. Springer, 1993.

[61] Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571. Springer, 2004.

[62] J. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970.

[63] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[64] Claude Sammut and Ranan B. Banerji. Learning concepts by asking questions. *Machine learning: An artificial intelligence approach*, 2:167–192, 1986.

[65] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www. gecode. org*, pages 11–13, 2006.

[66] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, (2015), 2010.

[67] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with Gecode, 2016.

[68] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Learning from learning solvers. In Michael Rueher, editor, *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 455–472. Springer, 2016.

[69] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. *Learning from Learning Solvers*, pages 455–472. Springer International Publishing, Cham, 2016.

[70] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Visual search tree profiling. *Constraints*, 21(1):77–94, Jan 2016.

[71] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243. Springer, 2009.

[72] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.

[73] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014.

[74] Lap Poon Rupert Tang. *Integrating top-down and bottom-up approaches in inductive logic programming: applications in natural language processing and relational data mining.* PhD thesis, 2003.

[75] Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 373–388. Springer, 2018.

[76] Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and Jean-Charles Régin. Constraint programming in opl. In *PPDP*, volume 1702, pages 98–116. Springer, 1999.

[77] Hantao Zhang and Mark E. Stickely. An efficient algorithm for unit propagation. *Proc. of AI-MATH*, 96, 1996.

[78] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.