Encoding Diverse Sudoku Variants as SAT Problems

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artifical Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Augusto B. Corrêa

Sebastian Schlachter
sebastian.schlachter@unibas.ch
2017-927-534

20.07.2022

Acknowledgments

I want to thank everyone who accompanied me during the writing of this thesis, especially my supervisor Augusto B. Corrêa, which always supported me with honest feedback and an open ear regarding questions or concerns. Further, I would like to sincerely thank Prof. Dr. Malte Helmert for giving me the opportunity to write this thesis on a subject that turned out to be incredibly engaging and joyful to work on.

Abstract

Sudoku has become one of the world's most popular logic puzzles, arousing interest in the general public and the science community. Although the rules of Sudoku may seem simple, they allow for nearly countless puzzle instances, some of which are very hard to solve. SAT-solvers have proven to be a suitable option to solve Sudokus automatically. However, they demand the puzzles to be encoded as logical formulae in Conjunctive Normal Form. In earlier work, such encodings have been successfully demonstrated for original Sudoku Puzzles. In this thesis, we present encodings for rather unconventional Sudoku Variants, developed by the puzzle community to create even more challenging solving experiences. Furthermore, we demonstrate how Pseudo-Boolean Constraints can be utilized to encode Sudoku Variants that follow rules involving sums. To implement an encoding of Pseudo-Boolean Constraints, we use Binary Decision Diagrams and Adder Networks and study how they compare to each other.

Table of Contents

1 Introduction

Sudoku Puzzles excite, as the rather simple task of filling out a grid with numbers becomes a demanding challenge just by stating a small set of rules that must be followed. For a 9×9 grid of cells which may already contain numbers, the original Sudoku rules every puzzle solver must follow are: A number with a value from 1 to 9 must be placed in each cell, and every number may appear only once per column, row and marked 3×3 box. An example of a normal Sudoku Puzzle and its solution is shown in Figures 1.1 and 1.2.

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

Figure 1.1: Normal Sudoku Puzzle [10]

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

Figure 1.2: Solution to Puzzle in Figure 1.1

Using these rules, the puzzle in its original form was first seen in the early 80s and established itself as one of the most popular logic puzzles in the last few decades. Since the mid-2000s, Sudokus also became an inherent part of the puzzle section in many newspapers and gained a fan community of puzzlers eager to develop and solve more challenging riddles. Around this time, researchers also started publishing first papers analysing Sudokus. For example, in 2006, it was shown by [6] that there are 6.671×10^{21} valid 9×9 Sudoku grids. In the same year, Lynce and Ouaknine published a paper [13] showing how Sudoku Puzzles can be encoded into logical formulas in a way that SAT-solvers can be used to find their solutions. SAT-solvers are suitable for solving Sudokus because most of the puzzles are "well-formed", which means they only have one solution that is deducible without ambitions. Or, as [13] states, "Such puzzles are meant to be solved without search, i.e., merely with reasoning."

In this thesis, we aim to go one step further and find encoding methods for Sudoku Variants that are based on original Sudoku but augment the rules with additional requirements that solutions must fulfil. Our source of choice for such rather unconventional Sudoku Variants and rules will be the book "Cracking The Cryptic Greatest Hits" (*CTCGH*) [10], which presents a collection of the most unique, entertaining but also demanding Sudoku Variants the puzzle community came up with until today. The book was published after a successful Kickstarter campaign by Mark Goodliffe and Simon Anthon, which run one of the most famous Youtube channels focused on Sudokus: "Cracking The Cryptic"[8]. In their videos, they present and solve puzzles sent to them from people all around the world, allowing them to create this phenomenal assortment of diverse Sudoku Variants.

To get a foretaste of the variants we will work with, one may solve the Sudoku depicted in Figure 1.3. The normal Sudoku rules apply, but instead of already filled out cells, seven so-called *Thermometers* are placed on the grid. For each thermometer, it must hold that starting from the bulb, the cell values along the thermometer can only strictly increase. As one will see, this already suffices to guarantee the uniqueness of the solution.

As we intend to make the encodings of different rules compatible with each other, it will be possible to encode and solve new Sudoku Variants formed by arbitrarily combining said rules. To test our encodings, though, we will use puzzle instances from CTCGH and state of the art SAT-solvers like MiniSat. Comparing the encodings and the performance of the SAT-solvers working on them, we will show that for most puzzle instances from CTCGH, SAT-solvers can find a solution within seconds. However, we will also find exceptions to this, with Sudoku Variants that require extensive encodings and comparably high solving times by the SAT-solvers, revealing that the task of encoding and solving these special Sudoku Variants is by no means a trivial one.

Because we will examine many Sudoku Variants with rules involving sums, we also want to further elaborate on the encoding of Pseudo-Boolean Constraints. Following the ideas of [5], we will show how Binary Decision Diagrams and Adder Networks can be used to encode Pseudo-Boolean Constraints and how the two methods compare to each other.

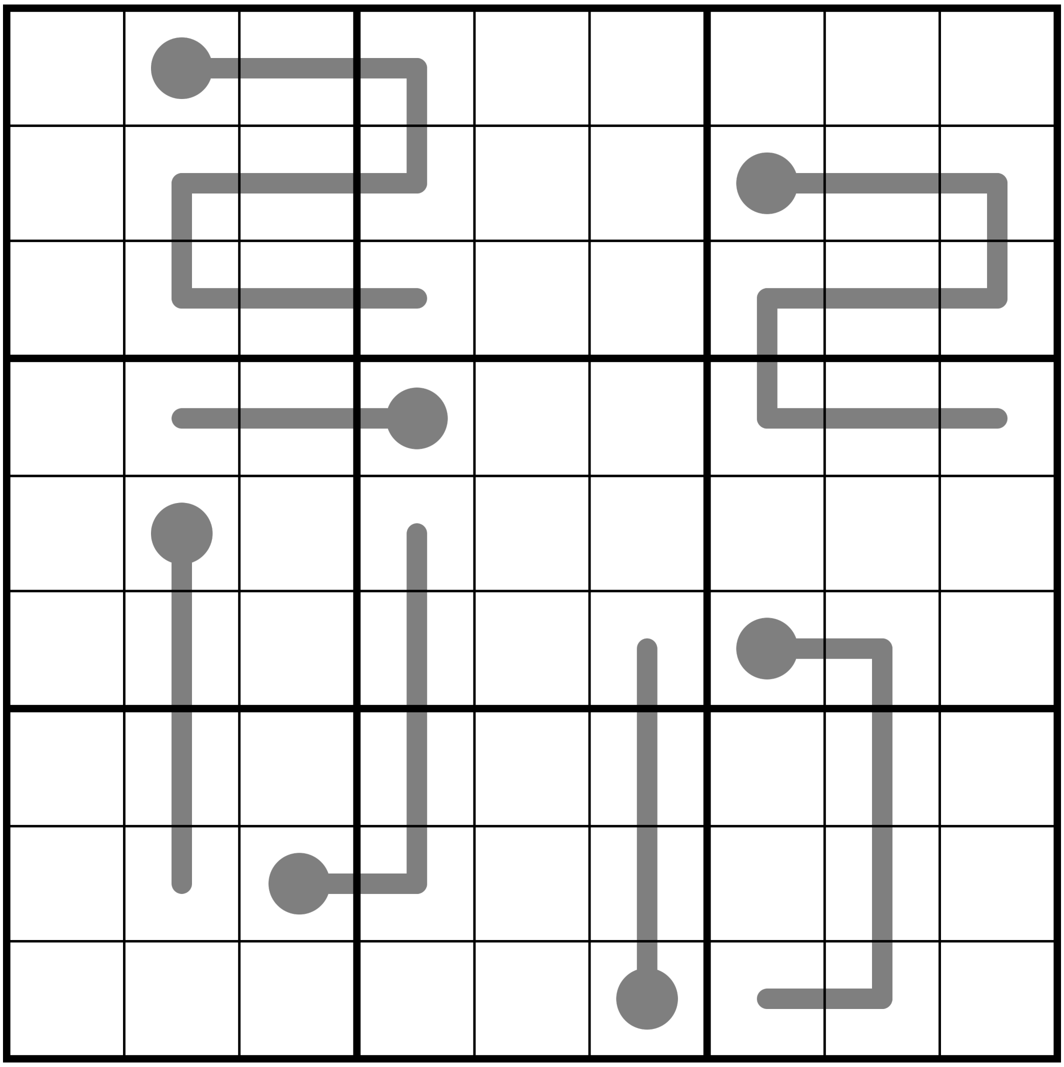


Figure 1.3: Example of a Sudoku Puzzle using Thermometers,
puzzle by Akash Doulani, CTCGH page 15 [10],
this is a corrected version, available at [9]

2 Background

Before we dive into how we can translate Sudokus into a language such that a computer can work on them, we first elaborate on some background knowledge and definitions needed to understand the used tools and formalisms.

2.1 Propositional Logic

Propositional logic is a language that provides a formal way of writing statements that can either be true or false. It is used in this thesis to describe and encode the specific rules of the different Sudoku Variants. This section will shortly introduce the common syntax and semantics.

Atoms (also called *atomic propositions*) are the smallest units used in propositional logic, and must have a truth value of true or false (often noted as digits 0 and 1).

Literals are atoms or their negation, so if x_1 is an atom, then x_1 and $\neg x_1$ are literals. Literals are said to be *positive* or *negative* respectively.

Formulas are compositions of one or multiple atoms and can be defined recursively:
Every atom is also a formula. If φ is a formula, then so is its negation $\neg\varphi$. If φ and ψ are formulas, then so is the conjunction $\varphi \wedge \psi$. If φ and ψ are formulas, then so is the disjunction $\varphi \vee \psi$.

Interpretations (also called truth assignments) are functions that assigns truth values to a set A of atoms $\mathcal{I} : A \to \{0, 1\}$. A formula φ over A holds (is true) under an interpretation $\mathcal{I}$ (written $\mathcal{I} \models \varphi$) following the semantical rules:

$$\begin{array}{lll} \mathcal{I} \models x_1 & \text{iff} & \mathcal{I}(x_1) = 1 \\ \mathcal{I} \models \neg x_1 & \text{iff} & \text{not } \mathcal{I} \models x_1 \\ \mathcal{I} \models (\psi \wedge \varrho) & \text{iff} & \mathcal{I} \models \psi \text{ and } \mathcal{I} \models \varrho \\ \mathcal{I} \models (\psi \vee \varrho) & \text{iff} & \mathcal{I} \models \psi \text{ or } \mathcal{I} \models \varrho \end{array}$$

Where ψ and ϱ are formulas and x_1 is an atom.

An interpretation for which a formula φ hold is called a *model* of φ.

Equivalence of two formulas φ and ψ is given if it holds for all interpretations $\mathcal{I}$ that, φ holds under $\mathcal{I}$ if and only if ψ holds under $\mathcal{I}$. The formulas are then called *logically equivalent* ($\varphi \equiv \psi$).

Implications / Biconditionals As one might have noticed, implication ($\rightarrow$) and biconditional ($\leftrightarrow$) have not been mentioned in the definition of formulas as they are abbreviations for more extended formulas that use $\vee$, $\wedge$ and $\neg$.

$$\begin{aligned} (\varphi \rightarrow \psi) &\equiv (\neg\varphi \vee \psi) \\ (\varphi \leftrightarrow \psi) &\equiv (\neg\varphi \vee \psi) \wedge (\neg\psi \vee \varphi) \end{aligned}$$

Clauses are disjunctions of literals (atoms and/or their negations). A formula that is a clause is true under interpretation $\mathcal{I}$ if one of its literals is true.

Conjunctive Normal Form (CNF) A formula is said to be in conjunctive normal form if it is a conjunction of clauses. For example, given the atoms a, b, c and d, the formulas φ, ψ and ϱ are in CNF:

$$\begin{aligned} \varphi &\equiv (a) \\ \psi &\equiv (a \wedge b) \\ \varrho &\equiv ((a \vee b) \wedge (c \vee d)) \end{aligned}$$

Also, it holds that every formula can be brought into CNF [19], and a formula in CNF can be noted as a set of sets of literals, for example, $\varrho \equiv \{\{a, b\}, \{c, d\}\}$.

2.2 SAT-Problems and SAT-Solvers

SAT-Problems (also called *Satisfiability Problems* or *Boolean Satisfiability Problems*) describe the problem of deciding if a formula of propositional logic is satisfiable (if there exists a model for it) or not. SAT-solvers are programs or algorithms that try to solve instances of this problem. They take a formula as input and return a boolean value (true or false) to indicate if a model exists or not. Most SAT-solvers also directly provide a model if they can find one. In the experiments of this thesis, multiple SAT-solvers are used, which are described in further detail in 2.2.3. As we will later see, the time to find solutions for particular problem instances varies between them. However, when it comes to computational complexity, it holds that SAT-Problems are in NP and that they are NP-Hard[1][12], so the runtime of the solvers may scale exponentially with the number of clauses and variables given to them as input.

2.2.1 DIMACS CNF File Format

The used SAT-solvers require the input formula to be in CNF. Further, they expect them to be described in the DIMACS CNF File Format, often just called DIMACS. The abbreviation DIMACS stands for Center for “Discrete Mathematics and Theoretical Computer Science”, which is a collaboration between Rutgers and Princeton University and research firms. The file format was utilised in the DIMACS Implementation Challenge 1993 and since then has become the common file format for SAT-Problems.

DIMACS CNF Files have the following Format:

- Atoms are represented as positive integers.
- Negative literals are represented as negative integers.
- The first line starts with the letter "p" and holds the problem description. It states the problem type, the highest integer used to describe an atom and the number of clauses.
- Clauses are represented as lists of their literals and are terminated by the number 0. White spaces or line breaks separate all literals and the ending 0s.
- Lines are interpreted as comments (ignored by the solver) if they start with the letter "c". Comments can be added everywhere in the file except inside the definition of a clause.

A DIMACS file describing the formula $\varphi \equiv (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ could look like this:

```
c some comment describing the problem
p cnf 3 4
 1 -2  0
 2  3  0
-1 -3  0
-1 -2  3  0
```

2.2.2 How SAT-Solvers solve

Given a formula, a SAT-solver must decide if it is satisfiable or not (We assume the formula is in CNF so it can be handled as a set of clauses). To do so, the solver tries to find a model by assigning truth values to atoms one by one. In a standard SAT-solver, this is not done randomly but by using the inference mechanism of *unit propagation*. A famous algorithm using this mechanism is the DPLL algorithm [2] which can be directly implemented into a SAT-solver. DPLL applies three rules:

1. If a clause contains only one literal (or multiple literals, but only one is still unbound and all the others do not make the clause true), the value that makes the literal true must be assigned. Otherwise, the clause and the formula would become false. If a value gets assigned, all clauses containing the corresponding true literal can be removed from the set of clauses. The corresponding false literals can be removed from all remaining clauses since they can not make them true.

2. If there is a literal that only appears in one polarity in all clauses (consistently positive/negative), its atom value can also be assigned to make the literals true.

3. If the set of clauses becomes empty, the formula is "satisfiable". If an empty clause is derived, the formula is "unsatisfiable". If no further assignments can be inferred and no decision can be made about satisfiability yet, a literal gets chosen. The algorithm would then continue in two branches, one where the chosen literal gets set to be true and one where it is set to be false. The formula is then satisfiable if and only if it is satisfiable under one of the two assumptions. Repeating this procedure of assigning values creates a Search-Tree of exponential size.

SAT-solvers can strongly differentiate in how they choose the next atom to assign a value to and how they learn from entering unsatisfiable branches in the search tree.

2.2.3 MiniSat and Sat4j

MiniSat is a lightweight SAT-solver that is based on "the ideas for conflict-driven backtracking, together with watched literals and dynamic variable ordering" as the original paper [4] states. Its original source code in C++ used less than 600 lines. The solver was further refined up to its current version 2.2, but for the experiments in this thesis we use version 1.4 for which precompiled binaries can be found at [3].

Sat4j is a Java library that provides a SAT-solver that can be directly called and run in Java. It does not provide the best performance, but because it is easy to use and integrate, it is well suited for early experimenting and testing. The used version is 2.3.4. The library itself and further details can be found at [18].

MiniSat+ (a particular version of MiniSat) and Sat4j provide native support for Pseudo-Boolean Constraints. However, this feature will not be used because this thesis aims to translate Sudoku Puzzles into a form that can be solved with arbitrary SAT-solvers.

2.3 Constraint Networks (CN)

Puzzles like Sudoku can be broken down into multiple constraints that must all hold for a solution to be correct. Problems like this can be described using constraint networks. The issue of finding a solution to a constraint network is called Constraint Satisfaction Problem or short CSP. Solutions for constraint networks can be found using Backtracking Search ([15], page 175).However, this thesis aims to find such solutions by first encoding the Constraints into SAT-Problems that arbitrary SAT-solvers can then solve. This section will shortly discuss how constraint networks are defined and how they can be translated directly to general SAT-Problems.

Constraint Networks can be described as a tuple of three components: $CN := \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$. $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of finite domains (one corresponding to each variable), and $\mathcal{C}$ is a set of constraints that describe the allowed values for variable subsets.

2.3.1 Unary Constraints

Constraints that only consider one variable are called unary constraints. They do not have to be explicitly written as part of $\mathcal{C}$ because they can also be seen as a domain restriction for a variable that can be represented by reducing the corresponding domain. Examples for a variable x_1 could be: "x_1 must be smaller than 10", "x_1 must be larger than 0", or "x_1 must be even".

