FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Graphical Environment for Creating Constraint Programming Models

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Martin Blöschl, B.Sc.
Matrikelnummer 1225362

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Nysret Musliu

Wien, 23. November 2017

Martin Blöschl   Nysret Musliu

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Graphical Environment for Creating Constraint Programming Models

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Martin Blöschl, B.Sc.
Registration Number 1225362

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

Vienna, 23rd November, 2017

Martin Blöschl Nysret Musliu

Erklärung zur Verfassung der Arbeit

Martin Blöschl, B.Sc.
Waldheimstraße 45 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. November 2017

Martin Blöschl

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisor, Priv.-Doz. Dr. Nysret Musliu. His remarks were very helpful and the meetings always provided new insights into the subject.

I would also like to thank my family and friends for supporting me during the time it took to write this thesis. In particular, I want to thank my girlfriend Lisa, who encouraged me during the most stressful times.

Kurzfassung

Viele wichtige Probleme aus der Praxis, etwa das „rotating workforce scheduling problem“ oder das „traveling salesperson problem“ können mit Constraint Programming modelliert werden. Gute Lösungen für diese Probleme können nicht nur große Kostenersparnisse für Firmen ermöglichen, sondern auch die Mitarbeiterzufriedenheit erhöhen.

Aus diesem Grund sind in den vergangenen Jahren viele (textbasierte) Constraint Programming Sprachen entstanden. Mit Hilfe dieser Sprachen kann man komplexe Probleme präzise formulieren. Dazu ist es aber nötig, die Syntax der jeweiligen Sprache zu beherrschen.

Für viele andere Programmierparadigmen, etwa für objektorientierte Programmierung, wurden graphische Sprachen entwickelt. Für Constraint Programming ist dies unseres Wissens nicht der Fall. Eine graphische Umgebung für Constraint Programming würde es erlauben, wichtige Probleme der Praxis zu modellieren, ohne die Syntax einer textbasierten Sprache zu kennen. Sie könnte auch hilfreich für erfahrene Entwickler sein, wenn sie schnelle und intuitive Modellierung ermöglicht.

In dieser Arbeit stellen wir eine neue und intuitive graphische Umgebung zur Erstellung von Constraint Programming Modellen vor. Wir zeigen einen neuen gitterbasierten Ansatz, mit dem es möglich ist, Variablen und Constraints zu erstellen und im zweidimensionalen Raum anzuordnen. Wir zeigen auch Möglichkeiten, häufig vorkommende Subprobleme wie Routen auf eine einfache grapische Art zu modellieren. Wir werden auch demonstrieren, wie solche graphische Modelle in die bekannte MiniZinc Sprache umgewandelt werden können, damit sie mit bestehenden Solvern gelöst werden können.

In der Evaluierung haben wir festgestellt, dass es mit der vorgestellten Umgebung möglich ist, viele bekannte Probleme wie das „rotating workforce scheduling problem“, das „social golfer problem“ und das „traveling salesperson problem with time windows“ zu modellieren. In manchen Instanzen haben wir beim Lösen Laufzeiten erreicht, die vergleichbar zu bestehenden Ansätzen sind.

Wir haben weiters herausgefunden, dass das Framework am besten für kleinere Instanzen mit niedriger Komplexität geeignet ist, da das Modell dann übersichtlich auf einem Bildschirm dargestellt werden kann. Wir können mögliche Anwendungen im Bereich der Bildung oder für den schnellen Entwurf von Modellen erkennen.

Abstract

Many practical problems of high importance such as the rotating workforce scheduling problem or the traveling salesperson problem can be modeled with constraint programming. Finding good solutions for those problems can enable great cost reduction for companies and even improve employee satisfaction.

Therefore, many powerful text-based constraint programming languages and corresponding solvers have emerged in the previous years. They allow concise formulation of problems but require that the user learns the specific syntax of the language.

For other programing paradigms such as object orientated programming, graphical languages have been proposed. To our knowledge, no general purpose graphical constraint programming language exists, that allows modeling without written formulas or constraints. Such a graphical constraint programming environment would not only allow important practical problems to be modeled without having to know the syntax of a text-based language, but could also be helpful to experienced developers if it allows fast and intuitive modeling.

In this thesis we propose a new graphical environment for creating constraint programming models. We present a grid-like approach to create variables and constraints and arrange them in the 2D space. We present various possibilities to make constraint programming in this framework as simple and intuitive as possible. We further present ways to model common subproblems in constraint programming, such as routes, in a graphical way. We also demonstrate how graphical models can be translated to the common MiniZinc language, so that they can be solved with existing constraint programming solvers.

In the evaluation we have found out that the proposed environment can be successfully used to model a variety of common constraint programming problems such as the rotating workforce scheduling problem, the social golfer problem and the traveling salesperson problem with time windows. In some problem instances, we achieved runtimes that were comparable to existing methods.

We have further found out that the framework is best suited for small instances with low complexity, as then the complete model can be displayed on an average computer screen without scrolling. We can see applications for the framework in the field of education, where it can help students learn the concepts of constraint programming.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Aims of this Thesis 2
1.3 Main Results 2
1.4 Structure of this Thesis 3

2 Theoretical Background and existing Approaches **5**
2.1 Constraint Satisfaction Problems 5
2.2 Constraint Optimization Problem 6
2.3 Solving Constraint Satisfaction Problems 7
2.4 MiniZinc Modeling Language 8
2.5 Existing Approaches of Graphical Programming Frameworks 9

3 A Graphical Environment **11**
3.1 Grid Representation of Problems 11
3.2 Developing a Framework 11
3.3 Model Creation and Solving 14
3.4 Functionality and Limits 14

4 Evaluation **35**
4.1 Sudoku 36
4.2 8 Queens 38
4.3 Social Golfer Problem 40
4.4 Rotating Workforce Scheduling Problem 42
4.5 Magic Hexagon 46
4.6 TSPTW 47
4.7 Traveling Tournament Problem 49
4.8 Simple Teacher Scheduling 52

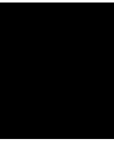
4.9 3-SAT 55
4.10 Creating Models for Variable Input 56
4.11 Finding the Right Solver 58
4.12 (Un-)Suitable Classes of Problems 60

5 Conclusion **63**

List of Figures **65**

Bibliography **67**

CHAPTER 1



Introduction

1.1 Motivation and Problem Statement

Constraint Programming is a declarative programming paradigm where variables and constraints are used to describe requirements that solutions have to fulfill. Constraint programming does not only have a strong theoretical foundation, but is widely used in practice, as it can be used for solving various important scheduling and planning problems [Bar99]. Those real-world problems include transit bus crew scheduling, aircraft scheduling, vehicle routing and physician scheduling [RVBW06].

Because of the high expressive power and practical relevance, many (text-based) constraint programming languages and solvers have emerged in the previous years. With those languages, it is possible to succinctly formulate even complex problems. It is however necessary to learn the syntax of the respective constraint programming language to be able to model problems.

For other programming paradigms, such as object-orientated programming, graphical languages and frameworks have been proposed [AB93] that allow the formulation of programs without having to know the precise syntax of a text-based language. Graphical languages are appealing because they provide visual feedback and allow expression without having to translate commands into abstract symbols [BA94] . There has been some progress made in the field of conceptual design [HOO03], where a constraint aided environment has been developed to help product designers. There are also approaches to assist in the creation of continuous domain constraint programming models [Vie15]. But, to our knowledge, no general purpose graphical framework for constraint programming exists, that allows the formulation of models without explicit text-based formulation of formulas or constraints.

Because of the expressive power and practical relevance of constraint programming, a graphical framework that can be used for the creation of models for common problems

without having to know the syntax of a language could be of great benefit. It could not only be helpful for beginners but if it allows fast and intuitive modeling, it could also be useful for those already experienced in constraint programming. For many problems there exist different possible formulations as constraint program. Therefore an environment that allows rapid creation of models could be very interesting for many practical applications, because it could make it easier to test different formulations.

Therefore, we want to investigate in this thesis if a graphical environment for constraint programming can be proposed that is general enough to model a variety of real-world problems and is nonetheless easy and intuitive to use. Since it is usually desired to solve the created constraint programming models, it is also necessary to examine how this can be done.

1.2 Aims of this Thesis

The aims of this thesis are:

- We want to examine if it is possible to create a graphical environment for constraint programming, since there exist many graphical languages for other computing paradigms such as object orientated programming.
- We will analyze if using such graphical framework does not only allow modeling of practical problems of high importance but can also be intuitive and simple to use.
- We will investigate how to translate the graphical models to a common constraint programming language, so that they can be solved with existing state of the art solvers.
- By implementing such a framework, we want to find out advantages and disadvantages of our approach and if the created models for different real-world problems can be solved in a reasonable time that is comparable to other existing approaches.

1.3 Main Results

The main results of this thesis are:

- We present a new graphical environment for creating constraint programming models. It allows users to create models of some problems without having to know a written constraint programming language. For this, we present a new grid-like approach to arrange variables and constraints in the 2D space.
- We have found a way to model common subproblems such as routes in a simple graphical way. This allows problems like the traveling salesperson problem to be modeled in a very intuitive way and without having to know a text-based constraint programming language.

- The proposed framework turned out to be capable enough to model a variety of common problems of high practical importance. We have found out that this environment can be successfully used to model instances of Sudoku, n-Queens and more complicated real-world problems such as the rotating workforce scheduling problem, the social golfer problem and the traveling salesperson problem with time windows. In some problem instances of the rotating workforce scheduling problem, we achieved runtimes that are comparable to existing methods.
- In empirical analysis, we discovered that the proposed grid based framework is best suited for modeling single instances of problems. It was however also possible to create models that can be configured to solve multiple different instances of some problem without having to recreate the complete model.
- As with other graphical programming environments, we have discovered that with increasing complexity of the problem, screen space becomes an issue. On a average sized (24 inch) computer monitor, approximately 375 variables can be displayed at once. If more variables are used, parts of the model can be accessed by scrolling.

1.4 Structure of this Thesis

In the second chapter, an introduction to the theoretical foundation of constraint programming will be given. We will also discuss existing approaches to create graphical programming environments.

In the third chapter, we propose a new framework. We will show the capabilities and limitations and how the framework is used.

In the last chapter, we will evaluate the framework by modeling common practical problems of high importance. We will provide runtimes for the solving process of some those models. We then compare the performance of solving to existing approaches.

Then, we will summarize the results, provide a conclusion and state yet open problems, that would be interesting to examine in future research.

CHAPTER 2

Theoretical Background and existing Approaches

In the following sections, an overview of the theoretical background of constraint programming will be presented. In particular, the constraint satisfaction problem and the constraint optimization problem will be specified. Then there will be a short explanation about how to solve a model and there will be an introduction to the MiniZinc Modeling Language. Finally, we will review existing literature.

2.1 Constraint Satisfaction Problems

The goal of a constraint satisfaction problem is to find an assignment of variables that satisfy a set of constraints. More formally [RN10], an instance of the constraint satisfaction problem is a triple $\langle X, D, C \rangle$, where $X = \{X_1, ..., X_n\}$ is a set of variables, $D = \{D_1, ..., D_n\}$ a set of nonempty domains and $C = \{C_1, ..., C_m\}$ a set of constraints. The domain of each variable X_i is D_i. A constraint C_i is a tuple $\langle t_i, R_i \rangle$ where t_i is a tuple with pairwise different elements of X of size k and R_i is a k-ary relation. An assignment is a mapping from the set of variables to their respective domains. A constraint C_i is satisfied with respect to an assignment, if the k-tuple of values assigned to variables t_i is in the relation R_i. An assignment is valid, if all constraints are satisfied.

Informally, a constraint problem can be seen in the following way: The variables can assume values defined in their respective domain. The constraints exist to exclude some solutions that have undesired properties. For example, one could specify a set of variables X of size 4. The domain of all variables in X could be $\{1, .., 4\}$ and we have a single constraint C_1. Suppose we want only those assignments to be valid, where all variables are mapped to a different element. C_1 would then consist of $t_1 = (x_1, ..., x_4)$ and $R_i = \{(1, 2, 3, 4), (2, 1, 3, 4), (3, 2, 1, 4), ..., (4, 3, 2, 1)\}$. It is also possible to use different

constraint(s) and still achieve the same set of valid assignments: It would for example also be possible to create a constraint for each pair of variables as t_i and set the corresponding $R_i = \{(1,2),(2,1),(1,3),...,(4,3),(3,4)\}$.

In the first example we had only a single constraint - the relation however was exponential in the amount of variables and the size of the domains. In the second example we had a quadratic number of constraints. The relations although were only quadratic in the size of the domains. In practice, one of the two formulations could be faster to solve.

Summarizing, there are different ways to specify (=to model) a constraint satisfaction problem instance. The performance difference of models in practice is reviewed in the section 'Evaluation'.

2.2 Constraint Optimization Problem

In the previous section, we discussed the constraint satisfaction problem. The question was if there is an assignment such that all constraints are satisfied. In practice, it is often the case that not all solutions are equally good.

For example, one could use constraint programming to determine if an assignment of workers is possible such that all break constraints of the workers are respected and all machines are manned. If this is the case, there even might be different schedules that satisfy all constraints. It could then be of interest to maximize the length of consecutive 'off'-blocks of the workers to improve worker satisfaction.

In principle, it would be possible to use the constraint satisfaction problem to find the optimal assignment: by repeated solving and slight adaptation of the model in each run. The first model would be the plain satisfaction problem asking if there is any solution for the model. The next model would then be altered to ask if there is a solution and the optimization function is as big or bigger than some fixed integer. With binary search, the computational overhead is within logarithmic bounds of the domain of the variable or optimization function to be optimized.

It is however more convenient to be able to directly specify a variable or optimization function that is supposed to be minimal or maximal instead of guiding the search process manually with repeated runs of the constraint satisfaction problem. In many modern constraint programming languages and solvers it is possible to directly specify optimization goals.

Since both the constraint satisfaction problem and the constraint optimization problem have strong practical relevance, we will deal with both problems in this thesis.

There exist multiple formal specifications of the constraint optimization problem [RVBW06]. One formulation [Dec03] is the following: In addition to the constraints and variables we define a global cost function F. Additionally, we define functions $F_1, ..., F_i$. Each function F_j is a function over some subset D_j of the set of variables X. An assignment of the variables is denoted with $\bar{a}$.

The global cost function with respect to an assignment $\bar{a}$ is then defined as:

$$F(\bar{a}) = \sum_{j=1}^{i} F_j(\bar{a}) \tag{2.1}$$

In above formula, $F_j(\bar{a})$ denotes that the function F_j is evaluated with respect to the scope (subset of variables) that the function was defined over. The functions F_j can be used to penalize undesired assignments of the variables.

2.3 Solving Constraint Satisfaction Problems

One important aspect of the field of constraint satisfaction is solving models. For fixed domain problems, the problems we will be dealing with in this thesis, solving is in theory simple [Bar99]. With an enumeration algorithm, all possible variable assignments could be enumerated and checked if the assignment satisfies all constraints.

This approach is however unsuitable for many problems in practice, since a model with only 50 variables with each a domain of size 10 will result in 10^{50} possibilities, much more than any current computing machine can enumerate in reasonable time.

One approach that avoids enumerating the complete search space is 'Backtracking'. The main idea is that instead of the simple enumeration method, partial assignments of some variables are fixed incrementally. After one or more variables are fixed it is tested if all constraints are satisfied (so far). If this is not the case, 'Backtracking' occurs. This means, the most recent variables are 'unfixed' until all constraints are satisfied.

Modern solvers combine many solution methods to achieve reasonable solving times. Other important methods besides 'Backtracking' are consistency techniques[Bar99]. The main idea is the following: If there are unary constraints (constraints that fix the value for one variable), all other domain elements of that variable can be removed. This concept is called node consistency. The same can be done for n-ary constraints. If a domain element of a variable does not occur in the relation of the domain at all, it can be removed. This concept is called arc consistency.

Another important technique is 'Forward checking'. With this method, the consistency methods described in the previous paragraph are applied to not yet instantiated variables. Again, like in 'Backtracking', the variables are assigned incrementally. For the domain of each unfixed variable, those domain elements are removed, that are inconsistent with the constraints and the partial assignment of the variables. If an unfixed variable has an empty domain, this means that there would be no way to complete the solution without unsatisfied constraints. 'Forward checking' can avoid that the solution space for some partial assignment is explored even though there is no possibility to complete the partial solution without conflicts.

Current state of the art solvers use those and many other advanced methods to accelerate the solving process. Since creating a good solver is a quite complicated task, we use existing solvers to find solutions to the models created by our framework.

2.4 MiniZinc Modeling Language

The MiniZinc Modeling Language [NSB+07] was developed to provide a uniform interface for constraint programming solvers. Previously, many solvers relied on an own modeling language that the user had to learn for that specific solver. The MiniZinc Modeling Language was created to overcome these limitations and provide a universal format for constraint programming models. MiniZinc can therefore also make it easier to compare different solvers with respect to a single model, as no adaptation of the model to the solver specific language is necessary. It is indeed the case that since the specification of the language 10 years ago, there exist many solvers that support the language. This is reason enough for us to use the language for our framework.

The MiniZinc package consists of two languages: First, there is the high level MiniZinc language that is meant to be used by the programmers/model creators. The MiniZinc language supports many high level constraints. There is for example a constraint that models the bin-packing problem.

Then, there is the FlatZinc language that is meant to be implemented by the solver. The FlatZinc language supports mostly low level arithmetic and logical constraints and is less convenient to write. The authors of the MiniZinc/FlatZinc languages provide tools that allow conversion of the MiniZinc to the FlatZinc language. Therefore, it is possible to write high level constraints in MiniZinc. Then, the completed model is translated to FlatZinc and subsequently solved by any solver that supports the FlatZinc language.

We will now describe the MiniZinc language in more detail, as this language will be used as the output language of our framework. A thorough description can be found in the paper where the language was first described [NSB+07]. The most important parts of the description for this thesis are:

In detail, a MiniZinc instance consists of a model and data, that can but don't necessarily have to be seperate files. The data file contains static assignments that determine the configuration of a specific instance. The model is the description of the problem containing the constraints and variables. In our framework, the output will be a single file containing both the data and the model.