2.3.2 Binary Constraints

Constraints that include two variables are called binary constraints. They get defined extensively as part of $\mathcal{C}$ by listing all allowed value pairs that the two variables could take. Examples for CSP variables x_1 and x_2 could be: "x_1 must be smaller than x_2", "x_1 or x_2 must be 9", or "$x_1 + x_2 \leq 12$".

2.3.3 N-ary Constraints

Constraints that include more than two variables are called N-nary constraints and get defined similarly to binary constraints in a CSP. Examples for variables x_1 to x_9 could be: "At most one CSP variable from x_1 to x_9 may be 5" or "The sum of the CSP variables x_1 to x_9 must be 45".

2.3.4 Common Encodings

There are many different ways to transform a CSP into a set of clauses that a SAT-solver can work on, two of the most common ones we want to elaborate on here.

Direct Encoding [20][7] We assign a SAT variable $x_{i,j}$ to all possible variable values j for all CSP variables i. *At-least-one* clauses ensure that a CSP variable i has at least one out of the d values assigned from its domain.

$$x_{i,1} \vee x_{i,2} \vee ... \vee x_{i,d}$$

At-most-one clauses ensure that a CSP variable i has at most one value assigned from its domain.

$$\neg x_{i,j} \vee \neg x_{i,k}$$
$$\forall x_{i,j}, x_{i,k} \in D_i \text{ s.t. } j \neq k$$

Conflict clauses ensure that no CSP variable value combinations are allowed that do not comply with the CSP's constraints. For example, The N-ary constraint "at most one CSP variable of x_1, x_2 and x_3 has value 5" would be transformed into the following conjunction of clauses:

$$(\neg x_{1,5} \vee \neg x_{2,5}) \wedge (\neg x_{1,5} \vee \neg x_{3,5}) \wedge (\neg x_{2,5} \vee \neg x_{3,5}).$$

Assuming the CSP has n variables that have a domain of size d_i, the transformation would in total lead to n at-least-one clauses and per variable to $\sum_{m=1}^{d_i}(m-1)$ at-most-one clauses.

For the constraints, however, we can only give an upper bound for the number of needed clauses per constraint equal to the product of the domain sizes of the contained variables.

Support Encoding [11][7] The same at-least-one and at-most-one clauses are used as in the direct encoding, but instead of conflict clauses, we add so-called support clauses. For all pairs of variables x_i and x_j in a constraint, we add a clause for all domain values of x_j. These clauses define the allowed values of the other variables of the constraint.However, not all these clauses must be necessary. They can be left away if the CSP variable value corresponding to $x_{j,d}$ is allowed with all values of CSP variable x_i. This is best shown in an example, assume all CSP variables have a domain of {1, 2, 3, 4, 5}, the N-ary constraint "at most one CSP variable of x_1, x_2 and x_3 has value 5" could then be transformed to the clauses:

$$\begin{gathered}
\neg x_{1,5} \lor x_{2,1} \lor x_{2,2} \lor x_{2,3} \lor x_{2,4} \\
\neg x_{2,5} \lor x_{1,1} \lor x_{1,2} \lor x_{1,3} \lor x_{1,4} \\
\neg x_{1,5} \lor x_{3,1} \lor x_{3,2} \lor x_{3,3} \lor x_{3,4} \\
\neg x_{3,5} \lor x_{1,1} \lor x_{1,2} \lor x_{1,3} \lor x_{1,4} \\
\neg x_{2,5} \lor x_{3,1} \lor x_{3,2} \lor x_{3,3} \lor x_{3,4} \\
\neg x_{3,5} \lor x_{2,1} \lor x_{2,2} \lor x_{2,3} \lor x_{2,4}
\end{gathered}$$

In this case further clauses like $\neg x_{1,1} \lor x_{2,1} \lor x_{2,2} \lor x_{2,3} \lor x_{2,4} \lor x_{2,5}$ can be added, but they are redundant because if the CSP variable x_1 takes the value 1 the constraint does allow all possible values for x_2. Generally, the support encoding requires more clauses than the direct encoding, but this does not make it inferior or necessarily slower for SAT-solvers [7].

2.3.5 Pseudo-Boolean Constraints (PBCs)

Constraints that describe equations like $w_1x_1 + w_2x_2 + ... + w_nx_n \leq K$ are called Pseudo-Boolean Constraints. The variables x_1 to x_n have boolean domains, the w_is are called weights, and K is called bound (or RHS). Both the weights and the bound must have integer values. If a variable x_i gets assigned a truth value of true/false, it gets valued 1/0 in the equation. PBCs can be transformed into clauses by first converting them to Binary Decision Diagrams (BDD), Adder Networks or Sorter Networks. Within this thesis, we will use the former two variants following the ideas of [5], which we will elaborate on further in section 4.1.

2.4 Sudoku

The family of logic puzzles called Sudoku is based on the Latin Squares Problem. In its current form, it was first published in the US in 1979 by Howard Garns as "Number Place". In 1984 the puzzle became popular in Japan, where it got the name "Sudoku" (translated: "digit-single") which is the name under which it later became famous around the world. A *normal* Sudoku Puzzle consists of a 9 × 9 grid, partially already filled with numbers from 1 to 9 named clues. Additionally, the grid is divided into smaller sub-areas called boxes, consisting of 3 × 3 cells. To solve the puzzle, one has to assign a value from 1 to 9 to each free cell so that each number is present in each row, column, and box. For most Sudoku instances, there is the additional property that they only have one solution. There must be at least 17 clues for this to hold, as McGuire et al. [14] show. An example of a Normal Sudoku Puzzle can be seen in Figure 2.1. For a more accessible annotation, we define a coordinate system on the Sudoku grid, which can be seen in Figure 2.2. With this coordinate system, a grid cell can be specified using a tuple of two numbers (x, y) which is how we reference specific cells from now on. Countless variations and rules can be added to the original Sudoku Puzzle to create new (and eventually harder) solving experiences. The book "Cracking The Cryptic Greatest Hits" [10] contains an extensive collection of Sudoku Variants, a few of which are introduced in the next chapter (3) and further elaborated during this thesis.

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

Figure 2.1: Example of a Sudoku Puzzle with 17 clues by McGuire et al. [14].

(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)
(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)
(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)
(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)
(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)
(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)	(8,6)	(9,6)
(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)	(8,7)	(9,7)
(1,8)	(2,8)	(3,8)	(4,8)	(5,8)	(6,8)	(7,8)	(8,8)	(9,8)
(1,9)	(2,9)	(3,9)	(4,9)	(5,9)	(6,9)	(7,9)	(8,9)	(9,9)

Figure 2.2: Coordinate system for grid cells.

3

Sudoku Variants and Rules

This chapter introduces the most common Sudoku Variants and rules used in CTCGH [10], which are all based on the normal Sudoku rules introduced in Section 2.4. Meaning the normal Sudoku rules still apply, in addition to the rules of the variants introduced in this chapter. The aim here is to give an overview, so the descriptions are on a high level and relatively informal. A formal definition and further details on how the constraints and types are encoded to CNF can be found in chapter 4.

3.1 Killer Sudoku

In a Killer Sudoku, the normal Sudoku rules apply. Additionally, some (not necessarily all) orthogonally adjacent cells are grouped in so-called cages. Each cell can be part of at most one cage. For each cage, two constraints must hold, firstly, the values of all its cells must have a certain sum (*target sum*), and within a cage, each value may only appear once. The cages are often marked on the grid with dashed lines or by colouring their cells, and a small number is placed in the top-left cell of a cage to specify its targeted sum. See Figure 3.1 for an example.

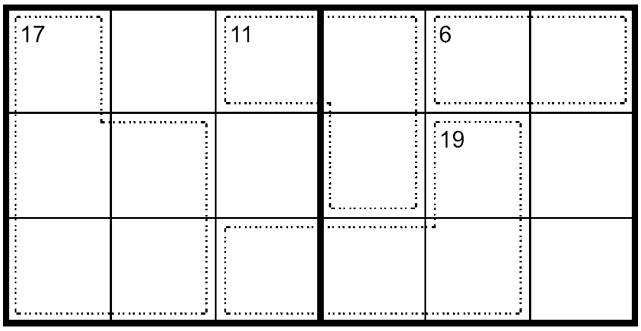


Figure 3.1: Section of a Killer Sudoku showing 4 cages, from CTCGH [10] page 36.

3.2 Thermometers

Thermometers placed on the Sudoku grid connect multiple adjacent cells sequentially (without branching). Each thermometer has a bulb cell marked with a filled circle, and its other cells are marked with a line outgoing from this circle. For a thermometer, it must hold that starting from the bulb, the cell values along the thermometer can only strictly increase. CTCGH [10] also contains a version called “Frozen Thermometers” that allows cell values along a thermometer to stay the same. Additionally, it differs from puzzle to puzzle instance, whether thermometers are allowed to overlap each other or not.
Thermometers are also part of the unique puzzle on page 52 of CTCGH [10]. In this puzzle, only three hint digits and the shape and orientation of 8 thermometers are given. It is then up to the solver to deduce the position of the thermometers, which can not overlap.

3.3 Sandwich Sums

Sandwich Sums are constraints that can be applied to rows, columns or cages. They require certain cells to have a particular sum, which is specified for each row and column at the grid’s edge or inside an affected cage. To fulfill a Sandwich Sum Constraint, it must hold that the sum of all cells lying between the cells containing the values 1 and 9 is equal to the specified target sum. The cells containing 1 and 9 are not included in the sum. Example: a row containing the numbers [2, 4, 8, 9, 6, 5, 1, 7, 9] has a sandwich sum of $6 + 5 = 11$. CTCGH [10] also includes a variation “Sandwich Sums - 1 to X”, where a target sum must be hit, but the second value that defines the sums border is not defined and must be deduced to a value of 2 to 9.

3.4 Secret Direction

The Secret Direction constraints is used in a unique Sudoku Puzzle described on page 41 of CTCGH [10]. Following an adventure’s backstory to find a buried treasure, the constraint requires a solution with a “secret path” from a shaded starting cell to the only cell with value 9, which is also the centre of a 3x3 box. The hidden path must be reconstructed in the following way, starting with the shaded cell; the current cell’s value describes the length of the next step, and the position of the 9 in the current 3x3 box defines the direction. Once a step in the path is found, one must apply the same rules to find the next one until the treasure is discovered.

3.5 Arrowheads

Arrowheads placed on the Sudoku grid specify the partial or total order that must be respected between two adjacent cell values.

3.6 Chess Moves

Some Sudoku Variants use constraints named after chess figures to describe the relationship between two different cells. In CTCGH [10], three of these constraints are introduced, which share the idea that if another cell can be reached from a cell by one chess move of a particular chess figure, these cells may not have the same value. The three constraints differ in the chess figure they use, as listed below.

- Anti-Knight: Cells that are one knight move away from each other may not contain the same value. A knight move consists either of one or two steps in one direction on the current column or row followed by either one or two steps in the then-current row or column. In total, the length must be three steps, and a column and row must be used.
- Anti-King: Cells that are one king move away from each other may not contain the same value. The set of cells that are one king move away from a cell is equal to the set of cells that are directly orthogonally and diagonally adjacent to it.
- Anti-Queen: Cells that are one queen move away from each other may not contain the same value. Cells that are one queen move away from a cell are the cells that lie on the same row, column or diagonal as a cell.

3.7 Fawlty Towers

This Sudoku Variant combines Killer Sudoku with the rules of Thermometers and is presented on page 49 of CTCGH [10]. A tower describes a group of cells which is always based at the lowest row (nr. 9) and rises to a certain height. The different towers are marked by the differently shaded cells. Furthermore, below each tower at the grid's edge stands a number representing its target sum. There are two types of towers, normal and faulty ones. In a normal tower, the cell values are strictly increasing starting from the base and adding all its cell values together results in the target sum. Differently, the numbers are not strictly increasing, or the sums do not match in a faulty tower, but not both at once. As an additional help, it is stated how many towers have non-increasing values and for how many towers the sums will not match. However, it is then up to the solver to deduce which towers belong to which type.

3.8 Nurikabe Sudoku

This Sudoku Variant is the most complex we will be looking at. To solve the Nurikabe Sudoku (page 85 in CTCGH [10]), one not only has to fill in all the numbers but also needs to define for each grid cell if it belongs to an island or the ocean. Islands are composed of multiple orthogonally adjacent grid cells, and within an island, the cell values may not repeat. Further, different islands cannot touch each other orthogonally, and an island must be built from at least three cells. All cells not part of an island belong to the ocean, a continuous area of orthogonally connected cells where value repetitions are allowed. There may be only one ocean, but no square of 2×2 cells may only contain ocean cells. In Nurikabe Sudoku, there are no hints in the form of already given numbers. Instead, some cells contain arrows which can face up, down, left and right. The value within a cell that contains an arrow must be equal to the number of cell types in the arrow's direction that are equal to the arrow cells type. This number does not include the arrow cell itself but includes cells separated from the arrow cell by the other cell type.

4
Encoding

As introduced in 2.2.1, the SAT-solvers expect a DIMACS file as input that describes a set of clauses. This chapter details how the different Sudoku Variants and their rules can be encoded into these clause sets. We elaborate on the different variants separately, but as long as there are no direct contradictions, the different variants and rules could be freely combined (which can be done by creating the union of the corresponding clause sets). In the following formulae, we use the notation $s_{x_n,...,x_0}$ to describe boolean variables, the corresponding integer numbers in the DIMACS files have the values $x_n * 10^n + ... + x_0 * 10^0$. For example, the literal $\neg s_{1,2,3}$ would be transformed to -123. There are two ways how we choose the name (number) for a variable during the encoding process:

1. We use increasing values starting from 1. These dynamically chosen values are, by example, used for variables necessary to encode PBCs. We use s_v, for $v \in \mathbb{N}$, to describe them in the formulae as we do not know the corresponding integers in advance.

2. We use fixed intervals of numbers to encode certain constraints. Here every digit of a number has a semantic meaning. The dynamically generated variables skip these fixed intervals. For example, a fixed interval is used for the boolean variables that describe the cell values of the Sudoku grid. The variable $s_{x,y,z}$ is true if and only if cell (x,y) has the value z assigned (as proposed in [13]). The other used fixed intervals are indicated in the corresponding sections of this chapter.

It is important to note that the numbers used as variable names are not "continuous". There are "gaps". For example, not all integers from 1 to 1000 are used. This is because the dynamic variables skip the entire interval from 111 to 999, and the fixed variables used to encode cell values will not use numbers like 400 because we start to count rows at 1, and the grid cells can only hold values from 1 to 9. The needed fixed intervals grow larger when encoding more complicated variants, and with them often also the "gaps". These gaps have no effect logically, but they can have a non-negligible effect on the time needed by the solvers, which is affected by the highest integer value used to describe a variable. However, as the "gaps" are the same when encoding Sudokus with the same rules, they should not make a difference when comparing solver-times.

4.1 Encoding of PBCs

In the following two subsections (4.1.1 and 4.1.2), we will elaborate on how PBCs can be encoded into clauses. As teased earlier, we will follow the ideas of [5] and use Binary Decision Diagrams and Adder Networks to perform this translation. The two methods approach their task rather differently, so we will compare their results and performance in chapter 5.

4.1.1 Encoding of PBCs using Binary Decision Diagrams

A binary decision diagram (BDD) is a directed graph that can be used to represent logical formulae and PBCs. Variables are considered in a fixed order. As discussed in [5], ordering them by decreasing weight values is generally reasonable. Every graph node corresponds to a sum and a subset of variables of the formula. The BDD has a root node corresponding to the empty variable set and sum 0. If a node is n edges away from the root, its variable set contains the n first variables, and such a node is said to be at depth n. An edge from a node u at depth k to a node v at depth $k+1$ corresponds to assigning a truth value to the $(k+1)$-th variable. Every node has at most two successors, one reachable via the edge that corresponds to assigning *true* and one via the edge that corresponds to assigning *false* to the next variable. Nodes store references to their successors as $true_{child}$ and $false_{child}$, respectively. Paths from the root to a node correspond to variable assignments to the variables in a node´s variable set and define the node's sum, which is equal to the sum of PBC weights that are multiplied by variables that are set to *true* by the assignment.

Terminal nodes however have no children. Terminal nodes are either nodes at depth l (with l equal to the total number of different variables in the formula) or nodes with a too high or too low sum regarding the variables already assigned in the paths to them.

An integer number is assigned to each node which can later be used to describe a boolean variable (called *extendable*). The *extendable* variable of a node is true if and only if the partial assignments that correspond to the paths from the root to it can be further extended to total assignments that respect the PBC the BDD represents.

The BDD can be built using a Breadth-First-Search starting in the root, shown as pseudocode in 4.1. The version we use for encoding PBCs differs from the one introduced in [5] because it is written to encode equations rather than inequations. Further, the used queue has additional functionalities: Given a node, it can check if it already contains a node with the same attribute values and can return said equal node. The *updated sums* of successor nodes either are the same as their predecessors (*false* was assigned) or are equal to the sum of their predecessor plus the weight value corresponding to the edge that led to them (*true* was assigned).

```
BuildBDD(PBC):
    queue = empty queue
    queue.append(createRoot())
    While not queue is empty:
        Node current = queue.first()
        If assigning next variable leads to a total assignment:
            Node cT = Create true successor node with updated sum
            Node cF = Create true successor node with updated sum
        Else:
            Node cT = Create true successor node with updated sum
            If cT.sum >= RHS:
                # true successor is a terminal node
            Else:
                # true successor is not a terminal node
                If cT not in queue:
                    queue.append(cT)
                Else:
                    cT = queue.get(cT)
            Node cF = Create false successor node with updated sum
            If cF.sum + sum of remaining weights < RHS:
                # false successor is a terminal node
            Else:
                # false successor is not a terminal node
                If cF not in queue:
                    queue.append(cF)
                Else:
                    cF = queue.get(cF)
        current.true_child = cT
        current.false_child = cF
    Return root
```

Algorithm 4.1: Pseudo Code of BDD construction

Once the BDD is built, we can transform it into clauses. In [5], it is explained how this can be achieved by treating the BDD network as a circuit of ITEs (if-then-else gates). However, it suffices to know that the BDD can be transformed by doing a second Breadth-First-Search starting from the root and that for each visited node that is not a terminal node, the following six implications must be added to the set of formulas [5]:

1. If the *extendable* variable of the current node is true and the variable corresponding to the leaving edges from this node is true, then it follows that the *extendable* variable of the $true_{child}$ node is true.

2. If the *extendable* variable of the current node is true and the variable corresponding to the leaving edges from this node is false, then it follows that the *extendable* variable of the $false_{child}$ node is true.

3. If the *extendable* variable of the current node is false and the variable corresponding to the leaving edges from this node is true, then it follows that the *extendable* variable of the $true_{child}$ node is false.

4. If the *extendable* variable of the current node is false and the variable corresponding to the leaving edges from this node is false, then it follows that the *extendable* variable of the $false_{child}$ node is false.

5. If the *extendable* variable of the $true_{child}$ node is true and the *extendable* variable of the $false_{child}$ node is true, then it follows that the *extendable* variable of the current node is true.

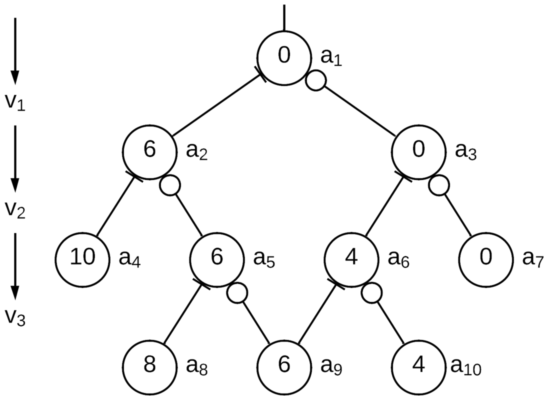
6. If the *extendable* variable of the $true_{child}$ node is false and the *extendable* variable of the $false_{child}$ node is false, then it follows that *extendable* variable of the current node is false.

Additionally, we add a clause that only contains the positive literal corresponding to the *extendable* variable of the root node. Also, when visiting a node during this second Breadth-First-Search, we only append its children that are not terminal nodes. For children that are terminal nodes, we add a clause to the set of clauses:

- If the child's sum is equal to the RHS, a clause containing a positive literal corresponding to the *extendable* variable of the child is added.

- If the child's sum is unequal the RHS, a clause containing a negative literal corresponding to the *extendable* variable of the child is added.

An example is shown in Figure 4.1 where the BDD and the clauses are depicted that are used to encode the PBC $6 * v_1 + 4 * v_2 + 2 * v_3 = 6$. Sums are written inside the nodes, and *extendable* variables are denoted as a_i for $i \in \{1, 2, ..., 10\}$. Edges with a circle correspond to assigning *false*. Edges with a line correspond to assigning *true*.



0.	$\{a_1\}$		
1.	$\{\neg a_1, \neg v_1, a_2\}$	from	$(a_1 \wedge v_1 \rightarrow a_2)$
	$\{\neg a_1, v_1, a_3\}$	from	$(a_1 \wedge \neg v_1 \rightarrow a_3)$
	$\{a_1, \neg v_1, \neg a_2\}$	from	$(\neg a_1 \wedge v_1 \rightarrow \neg a_2)$
	$\{a_1, v_1, \neg a_3\}$	from	$(\neg a_1 \wedge \neg v_1 \rightarrow \neg a_3)$
	$\{\neg a_2, \neg a_3, a_1\}$	from	$(a_2 \wedge a_3 \rightarrow a_1)$
	$\{a_2, a_3, \neg a_1\}$	from	$(\neg a_2 \wedge \neg a_3 \rightarrow \neg a_1)$
2.	$\{\neg a_2, \neg v_2, a_4\}$	from	$(a_2 \wedge v_2 \rightarrow a_4)$
	$\{\neg a_2, v_2, a_5\}$	from	$(a_2 \wedge \neg v_2 \rightarrow a_5)$
	$\{a_2, \neg v_2, \neg a_4\}$	from	$(\neg a_2 \wedge v_2 \rightarrow \neg a_4)$
	$\{a_2, v_2, \neg a_5\}$	from	$(\neg a_2 \wedge \neg v_2 \rightarrow \neg a_5)$
	$\{\neg a_4, \neg a_5, a_2\}$	from	$(a_4 \wedge a_5 \rightarrow a_2)$
	$\{a_4, a_5, \neg a_2\}$	from	$(\neg a_4 \wedge \neg a_5 \rightarrow \neg a_2)$
	$\{\neg a_4\}$		
3.	$\{\neg a_3, \neg v_2, a_6\}$	from	$(a_3 \wedge v_2 \rightarrow a_6)$
	$\{\neg a_3, v_2, a_7\}$	from	$(a_3 \wedge \neg v_2 \rightarrow a_7)$
	$\{a_3, \neg v_2, \neg a_6\}$	from	$(\neg a_3 \wedge v_2 \rightarrow \neg a_6)$
	$\{a_3, v_2, \neg a_7\}$	from	$(\neg a_3 \wedge \neg v_2 \rightarrow \neg a_7)$
	$\{\neg a_6, \neg a_7, a_3\}$	from	$(a_6 \wedge a_7 \rightarrow a_3)$
	$\{a_6, a_7, \neg a_3\}$	from	$(\neg a_6 \wedge \neg a_7 \rightarrow \neg a_3)$
	$\{\neg a_7\}$		
4.	$\{\neg a_5, \neg v_3, a_8\}$	from	$(a_5 \wedge v_3 \rightarrow a_8)$
	$\{\neg a_5, v_3, a_9\}$	from	$(a_5 \wedge \neg v_3 \rightarrow a_9)$
	$\{a_5, \neg v_3, \neg a_8\}$	from	$(\neg a_5 \wedge v_3 \rightarrow \neg a_8)$
	$\{a_5, v_3, \neg a_9\}$	from	$(\neg a_5 \wedge \neg v_3 \rightarrow \neg a_9)$
	$\{\neg a_8, \neg a_9, a_5\}$	from	$(a_8 \wedge a_9 \rightarrow a_5)$
	$\{a_8, a_9, \neg a_5\}$	from	$(\neg a_8 \wedge \neg a_9 \rightarrow \neg a_5)$
	$\{\neg a_8\}$		
5.	$\{\neg a_6, \neg v_3, a_9\}$	from	$(a_6 \wedge v_3 \rightarrow a_9)$
	$\{\neg a_6, v_3, a_{10}\}$	from	$(a_6 \wedge \neg v_3 \rightarrow a_{10})$
	$\{a_6, \neg v_3, \neg a_9\}$	from	$(\neg a_6 \wedge v_3 \rightarrow \neg a_9)$
	$\{a_6, v_3, \neg a_{10}\}$	from	$(\neg a_6 \wedge \neg v_3 \rightarrow \neg a_{10})$
	$\{\neg a_9, \neg a_{10}, a_6\}$	from	$(a_9 \wedge a_{10} \rightarrow a_6)$
	$\{a_9, a_{10}, \neg a_6\}$	from	$(\neg a_9 \wedge \neg a_{10} \rightarrow \neg a_6)$
	$\{a_9\}$		
	$\{\neg a_{10}\}$		

Figure 4.1: BDD and clauses to encode $6 * v_1 + 4 * v_2 + 2 * v_3 = 6$

4.1.2 Encoding of PBCs using Adder Networks

Adder Networks are built from Full and Half Adder nodes. A Full Adder (FA) is a node with three inputs and two outputs. A Half Adder (HA) is a node with two inputs and two outputs. The input and output values are binary (*true*/*false*). The outputs are name *sum* and *carry*, and their values are computed as follows. If at least two inputs are true, the carry output is true. Otherwise, it is false. So the carry value is already determined if two of three possible inputs are given. If one or three inputs are true, the sum output is true. Otherwise, it is false. This behaviour can be encoded into clauses, and as the name implies, it allows Adder Networks to compute sums over binary numbers. Binary numbers are Strings of bits, where each bit is either 1 (*true*) or 0 (*false*). The bit at position k represents a value of 2^k and is called k-bit (This notation differs from the one used in [5]).

An Adder Network used to sum numbers is explained best layer by layer. When adding binary numbers, the summand k-bits get fed into layer k. Layer k then also takes the carry values from the previous layer ($k - 1$) as input for its nodes except if k is 0. A layer's sum outputs are fed as input to adder nodes in the same layer (potentially requiring additional nodes) until only one input value remains. The remaining value of layer k then corresponds to the value of the k-bit of the overall addition result. The carry outputs of a layer are "carried over" to the next layer, where they become inputs to the adder nodes. The last layer has only one input, and its value gets directly treated as the output of this layer. Figure 4.2 shows an example network that can add three numbers, A, B and C, with values from 0 to 7 (3 bit long numbers).

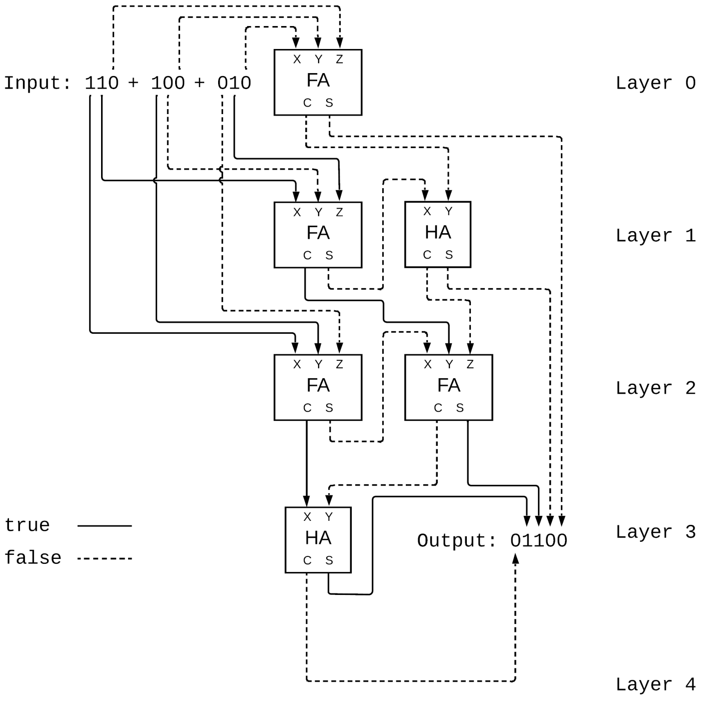


Figure 4.2: Adder Network computing $6 + 4 + 2 = 12$

As shown in [5], Adder Networks can be used to encode PBC. The weights of the PBC are taken as inputs for the network. Every input bit gets associated with the boolean variable that is multiplied by the weight they encode in the PBC. In practice, this is accomplished by feeding the boolean variable into the network directly (in the case of a *true* bit) or by feeding nothing into the network (in the case of a *false* bit). So if a boolean variable on the LHS of a PBC is *false*, all the input bits of the corresponding weights are interpreted as 0 by the network. The outputs of the adder nodes also get boolean variables assigned, including those outputs corresponding to the bit values of the overall result. These values should be equal to the corresponding bit values of the PBC's RHS as the PBCs we want to encode are equations. To enforce this, the values get compared, and corresponding unit clauses are added to the result set of clauses.

The encoding procedure (an adapted version of the algorithm originally proposed in [5]) is depicted as pseudocode in Figure 4.2. What makes the algorithm presented in [5] so efficient is that only the needed adder nodes are created, depending on how many *true* inputs there are in a layer. This dynamic generation keeps the number of adder nodes that actually get transformed into clauses as small as possible. For example, to encode the PBC $6 * v_1 + 4 * v_2 + 2 * v_3 = 6$, only two adder nodes (One Half Adder in Layer-1 and one Full Adder in Layer-2) must be transformed into clauses even though seven adders are needed to add up the numbers 6, 4 and 2 (see Figure 4.2).

```
AdderNetworkEncoder(PBC):
    clauseSet = empty Set
    Translate PBC weights to binary
    Translate RHS to binary
    bucket = Create empty map, integer —> set
    For every PBC weight:
        For every k-bit of the weight:
            If the bit is true:
                If bucket has no key == k:
                    bucket[k] = new set
                Add variable that is multiplied with current weigth
                in the PBC to bucket[k]
    networkOut = empty list
    k = 0
    While bucket has keys >= k:
        While bucket has no key == k:
            Append false-variable to networkOut
            k++
        While bucket[k] contains at least 3 variables:
            Create Full Adder with 3 variables of the k-bucket as input
            Add Full Adder clauses to clauseSet
            Create (k+1)-bucket if not present
            Put the Full Adders carry variable in bucket[k+1]
            Put the Full Adders sum variable in bucket[k]
        While bucket[k] contains at least 2 variables:
            Create Half Adder with 2 variables of the k-bucket as input
            Add Half Adder clauses to clauseSet
            Create (k+1)-bucket if not present
            Put the Half Adders carry variable in bucket[k+1]
            Put the Half Adders sum variable in bucket[k]
        Append the one remaining variable of bucket[k] to networkOut
        k++
    While networkOut.length < RHS.length:
        Append false-variable to networkOut
    While networkOut.length > RHS.length:
        Append false-variable to RHS
    For every RHS bit:
        If bit is true:
            Add a clause that contains a positive literal
            of the corresponding networkOut variable to clauseSet
        Else:
            Add a clause that contains a negative literal
            of the corresponding networkOut variable to clauseSet
    Return clauseSet
```

Algorithm 4.2: Pseudocode of PBC encoding using an Adder Network

During algorithm 4.2, Half and Full Adders are transformed into clauses, which can then be added to the result set of clauses that the procedure returns. Assuming the input variables of an adder are x, y and z, the corresponding output variables are s for the sum and c for the carry. Then the needed implications (as presented in [5] for Full Adders) to logically describe the adder nodes are as shown in Figure 4.3.

Full Adder clauses:

$$\begin{array}{lll}
\{x, y, z, \neg s\} & \text{from} & (\neg x \wedge \neg y \wedge \neg z) \rightarrow \neg s \\
\{x, \neg y, \neg z, \neg s\} & \text{from} & (\neg x \wedge y \wedge z) \rightarrow \neg s \\
\{\neg x, y, \neg z, \neg s\} & \text{from} & (x \wedge \neg y \wedge z) \rightarrow \neg s \\
\{\neg x, \neg y, z, \neg s\} & \text{from} & (x \wedge y \wedge \neg z) \rightarrow \neg s \\
\{\neg x, \neg y, \neg z, s\} & \text{from} & (x \wedge y \wedge z) \rightarrow s \\
\{\neg x, y, z, s\} & \text{from} & (x \wedge \neg y \wedge \neg z) \rightarrow s \\
\{x, \neg y, z, s\} & \text{from} & (\neg x \wedge y \wedge \neg z) \rightarrow s \\
\{x, y, \neg z, s\} & \text{from} & (\neg x \wedge \neg y \wedge z) \rightarrow s \\
\{\neg x, \neg y, c\} & \text{from} & (x \wedge y) \rightarrow c \\
\{\neg x, \neg z, c\} & \text{from} & (x \wedge z) \rightarrow c \\
\{\neg y, \neg z, c\} & \text{from} & (y \wedge z) \rightarrow c \\
\{x, y, \neg c\} & \text{from} & (\neg x \wedge \neg y) \rightarrow \neg c \\
\{x, z, \neg c\} & \text{from} & (\neg x \wedge \neg z) \rightarrow \neg c \\
\{y, z, \neg c\} & \text{from} & (\neg y \wedge \neg z) \rightarrow \neg c
\end{array}$$

Half Adder clauses:

$$\begin{array}{lll}
\{x, y, \neg s\} & \text{from} & (\neg x \wedge \neg y) \rightarrow \neg s \\
\{x, \neg y, s\} & \text{from} & (\neg x \wedge y) \rightarrow s \\
\{\neg x, y, s\} & \text{from} & (x \wedge \neg y) \rightarrow s \\
\{\neg x, \neg y, s\} & \text{from} & (x \wedge y) \rightarrow \neg s \\
\{x, y, \neg c\} & \text{from} & (\neg x \wedge \neg y) \rightarrow \neg c \\
\{x, \neg y, \neg c\} & \text{from} & (\neg x \wedge y) \rightarrow \neg c \\
\{\neg x, y, \neg c\} & \text{from} & (x \wedge \neg y) \rightarrow \neg c \\
\{\neg x, \neg y, c\} & \text{from} & (x \wedge y) \rightarrow c
\end{array}$$

Figure 4.3: Clauses for Full and Half Adders.
(Inputs: x,y,z Sum: s Carry: c)

4.2 Normal Sudoku

The normal Sudoku rules as introduced in 2.4 can be broken down into the five constraints shown in Table 4.1, which can be encoded into clauses using the variable $s_{x,y,z}$, which is true iff cell (x,y) has value z. The variable $s_{x,y,z}$ will be used further during the later shown formulae of other Sudoku Variants as the cell values play an important role in all of them. The encoding formulated in Table 4.2 can be seen as a direct encoding using at-least-one and at-most-one clauses and was proposed by [13] where it is called the minimal encoding.