In our thesis, we will only use variables of type integer. Those can be created by specifying a domain or without domain, where they can assume any value between -2147483646 and 2147483646.

We will also use predicates, which are similar to predicates in first order logic. A predicate over some variables can evaluate to true or false given an interpretation. A constraint can consist of a predicate or a connection of predicates (also similar to first order logic) that has to evaluate to true in any solution of a model.

Additionally, we will use arrays, which are fixed in size and contain variables. Some predicates can be defined over arrays for concise formulations. Sets of variable size containing variables will also be used in our framework.

An example for the MiniZinc syntax is a file containing the following lines:

```
var 1..4:  a;
var 1..4:  b;
var set of int:  set_01 = {a,b};
array[1..2] of var int:  array_01 = [a,b];
constraint array_01[1] > array_01[2];
constraint 4 in set_01;

solve satisfy;
```

In the first two lines, two variables named a and b are defined, both with domain $\{1, 2, 3, 4\}$. In the third line, a set containing the values of the previous two variables is defined. In the fourth line, an array of size 2 is defined where the first element is the variable a and the second element is the variable b. In the fifth line, a constraint is defined that contains the predicate $array_01[1] > array_01[2]$. This implicitly defined predicate returns true, if and only if the value of the first element (in our case a) is greater than the value of the second element (in our case b) of the array. The sixth line contains a constraint that is defined over a predicate that returns true if and only if the value 4 is contained in the set. The last line specifies that we want to find a solution to this instance, but do not want to define an optimization function.

When we solve this problem using existing MiniZinc solvers like Gecode, we could get the following instantiation as a result: a assumes 4 and b assumes 1. This instantiation satisfies all constraints and respects the domains of the variables: The value of the first element of the array, namely the variable a is 4 and thus bigger than the value of the second element in the array (the value of b is 1). The second constraint is also satisfied since the value 4 is contained in the sets of values that the variables a and b assume in the interpretation: $4 \in \{4, 1\}$. As expected, the interpretation we got from the solver is a solution for our model.

2.5 Existing Approaches of Graphical Programming Frameworks

In 2015, Nelson Viera [Vie15] has created a graphical user interface for continous constraint satisfaction problems. He described a way to represent the solution space and provided a way to design and visualize the constraints and variables. The constraints can be specified using formulas. For example, one constraint could look like this: $8x^4 - 12y < 6z$. His user interface displays the relations of variables and constraints very well. Viera's approach focuses on the continuous domain.

There have been efforts to guide the search space traversal of constraint programs in a graphical way [FSC04]. Those kinds of tools are very useful to improve solving runtime.

There exist a number of general purpose graphical programming languages [Hil92]. Some of those languages share common advantages and disadvantages. One of the advantages of a graphical programming language is for example the fact that there is no syntax that has to be learned to be able to write programs. Disadvantages include the fact that many graphical programming languages use a lot of screen space [Hil92].

Some progress in the field of graphical constraint programming with strong focus on a specific practical application has been made: Alan Holland et. al. [HOO03] have developed a constraint aided environment for conceptual design. In product development, it is often the case that precise specifications have to be respected. The environment can be used to model constraints, restrictions, dependencies of objects in a constraint based way. A prototype of the environment was developed to be compatible to Autodesk Inventor, a graphical 3D-CAD software. The environment also aims to be similar to use as existing modeling tools and uses graphical elements as user interaction interface. The specification of the constraints is done with help of graphical elements.

In the paper 'Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm' [BAWD+01], the authors Burnett et. al present a spreadsheet based language. A spreadsheet language is defined to be a language that is based on a spreadsheet with formulas within the cells. The language Forms/3 is designed to overcome limitations of previous spreadsheet languages (like lack of data abstraction features) and still aims to be usable for users that are not formally trained computer programmers. Burnett and Ambert [BA94] state that visual programming is appealing because it allows the expression by sketching, pointing or demonstrating and does not require translation into abstract symbols.

A recent case study [SLRGVC16] evaluated the integration of a visual programming language in the curriculum of five different elementary schools in Spain. The language that was worked with was 'Scratch'. It allows the creation of 'interactive stories, games and animations' [SLRGVC16]. The researchers recommended the implementation of such a course, as they have determined positive results after evaluation.

CHAPTER 3

A Graphical Environment

In this chapter, a graphical environment for creating constraint programming models is proposed. We present the theoretical foundation of such a framework and examine the advantages and disadvantages of our approach. We will then give a detailed evaluation and provide possible applications in the next chapter.

3.1 Grid Representation of Problems

One concrete example for a problem that can be represented as a grid is the machine/worker assignment problem. Let us say there are a number of machines and an equal number of workers. Each worker has to be assigned to a machine based on the capabilities of the worker. One natural representation for such problem is a simple table or grid where the first column is a numbered list of machines and the second column contains the assigned worker. Each row represents the assignment of one worker to one machine.

Scheduling, pickup and delivery and timetabling are some very common problems and have a natural representation as table or grid [FFH+01]. Even papers as early as 1969 [PWW69] use a grid-like approach for representation (see Figure 3.1).

Since many problems have a natural representation as a grid, we will use a grid as the base of our graphical framework. We aim to conceptualize an intuitive and easy to use environment that is able to model a number of important real-world problems.

3.2 Developing a Framework

In this thesis, a framework[1] will be developed that allows creation of constraint programming models in a grid based way. To be more precise, the idea is to let the user specify

[1] The reference implementation can be found here: https://github.com/MartinBlo/graphicalCSP

Figure 3.1: The solution representation of a scheduling problem. Recreated from the paper: [PWW69]

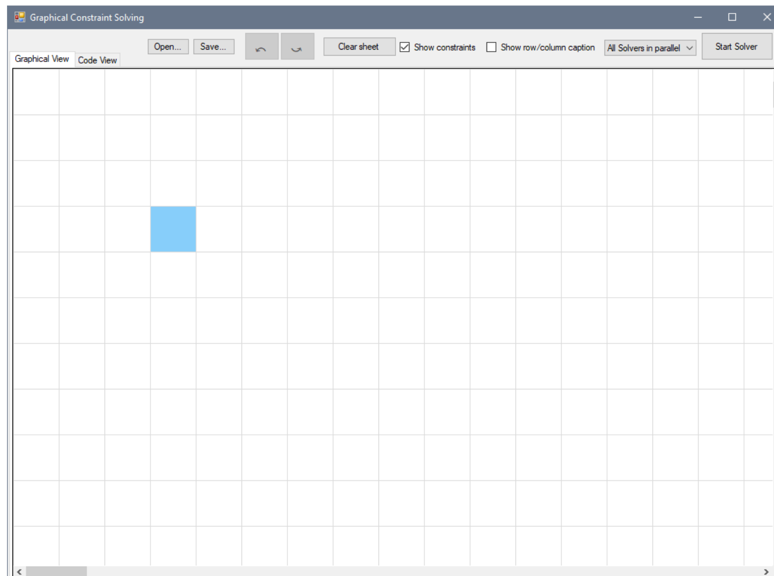
variable	1	2	3	4	5	6	7	8	9
x11t	-	-	-	1	0	-	-	-	
x12t	-	-	-	-	-	-	1	0	
x13t	-	-	-	0	0	0	1	0	
x1t	-	-	-	-	-	-	-	1	
x21t	-	-	-	1	0	0	0	-	-
x22t	-	-	-	0	0	1	0	0	0
x23t	-	-	-	-	-	1	0	0	0
x2t	-	-	-	-	-	-	1	1	1
x31t	-	-	-	-	-	-	1	0	0
x32t	-	-	-	0	0	0	1	0	0
x3t	-	-	-	-	-	-	-	1	1

the variables and the spatial order of the variables first and then let the user subsequently add constraints until the created solution fulfills all properties that are necessary . Our framework will allow the user to specify both variables and constraints in a graphical manner. We will also incorporate common graphical elements such as drag and drop to make model creation as easy and intuitive as possible.

A grid of unlimited size will be the base of our framework. Every grid cell can but does not necessarily contain a variable. A grid cell can either be an empty cell or contain a fresh variable.

In a lot of problems, there are numerous variables representing a similar type of value, such as an assignment or a state. In some existing constraint programming frameworks and languages, creating multiple variables is done with loops or arrays. While this allows very concise formulation, it is necessary to remember the correct syntax [KCC+02]. Our

Figure 3.2: The main window of our implementation of the graphical constraint programming framework.



approach will be to let users create similar variables by selecting a subset of the grid and specifying the desired domain in a graphical way. This prevents syntax errors and is also very fast and intuitive for the user to do.

The constraints will be incorporated as connections between variables. As described in the chapter 'Theoretical Background and existing Approaches', each constraint consists of variables and a relation that describes allowed assignments of those variables. As a relation over n elements can consist of an exponential number of elements that are in relation, an implicit specification of constraints is often used in practice.

An example: A common constraint would be to specify that certain variables are 'all different', meaning that they all have a pairwise different value. This specification is more concise than listing all assignments for the variables such that they are 'all different'.

Instead of writing the constraint in code, with our framework one will be able to select the involved variables of a constraint with common graphical user interface methods such as dragging with a mouse. Then it will be possible to create a constraint by selecting involved variables and the type and to configure the constraint.

One benefit of such approach is that for basic constraints that require no further configuration (such as 'all different', 'all equal', etc.) it is impossible to create models that are syntactically incorrect, as no code has to be written.

Summarizing, we have described a graphical framework whose aim it is to be not only intuitive and easy to learn but also to prevent syntax errors by relying on a graphical user interface instead of textual specification of the model. In figure 3.2 one can see the

main window of our implemented framework. We will go into more detail about the individual components in the following sections.

3.3 Model Creation and Solving

Once the graphical model is created, it is usually desired to find a solution. There exist a vast amount of constraint programming solvers [NSB+07] that are compatible with the MiniZinc format.

Therefore, we created a translator that converts each part of the graphical model to standardized MiniZinc code. The MiniZinc modeling language [NSB+07] was created to provide a constraint programming language that can be used to describe constraint programming problems and is supported by many solvers. Compatible solvers include Gecode, Google OR-Tools and Chuffed Solver.

One advantage of this approach over creating an own solver is that we can use very fast and highly optimized solvers and can fully concentrate on creating a graphical user interface for the model creation.

Integrated into the user interface of our framework is also a module for communicating with the solvers that oversees the solving and displays solutions (if found).

In the next section, the structure and functionality of the framework will be reviewed and each part will be described in more detail.

3.4 Functionality and Limits

The main functionality of the framework is to let the user create constraint programming models of problem instances. To achieve this, variables and constraints have to be created. In the subsections after the general structure, we will describe how this is achieved and which constraint can be created.

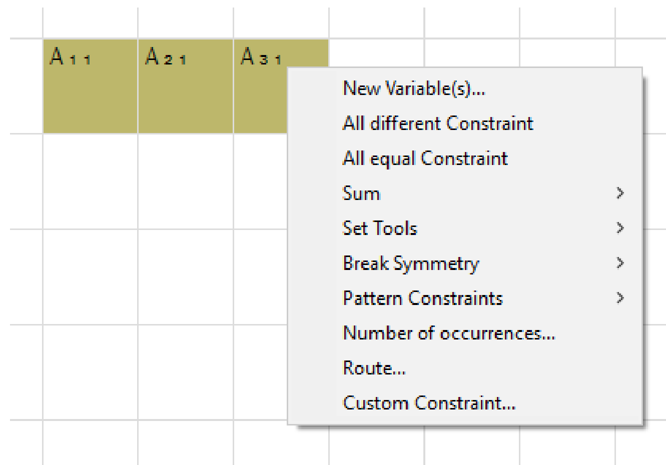
3.4.1 General Structure of the Framework

Our framework consists of the following parts:

- A graphical user interface for specifying variables and constraints
- A translator that translates the created variables and constraints to MiniZinc code
- A module for managing the solving process, reporting of solutions or failure and for the communication to the solvers

In the subsections below, the first part of the framework, namely the user interface, will be described thoroughly. It was one of the main challenges of this framework to develop

Figure 3.3: The context menu allows creation of variables and constraints.



a user interface that allows easy and intuitive creation of constraint programming models. First, we will describe how to create variables and constraints. Then, we will show how to specify optimization goals. Finally, other important graphical features will be described.

The rest of this section describes the translator module and the module for managing the solving process.

3.4.2 Variables

With our implementation it is possible to create variables of integer type and any fixed domain. We excluded continuous variables in our framework.

As described in the previous section, it is not necessary to create each variable individually but one can select a subset of the grid using a mouse and specify the desired domain for the variables to be created.

To be more specific, the user selects a particular set of cells and clicks the right button of the mouse to open the context menu. One can then use the function 'Create Variable' in the context menu. A separate window appears where the user is asked for specification of the domain and to provide a name for the variables to be created. In figure 3.3, one can see the context menu where the options to create the constraints are available.

The variable name is indexed using an x and y index. For example: if one were to create 21 variables as a 3 by 7 block and the name 'VAR', the individual variables would be named $VAR_{1,1}$ to $VAR_{3,7}$. In our framework, the names will be used in the non-formatted way, so for example $VAR_{1,1}$ corresponds to VAR_1_1.

The Minizinc code that is created for each variable is:

```
var DomainStart..DomainEnd:  VariableName;
```

Figure 3.4: Some variables that were created in the grid.

X_{11}	X_{21}	X_{31}	X_{41}	X_{51}	X_{61}	X_{71}
X_{12}	X_{22}	X_{32}	X_{42}	X_{52}	X_{62}	X_{72}
X_{13}	X_{23}	X_{33}	X_{43}	X_{53}	X_{63}	X_{73}
X_{14}	X_{24}	X_{34}	X_{44}	X_{54}	X_{64}	X_{74}
X_{15}	X_{25}	X_{35}	X_{45}	X_{55}	X_{65}	X_{75}

In above code, DomainStart is replaced with the lower bound of the domain and DomainEnd is replaced with the upper bound of the domain. VariableName is replaced with the user defined name of the variable plus indices. Domains with 'holes', for example {1,4,5} are not supported by our framework out-of-the-box but can be accomplished by manually excluding the undesired values using constraints.

In figure 3.4, one can see the grid containing 7*5 variables in a rectangular block. The indices and the name are displayed in the cell.

3.4.3 Implemented Constraints

One of the main parts of our graphical framework is easy creation and duplication of constraints. This is important for many problems in practice, as their models usually consist of a lot of constraints.

We will demonstrate on the following example, why duplicating constraints can be useful: One very commonly used example in demonstrating constraint programming is Sudoku. For those unfamiliar with this puzzle, Sudoku is a 9 by 9 grid where numbers from 1 to 9 must occur in each row, in each column and in all 9 adjacent 3 by 3 blocks. In Sudoku, there are 27 constraints, as there are 9 rows, 9 columns and 9 blocks. The constraints regarding the rows (and columns, blocks) are very similar except for the involved variables. It would be very tedious to create all constraints individually in our framework. This is why we present a drag and drop concept for constraint duplication.

In the enumeration below we will present the constraints we support in our reference implementation. For each constraint, we will also describe how and if it has to be

configured and how drag and drop duplication works.

- The 'All-Equal' constraint. This constraint forces all involved variables (see section 'Theoretical Background and existing Approaches') to assume equal value in all solutions of the constraint programming model.

  In our framework, creating such constraint is easy: the user selects the desired variables that he or she wants to have equal value and in the context menu (see: [SP98]) selects 'All-Equal'.

  Duplicating such constraint is done in the following way: Once the constraint is created, a label appears that the user can drag. Dragging the label will create a new constraint for each cell the label is moved to (Manhattan distance is used). For example: In a row, the first and the last variables are set equal. By dragging the label down, a constraint for each row below is created (in each row the first and last element are set equal now).

  The generated MiniZinc code of the constraint for three variables that are already defined looks like this:

  ```
  constraint all_equal([A_3_1,A_2_1,A_1_1]);
  ```

- The All-Different constraint. This is a very important and well studied constraint [Hoe01]. The intention is to force all involved variables to have a pairwise different value in a solution.

  Some paragraphs above we explained Sudoku, where the All-Different constraint can be used. Another practical example would be an assignment problems. There, it is often the case that one element (e.g machine) has to be assigned to some other element (e.g. to a worker). The list of assigned machines will be 'All-Different', since it is in our example not desired to assign a machine to two workers.

  The 'All-Different' constraint is created in a very similar way as above 'All-Equal' constraint, namely by selecting the desired variables and selecting the constraint in the context menu. Duplication also works with the label that was described in the previous constraint.

  The generated MiniZinc code for three variables looks like this:

  ```
  constraint all_different([A_3_1,A_2_1,A_1_1]);
  ```

  In figure 3.5, one can see a block of variables, where the first row of variables is involved in an 'All-Different' constraint. In our framework, each constraint is displayed with a rectangle containing all involved variables. The color is dependent on the type of constraint. In our implementation, the 'All-Different' constraint is displayed in green. On the right side of the constraint, the label is shown that can be used to duplicate the constraint to other rows.

- The 'Sum equal 1 constraint' forces the sum of all involved variables to be exactly 1. This constraint is useful when one wants to model a 1 to n assignment problem.

Figure 3.5: The All-Different constraint (first row, green).

For example, it might be necessary to model the location of an object. Then, it would be possible to create variables for each location with domain $\{0, 1\}$. The intended meaning is that a variable being one represents that the object has the respective location. Only one location at a time is possible for the object. One could create a 'Sum equal 1 constraint' for the variables that model the location of the object to achieve the desired effect of the object having precisely one location.

Creating such a constraint is done in the same way as described above, namely by selecting the variables and the desired constraint in the context menu. Duplication also works the same.

The generated MiniZinc code for three variables looks like this:

```
constraint (A_3_1+A_2_1+A_1_1) = 1;
```

- The 'Sum less or equal 1 constraint' is very similar to the 'Sum equal 1 constraint'. The only difference is that the sum can be 1 or less than one.

  This constraint is useful when one wants to model an event that can happen, but does not necessarily has to. Again, one can create a variable for each possible outcome with the domain $\{0, 1\}$. If one then creates a 'Sum less or equal 1 constraint' with the variables of all events, in every solution at most one event will have happened (at most one variable will have value 1).

  Creation and duplication of this constraint is done in the same way as the previous constraints.

  The generated MiniZinc code for three variables looks like this:

  ```
  constraint (A_3_1+A_2_1+A_1_1) < 2;
  ```

- The 'Custom sum constraint'. This constraint is a generalization of the previous two constraints. The sum of some variables can be equal, unequal, larger or smaller than some value or variable.

  This constraint is useful in a variety of applications. For example, one could easily model some products with variables of the domain $\{0, 1\}$. The intended meaning of the variables will be if the respective product will be purchased (1) or not purchased (0). One could then use a 'Custom sum constraint' to specify that the sum of purchased products should be less than a specific value or variable that might then depend on a budget.

  Creating this constraint is more difficult, as some parameters have to be specified. Similar to the previous constraints, the involved variables have to be selected and the constraint is created by selecting it in the context menu. Then a new window appears where some parameters are specified. First, the operator is defined as either '=', '!=', '<=', '>=' using a drop-down list. After that, the desired value is specified, for example to an integer like '5' or another variable or even a combination (e.g. variable minus 1). One fully specified constraint would then be for example: The sum of the selected variables is less than or equal to 6.

  Duplicating the 'Custom sum constraint' is done in the following way: Like in the previous constraints, dragging a label that appears after constraint creation will duplicate the constraints. Note that if the value that the sum should be equal (or less, etc.) is a variable, the framework will also change this variable accordingly in the duplication process.

  The generated MiniZinc code for three variables looks like this:

  ```
  constraint (A_3_1+A_2_1+A_1_1) op val;
  ```

  where op is the operator and val the value that the sum should be compared to.

- Create 'Set'. In the strict sense, this is not a constraint but more 'syntactic sugar' that is needed for other constraints. Creating a set of variables has the intended meaning of creating a set of the values of the variables in the solution. For example, if we specify two variables and select them to be in the set, the set contains precisely the value (if they assume the same value) or values (if they assume a different value) that they assume in a solution.

  Set operations can be used in a variety of contexts to model problems succinctly. One example, that will also be dealt with in more detail in the next chapter, is the 'Social Golfer Problem'. In this problem, the goal is to avoid that players play against each other more than once. In a mathematical sense, the groups can be seen as sets and we want the pairwise intersection of all sets to be limited to 1.

  Another example would be an assignment problem for groups, where the intersection of the set of group-leaders and each group be exactly one. Then, each group will contain exactly one group leader.

Creating a set is done exactly in the same way as creating a constraint, namely by selecting the desired variables and selecting 'Set' in the context menu. Duplicating the set also works as previously described using the drag label.

The generated MiniZinc code for a set of three variables looks like this:

```
var set of int:  set_35619075 = {A_3_1,A_2_1,A_1_1};
```

Note that in the MiniZinc modeling language it is necessary to name the set, this is done automatically with an unique identifier that is invisible to the user of our framework.

- The 'Set intersection at most 1' constraint. This constraint involves sets instead of variables like the previous constraints. It forces the pairwise intersection of all involved sets to have a cardinality of at most 1. If one wanted to formulate such constraint in the formal way described in the chapter 'Theoretical Background and existing Approaches', the following way is possible: The variables of the constraint are all individual variables that occur in the sets, the relation is simply the enumeration of all possible assignments such that the set intersection of all pairwise sets has cardinality of at most one.

  The constraint is created by selecting variables and then creating the constraint from the context menu. All sets that contain at least one of the selected variables will then have a pairwise intersection cardinality of at most 1.

  The generated MiniZinc code for two sets looks like this:

  ```
  constraint at_most1([set_52136226,set_35619075]);
  ```

  Other set constraints, like demanding the pairwise set intersection cardinality to be at least, at most or (un-)equal to some value or variable are in theory also possible to specify, albeit not part of our reference implementation.

- 'Break symmetry by ascending order'. This constraint is useful to create a model that can be solved faster. The constraint forces the values of the selected variables to be in ascending order. In some real world problems there exist multiple identical solutions that just have permuted order. For example, if we look at some set of a problem that we used as example in the previous two constraints, a set is not different if the order is reversed. Nonetheless, the search space is much larger if symmetric solutions are not eliminated.

  A concrete example: If there are four variables with the same domain and the sum of those variables $v_1, ..., v_4$ has to satisfy some property (e.g. being equal to an integer), there are a lot of solutions that are identical to each other with exception of the variable order. Any permutation of the assignments would be equally good as any other, as permutation does not change the sum of the variables. The 'Break symmetry by ascending order' constraint forces the assignment of the involved variables to be ascending, i.e the value of v_1 be lower or equal to the value of v_2, the value of v_2 be lower or equal to the value of v_3, etc.

A detailed evaluation of this constraint and when it is useful will be given in the chapter 'Evaluation'.

Creating and duplicating the 'Break symmetry by ascending order' is done in the same way as the 'All-equal constraint' and all other constraints without parameters.

The generated MiniZinc code for three variables looks like this:

```
constraint decreasing([A_3_1,A_2_1,A_1_1]);
```

Other symmetry breaking methods do exist in practice [GS00] but are not part of our our reference implementation. Such methods are very interesting as they can possibly reduce solving time drastically but are not part of our reference implementation of the framework.

- The 'Forbidden pattern' constraint. This constraint is often used in scheduling context. It excludes specific sequences of values to occur in a sequence of variables.

  For example, if we had a table for a shift assignment problem and each row represents the shifts and breaks of a single worker, we could use this constraint to specify illegal patterns. It might be desirable to disallow the worker to work an evening shift and on the next day the morning shift.

  To make it as easy as possible to create such a complex constraint, a simple language is used. First, the user has to select the desired variables. Then, it is possible to specify forbidden patterns one by one.

  The syntax for a single pattern is the following: $([operator]value\ blankspace)*$ which means that an arbitrary number of groups of operators, values and spaces can occur after each other. The intended meaning is that the line is parsed and any complete match is forbidden. In above specification, value can be a single integer. The idea is that integers are written after each other, separated with a blank space, to describe a concrete sequence of assignments that is undesired in a sequence of variables.

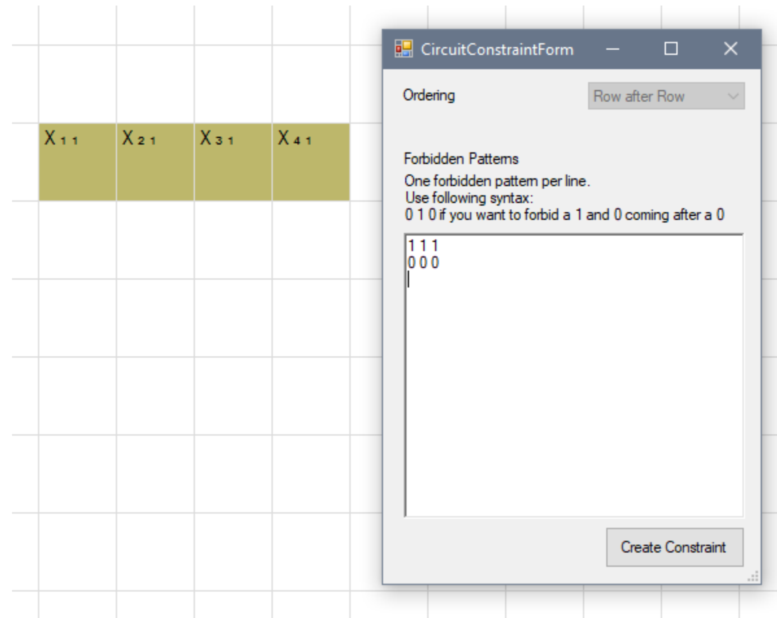
  An example: The constraint is configured with a single pattern that is '1 0 1 '. Then, three consecutive variables in the ordered list of variables of the constraint having precisely the values '1','0' and '1' are forbidden.

  Each forbidden pattern is separated with a new line. It remains to explain the meaning of [*operator*] in the syntax. The square brackets mean that the operator is optional. The operator can be one of the following strings: '~', '<', '<=', '>', '>=' which have the interpretation unequal, smaller, smaller or equal, larger, larger or equal.

  If one wanted to prohibit the sequence '3' followed by anything larger than '3' in a sequence of variables, the specification would be: '3 >3'. This succinct representation allows to express complex constraints in a straight forward way.

  Duplication of this constraint is done in the same way as for the previous constraint, by dragging the label that is displayed once the constraint is created.

Figure 3.6: The forbidden pattern constraint can be used for defining undesired patterns in sequences.



The generated code for 5 variables and the pattern '0 1' that is forbidden looks like this:

```
constraint A_1_1 != 0 \/ A_2_1 != 1;
constraint A_2_1 != 0 \/ A_3_1 != 1;
constraint A_3_1 != 0 \/ A_4_1 != 1;
constraint A_4_1 != 0 \/ A_5_1 != 1;
```

The code makes it unable for two consecutive variables to be 0 and 1 because of the clauses 'either one of the two consecutive variables must not be 0 or 1 respectively'.

In figure 3.6, one can see the interface that is used to specify the forbidden patterns.

- The 'Cyclic forbidden pattern' constraint. This constraint is very similar to the last constraint. The only difference is that the pattern must also not occur in the cyclic continuation of the list of the variables. In other words, the successor of the last variable is again the first variable. This leads to additional restriction of the search space and is necessary in some applications. In cyclic scheduling problems, forbidden shift sequences can be modeled with this constraint.

  Creation and duplication of this constraint is done in the same way as the previous constraint.

  The generated code for 5 variables and the pattern '0 1' that is forbidden looks like this:

```
constraint A_1_1 != 0 \/ A_2_1 != 1;
constraint A_2_1 != 0 \/ A_3_1 != 1;
constraint A_3_1 != 0 \/ A_4_1 != 1;
constraint A_4_1 != 0 \/ A_5_1 != 1;
constraint A_5_1 != 0 \/ A_1_1 != 1;
```

The only difference to the previous constraint is an additional line (last line) that continues the constraint from the last element to the first.

- The 'Forward' constraint. This constraint can be used to specify desired sequences in an ordered list of variables. The difference to the previous two constraints is that the formulation is done in a positive way. In the previous two constraints, the user had to specify patterns that were unwanted, e.g. a 0 following a 1. In this constraint, the user can formulate implications what values the variables have to assume if some conditions are given. In some cases, both the positive and the negative formulation method allow succinct formulation:

  If we wanted to specify that after each 3, a 4 should occur in a sequence of variables, we could use the negative way of formulating this constraint: '3 ~4' is undesired, or the positive way (for now informally): if variable x has value 3, then the next variable x+1 should have value 4.

  In some cases, the positive formulation is shorter:

  If we want to specify an implication where the right side consists of more than one element, e.g. $a \rightarrow b \wedge c \wedge, ..., \wedge z$, then a negative formulation would need an exponential overhead, as each wrong assignment of the right side would have to be excluded individually.

  This advantage makes it worth including the positive formulation of patterns. The syntax for the positive formulation is the following:

  Each pattern is written in a new line. The patten syntax is:

  $L_1\ operator_1\ a_1,\ ...\ ,\ L_n\ operator_n a_n\ \rightarrow\ R_1\ operator_{r1}\ b_1\ ,\ ...\ ,\ R_m\ operator_{rm}\ b_m$

  In above formula, L_i and R_i stand for variables on the left and right side of the implication. Note that we do not use the variable name but just the character 'C' plus an arbitrary index that is relative to the other indices to create an order. This notation is used to be able to specify the 'next' and 'previous' variable in the sequence of variables. $operator_i$ and $operator_{r1}$ are operators and can assume one of the following values: '=', '~', '<', '<=', '>', '>='. a_i and b_i stand for values the variables will be compared to.

  A practical example that uses above syntax: 'C0 = 5, C2 > 5 -> C4 = 4, C5 = 8' which has the following intended meaning: An ordered list of variables is specified. In this list, if some variable (C0) assumes 5 and the successor of the successor of that variable (C2) is bigger than 5, then (→) the successor of the successor of C2, namely C4 has to be 4 and the successor of C4, namely C5 has to assume 8.

This constraint is very useful in scheduling contexts, for example if some sequence of shifts has to be followed.

Creating a constraint is done in a similar way as the 'Forbidden pattern' constraint, namely by selecting the desired variables and entering the patterns in a text field. Duplicating the constraint is done in the usual way by dragging the label.

The generated code for 5 variables and the constraint 'C0 = 1 -> C1 = 1', meaning that the successor of a 0 must be a 1, looks like this:

```
constraint A_1_1 = 0 -> A_2_1 = 1;
constraint A_2_1 = 0 -> A_3_1 = 1;
constraint A_3_1 = 0 -> A_4_1 = 1;
constraint A_4_1 = 0 -> A_5_1 = 1;
```

This code works in the following way: For each variable, there is a clause that if it has the value 0, then the successor of that variable must have the value 1.

- The 'Cyclic forward' constraint. This constraint is very similar to the 'Forward constraint' with the difference that the variables are treated in a cyclic way The successor of the last variable in the list of involved variables is the first variable.

  This constraint is also useful for scheduling applications, especially when planning cyclic work patterns for example. A concrete problem would be 'rotating workforce scheduling'.

  Creating and duplicating this constraint is done in the same way as the previous non-cyclic variant of this constraint.

  The generated code for 5 variables and the constraint 'C0 = 1 -> C1 = 1', meaning that the successor of a 0 must be a 1, looks like this:

```
constraint A_1_1 = 1 -> A_2_1 = 1;
constraint A_2_1 = 1 -> A_3_1 = 1;
constraint A_3_1 = 1 -> A_4_1 = 1;
constraint A_4_1 = 1 -> A_5_1 = 1;
constraint A_5_1 = 1 -> A_1_1 = 1;
```

  The only difference to the previous (non-cyclic) constraint is that there is an additional line at the end that includes the first element being the successor of the last element.

- The 'Number of occurrences' constraint. This constraint is useful in a variety of contexts and also studied very well in academia [FA03]. It allows to specify the number of times a specific value has to occur in the solution for a set of variables.

  To be more precise, the parameters of this constraint are: A set of variables, the value to count, an operator and a count value or variable. The value to count can be any integer or variable name. The operator can be one of the following: '=', '!=', '<=', >=' meaning that the number of occurrences has to be equal, different,

smaller or equal, larger or equal to some value or variable. The count value or variable can either be an integer or some existing variable name.

A concrete problem, where this constraint is useful, is the nurse scheduling problem. Each day it might me necessary that for example at least 4 nurses are present in the day shift, at least 3 nurses are present in the evening shift and at least 2 nurses are present in the night shift. This can be easily modeled using 'Number of occurrences' constraints: Each shift type is modeled with a specific integer, e.g 1 for day shift. Then the number of ones occurring in the set of variables that model the type of shifts of all nurse for a single day has to be at least 4.

Creating this constraint works in the following way: First, the involved variables are selected. Those are the variables where the number of occurrences are counted. When the 'Number of occurrences' constraint is selected in the context menu, a small window is opened where one can configure the remaining parameters (value to count, operator and count value/variable).

The code generated for three variables (involved variables) and the number of occurrences of 1 (variable to count) to be less or equal (operator) than 5 (count variable) is the following:

```
constraint count([A_3_1,A_2_1,A_1_1], 1) <= 5;
```

- The 'Route' constraint. This constraint is a high level constraint that can be used for routing purposes and many other applications. It is useful in any case where the user wants to find an optimal sequence.

  To be more precise, the constraint has the following functionality: An ordered list of variables is defined. The domain of the variables represent all possible elements of the sequence, and the value of the variables are the order in the sequence. Each domain element has a specific distance (or cost, depending on the actual problem formulation) to any other domain element. This means, that a distance (or cost) of the complete sequence can be calculated. This distance can be chosen to be a constant or a variable, and if it is chosen to be a variable can be minimized in a second step.

  In some sense, this is a generalization of the traveling salesperson problem, where not necessarily all cities have to be visited, the first and the last city do not necessarily have to be identical and the distances can be asymmetric and non-euclidean.

  This constraint does not only allow formulation of variants of the traveling salesperson but also allows modeling of vehicle routing problems and modeling of dependencies in scheduling problems.

  One concrete example: The shortest path between two cities is wanted. The path is a permutation of a subset of intermediate cities $c_1, ..., c_5$. Furthermore, it is undesired that c_3 is the successor or the successor of the successor of c_2.

  In our model, the integer values 1 to 5 stand for the respective cities c_i. The route constraint can then be used to formulate the search for the shortest path. This is

done by entering the distances between all cities and setting the total cost of the route to be equal to a variable that is then minimized (see subsection Optimization Goals). The avoidance of c_3 following immediately or after one other city after c_2 can be modeled with the (non-cyclic)'Forbidden pattern constraint': '3 2' and '3 ~2 2'.

In our framework, the 'Route' constraint is created in the following way: First, a set of variables is selected. The constraint is selected in the context menu. In a separate window, the user is asked to specify the distance between all domain elements and what the sum should be equal to. This can be either a constant value or a variable. Since there a quadratic amount of distances in the number of domain elements, there are two tools to make this process of entering the distances less time consuming. First, it is possible to set all distances to symmetric, which halves the amount of distances that have to be entered. This is only useful if the problem formulation has symmetric distances.

The second tool to make distance entering easier is a CSV-Import function. This means, it is possible to use an external file that contains the distances for this constraint. We use a format that is common in academia for specifying distances: A distance matrix that is separated with spaces horizontally and with line breaks vertically.

Importing distances from adjacent lists is also interesting, in particular when a graph is sparse, but is not part of our reference implementation of the framework.

Duplication of this constraint is done in the same way for as the previous constraints - by dragging the label that appears when the constraint is created. It is worth noting that if the route length is set to be equal to a variable, this variable is also changed respectively when the constraint is duplicated to other parts of the grid.