Constraint	Formula	#Clauses
At least one number from 1 to 9 appears in each grid cell.	(S-i)	81
Every number appears at most once per row.	(S-ii)	2916
Every number appears at most once per column.	(S-iii)	2916
Every number appears at most once per box.	(S-iv) and (S-v)	2916
Every cell that contains a hint can only have that value.	(S-vi)	1/hint

Table 4.1: Constraints of Normal Sudoku.

$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigvee_{z=1}^{9} s_{x,y,z}$	(S-i)
$\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{x=1}^{9}\bigwedge_{i=x+1}^{9} \neg s_{x,y,z} \vee \neg s_{i,y,z}$	(S-ii)
$\bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{i=y+1}^{9} \neg s_{x,y,z} \vee \neg s_{x,i,z}$	(S-iii)
$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=y+1}^{3} \neg s_{(3*i+x),(3*j+y),z} \vee \neg s_{(3*i+x),(3*j+k),z}$	(S-iv)
$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=x+1}^{3}\bigwedge_{l=1}^{3} \neg s_{(3*i+x),(3*j+y),z} \vee \neg s_{(3*i+k),(3*j+l),z}$	(S-v)
$s_{x,y,input_{x,y}}$, for every (x,y) s.t. $\text{input}_{x,y} \neq 0$	(S-vi)

Table 4.2: Formulae of clauses, Normal Sudoku.

4.3 Anti-Knight

To encode the Anti-Knight rule, one must ensure that for each grid cell (x,y), it is forbidden to have the same value as its Knight-Neighbours (cells that are a knights distance away). The one constraint needed to formulate the Anti-Knight rule is stated in Table 4.3. The corresponding formula to encode it into clauses is shown in Table 4.4. The set of Knight-Neighbours to cell (x,y) can be defined as:

$$N(x, y) = \{(i, j) \mid \text{cell } (i, j) \text{ one knight distance from cell } (x, y)\}.$$

Constraint	Formula	#Clauses
Cells that are one knight distance apart (neighbours) must have different values.	(AK-i)	2016

Table 4.3: Constraints of Anti-Knight rule.

$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{(i,j)\in N(x,y)} \bigwedge_{z=1}^{9} \neg s_{x,y,z} \vee \neg s_{i,j,z}$	(AK-i)

Table 4.4: Formulae of clause, Anti-Knight rule.

4.4 Killer Sudoku

Using PBCs: For every killer cage of the input, we create a list of its cells. From every list, a PBC is created as follows: For every cell (x,y) of a cage, we add $\sum_{i=1}^{9}(s_{x,y,i} * i)$ to the left-hand side of the PBC. The right-hand side of the PBC is set to the target sum that was given as input. The different PBCs (one for every killer cage) can then be encoded into clauses, as explained in 4.1.1 and 4.1.2.

Using PBCs + Combinations: The PBC approach can be further optimized, because, given a fixed number of summands, not all values from 1 to 9 can be used to achieve a particular sum. For example, if a cage has a target sum of 8 and consists of three cells, the number of possible value combinations to achieve the target sum is fairly limited. There are only two possible value combinations $1 + 2 + 5 = 8$ and $1 + 3 + 4 = 8$, so the allowed values the cells could take are 1, 2, 3, 4 and 5. When constructing the PBC, this knowledge can be used to reduce the number of variable-value products on the left-hand side of the equation. For every cell in a cage, we only add the variable times the corresponding value (to the left-hand side) if the value is an allowed one.

Using Combinations: Another possibility is to completely abandon PBCs and exploit that only certain value combinations are possible given a cage with a fixed target sum and fixed number of cells that belong to it. To encode this every combination is given a corresponding variable s_v, for $v \in V_g \subset \mathbb{N}$, which is true iff the corresponding combination is used in a certain cage. The set of all cages is annotated as G, and the set of all possible combinations to achieve the target sum of a cage $g \in G$ is denoted as C_g. The constraints needed to encode the Killer Sudoku rules without the use of PBCs are stated in Table 4.5, further the formulae that encode these constraint to clauses are described in Table 4.6.

Constraint	Formula
For every cage $g \in G$ and possible combination $c \in C_g$ it holds that, either the cage's target sum is not achieved using combination c or every cage cell contains at least one value of the combination c.	(K-i)
In every cage $g \in G$ at least one combination $c \in C_g$ is used.	(K-ii)
In every cage $g \in G$ at most one combination $c \in C_g$ is used.	(K-iii)
Every value from 1 to 9 appears at most once within the cells of a cage $g \in G$.	(K-iv)

Table 4.5: Constraints of Killer Sudoku rules.

$$\bigwedge_{g:G}\bigwedge_{c:C_g}\bigwedge_{(x,y):g} -s_v \vee \bigvee_{z:c} s_{x,y,z} \tag{K-i}$$

$$\bigwedge_{g:G}\bigvee_{v:V_g} s_v \tag{K-ii}$$

$$\bigwedge_{g:G}\bigwedge_{v':V_g}\bigwedge_{v'':V_g} \neg s_{v'} \vee \neg s_{v''} \qquad \text{with } v' < v'' \tag{K-iii}$$

$$\bigwedge_{g:G}\bigwedge_{(x_i,y_i):g}\bigwedge_{(x_j,y_j):g}\bigwedge_{z=1}^{9} \neg x_i y_i z \vee \neg x_j y_j z \qquad \text{with } (x_i, y_i) \neq (x_j, y_j) \tag{K-iv}$$

Table 4.6: Formulae of clauses, Killer Sudoku rules.

4.5 Arrowheads

Arrowheads demand a total or partial order between two cells. The sets of all tuples of two cells under total or partial order are respectively noted as TO and PO. The first cell (x_1, y_1) of such a tuple $((x_1, y_1), (x_2, y_2)) \in TO \cup PO$ shall have a smaller (or equal) value than the second one (x_2, y_2). To enforce this, the constraints shown in 4.7 are encoded into clauses using the support encoding (see Table 4.8). In formula AH-i, the values of z_1 go from 1 to 9, which ensures that the first cell of a tuple can not have a value of 9. In formula AH-ii, on the other hand, z_1 iterates from 2 to 9 because if the first cell has a value of 1, all values from 1 to 9 are allowed for the second cell.

Constraint	Formula
For every $((x_1, y_1), (x_2, y_2)) \in TO$, either cell (x_1, y_1) has not value $z_1 \in \{1, .., 9\}$ or cell (x_2, y_2) has value $z_2 > z_1, z_2 \in \{2, .., 9\}$.	(AH-i)
For every $((x_1, y_1), (x_2, y_2)) \in PO$, either cell (x_1, y_1) has not value $z_1 \in \{2, .., 9\}$ or cell (x_2, y_2) has value $z_2 \geq z_1, z_2 \in \{1, .., 9\}$.	(AH-ii)

Table 4.7: Constraints of Arrowhead rules.

$$\bigwedge_{((x_1,y_1),(x_2,y_2)):TO} \bigwedge_{z_1=1}^{9} \neg s_{x_1,y_1,z_1} \vee \bigvee_{z_2=z_1+1}^{9} s_{x_2,y_2,z_2} \tag{AH-i}$$

$$\bigwedge_{((x_1,y_1),(x_2,y_2)):PO} \bigwedge_{z_1=2}^{9} \neg s_{x_1,y_1,z_1} \vee \bigvee_{z_2=z_1}^{9} s_{x_2,y_2,z_2} \tag{AH-ii}$$

Table 4.8: Formulae of clauses, Arrowhead rules.

4.6 Thermometers (Hidden)

As the Thermometer rule also demands a total (or partial) order between the cells of the thermometers, one can reuse the rules and formulas shown for Arrowheads in 4.5. Starting from a thermometers bulb, one can conceptually place arrowheads between every cell pair along the thermometer, enforcing a total (or partial) order between all cells.

Further constraints and variables are needed for the more complicated puzzle, where the solver must also deduce the thermometer positions. When encoding this puzzle, we distinguish two main cases: if the entire thermometer number t fits inside the 9×9 grid when its bulb is placed at (x, y), the clauses of TH-iv, TH-v and TH-vi are added. Otherwise the clauses of TH-vii are added. The corresponding constraints are defined in Table 4.10, and the formulae in Table 4.11 show how to encode them into clauses. Details on the used notation and variables within the formulae can be found in Table 4.9.

Notation and Variables:		
T	=	List of given thermometers
$T.size$	=	Number of given thermometers
$T[t]$	=	Thermometer number t in list of all thermometers
$T[t].size$	=	Number of cells in thermometer number t
$\chi(t, x, y, d)$	=	x-coordinate of the cell that is at depth d of the thermometer number t if its bulb is placed in cell (x, y)
$\Upsilon(t, x, y, d)$	=	y-coordinate of the cell that is at depth d of the thermometer number t if its bulb is placed in cell (x, y)
$Arrowhead(t, x, y, d)$	=	Set of clauses needed to encode that the cell at depth d of thermometer number t is smaller than the cell at depth $d + 1$, given the thermometer has its bulb in cell (x, y).
$s_{t,d,x,y}$	:	Is true if and only if the cell at depth d of thermometer number t is located in grid cell (x, y). t ∈ $\{1, .., 81\}$ d,x,y ∈ $\{1, .., 9\}$ Digits : ttdxy Range: : 01111 to 81999 In practice t has an offset of +10 to avoid the leading 0. So the actual variable range is: 11111 to 91999.

Table 4.9: Notations and variables, Thermometers-Hidden rules.

Constraint	Formula
Every cell of every thermometer is located in at least one grid cell.	(TH-i)
At most one thermometer cell is located in one grid cell.	(TH-ii) and (TH-iii)
If the cell at depth d of thermometer number t is located in grid cell (x, y), then the thermometer cell at depth $d + 1$ must be located next to it (corresponding to the thermometer's shape), except if $d =$ T[t].size.	(TH-iv)
If the cell at depth d of thermometer number t is located in grid cell (x, y), then the thermometer cell at depth $d - 1$ must be located next to it (corresponding to the thermometer's shape), except if $d = 0$.	(TH-v)
If the cell at depth d of thermometer number t is located in grid cell (x, y), then the cell value of (x, y) must be lower than that of the thermometer cell at depth $d + 1$.	(TH-vi)
If not the entire thermometer with number t is inside the 9×9 when its bulb is placed at cell (x, y), then the cell at depth d of thermometer number t can not be located at the cell it covers in this situation.	(TH-vii)

Table 4.10: Constraints of the Thermometers-Hidden rules.

$$\bigwedge_{t=1}^{T.size}\bigwedge_{d=1}^{T[t].size}\bigvee_{x=1}^{9}\bigvee_{y=1}^{9} s_{t,d,x,y} \qquad \text{(TH-i)}$$

$$\bigwedge_{t=1}^{T.size}\bigwedge_{d=1}^{T[t].size}\bigwedge_{k=d+1}^{T[t].size}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9} \neg s_{t,d,x,y} \vee \neg s_{t,k,x,y} \qquad \text{(TH-ii)}$$

$$\bigwedge_{t=1}^{T.size}\bigwedge_{k=t+1}^{T.size}\bigwedge_{d=1}^{T[t].size}\bigwedge_{l=1}^{T[t].size}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9} \neg s_{t,d,x,y} \vee \neg s_{k,l,x,y} \qquad \text{(TH-iii)}$$

$$\bigwedge_{t=1}^{T.size}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{d=1}^{T[t].size-1} \neg s_{t,d,\chi(t,x,y,d),\Upsilon(t,x,y,d)} \vee s_{t,d+1,\chi(t,x,y,d+1),\Upsilon(t,x,y,d+1)} \qquad \text{(TH-iv)}$$

$$\bigwedge_{t=1}^{T.size}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{d=1}^{T[t].size-1} s_{t,d,\chi(t,x,y,d),\Upsilon(t,x,y,d)} \vee \neg s_{t,d+1,\chi(t,x,y,d+1),\Upsilon(t,x,y,d+1)} \qquad \text{(TH-v)}$$

$$\bigwedge_{t=1}^{T.size}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{d=1}^{T[t].size-1} \neg s_{t,d,\chi(t,x,y,d),\Upsilon(t,x,y,d)} \vee Arrowhead(t,x,y,d) \qquad \text{(TH-vi)}$$

$$\bigwedge_{t=1}^{T.size}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{d=1}^{T[t].size} \neg s_{t,d,\chi(t,x,y,d),\Upsilon(t,x,y,d)} \qquad \text{with } \chi(\cdot),\Upsilon(\cdot)\in[1,9] \qquad \text{(TH-vii)}$$

Table 4.11: Formulae of clauses, Thermometers-Hidden rules.

4.7 Sandwich Sum

Sandwich Sums can be given for rows and columns. We will elaborate on the row constraints from which the formulas for the column constraints can be derived by swapping x and y, respectively. Similar to the Killer Sudoku rules, multiple optimizations can be (incrementally) made when encoding the Sandwich Sum rules.

Using PBCs: Assuming that all lengths of sandwiches are possible to achieve a certain sum. We must encode all possible sandwich lengths (0 to 7) and all corresponding positions for the cells with values 1 and 9. Also, we encode a PBC for every possible sandwich length and position. The LHS of said PBC is $\sum_{i=1}^{9}(s_{x,y,i} * i)$, and the RHS is set to the current rows sandwich sum. Since the sandwich can only be at one position in a row at once, only one of all the PBC will be true. To ensure that a PBC's clauses are satisfied if the sandwich is not in its corresponding position, we add the negative literal of the corresponding s_v to each clause (see formulae SW-vii, SW-viii and SW-ix in Table 4.14). To encode the Sandwich Sum rules for rows without the incorporation of further knowledge, the following constraints/formulas are used: SW-i, SW-ii, SW-vii, SW-x and SW-xi.

Using PBCs + Combinations for Lengths: Given the sandwich sum of a row, one can already make statements about the number of cells involved in the sum. For example, given a sandwich sum of 8, one can conclude that the sandwich has either length 3, 2, or 1 (contains 3, 2 or 1 cells, not counting the border cells with values 1 and 9). This already reduces the number of combinations of sandwich lengths and positions and can be utilized to encode the rules using the constraints/formulas: SW-iii, SW-iv, SW-v, SW-vi, SW-viii, SW-x and SW-xi.

Using Combinations for PBCs + Combinations for Lengths: Given the sandwich sum of a row, one can not only make statements about the number of cells involved in the sum but also about the possible values of these cells given a certain sandwich length (Similar to the possible cell values in a killer cage given the cages sum and size). This reduces the number of summands of the LHS of PBCs and can be combined with the knowledge about possible sandwich lengths. To encode the Sandwich Sum rules using this additional knowledge, the following constraints/formulas: are used: SW-iii, SW-iv, SW-v, SW-vi, SW-ix, SW-x, SW-xi and SW-xii.

The different constraints and formulae used for these three encoding versions are listed in Tables 4.12 and 4.14, respectively. Notation and variables used in the formulae are further explained in Table 4.13.

Constraint on rows	Formula
If the sandwich of a row is at a certain position, the cells with values 1 and 9 must be positioned at its left and right end.	(SW-i) and (SW-ii)
If the sandwich of a row (with compatible length regarding the sandwich sum) is at a certain position, the cells with values 1 and 9 must be positioned at its left and right end.	(SW-iii) and (SW-iv)
The cells with values 1 and 9 can not have a certain number of cells between them, if the values from these number of cells can not add up to the rows sandwich sum.	(SW-v) and SW-vi)
If the sandwich of a row is at a certain position, then the corresponding cells must add up to the row's sandwich sum.	(SW-vii)
If the sandwich of a row (with compatible length regarding the sandwich sum) is at a certain position, then the corresponding cells must add up to the row's sandwich sum.	(SW-viii)
If the sandwich of a row (with compatible length regarding the sandwich sum) is at a certain position, then the corresponding cells must add up to the row's sandwich sum using only compatible values regarding the sandwich sum.	(SW-ix)
The sandwich must be in at least one position.	(SW-x)
The sandwich must be in at most one position.	(SW-xi)
If the sandwich of a row is at a certain position, the cell values of the cells that are added up to the sandwich sum can not be incompatible regarding the sandwich sum.	(SW-xii)

Table 4.12: Constraints of Sandwich Sum rules.