The generated code for four variables each with domain $\{1, ..., 4\}$ and the distances between all places being 1 looks like this:

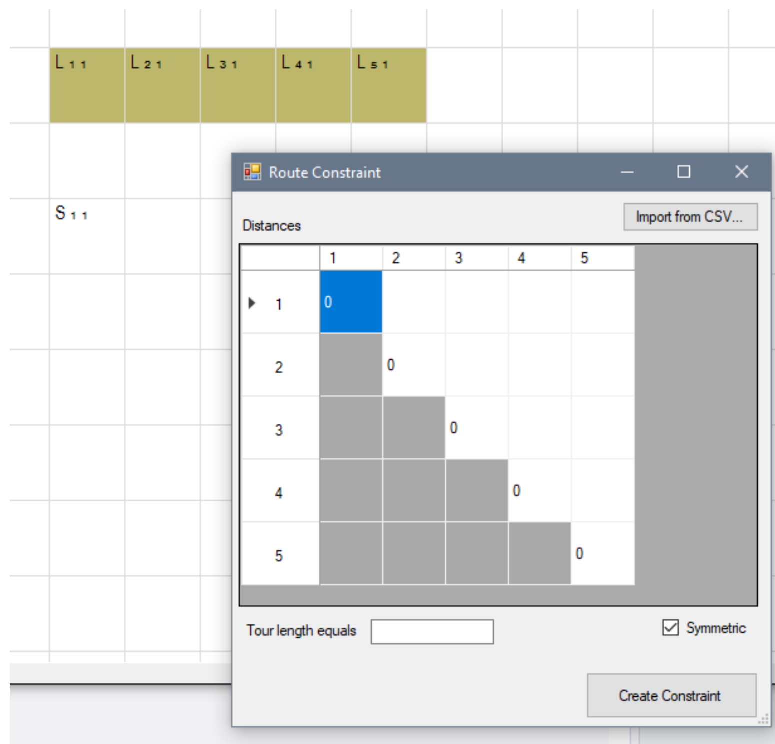
```
array[int,int] of int:  d_15832433 = array2d(1..4, 1..4,
[| 0, 1, 1, 1| 1, 0, 1, 1| 1, 1, 0, 1| 1, 1, 1, 0|]);

constraint (d_15832433[A_1_1 , A_2_1] + d_15832433[A_2_1 ,
A_3_1] + d_15832433[A_3_1 , A_4_1]) = R_1_1;
```

What is done here is the following: First, a 2d array containing all distances is created (distance-matrix). An unique identifier for this array is created that is invisible for the end-user but necessary for MiniZinc.

The second part of the code looks up the distances between two values in the distance-matrix, adds all distances and sets the sum equal to some value or variable, in this case the variable R_1_1.

In the figure 3.7, one can see the interface that is used to define the distances of the route constraints. Since in this case, the distances are selected to be symmetric,

Figure 3.7: The route constraint allows modeling of various routing and distance problems.



approximately half of the matrix is greyed-out and does not have to be filled manually.

- The 'Custom' constraint. This constraint is used to ensure that any aspects of a problem that can only be modeled succinctly with code can also be modeled that way in our framework. To be more precise, this constraint allows any MiniZinc code to be entered. This code is then inserted into the generated model.

  First, all variables that are involved in this constraint have to be selected. This selection will only be used for visualization purposes in the grid. It is then possible to enter code that will be included in the generated textual model. Duplication of this constraint is possible but in most cases not useful, as no dynamic alteration of the variables takes place if the constraint is duplicated to another part of the grid.

This concludes the list of available constraints in our framework. There exist numerous more constraints in the MiniZinc language, but above constraints are sufficient for some problems as the chapter 'Evaluation' will show.

3.4.4 Fixed Variables

Providing an intuitive way of fixing the value of variables is an important tool not only for debugging but can also be used for solution exploration and other purposes. In some sense, fixing the value of a variable is just a constraint over one variable where the relation contains only one element - the desired fixed value.

Our framework provides an easy way to add and remove such fixings of variables that is worth noting here. Once a grid cell contains a variable, double clicking it will take the user to a small window that allows him or her to enter the value that the variable has to assume in any solution.

The cell contains now the specified value in red - meaning it cannot assume any other value. If the fixed value is not desired anymore, the value can be removed with another double click on the cell and the variable is a regular variable again.

This feature is important for using the framework, as it allows rapid debugging. A concrete use case: The selected solver (see subsection Output and Solving) takes a very long time for solving and does not report a solution within acceptable time. The user thinks that the model should have at least one specific solution that fulfills all constraints and is not completely sure if the model is faulty. The user can now fix all variables to values that should satisfy the model. If the solver then reports unsatisfiable, one can be sure that the model is faulty.

Furthermore, the function of fixing variables can be used to create models for variable input for some problems, where the actual instance is determined by some fixed variables. See the chapter 'Evaluation' and in particular the subsection 'Creating models for variable input' for concrete examples.

3.4.5 Optimization Goals

As we discussed in the chapter 'Theoretical Background and existing Approaches', there are two different problems in constraint programming. Either, it is desired to find any variable assignment that satisfies all constraints (constraint satisfaction problem), or there is a specific global cost function that has to be minimized or maximized.

In practice, solutions are not always equally good. A valid schedule for a school could be optimized by taking personal teacher preferences into account, e.g on which day he or she wants to work longer or shorter.

Until now, or framework had no possibility to define such 'soft constraints' that can be satisfied but don't necessarily have to be. We will now present a way to define a cost function that the solvers will minimize or maximize.

In our framework, individual variables are minimized or maximized. In some sense, the variables selected to be optimized are the images of the functions F_i that we described in the chapter 'Theoretical Background and existing Approaches'. The function itself has to be specified using constraints.

Figure 3.8: Cells containing variables that are set as optimization goal are highlighted in color.

A_{11}	A_{21}	A_{31}
8	9	0

This means, that the user has to define constraints so that a variable or multiple variables represent the quality or penalty of the solution. A quick example: One could count the number of times some condition is not satisfied and use this number as quality criterion. Another possibility would be to use the total length of a tour (see constraint 'Route') as a quality criterion.

Once a variable is set up so that it represents part of the quality of a solution, one can use the context menu of this cell and select 'Minimize / Maximize this variable'. Then there are three options:

1. 'Minimize this variable'. Once selected, the solution quality is higher, the lower the value of this variable is. This option makes sense to be used for example with a variable that represents the total length of a route, as it is usually desired to minimize the route length.
2. 'Maximize this variable'. Once selected, the solution quality is higher, the higher the value of this variable is. One example, where this option is useful is the number of satisfied 'soft constraints'.
3. 'Bring variable close to'. When this option is selected, a new window appears where the user has to enter an integer value. The solution quality is higher, the closer the value of the variable is to the entered integer. This option is useful when one wants a variable to have a specific value, and if this is not possible as least to have a value as close as possible to the desired value.

When a optimization goal is set up, the cell that is involved is highlighted in color. This is useful, as one can immediately see the quality of the solution. In figure 3.8, one can see a variable being highlighted because it is defined as an optimization goal.

It is also possible to set multiple optimization goals. For example, one variable could be minimized while another one will be optimized. Currently all optimization goals are weighted equally. To be more precise, the following definition is used to calculate global cost function for the solution, taking into account all optimization goals:

$$q = 1 - \sum_{n=0}^{N} |g_n - v_n| \tag{3.1}$$

In above formula, q stands for the quality of the solution (higher is better). For each optimization goal, the distance to the optimal value of that goal. There must be a single optimal value for each optimization goal, as we are working with fixed domain integers and either minimization (then the optimal value is the lower bound of the domain of the respective variable), maximization (then the optimal value is the upper bound of the domain of the respective variable) or bringing a variable close to some value (then the optimal value is precisely that value). We can thus sum the distances of all goals to their actual value and use 1 minus that sum so we get a quality indicator that is better the higher.

In our framework, it is currently not possible to define different weights for the optimization goals directly. It is however possible to model the weight by specifying another variable to be a multiple of the original variable that models the image of F_i.

3.4.6 Other Features

In our reference implementation of the framework, we included many other features that make it easier to use. In this subsection, we will briefly describe some of them and why they are useful for creating constraint programming models.

1. Undo / Redo. Being able to undo and redo certain actions is a useful feature in many commercial and non-commercial applications. We thus implemented this functionality within our framework. The basic idea is the following:

   All actions (creating a constraint, variable, optimization goal,...) are put on a stack. Once the user wants to 'undo', the first element of the stack is popped and put on top of the 'redo' stack. This behavor is repeated for each time, the 'undo'-button is clicked by the user. If a new action is executed (e.g. a new constraint is created), the redo stack is cleared. If the redo-stack contains elements (i.e some action has been undone and no new action has been executed in the meantime), the redo button is available to click. Pressing it will move the topmost element from the redo-stack to the undo stack, meaning the action has been re-done.

2. Display Constraints. In our reference implementation, this function is realized as a check-box. If it is checked, all constraints that were created in the current model are displayed with color. If the check-box is unchecked, no constraints are visualized. This function is useful if the model has been created and the user wants to see the variables without distraction by the constraints.

3. Open / Save. Our implementation allows it to save and open constraint programming models. This means, once a model is created it can be saved to the disk and reused later. It is interesting to note that the saved file does not contain the MiniZinc code as a whole but the individual information about all created variables, constraints and in which cells they are located. This means it is possible to continue working with the model or create two versions from one base model by saving and altering.

The saved file of a very simple instance containing three variables, an 'All-Different'-Constraint involving all three variables and no optimization goal looks like this:

```
CONSTRAINT PROGRAMMING INSTANCE
-VARIABLES-
A_1_1|int_type|0|9|A|1|1|2|2|
A_2_1|int_type|0|9|A|2|1|3|2|
A_3_1|int_type|0|9|A|3|1|4|2|
-CONSTRAINTS-
GraphicalConstraintProgramming.AllDifferentConstraint|A_3_1|
A_2_1|A_1_1
-OPTIMIZATION GOALS-
```

The beginning of any file is a line containing the string 'CONSTRAINT PROGRAMMING INSTANCE'. A line containing just the string '–VARIABLES–' denotes the beginning of the section containing all variables. Each variable is defined in a new line with the following contents, seperated by the pipe-character: full name including indices, type, start of domain, end of domain, name without indices, x index, y index, x index of the cell in the grid containing the variable, y index of the cell containing the variable.

The section containing all constraints starts with a line with the string '–CONSTRAINTS–'. Each line contains the definition for a single constraint. As before, the parameters of the constraint are separated with a pipe-character. In this case, the first part contains the name of the constraint. For the 'All-Different' constraint, the necessary parameters are just the involved variables, that are separated with a pipe-character.

The last section deals with the defined optimization goals. Each optimization goal is written in a seperate line. The parameters are: the variable name of the variable to optimize, the optimal value for that variable and the weight of this optimization goal (which is always 1 in the current implementation).

4. Show Row / Column Caption. This feature can be activated with a check-box. Once checked, the grid is displayed with numbering for the rows and columns. This is useful if one wants to refer to a specific cell.

5. Code view. The code view can be opened by clicking a tab in the upper right corner of the main window. The grid view is changed to the textual view of the generated MiniZinc code. Not only is this valuable when trying to debug a model, but can also be helpful when trying to learn the MiniZinc syntax using this framework. In figure 3.9 one can see a screenshot of the code view. The code shows the model consists of four variables and a 'Sum equals 1' constraint. No optimization goal is set, thus the code ends with the command 'solve satisfy;'.

   Of coure, the code can also be copied in an external editor and modified and used with MiniZinc solvers directly. It could make sense to write some parts of a model using our graphical framework and completing the model in a code editor.

Figure 3.9: The code view can be used to examine the generated code.

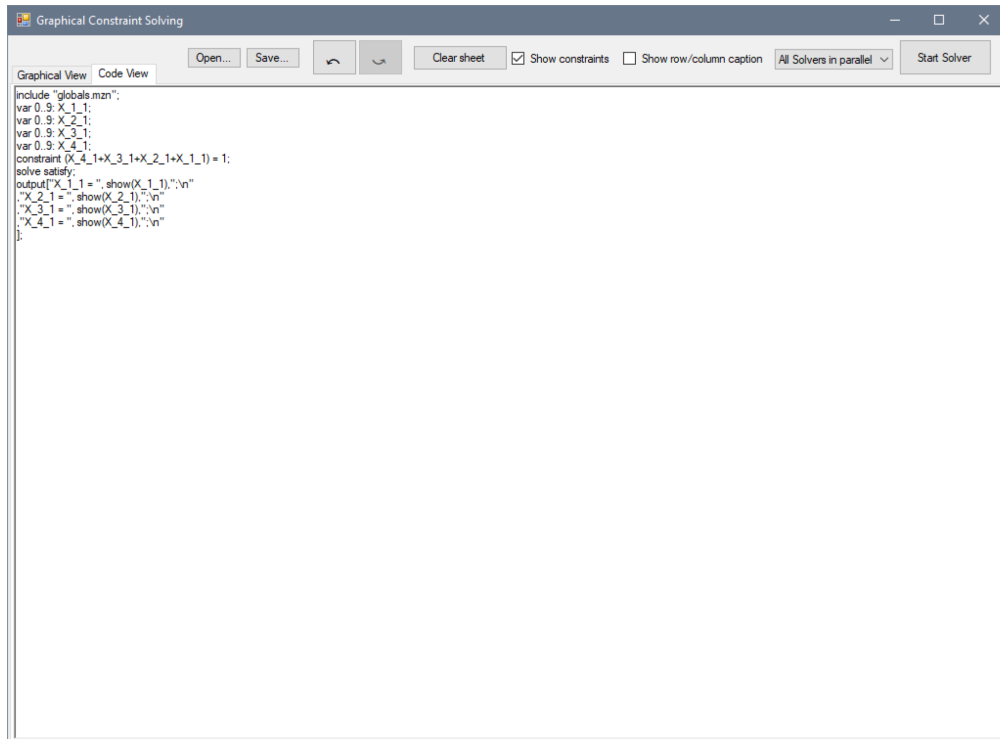
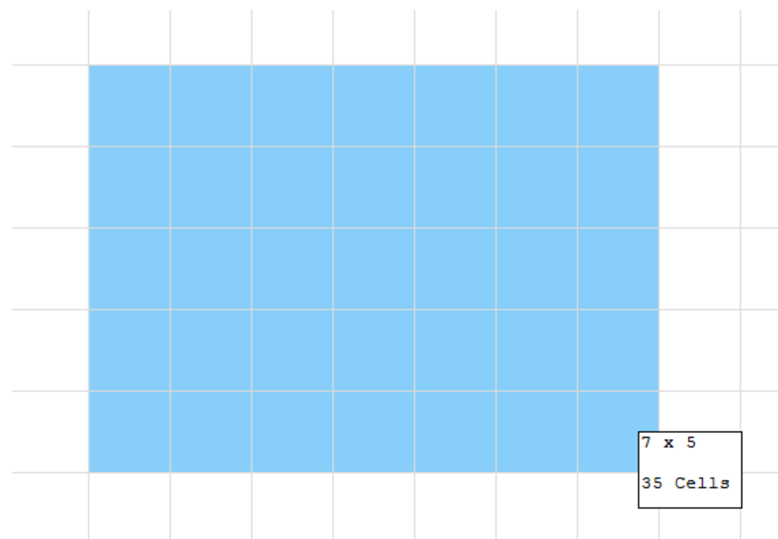


Figure 3.10: The number of cells and the width and height of the selected rectangle are displayed.



6. Selected Cells count. When selecting cells in the grid-view (default view), a small label appears displaying the total number of selected cells. If the selection is rectangular, the width and height (in cells) of that rectangle are displayed too. This could be helpful when someone wants to create a large number of variables like a 14*7 grid but does not want to count the cells one by one. A screenshot of this feature can be seen in figure 3.10.

3.4.7 Output and Code Generation

The translator module that creates code from our graphical model is essential so that we can use existing well working solvers to find solutions for our constraint programming models. To be more precise, this module works in the following way:

A header containing all necessary imports is written. In detail, the used code is:

```
include "globals.mzn";
```

Then, all code snippets of the variables are created and added after the header. In the next step, all constraints are translated to code and also added to the code. In the final step, the optimization goals and output annotations for the are added. The annotations ensure that the output is formatted in a way that our framework work with. This is necessary as we want to display the values of the variables of a solution in the grid. If no additional optimization goal is provided, the code (for a model with one variable that is specified above) concludes with the following lines:

```
solve satisfy;
output["A_1_1 = ", show(A_1_1),";"]
```

The last line is the line that ensures correct formatting the output and contains all variables that exist in the model.

The result is a valid MiniZinc code file that is saved on the hard-disc in a temporary location.

3.4.8 Solving

After a constraint programming model is created, the next step is to try so find a solution that satisfies all constraints and the domain restriction of the variables. As we discussed in the previous sections, it makes sense to use existing well established solvers.

As our framework creates models in the MiniZinc language, using existing solvers to solve them is not complicated. In the right upper corner, there is a drop-down gui-element that lets the user chose between different common solvers and the option 'All solvers in parallel'.

If a single solver is selected and the 'Solve'-button is clicked, the model is passed to that solver and an answer is awaited.

Figure 3.11: The value of a variable in the solution is displayed in the corresponding cell.

If 'All solvers in parallel' is selected, the model is passed to all available solvers. Once a solver that finds a model to be either 'unsatisfiable' or finds a solution and reports it to the framework, all other solvers are terminated. Since all solvers excel in different areas, running multiple solvers in parallel might not lead to the best runtime but to an acceptable runtime in most cases. In the chapter 'Evaluation' we will describe the benefits of that approach in more detail and with concrete runtimes and examples.

If 'unsatisfiable' is reported, the user is notified with a dialog window reporting that. If a solution is found, it is automatically parsed and the values of the variables in the solution are displayed in the respective cells. This means, the original spatial arrangement of the variables is maintained.

In the figure 3.11, one can see a solution for a model with 5 variables, all with domain $\{0, ..., 9\}$ and a single 'All-different'-constraint containing all variables.

3.4.9 Limitations

There are a number of MiniZinc constraints that are not realized in this framework. Examples would be the lexicographic functions and high level constraints like bin packing or some specialized scheduling constraints. Note that it is possible to use those constraints by entering the code in the 'custom constraint'.

We also did not include arithmetic operators like exponentiation, absolute value or modulo calculations. Those operation could be included in further iterations of our framework, but were not necessary for the problems we work with in the evaluation.