Notation and Variables:		
$PBC(x_1, x_2, y, sum)$	=	Set of clauses needed to encode that the cells between (x_1, y) and (x_2, y) have values $\in \{1, ..., 9\}$ that add up to *sum*.
$PBC(x_1, x_2, y, sum, Val)$	=	Set of clauses needed to encode that the cells between (x_1, y) and (x_2, y) have values $\in$ *Val* that add up to *sum*.
$L(y)$	=	Set of numbers corresponding to the sizes of cell sets for which it is possible to achieve the sandwich sum of row y, given that each cell can only have values from 1 to 9. $L(y) \subseteq \{0, 1, ..., 7\}$
$\bar{L}(y)$	=	$\{0, ..., 7\} \setminus L(y)$
$Z(y, l)$	=	Set of integer values for which it is possible to achieve the sandwich sum of row y if l different one of them are added. $Z(y, l) \subseteq \{1, ..., 9\}$
$\bar{Z}(y, l)$	=	$\{1, ..., 9\} \setminus Z(y, l)$
$\mathcal{S}(x, y, l)$	=	Dynamically assigned variable s_v for each combination of x, y and l that is true if the sandwich of row y has its borders in cell (x, y) and cell $(x + 1 + l, y)$. $v \in V_y \subset \mathbb{N}$

Table 4.13: Notations and variables, Sandwich Sum rules.

$$\bigwedge_{y=1}^{9}\bigwedge_{l=0}^{7}\bigwedge_{x=1}^{9} \neg s_{x,y,1} \vee \neg s_{(x+l+1),y,9} \vee \mathcal{S}(x,y,l) \tag{SW-i}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l=0}^{7}\bigwedge_{x=1}^{9} \neg s_{x,y,9} \vee \neg s_{(x+l+1),y,1} \vee \mathcal{S}(x,y,l) \tag{SW-ii}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:L(y)}\bigwedge_{x=1}^{9} \neg s_{x,y,1} \vee \neg s_{(x+l+1),y,9} \vee \mathcal{S}(x,y,l) \tag{SW-iii}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:L(y)}\bigwedge_{x=1}^{9} \neg s_{x,y,9} \vee \neg s_{(x+l+1),y,1} \vee \mathcal{S}(x,y,l) \tag{SW-iv}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:\bar{L}(y)}\bigwedge_{x=1}^{9-l-1} \neg s_{x,y,1} \vee \neg s_{(x+l+1),y,9} \tag{SW-v}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:\bar{L}(y)}\bigwedge_{x=1}^{9-l-1} \neg s_{x,y,9} \vee \neg s_{(x+l+1),y,1} \tag{SW-vi}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l=0}^{7}\bigwedge_{x=1}^{9}\bigwedge_{\varphi\in PBC(x,x+l+1,y,sum)} \varphi \vee \neg\mathcal{S}(x,y,l) \tag{SW-vii}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:L(y)}\bigwedge_{x=1}^{9}\bigwedge_{\varphi\in PBC(x,x+l+1,y,sum)} \varphi \vee \neg\mathcal{S}(x,y,l) \tag{SW-viii}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:L(y)}\bigwedge_{x=1}^{9}\bigwedge_{\varphi\in PBC(x,x+l+1,y,sum,Z(y,l))} \varphi \vee \neg\mathcal{S}(x,y,l) \tag{SW-ix}$$

$$\bigwedge_{y=1}^{9}\bigvee_{v:V_y} s_v \tag{SW-x}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{v':V_y}\bigwedge_{v'':V_y} \neg s_{v'} \vee \neg s_{v''} \qquad \text{with } v' < v'' \tag{SW-xi}$$

$$\bigwedge_{y=1}^{9}\bigwedge_{l:L(y)}\bigwedge_{x=1}^{9}\bigwedge_{z:\bar{Z}(y,l)}\bigwedge_{x'=x+1}^{x+l} \neg s_{x',y,z} \vee \neg\mathcal{S}(x,y,l) \tag{SW-xii}$$

Table 4.14: Formulae of clauses, Sandwich Sum rules.

4.8 Secret Direction

To encode the secret direction rules, we introduce a variable that is true iff a cell (x, y) is part of the hidden path at a certain depth d. We define the initial cell to have a depth of 0, and its successors in the path have depths 1, 2, et cetera. The depth can be used to ensure that every cell can only be in the path once, those prohibiting loops. That loops can not be part of the solution path can be inferred from the definition of the next step in a path, which is always determined by the position of the value 9 in the current (3×3)-box and the value of the current path cell, making it impossible that one cell of the path has two different successors. So loops cannot be part of the solution path because it would not be possible to break out of them.

From the fact that the position of the value 9 is needed to determine the direction in which the successor of a cell in the path lays, we can also derive that if the center of a (3×3)-box has value 9, all other cells of that box cannot be in the path, because for them no successor direction would be defined.

Since the Sudoku grid has only a size of 9×9 and the cell value of a cell in the path determines the distance to its successor, we can conclude that cells with a value of 9 can not be part of the path (except as final cell) because after them the path would step outside the grid.

Combining the previous two deductions, we can compute a sufficiently low upper bound for the maximal path length of $81 - 8 - 8 = 65$. Given these analyses, the constraints shown in 4.15 can be formulated and encoded into clauses as shown in 4.17. Details about the used notation and helping functions can be found in 4.16.

Constraint	Formula
At least one (3×3)-box has a cell with value 9 as center.	(SD-i)
At most (3×3)-box has a cell with value 9 as center.	(SD-ii) and (SD-iii)
A (3×3)-box center cell that has value 9 is part of the path in some depth.	(SD-iv)
A cell can be at most once in the path (can only be in at most one depth).	(SD-v)
The path can have at most one cell in every depth.	(SD-vi) and (SD-vii)
Cells that are in the path, and are center cells of a (3×3)-box, have not the value 9. (All cells with value 9, that are in the path are center cells.)	(SD-viii)
The non center cells of the (3×3)-box with a center cell that has value 9, are not in the path.	(SD-vi) and (SD-ix)
If a cell (x, y) is not the center cell of a (3×3)-box, has value z, is in the path at depth d, and the the position of the 9-valued cell in the same (3×3)-box hints in a certain direction. Then the cell (x_s, y_s) that is z steps in certain direction away from (x, y) is also in the path, at layer $(d + 1)$.	(SD-xi)

Table 4.15: Constraints of Secret Direction rules.

Notation and Variables:		
M	=	{(2,2), (2,5), (2,8), (5,2), (5,5), (5,8), (8,2), (8,5), (8,8)}
Ψ	=	{(1,1), (1,2), (1,3), (2,1), (2,3), (3,1), (3,2), (3,3)}
$\mathcal{G}$	=	$\{(a,b) \mid a,b \in \{1,2,3,4,5,6,7,8,9\}\}$
$\mathcal{F}_1(x_b, y_b, x', y', d, x_n, y_n, z)$	=	$\begin{cases} (\neg s_{1,(3*x_b+x'),(3*y_b+y'),d} \vee \neg s_{(3*x_b+x_n),(3*y_b+y_n),9} \vee \neg s_{x,y,z}) & (x_s, y_s) \notin \mathcal{G} \\ (\neg s_{1,(3*x_b+x'),(3*y_b+y'),d} \vee \neg s_{(3*x_b+x_n),(3*y_b+y_n),9} \vee \neg s_{x,y,z} \vee s_{1,succ_x(x_b,x',x_n,z),succ_y(y_b,y',y_n,z),(d+1)}) & (x_s, y_s) \in \mathcal{G} \end{cases}$
$succ_x(x_b, x', x_n, z)$	=	$\begin{cases} (3*x_b+x')-z & x_n = 1 \\ (3*x_b+x') & x_n = 2 \\ (3*x_b+x')+z & x_n = 3 \end{cases}$
$succ_y(y_b, y', y_n, z)$	=	$\begin{cases} (3*y_b+y')-z & y_n = 1 \\ (3*y_b+y') & y_n = 2 \\ (3*y_b+y')+z & y_n = 3 \end{cases}$
$s_{1,x,y,d}$	:	Is true if and only if the cell (x,y) is in the path, at depth d. d $\in$ $\{0,..,64\}$ x,y $\in$ $\{1,..,9\}$ Digits : $1xydd$ Range : 11100 to 19964

Table 4.16: Notations and variables, Secret Direction rules.

$$\bigvee_{x\in\{2,3,8\}} \bigvee_{y\in\{2,3,8\}} s_{x,y,9} \tag{SD-i}$$

$$\bigwedge_{x\in\{2,3,8\}} \bigwedge_{y\in\{2,3,8\}} \bigwedge_{\substack{k\in\{3,8\}\\ k>y}} \neg s_{x,y,9} \vee \neg s_{x,k,9} \tag{SD-ii}$$

$$\bigwedge_{x\in\{2,3,8\}} \bigwedge_{y\in\{2,3,8\}} \bigwedge_{\substack{k\in\{3,8\}\\ k>x}} \bigwedge_{l\in\{2,3,8\}} \neg s_{x,y,9} \vee \neg s_{k,l,9} \tag{SD-iii}$$

$$\bigwedge_{x\in\{2,3,8\}} \bigwedge_{y\in\{2,3,8\}} \neg s_{x,y,9} \vee \bigvee_{d=0}^{64} s_{1,x,y,d} \tag{SD-iv}$$

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{d=0}^{63} \bigwedge_{k=d+1}^{64} \neg s_{1,x,y,d} \vee \neg s_{1,x,y,k} \tag{SD-v}$$

$$\bigwedge_{d=0}^{64} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{k=y+1}^{9} \neg s_{1,x,y,d} \vee \neg s_{1,x,k,d} \tag{SD-vi}$$

$$\bigwedge_{d=0}^{64} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{k=x+1}^{9} \bigwedge_{l=1}^{9} \neg s_{1,x,y,d} \vee \neg s_{1,k,l,d} \tag{SD-vii}$$

$$\bigwedge_{d=0}^{64} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \neg s_{1,x,y,d} \vee \neg s_{x,y,9} \qquad \text{s.t. } (x,y) \notin M \tag{SD-viii}$$

$$\bigwedge_{x\in\{2,3,8\}} \bigwedge_{y\in\{2,3,8\}} \bigwedge_{x'=-1}^{1} \bigwedge_{y'=-1}^{1} \bigwedge_{d=0}^{64} \neg s_{x,y,9} \vee \neg s_{1,(x+x'),(y+y'),d} \qquad \text{s.t. } (x',y') \neq (0,0) \tag{SD-ix}$$

$$s_{1,x_r,y_r,0} \tag{SD-x}$$

$$\bigwedge_{x_b=0}^{2} \bigwedge_{y_b=0}^{2} \bigwedge_{x'=1}^{3} \bigwedge_{y'=1}^{3} \bigwedge_{d=0}^{63} \bigwedge_{(x_n,y_n):\Psi} \bigwedge_{z=1}^{8} \mathcal{F}_1(x_b, y_b, x', y', d, x_n, y_n, z) \tag{SD-xi}$$

Table 4.17: Formulae of clauses, Secret Direction rules.

4.9 Fawlty Towers

To encode the Fawlty Tower rules, we introduce three new variables for each tower, which are true iff a tower is faulty, has increasing values or if the cell values of a tower add up to the indicated sum, respectively. Details about these three variables can be found in Table 4.20. We can encode the needed increasing values of a tower by placing arrowheads between each neighbouring pair of its cells. Arrowheads can be encoded as shown in 4.5. To encode that the cells of a tower must have values that add up to the indicated sum, we can place a killer cage over them and set the cage's sum to the indicated one. The killer cage can then be encoded, as explained in 4.4. Further, we can use PBCs to ensure that the correct number of towers is faulty, has increasing values or has cell values that add up to the indicated sum. The three PBCs used for this are detailed in Table 4.19. The constraints derived from the Fawlty Tower rules are shown in Table 4.18, and the formulae to encode these constraints into clauses are displayed in Table 4.21.

Constraint	Formula
A tower in column x is faulty if and only if either the sum of its cell values is not equal to the indicated one for column x or its cell values are not increasing (but not both at once).	(FT-i) and (FT-ii)
If a tower has increasing cell values, all the clauses of the corresponding arrowheads placed between its cells must be true.	(FT-iii)
If the cells of a tower in column x have values that add up to the sum indicated for column x, all the clauses of the corresponding killer cage must be true.	(FT-iv)
The number of faulty towers must be equal to the one given in the puzzle description.	(FT-v)
The number of towers with increasing cell values must be equal to the one given in the puzzle description.	(FT-vi)
The number of towers with cell values that add up to the sum indicated for the corresponding column must be equal to the one given in the puzzle description.	(FT-vii)

Table 4.18: Constraints of Fawlty Tower rules.

Notation:		
$Arrowhead(x, y, x', y')$	$=$	Set of clauses needed to encode that the cell (x, y) contains a smaller value than the cell (x', y') as if there were an arrowhead placed between (x, y) and (x', y'). Arrowheads can be encoded as explained in 4.5.
$Killer(x, y, x', y')$	$=$	Set of clauses needed to encode that the cells between (x, y) and (x', y') (including (x, y) and (x', y')) have values that add up to the indicated sum for the tower in column x. This encoding is similar to the one for killer cages which is explained in 4.4.
PBC_{faulty}	$=$	Set of clauses needed to encode a PBC, which ensures that the number of columns containing a faulty tower matches the one stated in the puzzle description. $LHS = \sum_{x=1}^{9} 1 * faulty(x)$ $RHS =$ Number of faulty towers demanded in puzzle description.
PBC_{inc}	$=$	Set of clauses needed to encode a PBC, which ensures that the number of columns containing a tower with increasing cell values matches the one stated in the puzzle description. $LHS = \sum_{x=1}^{9} 1 * inc(x)$ $RHS =$ Number of increasing towers demanded in puzzle description.
PBC_{sum}	$=$	Set of clauses needed to encode a PBC, which ensures that the number of columns containing a tower with cell values that add up to the correspondingly indicated sum matches the one stated in the puzzle description. $LHS = \sum_{x=1}^{9} 1 * sum(x)$ $RHS =$ Number of towers with a cell value sum matching the indicated one.
$h(x)$	$=$	Height (number of contained cells) of tower in column x.

Table 4.19: Notation, Fawlty Tower rules.

Variables:

$faulty(x)$ = $s_{1,1,x,0}$
True iff the tower in column x is a faulty tower.
$x \in \{1,..,9\}$
Digits : $11x0$
Range : 1110 to 1190

$inc(x)$ = $s_{1,1,x,1}$
True iff the cells of the tower in column x is contain increasing values from the bottom to the top.
$x \in \{1,..,9\}$
Digits : $11x1$
Range : 1111 to 1191

$sum(x)$ = $s_{1,1,x,2}$
True iff the cell values of the tower in column x add up to the indicated value.
$x \in \{1,..,9\}$
Digits : $11x2$
Range : 1112 to 1192

Table 4.20: Variables, Fawlty Tower rules.

$$\bigwedge_{x=1}^{9} (\neg faulty(x) \vee sum(x) \vee inc(x)) \wedge (\neg faulty(x) \vee \neg sum(x) \vee \neg inc(x)) \qquad \text{(FT-i)}$$

$$\bigwedge_{x=1}^{9} (faulty(x) \vee sum(x) \vee \neg inc(x)) \wedge (faulty(x) \vee \neg sum(x) \vee inc(x)) \qquad \text{(FT-ii)}$$

$$\bigwedge_{x=1}^{9} \bigwedge_{y=11-h(x)}^{9} \bigwedge_{\varphi \in Arrowhead(x,y,x,y-1)} \neg inc(x) \vee \varphi \qquad \text{(FT-iii)}$$

$$\bigwedge_{x=1}^{9} \bigwedge_{y=11-h(x)}^{9} \bigwedge_{\varphi \in Killer(x,10-h(x),x,9)} \neg sum(x) \vee \varphi \qquad \text{(FT-iv)}$$

$$PBC_{faulty} \qquad \text{(FT-v)}$$

$$PBC_{inc} \qquad \text{(FT-vi)}$$

$$PBC_{sum} \qquad \text{(FT-vii)}$$

Table 4.21: Formulae of clauses, Fawlty Tower rules.

4.10 Nurikabe Sudoku

As one part of the solution to a Nurikabe Sudoku is marking island and ocean cells, we introduce corresponding variables that encode if a cell (x, y) is part of the ocean or part of island n (further detailed in Table 4.25 as $\mathcal{O}(x, y)$ and $\mathcal{I}(n, x, y)$). We note that there is only one ocean but multiple islands. Since an island must consist of at least three orthogonally adjacent cells and there are only 81 cells in the grid, a safe upper bound for the number of islands can be calculated by $81/3 = 27$. However, this number can be lowered because of the additional rule that islands may not touch each other orthogonally and that there is only one ocean. Considering this, the maximum number of islands found (by hand) to fit in the grid is 13, but we will use an upper bound of 14 for the encoding just to be sure. An example of how 13 islands can be placed in the grid that complies with the rules of Nurikabe Sudoku can be seen in Figure 4.4.

The rule that an island must consist of at least three orthogonally adjacent cells can be broken down to each individual cell of an island by stating that each cell that is part of an island must be in a *constellation* (a set) with two other neighbouring cells that are part of the same island. For such a constellation, it must hold that one of the three cells is orthogonally adjacent to the other two cells. As shown in Figure 4.6, there are 18 different possible constellations that one cell could be part of. To encode this rule, we introduce a new variable which is further explained in Table 4.25 in the definition of $\mathcal{N}(c, n, x, y)$.

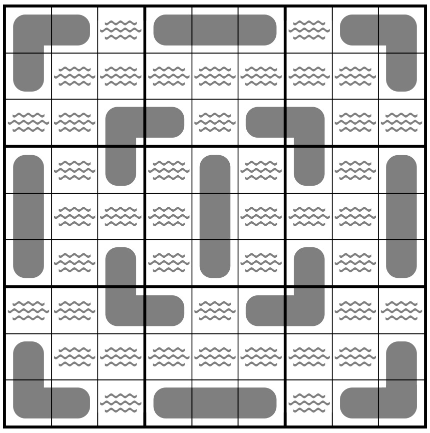


Figure 4.4: E.g. with 13 islands

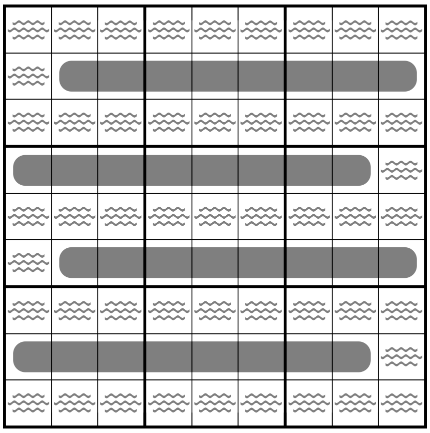


Figure 4.5: E.g. with max. flood depth 49

Further, we must enforce that islands are continuous. So from every cell that is part of an island n it must be possible to reach any other cell of island n, without crossing an ocean cell. This reachability can be enforced by constraints describing a Floodfill methodology. However, we will call this *walk* as the word *flood* is better suited to describe the similar idea used for the ocean cells. A walk is defined by its corresponding island n and its source (x_s, y_s), which is said to be at depth 1 of the walk. Outgoing from the source, orthogonally adjacent cells are walked with increasing depth numbers. As we model a Floodfill multiple cells can be at the same depth of a walk. At depth 9, the walk will have covered all cells of an island because the rules state that there are no value repetitions within an island which limits the size of an island to at most nine cells. The variable used to encode walks is further described in Table 4.25 in the definition of $\mathcal{W}(d, n, x_s, y_s, x, y)$.

The rules also dictate that there may only be one continuous ocean, which can be enforced with an approach similar to the one used for the continuousness of islands. However, as value repetitions are allowed within the ocean, its size is not limited to nine cells.The depth needed by a flood to cover all ocean cells is maximized if there are as few cells as possible in every depth. With only one cell in each depth, an upper bound of 81 can be given as the grid contains a total of 81 cells. But this number can be lowered as more than one cell will be at the same depth in practice. After experimenting with possible island placements by hand, we found an upper limit for the flood depth of 49. An Example for an island placements that requires this flood depth can be seen in Figure 4.5. The variable used to encode floods is further described in Table 4.25 in the definition of $\mathcal{F}(d, x_s, y_s, x, y)$.

The constraints to encode Nurikabe Sudoku are stated in the Figures 4.22, 4.23 and 4.24. How the constraints are encoded into clauses is shown in the Figures 4.28, 4.29 and 4.30.

Constraint	Formula
No (2×2)-square can only consist of ocean cells.	(NK-i)
Cells that are part of the same island can not have the same value.	(NK-ii) and (NK-iii)
Island cells of different islands can not touch each other orthogonally.	(NK-iv) and (NK-v)
Island cells must be in at least one constellation with two neighbouring cells.	(NK-vi)
If cell (x, y) is part of island n and is in a constellation that includes the two neighbouring cells (x', y') and (x'', y'') then the two neighbouring cells must be part of the same island n.	(NK-vii) and (NK-viii)
Cells must be part of the ocean or part of at least one island.	(NK-ix)
Cells that are part of the ocean, are not part of any island.	(NK-x)
Cells can be part of at most one island.	(NK-xi)

Table 4.22: Constraints-1 of Nurikabe Sudoku rules.

An ocean cell (x, y) must be in at least one depth of any flood that has a source (x_s, y_s) which is also an ocean cell.	(NK-xii)
Cells can be in at most one depth per flood with source (x_s, y_s).	(NK-xiii)
Cells that are not part of the ocean are not flooded.	(NK-xiv)
If cell (x,y) is part of the ocean, it is in depth 1 of the flood with source (x,y).	(NK-xv)
Only the source cell of a flood can be in depth 1 of a flood (where it is source).	(NK-xvi)
If a cell (x, y) is in depth d of a flood with source (x_s, y_s), and $(x, y) \neq (x_s, y_s)$, and the cell (x_s, y_s) is part of the ocean, then there must be an orthogonally adjacent cell to (x, y) which is in depth $d - 1$ of the flood with source (x_s, y_s).	(NK-xvii)
If a cell (x, y) is part of island n, then the cell (x,y) is at depth 1 of the walk on island n with source (x,y).	(NK-xviii)
Only the source cell of a walk can be in depth 1 of a walk (where it is source).	(NK-xix)
If a cell (x, y) is in depth d of a walk on island n with source (x_s, y_s), and $(x, y) \neq (x_s, y_s)$, and the cell (x_s, y_s) is part of island n, then there must be an orthogonally adjacent cell to (x, y) which is in depth $d - 1$ of the walk on island n with source (x_s, y_s).	(NK-xx)
If cell (x, y) is part of island n and cell (x_s, y_s) is part of island n, then cell (x, y) must be in at least one depth of the walk on island n that has source (x_s, y_s).	(NK-xxi)
A cell (x, y) can be in at most one depth of a walk on an island n that has source (x_s, y_s).	(NK-xxii)
Cells that are not part of an island n can not be in any depth of a walk on island n.	(NK-xxiii)

Table 4.23: Constraints-2 of Nurikabe Sudoku rules.

If a cell that is part of the ocean contains an arrow, and has value z, then there are z cells that are part of the ocean in the direction the arrow points.	(NK-xxiv)
If a cell that is not part of the ocean contains an arrow, and has value z, then there are z cells that are not part of the ocean in the direction the arrow points.	(NK-xxv)
There is no solution, if a cell that is part of the grid's edge contains an arrow that points directly outside the grid.	(NK-xxvi)

Table 4.24: Constraints-3 of Nurikabe Sudoku rules.

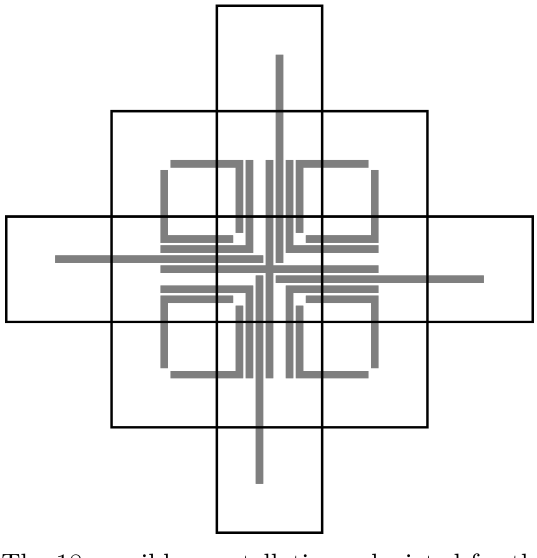


Figure 4.6: The 18 possible constellations, depicted for the center cell

Variables:		
$\mathcal{O}(x, y)$	$=$	$s_{1,5,0,0,0,x,y}$ True iff cell (x, y) is part of the ocean. x, y $\in$ $\{1, .., 9\}$ Digits : $15000xy$ Range : 1500011 to 1500099
$\mathcal{I}(n, x, y)$	$=$	$s_{1,4,0,n,x,y}$ True iff cell (x, y) is part of island n. x, y $\in$ $\{1, .., 9\}$ n $\in$ $\{1, .., 14\}$ Digits : $140nnxy$ Range : 1400111 to 1401499
$\mathcal{F}(d, x_s, y_s, x, y)$	$=$	$s_{2,d,x_s,y_s,x,y}$ True iff cell (x, y) is in flood-depth d of the flood with source (x_s, y_s). x, y $\in$ $\{1, .., 9\}$ x_s, y_s $\in$ $\{1, .., 9\}$ d $\in$ $\{1, .., 49\}$ Digits : $2ddx_sy_sxy$ Range : 2011111 to 2499999
$\mathcal{W}(d, n, x_s, y_s, x, y)$	$=$	$s_{1,d,n,x_s,y_s,x,y}$ True iff cell (x, y) is in depth d of the walk on island n with source (x_s, y_s). x, y $\in$ $\{1, .., 9\}$ x_s, y_s $\in$ $\{1, .., 9\}$ d $\in$ $\{1, .., 9\}$ n $\in$ $\{1, .., 14\}$ Digits : $1dnnx_sy_sxy$ Range : 11011111 to 19149999
$\mathcal{N}(c, n, x, y)$	$=$	$s_{1,0,c,n,x,y}$ True iff cell (x, y) belongs to island n, and has two adjacent neighbours that also belong to island n, and that are in constellation c with cell (x, y). c $\in$ $\{1, .., 18\}$ n $\in$ $\{1, .., 14\}$ x, y $\in$ $\{1, .., 9\}$ Digits : $1ccnnxy$ Range : 1010111 to 1181499

Table 4.25: Variables, Nurikabe Sudoku rules.

Notation:		
$\mathcal{A}$	$=$	$\{(x,y) \mid (x,y)$ is grid cell that contains an arrow$\}$
$\mathcal{B}(x,y)$	$=$	$\{(x+1,y),(x-1,y),(x,y+1),(x,y-1)\}$
$\mathcal{G}$	$=$	$\{(a,b) \mid a,b \in \{1,2,3,4,5,6,7,8,9\}\}$
$\mathcal{F}_2(d,x,y,x_s,y_s)$	$=$	$\neg\mathcal{O}(x_s,y_s) \vee \bigvee_{(x',y'):\mathcal{B}(x,y)} \mathcal{F}(d-1,x_s,y_s,x',y') \;\; s.t. \;\; (x',y') \in \mathcal{G}$
$\mathcal{F}_3(d,n,x,y,x_s,y_s)$	$=$	$\neg\mathcal{I}(n,x_s,y_s) \vee \bigvee_{(x',y'):\mathcal{B}(x,y)} \mathcal{W}(d-1,n,x_s,y_s,x',y') \;\; s.t. \;\; (x',y') \in \mathcal{G}$
$\eta(c,x,y)$	$=$	$\begin{cases} ((x,y-1),(x,y-2)) & c=1 \\ ((x,y-1),(x+1,y-1)) & c=2 \\ ((x+1,y),(x+1,y-1)) & c=3 \\ ((x+1,y),(x+2,y)) & c=4 \\ ((x+1,y),(x+1,y+1)) & c=5 \\ ((x,y+1),(x+1,y+1)) & c=6 \\ ((x,y+1),(x,y+2)) & c=7 \\ ((x,y+1),(x-1,y+1)) & c=8 \\ ((x-1,y),(x-1,y+1)) & c=9 \\ ((x-1,y),(x-2,y)) & c=10 \\ ((x-1,y),(x-1,y-1)) & c=11 \\ ((x,y-1),(x-1,y-1)) & c=12 \\ ((x,y-1),(x+1,y)) & c=13 \\ ((x,y-1),(x,y+1)) & c=14 \\ ((x,y-1),(x-1,y)) & c=15 \\ ((x+1,y),(x,y+1)) & c=16 \\ ((x-1,y),(x+1,y)) & c=17 \\ ((x-1,y),(x,y+1)) & c=18 \end{cases}$

Table 4.26: Notations, Nurikabe Sudoku rules.

$PBC_O(x,y,z)$ = Set of clauses needed to encode that in the direction where the arrow of cell (x,y) points, there are z cells that are part of the ocean.

$$\text{LHS} = \begin{cases} \sum\limits_{\substack{k\in\{1,\ldots,8\}\\ k<y}} 1 * \mathcal{O}(x,k) & (x,y) \text{ contains} \uparrow \\ \sum\limits_{k=y+1}^{9} 1 * \mathcal{O}(x,k) & (x,y) \text{ contains} \downarrow \\ \sum\limits_{\substack{k\in\{1,\ldots,8\}\\ k<x}} 1 * \mathcal{O}(k,y) & (x,y) \text{ contains} \leftarrow \\ \sum\limits_{k=x+1}^{9} 1 * \mathcal{O}(k,y) & (x,y) \text{ contains} \rightarrow \end{cases}$$

$$\text{RHS} = z$$

$PBC_I(x,y,z)$ = Set of clauses needed to encode that in the direction where the arrow of cell (x,y) points, there are z cells that are not part of the ocean.

$$\text{LHS} = \begin{cases} \sum\limits_{\substack{k\in\{1,\ldots,8\}\\ k<y}} 1 * \neg\mathcal{O}(x,k) & (x,y) \text{ contains} \uparrow \\ \sum\limits_{k=y+1}^{9} 1 * \neg\mathcal{O}(x,k) & (x,y) \text{ contains} \downarrow \\ \sum\limits_{\substack{k\in\{1,\ldots,8\}\\ k<x}} 1 * \neg\mathcal{O}(k,y) & (x,y) \text{ contains} \leftarrow \\ \sum\limits_{k=x+1}^{9} 1 * \neg\mathcal{O}(k,y) & (x,y) \text{ contains} \rightarrow \end{cases}$$

$$\text{RHS} = z$$

(As we only defined PBCs for positive literals, we must in practice introduce an additional variable s_v for each $\neg\mathcal{O}(x,y)$. The corresponding variable s_v is defined to be true iff $\neg\mathcal{O}(x,y)$ is true (iff $\mathcal{O}(x,y)$ is false).)

Table 4.27: PBCs, Nurikabe Sudoku rules.

$$\bigwedge_{x=1}^{8} \bigwedge_{y=1}^{8} \neg\mathcal{O}(x,y) \vee \neg\mathcal{O}(x+1,y) \vee \neg\mathcal{O}(x,y+1) \vee \neg\mathcal{O}(x+1,y+1) \qquad \text{(NK-i)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{k=y+1}^{9} \neg\mathcal{I}(n,x,y) \vee \neg\mathcal{I}(n,x,k) \vee \neg s_{x,y,z} \vee \neg s_{x,k,z} \qquad \text{(NK-ii)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{k=x+1}^{9} \bigwedge_{l=1}^{9} \neg\mathcal{I}(n,x,y) \vee \neg\mathcal{I}(n,k,l) \vee \neg s_{x,y,z} \vee \neg s_{k,l,z} \qquad \text{(NK-iii)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{8} \bigwedge_{y=1}^{9} \bigwedge_{\substack{k=1\\k\neq n}}^{14} \neg\mathcal{I}(n,x,y) \vee \neg\mathcal{I}(k,x+1,y) \qquad \text{(NK-iv)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{\substack{k=1\\k\neq n}}^{14} \neg\mathcal{I}(n,x,y) \vee \neg\mathcal{I}(k,x,y+1) \qquad \text{(NK-v)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \neg\mathcal{I}(n,x,y) \vee \bigvee_{\substack{c=1\\((x',y'),(x'',y''))=\eta(c,x,y)}}^{18} \mathcal{N}(c,n,x,y) \qquad \text{s.t.}\substack{(x'',y'')\in\mathcal{G}\\(x',y')\in\mathcal{G}} \qquad \text{(NK-vi)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{\substack{c=1\\((x',y'),(x'',y''))=\eta(c,x,y)}}^{18} \neg\mathcal{N}(c,n,x,y) \vee \mathcal{I}(n,x',y') \qquad \text{s.t.}\substack{(x'',y'')\in\mathcal{G}\\(x',y')\in\mathcal{G}} \qquad \text{(NK-vii)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{\substack{c=1\\((x',y'),(x'',y''))=\eta(c,x,y)}}^{18} \neg\mathcal{N}(c,n,x,y) \vee \mathcal{I}(n,x'',y'') \qquad \text{s.t.}\substack{(x'',y'')\in\mathcal{G}\\(x',y')\in\mathcal{G}} \qquad \text{(NK-viii)}$$

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \mathcal{O}(x,y) \vee \bigvee_{n=1}^{14} \mathcal{I}(n,x,y) \qquad \text{(NK-ix)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \neg\mathcal{O}(x,y) \vee \neg\mathcal{I}(n,x,y) \qquad \text{(NK-x)}$$

$$\bigwedge_{n=1}^{14} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{k=n+1}^{14} \neg\mathcal{I}(n,x,y) \vee \neg\mathcal{I}(k,x,y) \qquad \text{(NK-xi)}$$

Table 4.28: Formulae-1 of clauses, Nurikabe Sudoku rules.