In our framework, it is not possible to specify variables that have a continuous domain. This restriction is necessary to ensure that any variable can be used in constraints that rely on a discrete domain. Because continuous variables are excluded in our framework, we did not include functions like rounding or type conversions.

Another function that is not fully included in our framework is the possibility to incorporate data files. While it is possible to read a csv-file for the 'Route' constraint, this is the only constraint for which that is possible. For commercial applications it would be of interest to have more possibilities to work with external data.

CHAPTER 4

Evaluation

In the previous chapter we introduced a graphical framework for creating constraint programming models. In this section, we will use the framework to create models for practical and well-known problems and evaluate them using different measures.

The following aspects of our chosen problems will be examined:

1. Is it possible to model a typical problem instance with our framework? If yes, how is it possible? If it is not possible to model the problem, what could be changes to the framework such that it will be possible to model the problem or is this infeasible?
2. What is the quality of the model? Is it possible to find a solution using existing solvers in reasonable time? The solving time will be compared to other models and solving techniques.
3. Is the modeling process straight forward and intuitive, as the framework claims to be? What could be extension of the framework that would improve the modeling process?
4. Are there different ways to model the problem within the framework? Which is the best way?

After the evaluation of the individual problems, we will ask if it is possible not only to model concrete instances but create a general 'solver' for any instances of some problems. We will also try to generalize the previous insights and try to formulate classes of problems for which the framework works very well and classes of problems that might be formulated better with other means.

We will also try to find out if the function 'Run all solvers in parallel' is useful in practice or if a single solver is dominant with respect to the models generated with our framework.

4.1 Sudoku

The first problem that we will be looking at is Sudoku. The problem description is the following:

Definition 1. *'A Sudoku square of order n consists of n^4 variables formed into a n^2 * n^2 grid with values from 1 to n^2 such that the entries in each row, each column and in each of the n^2 major n * n blocks are alldifferent.' [HPS05].*

Above definition is the definition for the generalized Sudoku square. The variant that we will be working with is $n = 3$.

Definition 2. *'A Sudoku problem (SP) consists of a partial assignment of the variables in a Sudoku square. The objective is to find a completion of the assignment which extends the partial assignment and satisfies the constraints.' [HPS05].*

One of the early approaches to apply constraint programming to Sudoku was by Simonis [Sim05], where he suggested the 'All-Different' constraint to be used for modeling. Constraint programming was also used by [MG06], [Lew07] and [CACM08] to solve Sudoku. Recent advances are a hybrid approach by [MW17] that combines constraint programming methods with iterated local search. Their approach aims to solve larger instances. In the evaluation they have concluded that their technique offers state of the art performance for Sudoku instances of order four and five.

In our framework, modeling the $n = 3$ Sudoku square is possible in the following way: First, a $9 * 9$ block of variables with domain $\{1, ..., 9\}$ is created. As we take from the problem definition, a total of 27 constraints have to be created. The variables in each row will be in an 'All-Different' constraint. It is sufficient to create the constraint for the first row and then using the drag-label (see previous section) to generate the other 8 constraints for the rows.

The constraints for the columns can be created in a similar fashion. A constraint is created for the first column and the other 8 column constraints are created by duplicating the original constraint with the drag-label.

Generating the 9 'All-Different' constraints for the $3 * 3$ boxes must by done individually, as duplicating would create superfluous constraints for boxes that don't necessarily have to be 'all-different'.

Once the model is created, we can solve the empty Sudoku square by pressing 'Solve', or we can fix some variables to solve a Sudoku with some numbers already filled out. This is the Sudoku problem that was defined above, that is also often published in newspapers, magazines, etc.

In figure 4.1, one can see the completed model for one configuration of the Sudoku problem, where some numbers are already fixed by us. Solving then adds the remaining numbers within a fraction of a second.

Figure 4.1: Sudoku problems can be modeled within our framework. In red: variables fixed by us.

X_{11} 5	X_{21} 8	X_{31} 2	X_{41} 4	X_{51} 3	X_{61} 1	X_{71} 6	X_{81} 7	X_{91} 9
X_{12} 1	X_{22} 4	X_{32} 6	X_{42} 7	X_{52} 8	X_{62} 9	X_{72} 3	X_{82} 2	X_{92} 5
X_{13} 9	X_{23} 7	X_{33} 3	X_{43} 6	X_{53} 2	X_{63} 5	X_{73} 4	X_{83} 1	X_{93} 8
X_{14} 8	X_{24} 2	X_{34} 4	X_{44} 3	X_{54} 1	X_{64} 6	X_{74} 5	X_{84} 9	X_{94} 7
X_{15} 3	X_{25} 9	X_{35} 5	X_{45} 2	X_{55} 4	X_{65} 7	X_{75} 1	X_{85} 8	X_{95} 6
X_{16} 6	X_{26} 1	X_{36} 7	X_{46} 5	X_{56} 9	X_{66} 8	X_{76} 2	X_{86} 3	X_{96} 4
X_{17} 4	X_{27} 5	X_{37} 1	X_{47} 8	X_{57} 7	X_{67} 2	X_{77} 9	X_{87} 6	X_{97} 3
X_{18} 7	X_{28} 6	X_{38} 9	X_{48} 1	X_{58} 5	X_{68} 3	X_{78} 8	X_{88} 4	X_{98} 2
X_{19} 2	X_{29} 3	X_{39} 8	X_{49} 9	X_{59} 6	X_{69} 4	X_{79} 7	X_{89} 5	X_{99} 1

We can thus claim that it is possible to formulate this problem within our framework.

To assess the quality of the model, we measure the time it takes to solve it. With all available solvers (Gecode, Chuffed Solver, OR-Tools), the time to solve our model is less than a second. We conclude that the quality of the model is sufficient for the $9x9$ Sudoku problem instances.

The modeling process of Sudoku within our framework is quite straight-forward. The only thing that is suboptimal is the creation of the constraints for the adjacent $3 * 3$ boxes. One possibility to improve this would be to specify a 'hot-key' that when pressed while dragging the drag-label, duplicates the constraints only to the location where the drag-label is released.

The way the $9x9$ Sudoku problem was modeled above seems to be one of the most reasonable ways to do that within constraint programming. Looking at other models[1], we can see that our approach is very similar to other formulations ('All-Different' constraints

[1] https://github.com/MiniZinc/libminizinc/blob/master/tests/examples/sudoku.mzn accessed on 22.10.2017

are used). Our model is also the same as in [Sim05], where 27 constraints for the rows, columns and blocks are used.

4.2 8 Queens

The 8 Queens problem is - like Sudoku - often used for demonstration and teaching purposes in the field of constraint programming. The setting is the following:

Definition 3. *Given an n * n chessboard, is it possible to place n queens on the board such that no queen threatens any other queen?*

It is proven [BS09] that the answer is 'yes' for any $n > 3$ and $n = 1$, and 'no' for $n = 2$ and $n = 3$. The more interesting question is thus providing a concrete solution for a specific n. We will try to find a solution for the most commonly used variant, $n = 8$.

One of the earliest approaches to solve n-queens with the means of constraint programming was by Mackworth in 1977 [Mac77]. In this paper the problem is already described as being a common problem for demonstration purposes.

One approach to model this problem with the proposed framework is the following: A $8*8$ grid with variables of domain $\{0, 1\}$ is created. The intended meaning is the following: A variable that has the value 0 in the grid stands for an empty cell of the chess board. A variable that has the value 1 in the solution stands for a cell that contains one of the 8 queens.

The constraints are added to model the properties of the queen: that they must be alone in each row, column and both diagonals going through a cell. It is relatively simple to model the first two properties: We add a 'Sum equals 1' constraint to the first row and duplicate the constraint for all other rows. The same is done for the columns: We add a 'Sum equals 1' constraint to the first column and duplicate it to the remaining 7 columns.

It remains to restrict the number of queens that can appear in the diagonals: No two queens are allowed in any of the 30 diagonals. We can create a 'Sum less or equal to 1'-constraint for the two diagonals going through the center of the grid and duplicate the constraints for the remaining variables. Unfortunately, we can only duplicate the constraint in one diagonal direction, we thus have to create the constraints for the two diagonals separately.

In figure 4.2, one can see what the finished and already solved model in our framework looks like.

We conclude that it is possible to model the problem within our framework. The solvers we used (Gecode, OR-Tools, Chuffed) were able to solve the model within less then a second. Nonetheless, in comparison to other formulation strategies (see below), our model seems relatively complex.

The modeling process is more or less straight forward and intuitive. The selection of the diagonal elements and the duplication is not as simple as in Sudoku. One thing that

Figure 4.2: The solved model shows one solution for the 8-queens problem.

Q11 0	Q21 0	Q31 0	Q41 0	Q51 0	Q61 0	Q71 0	Q81 1
Q12 0	Q22 1	Q32 0	Q42 0	Q52 0	Q62 0	Q72 0	Q82 0
Q13 0	Q23 0	Q33 0	Q43 1	Q53 0	Q63 0	Q73 0	Q83 0
Q14 1	Q24 0	Q34 0	Q44 0	Q54 0	Q64 0	Q74 0	Q84 0
Q15 0	Q25 0	Q35 0	Q45 0	Q55 0	Q65 0	Q75 1	Q85 0
Q16 0	Q26 0	Q36 0	Q46 0	Q56 1	Q66 0	Q76 0	Q86 0
Q17 0	Q27 0	Q37 1	Q47 0	Q57 0	Q67 0	Q77 0	Q87 0
Q18 0	Q28 0	Q38 0	Q48 0	Q58 0	Q68 1	Q78 0	Q88 0

could improve the modeling process would be the addition of 'hot-keys', for example for a diagonal selection or duplication.

As we observed above, our framework generates a lot of code for a relatively simple problem. More concise formulations are possible with text-based constraint programming languages.

One approach[2] works the following way: Instead of a binary formulation the solution is encoded as 8 ordered variables with domain $\{1, ..., 8\}$ indicating the position of the queens in the respective row. Additionally, all variables must be different, as the queens cannot be below or above each other. Diagonal violations are dealt with by specifying that the horizontal and vertical distance from any queen to any other queen must be different, i.e queens placed at $(3, 4)$ and $(4, 5)$ are not allowed, as their distances would be $(1, 1)$ and thus the horizontal and vertical difference would be identical. To be more precise, the absolute distance of two variables in the order must not be equal to the absolute difference of their values.

While it is possible to add above concise constraint as a custom constraint, in this case there is no benefit of using our framework in comparison to writing MiniZinc code directly.

[2] http://www.csplib.org/Problems/prob054/models/nqueens.essence.html accessed on 20.10.2017

4.3 Social Golfer Problem

The description for the original problem [GW99] is the following:

Definition 4. *32 golfers play in groups of four once a week. Is it possible to play for 10 weeks in such groups that no two golfers play against each other more than once?*

We will first model a different instance, namely with 20 golfers that play for 4 weeks.

Early approaches to model the social golfer problem with constraint programming were by Flener et. al. [FFH+01] who suggested a matrix formulation. In their formulation, matrices of decision variables are used. The authors also claim that many problems have a natural model as 2-d matrix, which also supports our claim that a grid of variables (which can be seen as a 2-d matrix) is a suitable base for our framework. Constraint programming was also used by [Har01], [RM05] and [Aze07] to solve the social golfer problem.

In our framework, we can use the set functions to model this problem succinctly. To be more precise, each group of golfers in each week is a set. Defining this is done in the following way: First, the variables for all golfers and all weeks are created. For our example, we will use 20 golfers that play in groups of 4 for 5 weeks. Thus we generate 5 times the 20 variables that represent which golfer plays in which group. The domain of each variable is $\{1, ..., 20\}$.

Then, we proceed by creating 5 groups of 4 as sets per week, resulting in 25 sets. To achieve this quickly, we can use the duplication function so we don't have to create all sets individually. The final step is to select all sets and selecting the constraint 'Set intersection at most 1'. Selecting sets is done in our framework by selecting a variable that is contained by the set. We have now restricted the solution space so that a solution is only valid when the pairwise set intersection of all sets is at most one, meaning no player can play against another player twice.

If we now try to solve this model, the solution is not found immediately with any of the solvers. Note that OR-Tools currently (as of October 2017) does not support set operations at all and is thus not able to work with our generated model. But even the other solvers (e.g. Gecode) do not find the solution instantaneously.

One reason for this is that because we are working with sets, there is a large amount of symmetrical solutions. Each group of each week can be rearranged in $4! = 24$ ways. The groups of each week can be rearranged in $5! = 120$ ways. Then, the weeks can be rearranged in $4! = 24$ ways and lead to the 'same' solution. Those are already approximately 70000 symmetrical solutions for each solution. And we did not even take isomorphic solutions that only differ in variable naming into account.

In the previous chapter, we introduced a constraint to break those symmetries to reduce the solution space. We will apply the 'Break symmetry by ascending order' constraint to accelerate the solving process.

Figure 4.3: The model and solution for the (20, 4, 5) social golfer problem.

G 1 1	G 2 1	G 3 1	G 4 1	G2 1 1	G2 2 1	G2 3 1	G2 4 1	G3 1 1	G3 2 1	G3 3 1	G3 4 1	G4 1 1	G4 2 1	G4 3 1	G4 4 1	G5 1 1	G5 2 1	G5 3 1	G5 4 1
1	3	7	16	1	4	6	10	1	17	19	20	1	5	11	13	1	8	12	14
G 1 2	G 2 2	G 3 2	G 4 2	G2 1 2	G2 2 2	G2 3 2	G2 4 2	G3 1 2	G3 2 2	G3 3 2	G3 4 2	G4 1 2	G4 2 2	G4 3 2	G4 4 2	G5 1 2	G5 2 2	G5 3 2	G5 4 2
2	4	8	17	2	9	13	19	2	11	16	18	2	3	6	12	2	5	7	10
G 1 3	G 2 3	G 3 3	G 4 3	G2 1 3	G2 2 3	G2 3 3	G2 4 3	G3 1 3	G3 2 3	G3 3 3	G3 4 3	G4 1 3	G4 2 3	G4 3 3	G4 4 3	G5 1 3	G5 2 3	G5 3 3	G5 4 3
5	6	9	18	3	8	11	15	3	9	10	14	4	14	18	19	3	13	17	18
G 1 4	G 2 4	G 3 4	G 4 4	G2 1 4	G2 2 4	G2 3 4	G2 4 4	G3 1 4	G3 2 4	G3 3 4	G3 4 4	G4 1 4	G4 2 4	G4 3 4	G4 4 4	G5 1 4	G5 2 4	G5 3 4	G5 4 4
10	11	12	19	5	14	16	17	4	5	12	15	7	9	15	17	4	9	11	20
G 1 5	G 2 5	G 3 5	G 4 5	G2 1 5	G2 2 5	G2 3 5	G2 4 5	G3 1 5	G3 2 5	G3 3 5	G3 4 5	G4 1 5	G4 2 5	G4 3 5	G4 4 5	G5 1 5	G5 2 5	G5 3 5	G5 4 5
13	14	15	20	7	12	18	20	6	7	8	13	8	10	16	20	6	15	16	19

First, we can fix the first week to the numbers 1, ..., 20 in ascending order by selecting the whole week and adding the 'Break symmetry by ascending order' constraint. This removes some of the isomorphic solutions that appear due to variable naming. The complete first week is thus fixed and can be solved without branching, because the solution space for those variables is of size 1.

We can proceed by removing symmetrical solutions from the other weeks. For each group and each week, we can select the players to be in ascending order. As stated above, this reduces the number of branches by a factor of 24 for each group and week.

We can then order the groups per week by their lowest numbered player. This removes a symmetry factor of 120 for each week. Additionally, this fixes the first player of each week to be player 1, as the player is guaranteed to be the first player in each group he plays and furthermore the group the player participates will be the first group per week.

As the final step, we can sort one arbitrary value of each group in ascending order, to reduce the solution space by an additional factor of 120 by fixing the order the weeks occur. With all those measures, we have reduced the solution space so much that the solvers (Chuffed, Gecode) report a valid solution on our average desktop computer used for evaluation nearly instantly. One possible solution can be seen in figure 4.3.

We can therefore answer the questions regarding the evaluation: A problem instance can be modeled within the framework. The initial model is created very fast, breaking symmetries is then far more time consuming.

The quality of the model is good, a solution for the $(20, 4, 5)$ instance can be found within a second, but only after symmetry breaking constraints are added. With the plain model, we could not get a result until timeout (1000 seconds).

In addition to the instance with 20 golfers and 5 weeks, we have used the framework to model the original instance with 32 golfers and up to 10 weeks. The runtimes of all configurations (1 to 10 weeks) can be seen in table 4.1. The test setup was the following: Each instance was modeled and solved with all solvers in parallel on our test machine (Intel Core2Quad Q6600, 6GB Ram). Each instance is tested 3 times and the average

Table 4.1: The runtimes for the social golfer problem instances with 32 players and up to 10 weeks.

Instance	Runtime in seconds w/o symmetry breaking	RT with symmetry breaking
32 players / 1 week	<1	<1
32 players / 2 weeks	<1	<1
32 players / 3 weeks	<1	<1
32 players / 4 weeks	<1	<1
32 players / 5 weeks	<1	<1
32 players / 6 weeks	>1000	9
32 players / 7 weeks	>1000	>1000
32 players / 8 weeks	>1000	>1000
32 players / 9 weeks	>1000	>1000
32 players / 10 weeks	>1000	>1000

runtime is used. A value of >1000 denotes that we were not able to solve the instance within 1000 seconds.

We can conclude our examination of this problem by noting the following findings: Instances with at most 20 golfers (100 variables) can be solved quickly. Regarding larger instances (up to 320 variables, 32 golfers, up to 10 weeks), we could only solve instances with up to 6 weeks. In our test setup, incorporating symmetry breaking methods only changed the runtime in one instance, where without symmetry breaking we did not get a result at all within our timeout.

The modeling process itself is mostly straight forward and intuitive. A really good addition would be mechanisms to automatically apply symmetry breaking features instead of having to create the constraints manually. On the other hand, those methods could be better added to the solver instead, making the improvements available to a wider audience.

Regarding other formulations, there exist various possibilities to formulate the social golfer problem as constraint satisfaction problem. However, there are often very refined models used, as the solution space of this problem grows very fast by increasing number of players, weeks and groups. We tried another formulation using a player-group assignment (i.e in which group each player plays for each week) but this model was not as efficient for the $(20, 4, 5)$ instance. We could not get a result within the timeout (1000 seconds).

4.4 Rotating Workforce Scheduling Problem

The Rotating Workforce Scheduling problem exists in several variations. The main problem setting is similar: a schedule for workers has to be created. There exist different shift types, typically 'day', 'evening' and 'night'. The planning horizon is usually one

or more weeks, after which the schedule of a single worker continues as the schedule of the next worker. There are constraints regarding the weekends, legal shift sequences and minimum available workers of each day.

Early approaches to model cyclic employee scheduling problems with constraint programming include a paper by Chan et al [CW01]. They use high level constraints like 'sequence', that make it possible to specify allowed patterns in a sequence of variables.

We consider the rotating workforce scheduling problem as defined in [MGS02]. Constraint programming was also used by [LP04] and [TM11] to solve the this problem.

We will use our framework to model a typical instance, namely instance 8 from the paper [Mus05], that was already used for the evaluation of previous methods. The properties of the instance are:

1. The number of workers is 16.
2. The total planning horizon is 1 week (=7 days).
3. A shift change from evening to day, night to evening, and night to day must be separated with at least one day off.
4. A shift must be worked at least twice.
5. The maximum number of consecutive work days is 7 and the minimum number is 3.
6. The number of consecutive rest days is between 2 and 4.
7. The maximum number of consecutive day shifts is 7, of consecutive evening shifts is 6 and of consecutive night shifts is 5.
8. The number of workers needed for each day is given in matrix where each row represents the demands for a specific shift type and each column represents a single day. In our case, the matrix is:

   5 5 5 5 5 2 0
   5 5 5 5 5 2 0
   3 3 3 3 2 0 3

We can model this instance in the following way:

A $7 * 16$ grid for the assignments of the workers for each day is created. The domain of the variables is $\{0, ..., 3\}$. In our model, 0 stands for no shift, 1 is the morning shift, 2 is the evening shift and 3 is the night shift.

Next, we want each shift to be manned according to our matrix every day. We create a 'Number of occurrences' constraint selecting the 16 variables representing the first day

and set the value we count to 1, the operator to '=' and the number of occurrences to 5. This means, the number of '1-shifts' (morning shifts) must be at least 5. We can duplicate this constraint by dragging it to the next 4 days. The same procedure is done for all shift demands.

For the other constraints, we create one 'Cyclic forbidden patterns' constraint. We add the following configuration text:

```
0 0 0 0 0
~0 0 ~0
0 ~0 0
0 ~0 ~0 0
~0 ~0 ~0 ~0 ~0 ~0 ~0 ~0
3 1
3 2
2 1
~1 1 ~1
~2 2 ~2
~3 3 ~3
1 1 1 1 1 1 1 1
2 2 2 2 2 2 2
3 3 3 3 3 3
```

The first line prevents 'off-blocks' to be of size 5 or more. The next line prevents 'off' blocks to be of size 1.

The third line prevents work blocks of size 1. The fourth line excludes work blocks of size 2. Together with the previous line, the constraint that work blocks must be at least of length 3 is modeled.

The fifth prevents work blocks to be of size 8 or more.

The next 3 lines forbid combinations of different work blocks to come after each other without a day off. The last 3 lines define the minimum and maximum block size of specific shift types.

The model we created above can be solved within approximately 3 seconds when we select the option 'All solvers in parallel'. This result is interesting, as it is comparable to other solving techniques [TM11]. In figure 4.4, we can see the model and solution for this instance.

We can thus conclude that it is possible to model an instance of this problem and solve it within reasonable time. The modeling process itself was quite straight forward, since we could use the 'Number of occurrences' constraint and the patterns were easily defined using the 'Cyclic forbidden pattern' constraints. One further extension of the framework could be an interface that allows the specification of patterns in a graphical instead of text based way.

Figure 4.4: The model and solution for the instance 8 of the rotating workforce scheduling problem.

W 1 1 3	W 2 1 3	W 3 1 3	W 4 1 3	W 5 1 0	W 6 1 0	W 7 1 3
W 1 2 3	W 2 2 3	W 3 2 3	W 4 2 0	W 5 2 0	W 6 2 0	W 7 2 0
W 1 3 1	W 2 3 1	W 3 3 1	W 4 3 2	W 5 3 2	W 6 3 2	W 7 3 0
W 1 4 0	W 2 4 0	W 3 4 0	W 4 4 1	W 5 4 1	W 6 4 1	W 7 4 0
W 1 5 0	W 2 5 0	W 3 5 0	W 4 5 1	W 5 5 1	W 6 5 1	W 7 5 3
W 1 6 3	W 2 6 0	W 3 6 0	W 4 6 2	W 5 6 2	W 6 6 2	W 7 6 0
W 1 7 0	W 2 7 3	W 3 7 3	W 4 7 3	W 5 7 3	W 6 7 0	W 7 7 0
W 1 8 2	W 2 8 2	W 3 8 2	W 4 8 3	W 5 8 3	W 6 8 0	W 7 8 0
W 1 9 1	W 2 9 1	W 3 9 1	W 4 9 2	W 5 9 2	W 6 9 0	W 7 9 0
W 1 10 2	W 2 10 2	W 3 10 2	W 4 10 2	W 5 10 2	W 6 10 0	W 7 10 0
W 1 11 2	W 2 11 2	W 3 11 2	W 4 11 0	W 5 11 0	W 6 11 0	W 7 11 0
W 1 12 1	W 2 12 1	W 3 12 1	W 4 12 1	W 5 12 1	W 6 12 0	W 7 12 0
W 1 13 2	W 2 13 2	W 3 13 2	W 4 13 2	W 5 13 2	W 6 13 0	W 7 13 0
W 1 14 1	W 2 14 1	W 3 14 1	W 4 14 1	W 5 14 1	W 6 14 0	W 7 14 0
W 1 15 1	W 2 15 1	W 3 15 1	W 4 15 1	W 5 15 1	W 6 15 0	W 7 15 0
W 1 16 2	W 2 16 2	W 3 16 2	W 4 16 0	W 5 16 0	W 6 16 0	W 7 16 3

The actual code of the model created is somewhat comparable to the model created others [TM11]. In fact, our approach of 'forbidden patterns' and the language used to model the patterns is somewhat similar to the existing language by [TM11].

For further evaluation, we tested our framework on all other instances from [Mus05] as well. In table 4.2, one can see the runtimes in seconds for each instance. A value of >1000 denotes that we were not able to solve the instance within 1000 seconds. The column captioned with 'Runtime with fixed start' denotes the runtimes for the model where some value is fixed as first variable to eliminate symmetric solutions. For example, if there is a demand of shift 1 of at least 1, we set the very first variable to one. As can be seen, this reduces the runtime in some cases. In general, the results and comparable to other solving methods (see [EM17]).

Table 4.2: The runtimes for 20 rotating workforce scheduling instances

Instance	No. of workers	Runtime in seconds	Runtime with fixed start
1	9	<1	<1
2	9	<1	<1
3	17	<1	<1
4	13	<1	<1
5	11	<1	<1
6	7	<1	<1
7	29	<1	<1
8	16	<1	<1
9	47	>1000	>1000
10	27	<1	<1
11	30	>1000	>1000
12	20	143	3
13	24	202	<1
14	13	<1	<1
15	64	>1000	>1000
16	29	<1	<1
17	33	<1	<1
18	53	<1000	<1
19	120	<1	<1
20	163	<1000	<1000

4.5 Magic Hexagon

We chose this problem because it is not as common as Sudoku or 8-queens but also an interesting mathematical problem. This problem is a number assignment problem with additional constraints. The formal definition is the following:

Definition 5. *'A magic hexagon of order n is an arrangement of close-packed hexagons containing the numbers $1, 2, ..., H_{(n-1)}$, where H_n is the nth hex number such that the numbers along each straight line add up to the same sum. In the magic hexagon of order n=3, each line (those of lengths 3, 4, and 5) adds up to 38.' [Wei17].*

We will demonstrate the common variant where $n = 3$. Modeling the problem is straight forward. We start by creating a grid of size $5*5$. The domain of the variables is $\{1, ..., 19\}$. We do not need the first and last variable in the first and last row. Further we do not need the first variable in the second and second to last row. We can fix those variables to 0 by hand (see previous chapter). The other variables are selected and an 'All-different' constraint is created.

It then remains to add some 'Custom sum constraints' to fix the sum of the variables in the rows, columns and diagonals to 38. This process is not complicated but tedious.

Solving this model is a matter of less then a second, which is not surprising since the number of variables that we use is only 19.

We can observe the following: It is possible to model the problem within our framework. The quality of the model is acceptable, as it is possible to solve it within a fraction of a second with all solvers.

The modeling process is intuitive albeit not that fast as other approaches (e.g. specification via code) as a lot of constraints have to be created individually. An extension of the framework could be the following: Instead of a grid it could be possible to create variables in other shapes and even freely arranged in the 2d space. A selection could be done in a 'lasso'-like manner (see also: [LH11]).

There may be different formulations of this problem, but most models[3] look very much like our generated model.

4.6 TSPTW

The Traveling Salesperson Problem with Time Windows (short TSPTW) is a generalization of the Traveling Salesperson Problem that incorporates time constraints for each location.

The description of the problem is:

Definition 6. *'In the Traveling Salesman Problem (TSP) a set of N cities (one of which is the depot) and their pairwise distances are given. The task is to find the shortest route that starts and ends at the depot and visits each city only once. In the Traveling Salesman Problem with Time Windows (TSPTW), additionally to the TSP, each city has to be visited and left within a given time interval.' [EGCT13].*

First, we will show how to model the problem. We will use a smaller instance that was already used in the academic context. As we have defined a constraint in our framework that can directly use distance-matrices from a csv-file, the López-Ibáñez-Blum format[4] will be used in our example. The concrete instance that is used is called 'LIB_test.tsptw' and can be found in the link provided the footnote.

The first step is to create a $6*1$ grid of variables named $C_{1,1}$ to $C_{6,1}$ with domain $\{1, ..., 5\}$. Those variables will represent the order that the cities are visited. The first and the last variables are fixed to the depot, namely the value 1. Then, an 'All-different' constraint is created for the first 5 variables. This will ensure that all cities are visited once.

Then, 6 variables named $L_{1,1}$ to $L_{6,1}$ are created to reflect the distance between consecutive cities in the order that is defined in the first 6 variables. For this, we create a 'Route'

[3] http://www.csplib.org/Problems/prob023/models/ accessed on 22.10.2017.

[4] https://acrogenesis.com/or-tools/documentation/user_manual/manual/tsp/first_tsptw_implementation.html accessed on 22.10.207

constraint between the first two variables ($C_{1,1}$, $C_{2,1}$) in the order and set the route length to be equal to $L_{2,1}$. We can duplicate this constraint horizontally such that all $L_{i,1}$ represent the distance between the locations $C_{i-1,1}$ and $C_{i,1}$.

An additional 6 variables called $T_{1,1}$ to $T_{6,1}$ are created to represent the total time after each city (the time when a city is left to the next one). This value $T_{i,1}$ is calculated to be larger than the sum of the previous time $T_{i-1,1}$ and the distance to the next location, $L_{i,1}$. The reason that the value can by anything greater than the sum and does not have to be exactly the sum is that it may be favorable or even necessary to wait and thus 'waste' time. We thus use a 'Custom sum constraint' and set the operator to '>='. This constraint has to be created once and can then duplicated using the drag label to all other $T_{i,1}$.

It remains to show how the time-constraints are modeled. We want the leave time of each city $C_{i,1}$ to be greater or equal to the earliest possible leave time for that city (that value is given in the instance file) and the leave time to be less or equal to the latest leave time for that city (that value is also given for each city). We can model this restriction with a 'forbidden pattern' constraint. We select the first unfixed city $C_{2,1}$ and the leave time $T_{2,1}$ using the following configuration of the 'forbidden pattern' constraint:

```
2 <197
2 >216

3 <147
3 >165

4 <242
4 >254

5 <56
5 >67
```

The intended meaning is the following: each line forbids a specific pattern to occur. Each pattern describes one time constraint for one city. The first line for example can be understood the following way. We do not want the pattern $2 < 197$ to occur in the ordered variables $C_{2,1}$ and $T_{2,1}$. This means, it is forbidden that the city is 2 and the leave time for that city is less than 197 (as this is demanded so in the instance file).

We can duplicate this constraint to all other cites and have thus modeled the time constraints. The now completed model can be seen in figure 4.5.

If one wanted to the model the decision variant of this problem (Is there a route that satisfies all constraints?) we are done. We can change this to the optimization variant (What is the route with the earliest arrival time at the depot after visiting all cities?) easily. We click to the time at the depot at the end, namely $T_{6,1}$ and select 'Minimize this variable'.

Figure 4.5: The Travelling Salesperson with Time Windows Problem can be modeled with our framework.

C_{11} 1	C_{21} 5	C_{31} 3	C_{41} 2	C_{51} 4	C_{61} 1
L_{11} 0	L_{21} 67	L_{31} 95	L_{41} 42	L_{51} 47	L_{61} 26
T_{11} 0	T_{21} 67	T_{31} 162	T_{41} 204	T_{51} 251	T_{61} 277

Using the solve function 'All solvers in parallel' we find the optimal solution for this problem within less then a second.

We have shown that we can in fact model the TSPTW within our framework. A result for smaller instances can be achieved in reasonable time (approximately 1 second on our machine).

The modeling process is in our opinion a little bit complicated, as there are a lot of different constraints. It is also necessary to decompose the problem into subproblems. Our framework could be improved by providing more high level constraints that are specialized for these kinds of problems.

The generated model is quite straight forward and relatively easy to understand. It may however be the case that the use of the 'forbidden pattern' constraint is not optimal, as the number of constraints needed for modeling the time windows is quadratic in the number of cities.

4.7 Traveling Tournament Problem

The Traveling Tournament Problem is closely related to the Traveling Salesperson Problem (see above) and various tournament scheduling problems. In fact, the problem deals with teams that play in different locations. The problem description is:

Definition 7. *'Given n teams with n even, a double round robin tournament is a set of games in which every team plays every other team exactly once at home and once away. A game is specified by an ordered pair of opponents. Exactly $2(n-1)$ slots or time periods are required to play a double round robin tournament. Distances between team sites are given by an n by n distance matrix D. Each team begins at its home site and travels to play its games at the chosen venues. Each team then returns (if necessary) to its home base at the end of the schedule. Consecutive away games for a team constitute a road trip; consecutive home games are a home stand. The length of a road trip or home stand is the number of opponents played (not the travel distance).' [ENT01].*

Easton et. al. [ENT01] also described how constraint programming can be used to solve this problem. They note that even instances of size 6 are already challenging. A recent method for the 'Traveling Tournament Problem' problem is a combination of integer programming and local search [GW16]. The authors claim that their approach was able to find many new best known solutions for larger instances.

We use a small instance with 4 cities/teams for our demonstration purposes[5].The distances can be also found in the footnote.

We can model this problem with our framework in the following way: We create a grid of $4 * 6$ variables that will represent the solution in the same way that Michael Trick (see footnote) used on his website. The domain of the variables will be $\{0, ..., 4\}$. The value 0 represents that the respective team plays at home. The values $1, ..., 4$ represent that a team plays a road game against the respective team. Since each team starts and ends their route at home, we will add additional 4 variables fixed to 0 above our grid and 4 variables fixed to 0 below our grid that represent the route.

Below all those variables we create 4 variables that represent the sum of each route of each team. The domain of those variables is $\{0, ..., 99999\}$ as the route can have arbitrary length. We chose the upper bound of the domain to be 99999 as this value is as least as large as 7 times the maximum of any distances. Then, another variable is created that is equal to the total travel time of all teams (again with large domain). That variable is selected to be minimized.

To correctly calculate the route length, we create a route constraint for each team selecting the variables that represent the route and the start and end (0). The distances can be imported from the csv given in the instance. Unfortunately, each route constraint is slightly different. The distance matrix has to be adapted such that the distances from 0 are equal to the distances from the current team. For team 1 for example, all distances from 0 are equal to all respective instances from 1, as 0 has the indented meaning of a home play. The sum of the route is set equal to the variable that represents the length of the route for each team.

It remains to add constraints such that the schedule is valid. First, the number of occurrences of 0 in the schedule of each team (without start and end) is 3. Furthermore, the number of each of the numbers $1, ..., 4$ is less or equal to 1. The number of occurrences of the own location (e.g. value 1 for team 1) is always 0, as a team cannot do a road game against itself.

Then we will model the fact that a team can only play a road game against another team if and only if that other team plays at home. We will create another $4 * 6$ variables with the domain $\{0, 1\}$ that represents if a team is played against in a road game. To model this, we create a 'number of occurrences' constraint. The number of occurrences of the values $1, .., 4$ is equal to the respective variable in the new grid and thus between 0 and 1. We then create 'forbidden pattern constraints' with the following content:

[5] http://mat.tepper.cmu.edu/TOURN/ accessed on 20.10.2017

```
~0 1
0 0
```

The first line states that if a city plays a road game, it cannot be played against in a road game (as the city is on tour). The second line states that if a city plays at home, another team must play against that city as a road game.

The constraint that no more than three consecutive home or road plays are allowed is not needed in this instance (as there are only a total of three home and three road plays) but could be modeled with a 'forbidden pattern' constraint containing the tour of each team:

```
~0 ~0 ~0 ~0
0 0 0 0
```

The first line states that 4 (or more) consecutive road games are forbidden, the second line states that 4 (or more) home plays are forbidden.

The only thing now remaining is the fact that repeaters are forbidden. This means, a team must not play against the team of the previous day.

In our model, incorporating this restriction is quite complicated. A quadratic (in the number of teams) number of constraints has to be created manually. A 'forbidden pattern constraint' is created for each pair of teams and the first two days. These constraints can then be duplicated to the rest of the days.

The final model can be seen in figure 4.6.

The model can be solved within less then a second, which is not surprising given that the instance contains only four teams. We also created a model for a larger instance containing 6 teams that was solved within 75 seconds on our machine.

Summarizing, we can state that it is possible to model the Traveling Tournament Problem with our framework. The quality of our model is reasonable, as we were able to solve smaller instances quite fast.

The modeling process is very tedious and probably a lot slower than creating the model in code. As we found out in the evaluation of the previous problems, models that require various different constraints are less suitable for our framework. Even adding additional high-level constraints would not solve the problem that the modeling process gets unhandy with increasing complexity.

Looking at existing models for this problem[6], we can state that our approach has many additional variables that are not needed with other approaches. It will very likely be the case that our approach is slower on larger instances, as the solution space is larger.

[6] http://csplib.org/Problems/prob068/models/TTPPV.mzn.html accessed on 20.10.2017

Figure 4.6: An instance of the Traveling Tournament Problem modeled within our framework.

S_{11} 0	S_{21} 0	S_{31} 0	S_{41} 0
T_{11} 0	T_{21} 0	T_{31} 1	T_{41} 2
T_{12} 0	T_{22} 3	T_{32} 0	T_{42} 1
T_{13} 0	T_{23} 1	T_{33} 0	T_{43} 3
T_{14} 3	T_{24} 4	T_{34} 0	T_{44} 0
T_{15} 2	T_{25} 0	T_{35} 4	T_{45} 0
T_{16} 4	T_{26} 0	T_{36} 2	T_{46} 0
E_{11} 0	E_{21} 0	E_{31} 0	E_{41} 0
SUM_{11} 2011	SUM_{21} 2011	SUM_{31} 2127	SUM_{41} 2127
			TOT_{11} 8276

O_{11} 1	O_{21} 1	O_{31} 0	O_{41} 0
O_{12} 1	O_{22} 0	O_{32} 1	O_{42} 0
O_{13} 1	O_{23} 0	O_{33} 1	O_{43} 0
O_{14} 0	O_{24} 0	O_{34} 1	O_{44} 1
O_{15} 0	O_{25} 1	O_{35} 0	O_{45} 1
O_{16} 0	O_{26} 1	O_{36} 0	O_{46} 1

4.8 Simple Teacher Scheduling

Teacher Scheduling is an assignment problem for the schedule of a school. In the simple variant, only one class is scheduled at a time.

The problem is defined in the following way: Each course (or teacher) has a specific domain, meaning that the class can only take place on specific times and days. There are restrictions that some courses must take place blocked and some courses have to be held separately (on a separate day for example).

All classes have to be assigned according to the constraints. Additionally, the number of courses that have priority (for example mathematics and languages) that take place in the morning should be maximized.

We will model this problem in the following way: For every possible date a lesson can take place a variable is created. We thus create a $7 * 10$ grid, meaning each lesson can possible take place from Monday to Sunday from 8:00 to 18:00. The domain of the variables is as large the number of different courses or teachers we want to schedule plus one, in our case 11 if we have 10 courses. We thus set the domain to $\{0, ..., 10\}$. The value 0 represents that the slot is unused (empty). If we want to exclude lessons to take place on the weekend in beforehand, one can either create a smaller grid with less slots

or fix the variables to 0.

Next, we will add the constraints that each course can only have specific dates or times. For this, we also need the number of times a specific course should be held per week.

We select all variables that represent slots that a single course (e.g. mathematics) can be held and create a 'Number of occurrences constraint'. We state that the number of occurrences of 1 (as this is the first course we model) has to be equal to 4 (the number of times mathematics is held per week).

If we do this for all other courses, we can be sure that all courses are assigned according to the time and date restrictions. We add another constraint where we fix the number of times the value '0' is assigned such that no superfluous assignments are done to empty slots.

We will continue by creating constraints for the restrictions that some courses only appear blocked or must not be held on the same day.

If a course has to occur in blocks of two, a 'forbidden pattern constraint' with the following content is created for each day:

```
~1 1 ~1
```

Above constraint will prevent that the course that corresponds to value 1 occurs as a single (non-blocked) lesson.

If we want to achieve the opposite, namely that a course can only appear as a single lesson, we can create the following 'forbidden pattern constraint':

```
1 1
```

Above constraint will prevent the course 1 to appear directly after itself. If we wanted to exclude a course to appear more than once per day, we can create a 'Number of occurrences constraint' and set the number of occurrences of 1 for example to be at most 1.

We can exclude small 'holes' in the schedule in a similar way. If we create a 'forbidden pattern constraint' with the content

```
~0 0 ~0
~0 0 0 ~0
```

Above configuration has the following intended meaning: The first line prevents 'holes' in the schedule of size one, i.e a single empty slot between two lessons. The second line prevents 'holes' of size two. In any solution of the model empty slots between courses must now be at least 3 hours long.

It remains to show how we optimize the schedule such that the number of courses with priority are mostly in the morning.

This can be done in the following way: All variables that count as morning are selected (e.g. the first 3 slots of each day). Then, a 'number of occurrences' constraint is created.

Figure 4.7: Simple Teacher Scheduling can be modeled and solved within our framework.

$C_{1\,1}$ 4	$C_{2\,1}$ 1	$C_{3\,1}$ 2	$C_{4\,1}$ 3	$C_{5\,1}$ 3	$C_{6\,1}$ 0	$C_{7\,1}$ 0		$P_{1\,1}$ 3
$C_{1\,2}$ 2	$C_{2\,2}$ 1	$C_{3\,2}$ 2	$C_{4\,2}$ 3	$C_{5\,2}$ 1	$C_{6\,2}$ 0	$C_{7\,2}$ 0		$P_{1\,2}$ 3
$C_{1\,3}$ 4	$C_{2\,3}$ 5	$C_{3\,3}$ 5	$C_{4\,3}$ 6	$C_{5\,3}$ 6	$C_{6\,3}$ 0	$C_{7\,3}$ 0		$P_{1\,3}$ 3
$C_{1\,4}$ 7	$C_{2\,4}$ 7	$C_{3\,4}$ 8	$C_{4\,4}$ 8	$C_{5\,4}$ 9	$C_{6\,4}$ 0	$C_{7\,4}$ 0		$TOT_{1\,1}$ 9
$C_{1\,5}$ 0	$C_{2\,5}$ 0	$C_{3\,5}$ 0	$C_{4\,5}$ 0	$C_{5\,5}$ 0	$C_{6\,5}$ 0	$C_{7\,5}$ 0		
$C_{1\,6}$ 0	$C_{2\,6}$ 0	$C_{3\,6}$ 0	$C_{4\,6}$ 0	$C_{5\,6}$ 0	$C_{6\,6}$ 0	$C_{7\,6}$ 0		
$C_{1\,7}$ 0	$C_{2\,7}$ 0	$C_{3\,7}$ 0	$C_{4\,7}$ 0	$C_{5\,7}$ 9	$C_{6\,7}$ 0	$C_{7\,7}$ 0		
$C_{1\,8}$ 0	$C_{2\,8}$ 0	$C_{3\,8}$ 0	$C_{4\,8}$ 10	$C_{5\,8}$ 11	$C_{6\,8}$ 0	$C_{7\,8}$ 0		
$C_{1\,9}$ 0	$C_{2\,9}$ 0	$C_{3\,9}$ 0	$C_{4\,9}$ 10	$C_{5\,9}$ 11	$C_{6\,9}$ 0	$C_{7\,9}$ 0		
$C_{1\,10}$ 0	$C_{2\,10}$ 0	$C_{3\,10}$ 0	$C_{4\,10}$ 0	$C_{5\,10}$ 0	$C_{6\,10}$ 0	$C_{7\,10}$ 0		

The number of occurrences of 1 (mathematics) must be equal to some fresh variable. The same is done for all other priority courses. Then, a variable that represents the sum of all those counting variables is created. By clicking right on that variable, one can select 'Maximize this variable'.

The resulting solution will then be a solution that satisfies each 'hard constraint', e.g. respecting the possible dates for each course and will also be optimal in terms of scheduling the priority courses as early as possible. In figure 4.7, one can see the completed model and the solution. With the function 'All solvers in parallel', we were able to solve the satisfiability variant of the problem within less than a second on our machine. For the optimization variant, it took approximately 10 minutes to get to the optimal solution.

If one wants to extend the problem to a more advanced model incorporating multiple classes, one could do this in the following way: The slots for all classes are created separately, Then, the variables are used to specify specific courses like mathematics. Constraints are created to account for the fact that for any given time only a specific number of teachers are available.

Since we used only a total of 50 variables (we fixed the weekends to all 0), solving the model was done in less than 10 seconds with all solvers (Gecode, Chuffed, OR-Tools). We conclude that our framework is suitable to model such school scheduling problems.

4.9 3-SAT

The 'Boolean Satisfiablity Problem' is of big importance in the field of computer science, as it can be used to prove NP-hardness of a problem. Here, we will show how an instance of '3-SAT' can be modeled within our framework. We have chosen '3-SAT' over the general 'Boolean Satisfiability Problem' because in '3-SAT', the clause length is fixed to 3 and this makes it easier to model with our framework. Nonetheless '3-SAT' is NP-complete and thus can be used to prove the NP-hardness of problems.

Definition 8. *Given a SAT-formula F, where each clause has a length of 3, is F satisfiable?*

Suppose we have an instance of '3-SAT' with 10 clauses and thus a maximum of 30 different variables. The instance can be modeled in the following way:

A grid of variables of size $3 * 10$ is created. Each variable has the domain $\{0, 1\}$. Then, for each different variable v_i in the original instance, the following is done:

1. All variables (in our framework) that represent the positive occurrence of v_i are selected and an 'All equal constraint' is created.
2. All variables (in our framework) that represent a negative occurrence of v_i are selected and an 'All equal constraint' is created.
3. If there are both positive and negative occurrences of a variable v_i: One variable that represents a negative occurrence of v_i is selected and a variable that represents a positive occurrence of v_i is selected and an 'All-Different constraint' is created.
4. For the first clause (3 variables) a 'Custom sum constraint' is created, stating that the sum of the variables must be at least 1. This constraint is duplicated for all other clauses.

The sum constraint forces each clause to have at least one variable assigned to 1, which corresponds to the truth assignment 'true'. The other constraints ('all equal') force all variables that occur more than once to have a consistent assignment. There is a solution for the model if and only if there is a solution for the original '3-SAT' instance, as we can show: A valid solution for the model is also a model for the original instance since each variable has exactly one assignment and each clause contains at least one literal that evaluates to 'true' (because the sum of the truth assignments is specified to be 1 or more). On the other hand, if there is a model for the '3-SAT' instance, a respective variable assignment for our model exists that satisfies all constraints.

Above proof (sketch) can be used to show that we can solve arbitrary concrete instances of the 3-SAT problem within our framework.

Modeling this problem is interesting in more a theoretical than a practical sense, as modern SAT solver would be faster at solving. Nonetheless, it is noteworthy that instances of the 3-SAT problem can be solved within our framework.

4.10 Creating Models for Variable Input

One of the disadvantages of our framework is the fact that it is - at least on first sight - not suitable for creating models that solve more than a single fixed instance. An example: it might be useful to have a solver that can solve scheduling problems for any amount of workers, machines and working days without recreating the whole model for each individual configuration.

Our framework relies on the fact that a single solution of fixed format is desired. Models that work with variable input on the other hand often have different outputs depending on the input. The output for a $n * n$ Sudoku for example is an $n * n$ matrix containing the solution.

In this section we will present two examples how we can overcome these limitations.

4.10.1 N-queens

This problem is a generalization of the 8-queens problem that we already have modeled above. The main difference is now that we will create a solver that can provide solutions for different given n. While it is in principle possible to create a solver within our framework that can solve arbitrarily large n, we will show how to create a solver for any given n between 1 and 14.

The first step is to create the solution space for the largest n. In our case we will create a $14 * 14$ grid. The sum of variables of each row and each column must be less or equal to 1, regardless of n. Our idea is it to force the solution for a given n to be in the upper left corner. We thus create additional $2n$ variables counting the number of queens in each row and column. We create constraints such that the ordered variables that represent the counting above must be descending. Then, we create another variable representing n and state that the total sum of the complete grid must be equal to n. The last step is to create constraints to force the sum of all diagonals to be at most 1.

The finished model can be seen in figure 4.8.

If we then fix the variable representing n to any value between 1 and 14, the solution will be a valid solution for the respective problem. In figure 4.8, n was set to 4. The resulting solution is a solution for the 4-queens problem. If we try other values, we quickly find out that there is a solution for $n = 1$, there is no solution for $n = 2$ and $n = 3$ and there is a solution for any n above 3. All configurations from $n = 1$ to $n = 14$ were solved in less than a second on our machine.

In theory it would be possible to create a model for instances of any maximal size. It is however the case that the framework gets unpractical for large models with many

Figure 4.8: A solver that solves the n-queens problem up to 14 can be created within our framework.

H 1 1 1	H 2 1 1	H 3 1 1	H 4 1 1	H 5 1 0	H 6 1 0	H 7 1 0	H 8 1 0	H 9 1 0	H 10 1 0	H 11 1 0	H 12 1 0	H 13 1 0	H 14 1 0		
Q 1 1 0	Q 2 1 1	Q 3 1 0	Q 4 1 0	Q 5 1 0	Q 6 1 0	Q 7 1 0	Q 8 1 0	Q 9 1 0	Q 10 1 0	Q 11 1 0	Q 12 1 0	Q 13 1 0	Q 14 1 0	V 1 1 1	N 1 1 4
Q 1 2 0	Q 2 2 0	Q 3 2 0	Q 4 2 1	Q 5 2 0	Q 6 2 0	Q 7 2 0	Q 8 2 0	Q 9 2 0	Q 10 2 0	Q 11 2 0	Q 12 2 0	Q 13 2 0	Q 14 2 0	V 1 2 1	
Q 1 3 1	Q 2 3 0	Q 3 3 0	Q 4 3 0	Q 5 3 0	Q 6 3 0	Q 7 3 0	Q 8 3 0	Q 9 3 0	Q 10 3 0	Q 11 3 0	Q 12 3 0	Q 13 3 0	Q 14 3 0	V 1 3 1	
Q 1 4 0	Q 2 4 0	Q 3 4 1	Q 4 4 0	Q 5 4 0	Q 6 4 0	Q 7 4 0	Q 8 4 0	Q 9 4 0	Q 10 4 0	Q 11 4 0	Q 12 4 0	Q 13 4 0	Q 14 4 0	V 1 4 1	
Q 1 5 0	Q 2 5 0	Q 3 5 0	Q 4 5 0	Q 5 5 0	Q 6 5 0	Q 7 5 0	Q 8 5 0	Q 9 5 0	Q 10 5 0	Q 11 5 0	Q 12 5 0	Q 13 5 0	Q 14 5 0	V 1 5 0	
Q 1 6 0	Q 2 6 0	Q 3 6 0	Q 4 6 0	Q 5 6 0	Q 6 6 0	Q 7 6 0	Q 8 6 0	Q 9 6 0	Q 10 6 0	Q 11 6 0	Q 12 6 0	Q 13 6 0	Q 14 6 0	V 1 6 0	
Q 1 7 0	Q 2 7 0	Q 3 7 0	Q 4 7 0	Q 5 7 0	Q 6 7 0	Q 7 7 0	Q 8 7 0	Q 9 7 0	Q 10 7 0	Q 11 7 0	Q 12 7 0	Q 13 7 0	Q 14 7 0	V 1 7 0	
Q 1 8 0	Q 2 8 0	Q 3 8 0	Q 4 8 0	Q 5 8 0	Q 6 8 0	Q 7 8 0	Q 8 8 0	Q 9 8 0	Q 10 8 0	Q 11 8 0	Q 12 8 0	Q 13 8 0	Q 14 8 0	V 1 8 0	
Q 1 9 0	Q 2 9 0	Q 3 9 0	Q 4 9 0	Q 5 9 0	Q 6 9 0	Q 7 9 0	Q 8 9 0	Q 9 9 0	Q 10 9 0	Q 11 9 0	Q 12 9 0	Q 13 9 0	Q 14 9 0	V 1 9 0	
Q 1 10 0	Q 2 10 0	Q 3 10 0	Q 4 10 0	Q 5 10 0	Q 6 10 0	Q 7 10 0	Q 8 10 0	Q 9 10 0	Q 10 10 0	Q 11 10 0	Q 12 10 0	Q 13 10 0	Q 14 10 0	V 1 10 0	
Q 1 11 0	Q 2 11 0	Q 3 11 0	Q 4 11 0	Q 5 11 0	Q 6 11 0	Q 7 11 0	Q 8 11 0	Q 9 11 0	Q 10 11 0	Q 11 11 0	Q 12 11 0	Q 13 11 0	Q 14 11 0	V 1 11 0	
Q 1 12 0	Q 2 12 0	Q 3 12 0	Q 4 12 0	Q 5 12 0	Q 6 12 0	Q 7 12 0	Q 8 12 0	Q 9 12 0	Q 10 12 0	Q 11 12 0	Q 12 12 0	Q 13 12 0	Q 14 12 0	V 1 12 0	
Q 1 13 0	Q 2 13 0	Q 3 13 0	Q 4 13 0	Q 5 13 0	Q 6 13 0	Q 7 13 0	Q 8 13 0	Q 9 13 0	Q 10 13 0	Q 11 13 0	Q 12 13 0	Q 13 13 0	Q 14 13 0	V 1 13 0	
Q 1 14 0	Q 2 14 0	Q 3 14 0	Q 4 14 0	Q 5 14 0	Q 6 14 0	Q 7 14 0	Q 8 14 0	Q 9 14 0	Q 10 14 0	Q 11 14 0	Q 12 14 0	Q 13 14 0	Q 14 14 0	V 1 14 0	

constraints. Here, we observed that our framework shares a disadvantage that can be found in many graphical programming frameworks. As the complexity of the problem rises, the more is screen space an issue. While n-queens for n up to 14 was no problem on our screen, instances up to 100 could only be solved with a lot of scrolling.

4.10.2 Rotating Workforce Scheduling

To create a model that can solve the problem with a variable number of workers, we again first have to fix the biggest instance we want to solve. We state that we want to solve instances with a planning horizon of 7 days and $1, ..., 14$ workers. We thus create $7 * 14$ variables called $W_{1,1}, ..., W_{7,14}$ with domain $\{0, ..., 3\}$. As before, each variable $W_{i,j}$ represents the type of a shift, 0 stands for a free day and $1, 2, 3$ for the day, evening and night shift.

Additionally, we create a variable for each worker that represents if the worker is active. These variables will change depending on the number of workers of the current instance.

14 variables called $A_{1,1}, ..., A_{1,14}$ with domain $\{0, ..., 1\}$ are thus created. We create constraints that force the values of each variable $A_{1,j}$ to be bigger than the value of $A_{1,j+1}$. This means, the active workers will be the ones that appear first in the grid $W_{1,1}, ..., W_{7,14}$.

To connect the variables representing if a worker is active with the actual assignment, we create another 14 variables that count the number of zeros (days off) that appear in the schedule of a specific worker. We will call those counting zero variables $CZ_{1,1}, ..., CZ_{1,14}$. We then create a 'forbidden pattern constraint for each pair $(A_{1,i}, CZ_{1,i})$. The content of the constraint is:

```
0 ~7
```

This means that if the worker is inactive (0), the count of zeros (days off) must be exactly 7. We create another single variable called $TW_{1,1}$ that counts the number of active workers. This variable can then be fixed to specify the instance. If it is for example set to 9, exactly 9 workers from top to bottom will be active and all other workers will have all shift assignments set to 0.

We then create a grid of $7 * 3$ variables that represent how often a shift has to occur each day. The 'Number of occurrences' constraint is used to achieve this. The variables can then conveniently be fixed using our framework.

To configure a specific instance of the Rotating Workforce Scheduling Problem, the procedure is the following. First the variable $TW_{1,1}$ is fixed to the number of workers of the instance. Then, the variables representing how often a shift has to appear each day are fixed according to the instance.

The last step is to create a 'Forbidden pattern' constraint selecting all active workers. The content of this constraint will be dependent on the instance specification. For example, one could enter

```
3 2
3 1
2 1
```

To forbid some pattern of shifts to occur. The final model that is already configured for instance 1[7] can be seen in figure 4.9.

4.11 Finding the Right Solver

Finding the right solver for a model is not an easy task. There is even a challenge whose aim it is to compare different MiniZinc solvers with respect to various problem [SBF10].

While using our framework, we have observed that using the option 'All solvers in parallel' is a sensible choice. The total runtime of running 4 solvers in parallel until the first solver

[7] http://www.dbai.tuwien.ac.at/staff/musliu/benchmarks/ accessed on 20.10.2017

Figure 4.9: A model for the Rotating Workforce Scheduling Problem that is configurable for different instances.

W 1 1 1	W 2 1 1	W 3 1 1	W 4 1 1	W 5 1 1	W 6 1 1	W 7 1 1		A 1 1 1	CZ 1 1 0
W 1 2 0	W 2 2 0	W 3 2 3	W 4 2 3	W 5 2 3	W 6 2 3	W 7 2 0		A 1 2 1	CZ 1 2 3
W 1 3 0	W 2 3 2	W 3 3 2	W 4 3 2	W 5 3 2	W 6 3 2	W 7 3 0		A 1 3 1	CZ 1 3 2
W 1 4 0	W 2 4 2	W 3 4 2	W 4 4 2	W 5 4 3	W 6 4 3	W 7 4 3		A 1 4 1	CZ 1 4 1
W 1 5 3	W 2 5 0	W 3 5 0	W 4 5 1	W 5 5 1	W 6 5 1	W 7 5 2		A 1 5 1	CZ 1 5 2
W 1 6 2	W 2 6 0	W 3 6 0	W 4 6 2	W 5 6 2	W 6 6 2	W 7 6 3		A 1 6 1	CZ 1 6 2
W 1 7 3	W 2 7 3	W 3 7 0	W 4 7 0	W 5 7 2	W 6 7 2	W 7 7 2		A 1 7 1	CZ 1 7 2
W 1 8 2	W 2 8 3	W 3 8 3	W 4 8 3	W 5 8 0	W 6 8 0	W 7 8 1		A 1 8 1	CZ 1 8 2
W 1 9 1	W 2 9 1	W 3 9 1	W 4 9 0	W 5 9 0	W 6 9 0	W 7 9 0	+	A 1 9 1	CZ 1 9 4
W 1 10 0	W 2 10 0	W 3 10 0	W 4 10 0	W 5 10 0	W 6 10 0	W 7 10 0		A 1 10 0	CZ 1 10 7
W 1 11 0	W 2 11 0	W 3 11 0	W 4 11 0	W 5 11 0	W 6 11 0	W 7 11 0		A 1 11 0	CZ 1 11 7
W 1 12 0	W 2 12 0	W 3 12 0	W 4 12 0	W 5 12 0	W 6 12 0	W 7 12 0		A 1 12 0	CZ 1 12 7
W 1 13 0	W 2 13 0	W 3 13 0	W 4 13 0	W 5 13 0	W 6 13 0	W 7 13 0		A 1 13 0	CZ 1 13 7
W 1 14 0	W 2 14 0	W 3 14 0	W 4 14 0	W 5 14 0	W 6 14 0	W 7 14 0		A 1 14 0	CZ 1 14 7

RQ 1 1 2	RQ 2 1 2	RQ 3 1 2	RQ 4 1 2	RQ 5 1 2	RQ 6 1 2	RQ 7 1 2		TW 1 1 9
RQ 1 2 2	RQ 2 2 2	RQ 3 2 2	RQ 4 2 3	RQ 5 2 3	RQ 6 2 3	RQ 7 2 2		
RQ 1 3 2	RQ 2 3 2	RQ 3 3 2	RQ 4 3 2	RQ 5 3 2	RQ 6 3 2	RQ 7 3 2		

finishes is on average approximately 2 times the runtime of just the fastest solver. This overhead is in our opinion a good trade-off to avoid cases as good as possible where a solver does not find a solution at all.

We did not do further research on this topic, as solver selection is in principle a problem that is independent from our framework.

4.12 (Un-)Suitable Classes of Problems

In the previous section we have seen a number of problems for which the framework is suitable for. In some cases, our framework provided an intuitive way of graphically modeling the problem instance.

All those problems that our framework was well suited for have the following properties in common:

1. There is a natural tabular or grid-like problem representation.
2. The total number of different constraints is as low as possible. This means, that for a problem that requires a lot of different constraints, the graphical framework we proposed quickly gets visually cluttered. In all above examples, we used a maximum of 10 different types of constraints (but those can occur more than once).
3. The constraints have some symmetry regarding the grid representation. This means, one can save time by duplicating the constraints instead of creating all constraints individually.
4. The constraints that are required to model the problem are 'low level', meaning that they are in the set of commonly used CP constraints [RVBW06].
5. The total number of variables of the instance is relatively small (<100). This allows the model to fit on a average computer screen without scrolling.
6. The constraints are of low complexity, meaning that there is no combination of different constraints. An example of a constraint that would be difficult to model with our framework would be the following: 'If the variables $A_1, ..., A_i$ are all different, then the variables $B_1, ..., B_j$ must be all at least as big as C_j.
7. The domain of the variables is fixed, meaning for a specific instance, one can determine the minimum and maximum possible value for each variable in beforehand.

A concrete example, for which our framework will be unsuitable is the 'Low Autocorrelation Binary Sequences' Problem [Pre00]. In this problem, the goal is to find a binary sequence that minimizes a given formula.

While it is possible to model this problem in our framework using the 'Custom constraint', there is no advantage in comparison to just programming plain MiniZinc Syntax. The

model for this problem consists of just 3 lines if formulated concisely[8], meaning the approach of specifying the model in a written way is likely to be the most efficient one.

[8] http://www.csplib.org/Problems/prob005/models/LowAutocorrelationBinarySequences.essence.html accessed on 22.10.2017

CHAPTER 5

Conclusion

In this thesis a new graphical environment for creating constraint programming models has been proposed.

The main goal of this environment is to allow users to create models of some problems without having to know a written constraint programming language. For this, we present a grid-like approach to arrange variables and constraints in the 2D space. The variables and constraints are created using common graphical user interface elements such as selecting elements with a mouse and the context menu. Various features are included to make the modeling process as simple as possible. We have found a way to model common subproblems such as routes in a simple graphical way.

In the evaluation we have found out that this environment can be successfully used to model a variety of common constraint programming problems such as the rotating workforce scheduling problem, the social golfer problem and the traveling salesperson problem with time windows. In some problem instances, we achieved runtimes that are comparable to existing methods.

In general, the framework is more suitable to model fixed instances of problems, as the grid itself is static. It was however also possible to create models that can be configured to solve multiple different instances of problems without having to recreate the complete model. One of those problems is the n-queens problem.

As with other graphical programming environments, we have noticed that with increasing complexity of the problem, screen space becomes an issue. On a average sized (24 inch) computer monitor, approximately 375 variables can be displayed at once. If more variables are used, parts of the model can be accessed by scrolling.

We have come to the conclusion that the framework is most suitable for modeling small instances with low complexity, as such instances can be presented on a computer screen without scrolling. Furthermore, smaller models can be solved without advanced

refinements like symmetry breaking of the model. We therefore suggest that our framework could be used for teaching and learning purposes to demonstrate the general idea of constraint programming with simple examples. We can also see the framework being beneficial for sketching and debugging purposes. It could make sense to use the framework to quickly model different constraint programming formulations, as one of the aims of our environment is to allow fast and simple modeling.

Future work on this topic could be experimental testing with students to see if the framework can be a benefit in teaching the topic of constraint programming. It would also be interesting to see if further refinements of our graphical environment, like additional high level constraints, could make even more problems solvable within our framework. It also would be interesting to see if automatic symmetry breaking methods could be included in our framework to make the solving process faster.

List of Figures

3.1 The solution representation of a scheduling problem. Recreated from the paper: [PWW69] . . . 12
3.2 The main window of our implementation of the graphical constraint programming framework. . . . 13
3.3 The context menu allows creation of variables and constraints. . . . 15
3.4 Some variables that were created in the grid. . . . 16
3.5 The All-Different constraint (first row, green). . . . 18
3.6 The forbidden pattern constraint can be used for defining undesired patterns in sequences. . . . 22
3.7 The route constraint allows modeling of various routing and distance problems. 27
3.8 Cells containing variables that are set as optimization goal are highlighted in color. . . . 29
3.9 The code view can be used to examine the generated code. . . . 32
3.10 The number of cells and the width and height of the selected rectangle are displayed. . . . 32
3.11 The value of a variable in the solution is displayed in the corresponding cell. 34

4.1 Sudoku problems can be modeled within our framework. In red: variables fixed by us. . . . 37
4.2 The solved model shows one solution for the 8-queens problem. . . . 39
4.3 The model and solution for the (20, 4, 5) social golfer problem. . . . 41
4.4 The model and solution for the instance 8 of the rotating workforce scheduling problem. . . . 45
4.5 The Travelling Salesperson with Time Windows Problem can be modeled with our framework. . . . 49
4.6 An instance of the Traveling Tournament Problem modeled within our framework. . . . 52
4.7 Simple Teacher Scheduling can be modeled and solved within our framework. 54
4.8 A solver that solves the n-queens problem up to 14 can be created within our framework. . . . 57
4.9 A model for the Rotating Workforce Scheduling Problem that is configurable for different instances. . . . 59

Bibliography

[AB93] Allen L. Ambler and Margaret M. Burnett. Visual programming languages from an object-oriented perspective. *ACM SIGPLAN OOPS Messenger*, 4(2):225, April 1993.

[Aze07] Francisco Azevedo. An Attempt to Dynamically Break Symmetries in the Social Golfers Problem. In *Recent Advances in Constraints*, volume 4651, pages 33–47. Springer Berlin Heidelberg, 2007.

[BA94] Margaret M. Burnett and Allen L. Ambler. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. *Journal of Visual Languages & Computing*, 5(1):29–60, March 1994.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, pages 555–564, 1999.

[BAWD+01] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A First-order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *J. Funct. Program.*, 11(2):155–206, March 2001.

[BS09] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, January 2009.

[CACM08] Broderick Crawford, Mary Aranda, Carlos Castro, and Eric Monfroy. Using Constraint Programming to solve Sudoku Puzzles. pages 926–931. IEEE, November 2008.

[CW01] Peter Chan and Georges Weil. Cyclical Staff Scheduling Using Constraint Logic Programming. In *Practice and Theory of Automated Timetabling III*, volume 2079, pages 159–175. Springer Berlin Heidelberg, 2001.

[Dec03] Rina Dechter. *Constraint processing.* Morgan Kaufmann Publishers, 2003.

[EGCT13] Stefan Edelkamp, Max Gath, Tristan Cazenave, and Fabien Teytaud. Algorithm and knowledge engineering for the TSPTW problem. pages 44–51. IEEE, April 2013.

[EM17] Christoph Erkinger and Nysret Musliu. Personnel Scheduling as Satisfiability Modulo Theories. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 614–621, 2017.

[ENT01] Kelly Easton, George Nemhauser, and Michael Trick. The Traveling Tournament Problem Description and Benchmarks. In *Principles and Practice of Constraint Programming — CP 2001*, volume 2239, pages 580–584. Springer Berlin Heidelberg, 2001.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of constraint programming.* Springer Science & Business Media, 2003.

[FFH+01] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kzltan, Ian Miguel, and Toby Walsh. Matrix Modelling. In *Proc. of the CP-01 Workshop on Modelling and Problem Formulation*, 2001.

[FSC04] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: A Generic Graphical User Interface for Constraint Logic Programming. *Constraints*, 9(4):241–262, October 2004.

[GS00] 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.

[GW99] Ian P. Gent and Toby Walsh. CSPlib: A Benchmark Library for Constraints. In *Principles and Practice of Constraint Programming – CP'99*, volume 1713, pages 480–481. Springer Berlin Heidelberg, 1999.

[GW16] Marc Goerigk and Stephan Westphal. A combined local search and integer programming approach to the traveling tournament problem. *Annals of Operations Research*, 239(1):343–354, April 2016.

[Har01] Warwick Harvey. Symmetry breaking and the social golfer problem. *Proceedings SymCon-01: Symmetry in Constraints, co-located with CP*, pages 9–16, 2001.

[Hil92] Daniel D Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, March 1992.

[Hoe01] W. J. van Hoeve. The AllDifferent Constraint: a Survey. In *Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

[HOO03] Alan Holland, Barry O'Callaghan, and Barry O'Sullivan. A Constraint-Aided Conceptual Design Environment for Autodesk Inventor. In *Principles and Practice of Constraint Programming – CP 2003*, volume 2833, pages 422–436. Springer Berlin Heidelberg, 2003.

[HPS05] Brahim Hnich, P Posser, and Barbara Smith. Modelling and Reformulating Constraints Satisfaction Problem. In *Proceedings of the 4th International Workshop Sitges (Barcelona), Spain*, 2005.

[KCC+02] Caitlin Kelleher, Dennis Cosgrove, David Culyba, Clifton Forlines, Jason Pratt, and Randy Pausch. Alice2: programming without syntax errors. In *User Interface Software and Technology*, 2002.

[Lew07] Rhydian Lewis. On the Combination of Constraint Programming and Stochastic Search: The Sudoku Case. In *Hybrid Metaheuristics*, volume 4771, pages 96–107. Springer Berlin Heidelberg, 2007.

[LH11] Jakob Leitner and Michael Haller. Harpoon selection: efficient selections for ungrouped content on large pen-based surfaces. page 593. ACM Press, 2011.

[LP04] Gilbert Laporte and Gilles Pesant. A general multi-shift scheduling system. *JORS*, 55(11):1208–1217, 2004.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, February 1977.

[MG06] Todd Moon and Jacob Gunther. Multiple Constraint Satisfaction by Belief Propagation: An Example Using Sudoku. pages 122–126. IEEE, July 2006.

[MGS02] Nysret Musliu, Johannes Gärtner, and Wolfgang Slany. Efficient generation of rotating workforce schedules. *Discrete Applied Mathematics*, 118(1-2):85–98, 2002.

[Mus05] Nysret Musliu. Combination of Local Search Strategies for Rotating Workforce Scheduling Problem. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1529–1530, 2005.

[MW17] Nysret Musliu and Felix Winter. A Hybrid Approach for the Sudoku Problem: Using Constraint Programming in Iterated Local Search. *IEEE Intelligent Systems*, 32(2):52–62, March 2017.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming – CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings*, pages 529–543. Springer Berlin Heidelberg, 2007.

[Pre00] Steven Prestwich. A Hybrid Search Architecture Applied to Hard Random 3-SAT and Low-Autocorrelation Binary Sequences. In *Principles and*

Practice of Constraint Programming – CP 2000, volume 1894, pages 337–352. Springer Berlin Heidelberg, 2000.

[PWW69] A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. Multiproject Scheduling with Limited Resources: A Zero-One Programming Approach. *Management Science*, 16(1):93–108, September 1969.

[RM05] Arathi Ramani and Igor L. Markov. Automatically Exploiting Symmetries in Constraint Programming. In *Recent Advances in Constraints*, volume 3419, pages 98–112. Springer Berlin Heidelberg, 2005.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Prentice Hall series in artificial intelligence. Prentice Hall, 3rd ed edition, 2010.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh, editors. *Handbook of constraint programming.* Foundations of artificial intelligence. Elsevier, 1st ed edition, 2006.

[SBF10] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, July 2010.

[Sim05] Helmut Simonis. Sudoku as a Constraint Problem. *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems, 2005*, 2005.

[SLRGVC16] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using “Scratch” in five schools. *Computers & Education*, 97:129–141, June 2016.

[SP98] V. Solimene and R. Provencal. *Menu control in a graphical user interface.* Google Patents, October 1998.

[TM11] Markus Triska and Nysret Musliu. A Constraint Programming Application for Rotating Workforce Scheduling. In *Developing Concepts in Applied Intelligence*, volume 363, pages 83–88. Springer Berlin Heidelberg, 2011.

[Vie15] Nelson Manuel Marques Vieira. *Graphical constraints: a graphical user interface for constraint problems, University of Madeira.* University of Madeira, 2015.

[Wei17] Eric W. Weisstein. Magic Hexagon. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/MagicHexagon.html, 2017.