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\neg\mathcal{O}(n,x_s,y_s)\vee\neg\mathcal{O}(n,x,y)\vee\bigvee_{d=1}^{49}\digamma(d,x_s,y_s,x,y) \tag{NK-xii}$$

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\bigwedge_{d=1}^{49}\bigwedge_{k=d+1}^{49}\neg\digamma(d,x_s,y_s,x,y)\vee\neg\digamma(k,x_s,y_s,x,y) \tag{NK-xiii}$$

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\bigwedge_{d=1}^{49}\mathcal{O}(x,y)\vee\neg\digamma(d,x_s,y_s,x,y) \tag{NK-xiv}$$

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\neg\mathcal{O}(x,y)\vee\digamma(1,x,y,x,y) \tag{NK-xv}$$

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\neg\digamma(1,x_s,y_s,x,y)\quad\text{with }(x,y)\neq(x_s,y_s) \tag{NK-xvi}$$

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{\substack{y_s=1\\(x,y)\neq(x_s,y_s)}}^{9}\bigwedge_{d=2}^{49}\neg\digamma(d,x_s,y_s,x,y)\vee\mathcal{F}_2(d,x,y,x_s,y_s) \tag{NK-xvii}$$

$$\bigwedge_{n=1}^{14}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\neg\mathcal{I}(n,x,y)\vee\mathcal{W}(1,n,x,y,x,y) \tag{NK-xviii}$$

$$\bigwedge_{n=1}^{14}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{\substack{y_s=1\\(x,y)\neq(x_s,y_s)}}^{9}\neg\mathcal{W}(1,n,x_s,y_s,x,y) \tag{NK-xix}$$

$$\bigwedge_{n=1}^{14}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{\substack{y_s=1\\(x,y)\neq(x_s,y_s)}}^{9}\bigwedge_{d=2}^{9}\neg\mathcal{W}(d,n,x_s,y_s,x,y)\vee\mathcal{F}_3(d,n,x,y,x_s,y_s) \tag{NK-xx}$$

$$\bigwedge_{n=1}^{14}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\neg\mathcal{I}(n,x_s,y_s)\vee\neg\mathcal{I}(n,x,y)\vee\bigvee_{d=1}^{9}\mathcal{W}(d,n,x_s,y_s,x,y) \tag{NK-xxi}$$

$$\bigwedge_{n=1}^{14}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\bigwedge_{d=1}^{9}\bigwedge_{k=d+1}^{9}\neg\mathcal{W}(d,n,x_s,y_s,x,y)\vee\neg\mathcal{W}(k,n,x_s,y_s,x,y) \tag{NK-xxii}$$

$$\bigwedge_{n=1}^{14}\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{x_s=1}^{9}\bigwedge_{y_s=1}^{9}\bigwedge_{d=1}^{9}\mathcal{I}(n,x,y)\vee\neg\mathcal{W}(d,n,x_s,y_s,x,y) \tag{NK-xxiii}$$

Table 4.29: Formulae-2 of clauses, Nurikabe Sudoku rules.

$$\bigwedge_{(x,y):\mathcal{A}} \bigwedge_{z=1}^{9} \bigwedge_{\varphi \in PBC_O(x,y,z)} \neg\mathcal{O}(x,y) \vee \neg s_{x,y,z} \vee \varphi \qquad \text{(NK-xxiv)}$$

$$\bigwedge_{(x,y):\mathcal{A}} \bigwedge_{z=1}^{9} \bigwedge_{\varphi \in PBC_I(x,y,z)} \mathcal{O}(x,y) \vee \neg s_{x,y,z} \vee \varphi \qquad \text{(NK-xxv)}$$

$\Box$ if the grid contains an arrow that points directly outside the grid. (NK-xxvi)

Table 4.30: Formulae-3 of clauses, Nurikabe Sudoku rules.

5 Experiments

This chapter elaborates on how the different encodings perform and how the different Sudoku Variants compare. The program to encode the various puzzle instances is written in Java. We also use Java to call the different solvers (introduced in 2.2.3) and to log the resulting data. All experiments are run on a personal computer using Windows 10. The PC has an AMD Ryzen 7 5800X 8-Core Processor (3.80 GHz) and 32 GB of RAM (from which the Java Virtual Machine (JVM) is allowed to use 28GB).

The configurations we test for each puzzle instance differ by:

- The used Solver: Sat4j or MiniSat
- The used PBC-Encoding: Binary Decision Diagrams or Adder Networks
- The used optimization level: what is encoded as PBC (only for Killer Sudoku)

Conducting runtime experiments on a personal computer using Java comes with the caveat that the reported times all include some noise because their executions may be affected by other processes running in the background or by JVM events like garbage collections. To mediate that, we run 60 instances for every experiment configuration and report the average runtime and corresponding standard deviations.

Early testing showed that the solving times depend heavily on the order in which clauses and literals are given to the solvers. This is critical because we store clauses in sets and treat them as sets themselves. So the order of clauses (and literals) is not strictly defined and can vary from run to run. To still obtain comparable results, we order the clauses by length and the literals by the absolute values that encode them. We arbitrarily decided to pass the clauses containing the most literals first and to sort the literals within a clause by increasing absolute values. This way, the solvers get the same encoding as input if a problem is encoded and solved multiple times. However, eliminating this additional noise comes at the cost of having to sort all the clauses after they are created.

Because we use a relatively small test set of different puzzle instances, it is important to note that assumptions made based on insights from the conducted experiments may not be applied to a general case. The reason for the limited size of the test set is that in the case of puzzles from CTCGH, they are very exotic and often unique. For Killer Sudokus Puzzles, many instances can be found, but writing the puzzle down into a format that the program can work on is very time-consuming as it has to be done manually.

5.1 Adder Networks vs. Binary Decision Diagrams

To analyze the Encoding of PBCs, we conducted further experiments using Killer Sudoku instances from the books “The Times Ultimate Killer Su Doku Book 14”[17] and “The Times Killer Su Doku (Book 18)”[16] both of which are the latest collections of Killer Sudoku Puzzles arranged by “The Times”. In these books, puzzles are further categorized into categories of hardness, which reach from the easiest, “Moderate”, to “Extra Deadly”, the hardest. With ten instances from each of the categories, “Moderate” and “Extra Deadly”, we can compare how Binary Decision Diagrams and Adder Networks perform when encoding PBCs.

As Plot 5.1 shows, there is a strong correlation between the number of clauses and the time needed to encode if BDDs are used. In the case of Adder Networks (Plot 5.2), it is harder to make such a statement, as some instances with few clauses require more time to encode compared to instances with fewer clauses. Furthermore, in this case, the variance in clauses needed by the instances is relatively low. So to reliably analyze the relation, additional samples with higher clause numbers would be necessary.

Looking at the ratio of solving time per clause (shown in Plots 5.3 and 5.4), it seems that in both cases, the variance in solving time between the different instances seems to increase the more clauses are needed to encode them.

Finally, when comparing the solving times (Plot 5.6), one can see that neither encoding with BDDs nor encoding with Adder Networks gives a clear advantage, as it depends on the puzzle instance, which method leads to a faster solving process. However, using Adder Networks has the advantage of being much faster in the encoding part itself, which can be seen in Plot 5.5.

Additionally, as one can see in the corresponding plots of this section, there also seems to be a clear trend that the Sudoku instances of the harder “Extra Deadly” category demand more clauses to encode them. However, this does not necessarily mean that they take longer to encode, or at least not if Adder Networks are used (see 5.2). If it comes to the time needed by the solver, it may seem intuitive that puzzles that are hard to solve for humans also take longer to be solved by a SAT-solver. Surprisingly this assumption which is by no means trivially given seems to be correct, as Plot 5.6 shows.

5.2 Optimization of Killer Sudoku Encoding

Using the same Killer Sudoku Instances as explained in the previous section and Adder Networks to encode PBCs, we can compare the encoding and solving times of the two optimization ideas introduced in 4.4 to the standard approach of using PBCs. Plots 5.7 and 5.9 show that only generating the PBC for possible value combinations given the cages size and target sum leads to a measurable but neglectable performance gain. However, encoding Killer Sudokus completely without using PBCs can result in large savings, especially in solving time (see 5.8 and 5.10).

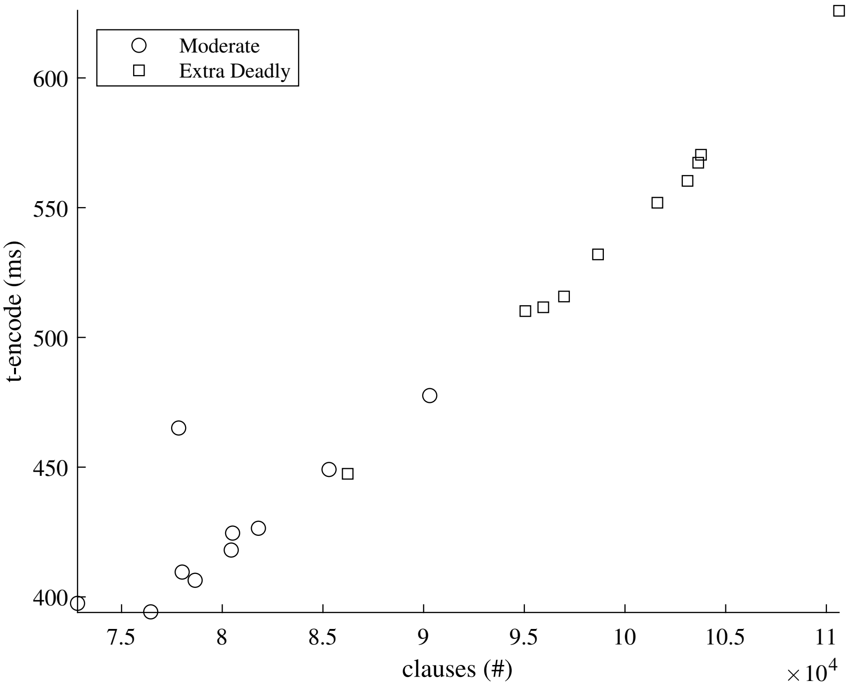
5.3 CTCGH Sudoku Variants compared using Sat4j and MiniSat

Table 5.1 shows the outcome of encoding and solving thirteen different Sudoku instances with Sat4j and MiniSat. For puzzles where PBCs were needed, Adder Networks were used to encode them. The shown puzzle instances are from CTCGH and combine different variants and rules, except for “Sudoku Man Of Mystery” and “The Road To Genius”, where only the normal Sudoku rules apply. An overview of which rules are present in each instance is given in Table 5.2.

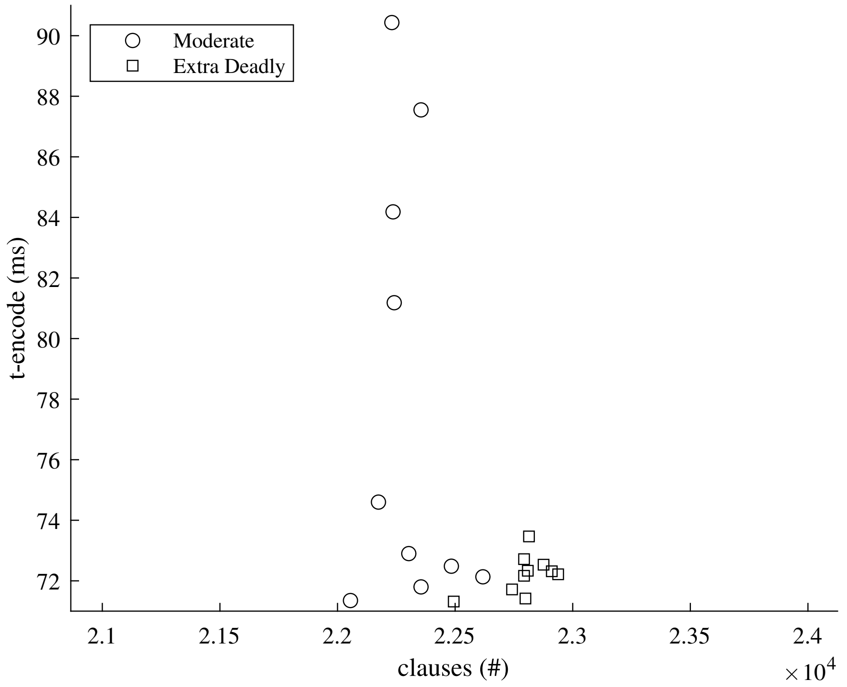
Studying Table 5.1, one can see that in most cases the solving times using MiniSat are significantly higher than the ones of Sat4j. This may have to do with the fact that the Sat4j library and its functions can be accessed directly by the Java program, whereas the MiniSat-Solver is called as an external executable. For all further experiments, we are only using Sat4j (which internally also contains a version of MiniSat). There are two exceptions to Sat4j always being faster. “The Original Sandwich” and “Nurikabe Sudoku” seem to perform much better with MiniSat, which we have no direct explanation for. As shown in Table 5.1, those are the puzzle instances that demand the highest number of clauses and variables, which also means they potentially require more memory (we measured usage of up to 27 GB in the case of “Nurikabe Sudoku”). Therefore, we think the external MiniSat executable might gain an advantage because of more efficient memory management than the Java-based solver. However, we do not have enough data to prove this claim, so MiniSat may prevail for other puzzle instances we did not test. Because the solving times for “Nurikabe Sudoku” using Sat4j are significantly higher, only ten runs were conducted instead of 60 like for the other puzzle instances.

When analyzing the constraints of “Nurikabe Sudoku”, we found that the high number of clauses needed is mainly caused by the constraints ensuring that the ocean and every island is continuous. The Formulae NK-xii, NK-xiii, NK-xiv, NK-xxi, NK-xxii and NK-xxiii produce more than 12 million clauses alone, which is almost ten times the number all other instances require combined.

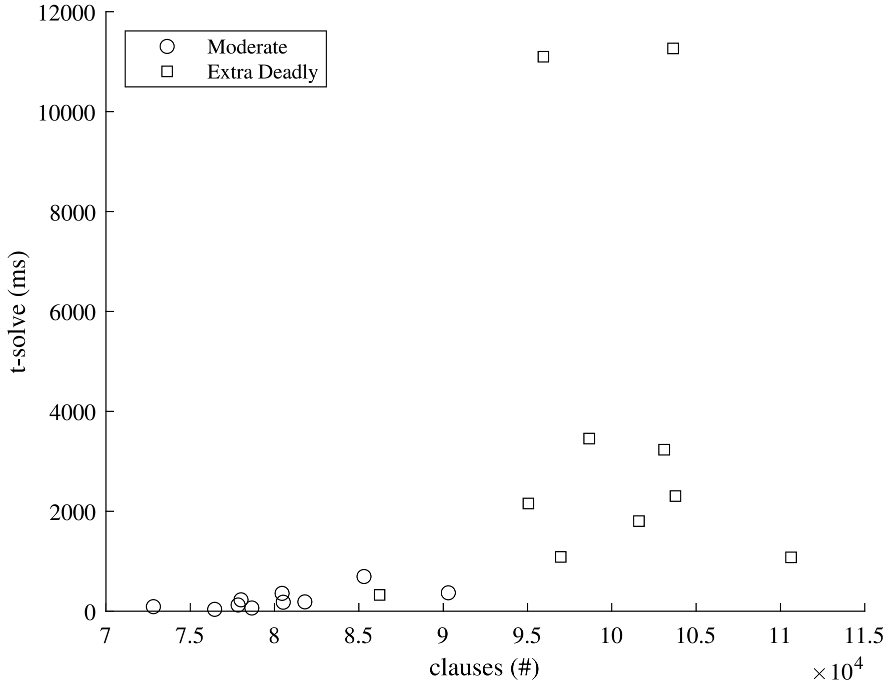
Comparing encoding to solving times (using Sat4j), it turns out that for instances without PBCs, the encoding times are all higher than the solving times, which does not hold if PBCs are involved, where it differs from instance to instance which part takes longer.



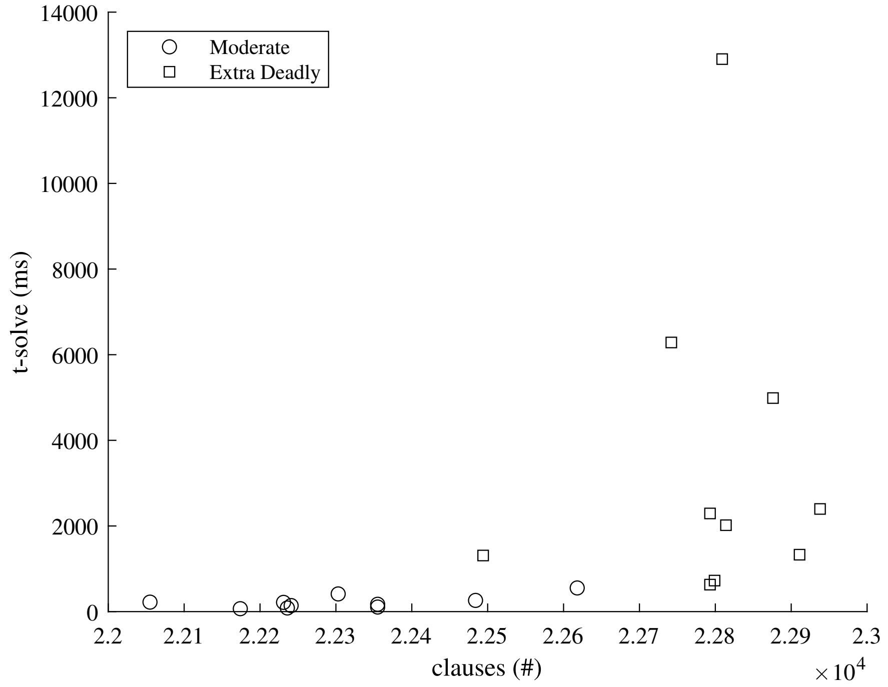
Plot 5.1: Encoding time per clause, using Binary Decision Diagrams



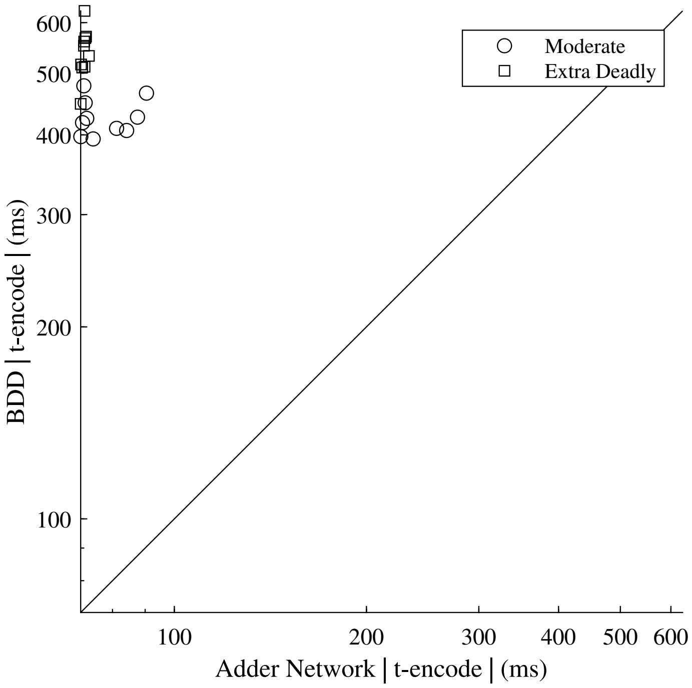
Plot 5.2: Average encoding time per clause, using Adder Networks



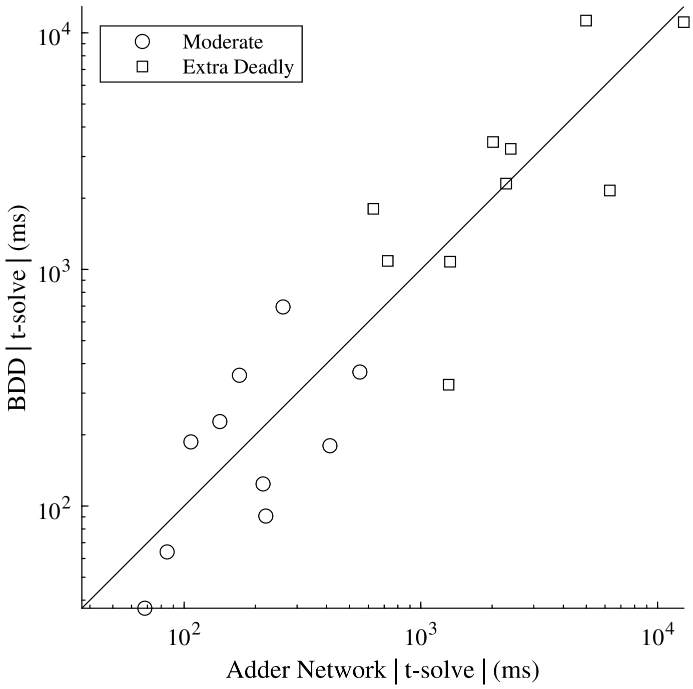
Plot 5.3: Average solving time per clause, using Binary Decision Diagrams



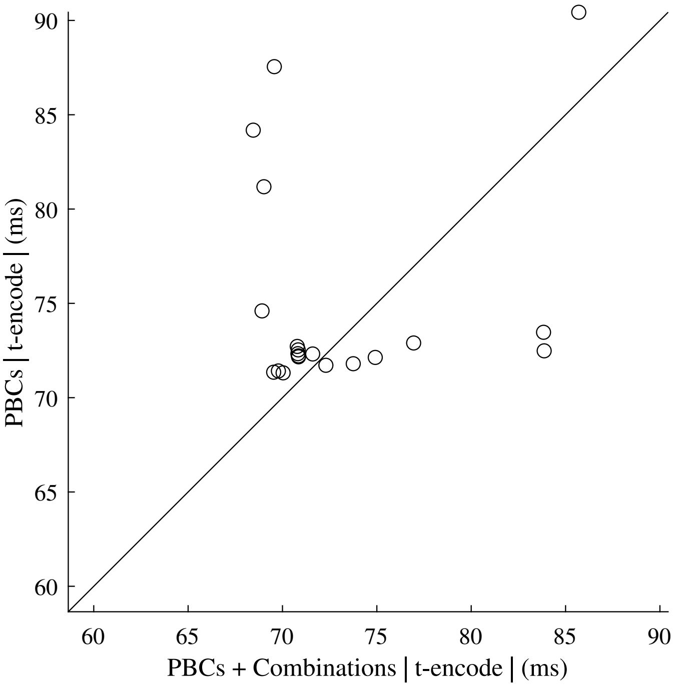
Plot 5.4: Average solving time per clause, using Adder Networks



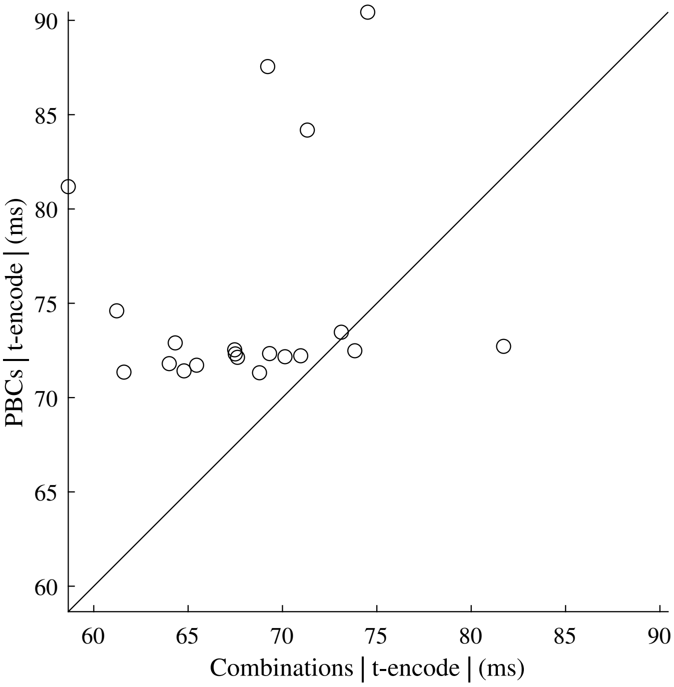
Plot 5.5: Average encoding time comparison, Binary Decision Diagrams vs. Adder Networks



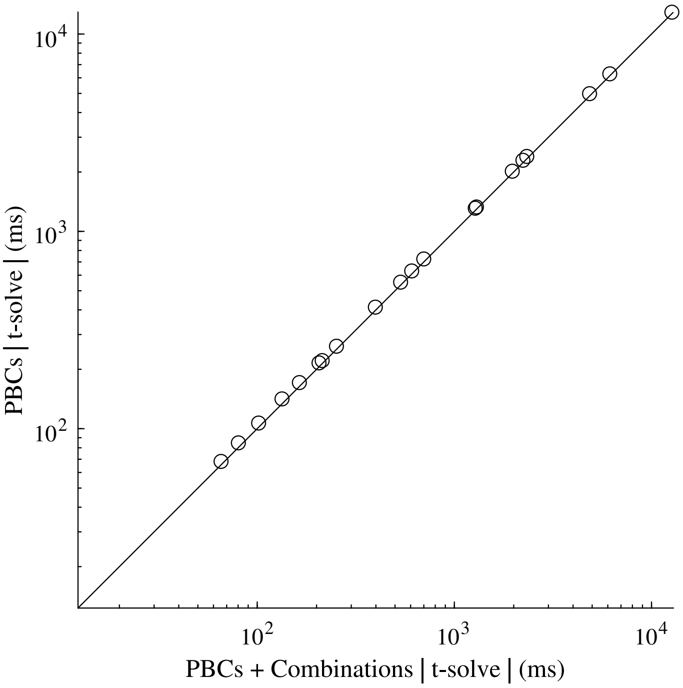
Plot 5.6: Average solving time comparison, Binary Decision Diagrams vs. Adder Networks



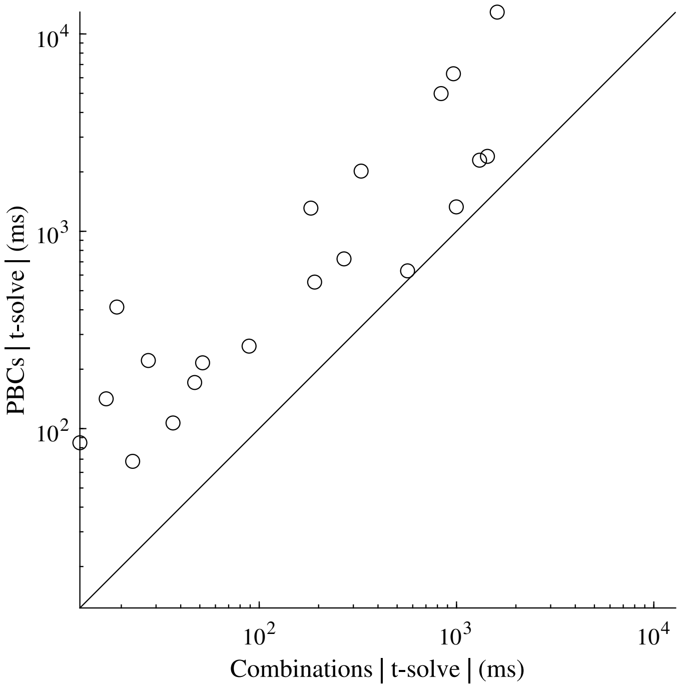
Plot 5.7: Average encoding time comparison, PBCs vs. PBCs + Combinations, using Adder Networks



Plot 5.8: Average encoding time comparison, PBCs vs. Combinations, using Adder Networks



Plot 5.9: Average solving time comparison, PBCs vs. PBCs + Combinations, using Adder Networks



Plot 5.10: Average solving time comparison, PBCs vs. Combinations, using Adder Networks

Variant	t-avg. (ms)			t-std. (ms)			#clauses	#variables
	encode	solve		encode	solve			
		Sat4j	MiniSat		Sat4j	MiniSat		
9 Marks The Spot	4428.02	170.90	1179.53	211.03	36.76	519.18	727636	5994
Chess Sudoku	29.05	3.23	1061.05	1.10	0.43	241.94	8912	729
Fawlty Towers	65.38	11.10	904.62	12.31	0.71	127.42	17632	2186
Frozen Picnic	168.32	14.92	886.33	3.99	0.81	51.47	40519	5387
Mark 1	28.60	8.18	903.05	2.12	0.50	52.05	8839	729
Nurikabe Sudoku	77406.68	4145268.40	413259.82	3341.18	536495.31	5633.74	13904145	1169013
Sudoku Man Of Mystery	23.18	1.97	876.67	0.93	0.71	6.85	7399	729
The Miracle Thermo	739.77	446.32	1671.62	261.78	24.74	112.12	138743	5265
The Original Sandwich	1677.18	15408.00	2059.25	26.28	384.06	12.22	302640	42635
The Pyramid	48.68	219.07	1434.53	2.23	3.46	3.36	15233	1679
Thermo 2020	24.23	1.97	1355.67	1.09	0.66	54.23	7659	729
Thermo Couples	27.98	3.80	1354.52	0.85	0.55	8.19	8884	729
Thermo Squares	23.92	17.95	1362.25	0.74	0.67	54.87	7692	729
The Road To Genius	22.95	1.97	1364.20	0.81	0.74	56.83	7401	729

Table 5.1: CTCGH Sudokus, solving with Sat4j and MiniSat, PBC-Encoding with Adder Networks

Instance | Rule	Normal Sudoku	Anti-Knight	Killer Sudoku	Arrowheads	Thermometers	Sandwich Sum	Secret Direction	Fawlty Towers	Nurikabe Sudoku
9 Marks The Spot	×						×		
Chess Sudoku	×	×			×				
Fawlty Towers	×							×	
Frozen Picnic	×			×	×	×			
Mark 1	×	×			×				
Nurikabe Sudoku	×								×
Sudoku Man Of Mystery	×								
The Miracle Thermo	×				×				
The Original Sandwich	×					×			
The Pyramid	×	×	×						
Thermo 2020	×				×				
Thermo Couples	×	×			×				
Thermo Squares	×				×				
The Road To Genius	×								

Table 5.2: Sudoku Puzzle instances and their rules.

6 Conclusion

This thesis demonstrated how different variants and rules of Sudoku Puzzles could be encoded as sets of logical clauses. We have seen that in most cases, SAT-solvers can find assignments that satisfy these sets of clauses in a relatively short time. We also found that solving Sudokus is by no means a trivial problem as the number of clauses and the time needed to solve "Nurikabe Sudoku" have shown. We have elaborated in detail on how Pseudo-Boolean Constraints can be used to encode constraints regarding sums, like in Killer Sudokus or for the Sandwich Sum rules. Specifically for Killer Sudoku instances, we found that both shown encoding methods Adder Networks and Binary Decision Diagrams (both proposed by [5]) have comparable performance, but that Adder Networks produce fewer clauses and are encoding the PBCs faster. Additionally, we have shown that encoding Killer Sudokus without PBCs can significantly reduce the time needed to solve them.

In future work, the proposed encoding methods for Sudoku rules could help to craft new puzzle instances of exotic variants like "9 Marks The Spot" or "The Miracle Thermo". Further, we came across multiple engaging questions for which we only estimated an answer or gave an upper bound, like what is the highest possible number of islands in "Nurikabe Sudoku" or how long can a hidden path in "9 Marks The Spot" be at maximum. Also teasing: CTCGH [10] contains many more Sudoku variants which could be analysed and encoded for SAT-solvers.

During our work, we only used a limited test set of puzzle instances, which all had to be rewritten by hand into a format our program could understand. In general, there seem to be no common test sets for non-original Sudoku Variants like Killer Sudoku, which may also has to do with the lack of a common file format that could be used to share more exotic puzzle variants. To facilitate future work, it would be desirable to agree on a standard file format specifically tailored for Sudokus. This would allow the puzzle and research community to build an openly available and directly usable database of Sudoku Puzzles, which would make the testing and analysis of new encoding and solving algorithms more reliable.

Bibliography

[1] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *ACM Symposium on Theory of Computing, STOC 1971*, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. ISBN 978-1-450-37464-4. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

[2] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, jul 1962. ISSN 0001-0782. doi: 10.1145/368273.368557. URL https://doi.org/10.1145/368273.368557.

[3] Niklas Eén and Niklas Sörensson. MiniSat. http://minisat.se/MiniSat.html. Accessed: 2022-05-14.

[4] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24605-3.

[5] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006. doi: 10.3233/sat190014. URL https://doi.org/10.3233/sat190014.

[6] Bertram Felgenhauer and Frazer Jarvis. Mathematics of Sudoku I. *Mathematical Spectrum*, 39:15–22, 2006.

[7] Ian P. Gent. Arc Consistency in SAT. In Frank van Harmelen, editor, *European Conference on Artificial Intelligence, ECAI 2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[8] Mark Goodliffe and Simon Anthony. Youtube Channel: Cracking The Cryptic. https://www.youtube.com/c/CrackingTheCryptic, . Accessed: 2022-07-15.

[9] Mark Goodliffe and Simon Anthony. Online solution for Sudoku Puzzle "THERMO 2020". http://ctcbook.com/gh015/, . Accessed: 2022-07-18.

[10] Mark Goodliffe and Simon Anthony. *Cracking The Cryptic Greatest Hits*. Coffeebean, 2021. ISBN 978-1-954958-00-5.

[11] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990. ISSN 0004-3702. doi: https://

doi.org/10.1016/0004-3702(90)90009-O. URL https://www.sciencedirect.com/science/article/pii/000437029090009O.

[12] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[13] Inês Lynce and Joël Ouaknine. Sudoku as a SAT Problem. In *International Symposium on Artificial Intelligence and Mathematics, AI&Math 2006, Fort Lauderdale, Florida, USA, January 4-6, 2006*, 2006. URL http://anytime.cs.umass.edu/aimath06/proceedings/P34.pdf.

[14] Gary McGuire, Bastian Tugemann, and Gilles Civario. There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. *Experimental Mathematics*, 23(2):190–217, 2014. doi: 10.1080/10586458.2013.870056. URL https://doi.org/10.1080/10586458.2013.870056.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, 4th Global ed.* Pearson, KAO Two KAO Park Hockham Way Harlow CM17 9SR United Kingdom, 2022. ISBN 978-1-292-40117-1.

[16] Puzzler Media Times Books. *The Times Killer Su Doku Book 18.* Times Books, HarperCollins Publishers, Westerhill Raod Bishopbriggs, Glasgow G64 2QT 02210, Scotland, 2022. ISBN 978-0-00-847276-4.

[17] Puzzler Media Times Books. *The Times Ultimate Killer Su Doku Book 14.* Times Books, HarperCollins Publishers, Westerhill Raod Bishopbriggs, Glasgow G64 2QT 02210, Scotland, 2022. ISBN 978-0-00-847268-9.

[18] Artois University and CNRS. Sat4j. https://www.sat4j.org/index.php. Accessed: 2022-06-09.

[19] Schöning Uwe. *Logic for Computer Scientists.* Birkhäuser, c/o Springer Science+Business Media LLC, 233 Spring Street Boston, New York, NY 10013, USA, 2008. ISBN 978-0-8176-4763-6.

[20] Toby Walsh. SAT v CSP. In Rina Dechter, editor, *Principles and Practice of Constraint Programming, CP 2000*, pages 441–456, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45349-9.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Encoding Diverse Sudoku Variants as SAT Problems

Name Assesor: Prof. Dr. Malte Helmert

Name Student: Sebastian Schlachter

Matriculation No.: 2017-927-534

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Möhlin, 20.07.22 Student: ______________________

Will this work be published?

□ No

☒ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: ______________________

Place, Date: ______________________ Student: ______________________

Place, Date: ______________________ Assessor: ______________________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .