UTRECHT UNIVERSITY

BACHELOR THESIS
BSC ARTIFICIAL INTELLIGENCE

How to Solve a Sudoku

A logical analysis and algorithmic implementation of strategically Solving Sudoku puzzles

Author:
Daan Luca DI SCALA

Student Number:
5937922

Supervisor:
Prof. dr. ir. Jan BROERSEN

Second reader:
Aleksei NAZAROV

A 7.5 ECTS Thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in

Artificial Intelligence

at the

Faculty of Humanities

April 13, 2020

Abstract

Daan Luca Di Scala

How to Solve a Sudoku

Sudokus are widely popular logical combinatorial puzzles. From the Sunday newspaper to World Championships, many people like to take a crack at the world-famous puzzle. Many sudoku solvers exist, often based on efficiency and therefore not taking human thinking steps and strategies into account. Consequently, there is currently not a solver which can logically explain its steps. Something which would be useful to aid human players in their own solving process.

Therefore the aim of this thesis is to answer whether it is possible to build a transparent logical algorithm which solves sudoku's based on human strategies, by analysing the logic behind these strategies. And if this is the case, what would such a solver look like? Because of this the aim of the thesis is twofold, consisting of a logical analysis and an algorithmic implementation. The Sudoku puzzle is formalized as a Modal Logic Problem. To approximate how human puzzlers complete Sudokus, thirteen of the most popular Sudoku solving strategies are gathered and formalized as Alethic Natural Deduction rules and their dependency and complexity is analysed. Based on this *Helping HAND*, a Heuristic Alethic Natural Deduction based Sudoku solver, is proposed. *Helping HAND* searches for optimal strategies by Weighted Graph Search and can solve Sudokus while being able to explain each step along the way. This algorithm proves to be a valuable step in the field of explainable AI.

Contents

Chapter 1

Introduction

Sūdoku: ***Suuji wa dokushin ni kagiru***
"The digits must remain single"[36]

Sudokus are incredibly popular puzzles with a seemingly easy goal: fill in the cells in a 9×9 grid such that in each row, column and block of 9 squares the digits 1 through 9 only appear once. But looks can be deceiving! Since the difficulty of the puzzle is caused by a combination of the amount of clues and their placement, it can be tough to evaluate its solvability at first glance. Sometimes one simple solving strategy is enough to solve a sudoku while other sudokus require a lot more thought. The challenge of solving sudoku's is what attracts me and many people[1]. The wide range of difficulty in which sudokus exist makes them a fun Sunday morning pastime for newspaper readers as well as a tough challenge for World Championship contestants[33,34]. Besides the fun, puzzling purpose, sudokus also make for a fascinating research subject.

Because of the logical nature of the problem, there has been a lot of research done on efficient logical sudoku solver algorithms. One factor as to why this is an interesting research topic for some logicians, is because the generalized form of sudoku puzzles, being puzzles of size $n \times n$, is NP-complete[35]. However, because of its sheer amount of possible configurations – there are approximately $7 \cdot 10^{21}$ possible different Sudoku grids[6] of Classic Sudokus – simple search algorithms, such as naïve backtracking[9], tend to be lacking the necessary speed and strength. Ultimately, it is a very difficult task to solve sudokus, both for humans and computers.

Because of this, proposed solvers offer some kind of clever search heuristic in addition to backtracking. There have been papers proposing efficient searching solvers such as Forward Checking and Limited Discrepancy searches[3], Chaotic Harmony search[2], SAT-based search[32], Minigrid based backtracking[12] and more[11,13,14,18,20]. While many of these approaches are quite successful at efficiently solving many sudoku puzzles by a computer, they don't tend to grant us insight into how humans tackle these problems.

However, the aim of this paper is to investigate how we as human players solve sudoku puzzles. Because in comparison to computers, humans have to approach these puzzle differently. This is, among other reasons, because humans are not great at keeping track of everything that is going on, they lack certain omniscience about the puzzles. Yet, many skilled players can solve the toughest of sudoku puzzles. The way algorithms find a solution is too different from how humans do, which makes it useless in helping humans solve these problems.

Therefore, my goal is not to tackle the sudoku problem as efficiently as possible, but based on a logically sound, yet understandable manner. This way I attempt to create a useful and transparent system which could help someone understand how to solve a sudoku. In other words, I will be trying to answer the research question whether it is possible to build a transparent logical algorithm which solves sudoku's based on human strategies, by analysing the logic behind these strategies. And if this is the case, what would such a solver look like?

In this thesis I will propose a new algorithm, based on human puzzle solving strategies. I will formalize the puzzle as an Alethic Modal Logic problem. I will gather the sudoku solving strategies from popular forums where people come together to write about and introduce new steps to tackle these puzzles and from earlier set theoretical research. From these sources I will extract and formalise strategies with increasing levels of difficulty. I will analyse how these strategies relate to each other and from this I will write an algorithm that not only solves a given sudoku, but can show its work in such a way it can be understood and replicated by humans.

1.1 Artificial Intelligence and Societal Relevance

While other proposed algorithms are merely efficient in finding a solution, the solver this paper proposes can actually explain the steps it takes. The research in explainable AI and transparent algorithms, which can explain taken steps is an important topic right now in the field of AI. Many "Black Box" techniques do deliver great results but have both practical and ethical limitations[8]. As this paper attempts to combine human reasoning strategies, logic and computer science for this purpose, it can be a stepping stone for further research on Explainable AI.

Furthermore, as this research is based on analysing the structure of logical reasoning, it includes useful societal applications. This process of making a solver and creating a program which exists to support a person achieve their goal can be applied to other fields. From work schedule planning to deduction in the court of law, in all fields where logical reasoning is used to support important decision making, this principle of logical analysis and algorithmic implementation could be useful. This is why investigating whether a logical helper can be manufactured could be helpful and applicable to many other fields where thinking rationally and strategically is required.

1.2 Structure of this Paper

In Chapter 2 the sudoku problem and its rules are formalized in such a way that the strategies can be efficiently defined. Then, the sudoku solving strategies are formalized and from them Natural Deduction[7] rules are deduced. In Chapter 3 it is discussed how from this a Solver is written and its capabilities are tested. Afterwards, the results are discussed in Chapter 4. It will also be reflected upon whether such system can be classified as explainable Artificial Intelligence and its societal relevance in the AI field and other fields is discussed.

Chapter 2

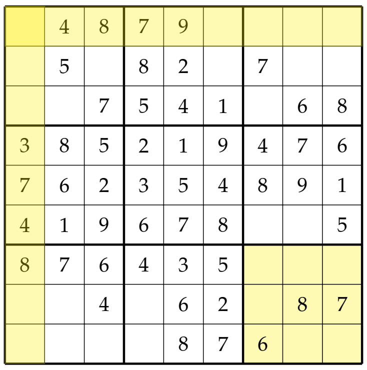
Formalisation

2.1 Theoretical Background

This section provides some background information on Sudoku Terminology (2.1.1), Alethic Modal Logic (2.1.2) and Natural Deduction (2.1.3), which will all be used throughout the rest of the thesis.

2.1.1 Sudoku Terminology

Before I can start with formally defining a Sudoku, some definitions on Sudoku are required. A *General Sudoku* is a $n \times n$ grid of squares, called *cells*. For a *Classic/Original Sudoku*, this n equals 9. Each *Proper Sudoku* only has one solution. From now on the term Sudoku is used to indicate a Classic, Proper Sudoku, unless otherwise stated. There will also sometimes be referred to the Sudoku as 'the puzzle' or 'the problem'.



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

FIGURE 2.1: Classic Sudoku Grid with highlighted Row, Column and Block

The grid of cells consists of n *rows/horizontals*, n *columns/verticals* and n $b \times b$ *boxes* of cells, with $b = \sqrt{n}$. These three different structures are called *houses/groups*. When generally talking about a column or row, *line* is used. In Classic Sudoku this means there are nine rows, nine columns and nine boxes (of size 3 by 3). See Figure 2.1.

There are n symbols which can be filled in the cells, called *digits/candidates*[1]. Each cell can contain only one digit. In the start state of the puzzle there is a certain amount of filled cells, which are called the *clues/givens*. Empty cells can be marked with the n optional digits to fill in. These markings are called *pencil marks*, often abbreviated to *PM's*. A completely filled Sudoku is called a *solution/goal*, when each house contains all digits exactly once.[17,24,29]

[1]As no calculation is involved in Sudokus, the usually used digits could be swapped with other symbols, such as letters or pictures.

2.1.2 Alethic Modal Logic

There are different ways of interpreting Modal Logic, which adds two unary modal operators to propositional logic: $\Box$ and $\Diamond$.[16] One of these interpretations is Alethic Logic, the logic of necessity and possibility[5,7]. In this logic the $\Box$ stands for 'necessarily' and the $\Diamond$ stands for 'possibly', respectively. From now on the terms Alethic (Modal) Logic and Modal Logic are used interchangeably, as this is the interpretation of modality used in the thesis. In Modal Logic, Kripke models are used to represent modality in terms of possible worlds[10]. A Kripke model is formally defined[16] as shown in Definition 2.1.1.

Definition 2.1.1 (Kripke Model).
A Kripke model is a tuple $M = \langle W, R, V \rangle$ such that

- W is a non-empty set of possible worlds,
- $R \subseteq (W \times W)$ is a binary relation on W
  If wRv we say that v is accessible from w.
- $V : W \to Pow(var)$ is a valuation for the set of atomic propositions var
  Proposition p is true in world w if $p \in V(w)$, and false in w if $p \notin V(w)$.

When it concerns such models, "$M, w \vDash p$" means "in model M, a certain proposition p is true in world w". As for the modal operators, "$M, w \vDash \Box p$" means that in model M, for all the worlds v reachable from w, a certain proposition p is true in world v. Or, in the Alethic interpretation, it is necessary that p. A similar rule applies to $\Diamond$, with "$M, w \vDash \Diamond p$" meaning that for at least one world v reachable from w, a certain proposition p is true in world v. Its Alethic interpretation is that it is possible that p.

2.1.3 Natural Deduction

I will be using Fitch style Natural Deduction[7,16], often shortened to ND, which is a deduction system for (modal) logic. These ND-rules are used to deduct a logical proof and usually look like what is shown in 2.2a. For the purpose of this thesis, the ND-rules I use will have a slightly different form, looking like the one in 2.2b. Both of these rules shown in 2.2 show a Natural Deduction rule, called Conjunction Introduction. While their meaning and use are similar, the form of 2.2b is used to state information about worlds in the model, which is useful when talking about the sudoku. The ND-rules used in the thesis differ from the original ND-rules in a second way, as they will often include accessibility relations –such as wRv– in its premises. This is to express information about the relation between worlds in the model.

$$\boxed{\begin{array}{l} \varphi \\ \psi \\ \dots \\ \varphi \wedge \psi \end{array}}$$

(A) Propositional $\wedge$I

$$\boxed{\begin{array}{l} M, w \vDash \varphi \\ M, w \vDash \psi \\ \dots \\ M, w \vDash \varphi \wedge \psi \end{array}}$$

(B) Modal $\wedge$I

FIGURE 2.2: $\wedge$I: Conjunction Introduction ND-rules. In 2.2a usual way. In 2.2b the modal way.

2.2 Formalisation of the Sudoku puzzle

To properly talk about the sudoku in a formal matter, I will now define an Alethic Kripke Model in such a way that it represents a Sudoku structure. This Model is pragmatically crafted in such a way that it correctly corresponds to the puzzle, which makes it an unusual Kripke Model, but a Kripke Model nonetheless. The formal definition of the Sudoku model, like in 2.1.1, is shown in Definition 2.2.1[2].

Definition 2.2.1 (Sudoku Kripke Model).
Our Sudoku model is a Kripke model $S = \langle W, R, D \rangle$ such that

- $W = C \cup P = \{c_1, ..., c_{81}\} \cup \{p_1, ..., p_9\}$, a union of C, the set of worlds representing Sudoku cells, and P, the set of possible pencil marks.
- $R = G \cup P = H \cup V \cup B \cup P$, a union of sets of different accessibility relations between these worlds, as defined in Definition 2.2.2.
- $D = \{1, ..., 9\}$ being a set of Digit valuations.
  Proposition d is true in world p if $d \in D(p)$, and false in p if $d \notin D(p)$.

The set of accessibility relations of this Kripke Model, R, is made up of four different sets of accessibility relations, which are called H, V, B, and P. These accessibility relations are explained in Definition 2.2.2.

Definition 2.2.2 (Accessibility Relations).

- $c_x G c_y$ marks a relation between two cells which are in the same Group, so either in the same Horizontal, Vertical or Block.
- $c_x L c_y$ marks a relation between two cells which are in the same Line, so either in the same Horizontal or in the same Vertical.
- $c_x H c_y$ marks a relation between two cells which are in the same Horizontal.
- $c_x V c_y$ marks a relation between two cells which are in the same Vertical.
- $c_x B c_y$ marks a relation between two cells which are in the same Block.
- $c_x P p_y$ marks a relation between a cell and a pencil mark.

Note that contrary to regular Kripke Models, where it would be required to have a powerset of valuations, now only the numbers 1 through 9 are needed as valuations. This is because –as by the rules of Sudoku– no world can contain more than one valuation. Because of this, instead of using $\vee$ (or), $\otimes$ (xor) is used. This is defined in 2.2.3.

Definition 2.2.3 (XOR).
$\varphi \otimes \psi$ means: either φ or ψ, but not both φ and ψ.

[2]Note that S is used instead of M, c_x / p_x instead of w, D instead of V and d instead of p. This is just to make it more clearly correspond to the terms *Sudoku*, *cells/PM's*, *digits* and *digit* respectively, but do not have any other impact.

Figure 2.3 shows a visual representation of the main part of the Kripke Model 2.2.1 of the sudoku. There are 81 worlds connected to each other through different accessibility relations. Each world corresponds to a cell in a sudoku. The different accessibility relations correspond to the horizontals (H), verticals (V) and blocks (B). Note that the accessibility relations shown in 2.3 are displayed in a transitive matter, which means that any cell that can access any other cell through one or multiple Group relations can also access each other through this group relation. For example, cell c_1 can reach c_{21} through a Block relation.

In logical terms, $c_x B c_y$ means that c_y is accessible from c_x by a Block relation, so c_x and c_y are in the same block. Also, $S, c_x \vDash \Box_V \varphi$ means that for all the worlds accessible from c_x by a V-relation, a formula φ holds, so in each of the cells in the same vertical a certain φ would be the case.

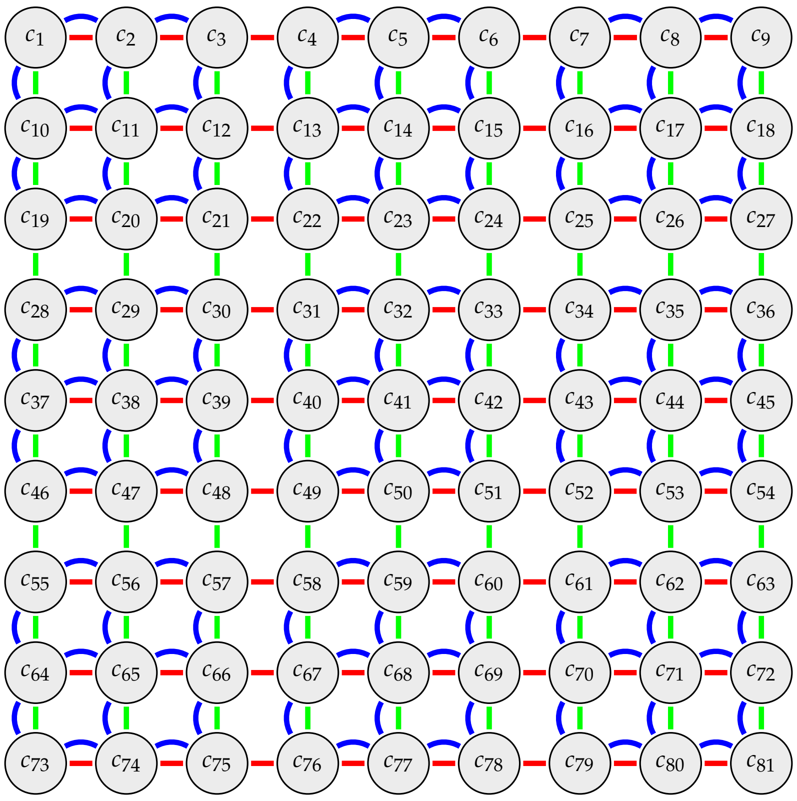


FIGURE 2.3: Sudoku represented as a Kripke Model. Blue lines represent transitive Block-relations, B. Red lines represent transitive Horizontal relations, H. Green lines represent transitive Vertical relations, V.

None of these cells contain any valuations, as filling in the cells works through Pencil Marks. Instead of considering each Pencil Mark as a direct proposition of each cell world, there will be referred to other worlds that contain these pencil marks valuations. This will be expressed through the use of another set of worlds, called Pencil mark worlds. This way of talking about it in terms of possibility and necessity is introduced.

Figure 2.4 shows how filling in the cells with Pencil Marks works. Note that, for clarity's sake, only one of the 81 cell worlds is shown. Not only can the sudoku cell worlds access each other, each cell world in the framework can potentially access one or multiple Pencil Mark worlds. As multiple options for a digit in a cell are considered, multiple accessibility relations between the two layers exist. Throughout the solving process, more and more pencil mark relations are being removed, until each cell only considers one pencil mark. In logical terms, when a certain PM, say 3, is considered to not be possible in a certain cell c, it would look like "$S, c \models \neg\Diamond_P 3$". This is because there is a direct correspondence between accessibility relation and possibility, as $c \not{P} p_d \leftrightarrow S, c \models \neg\Diamond_P d$. When there is no accessibility relation between cell c and pencil mark p_d (which holds proposition d) then it is not possible to find a d through a P-relation, and vice versa.

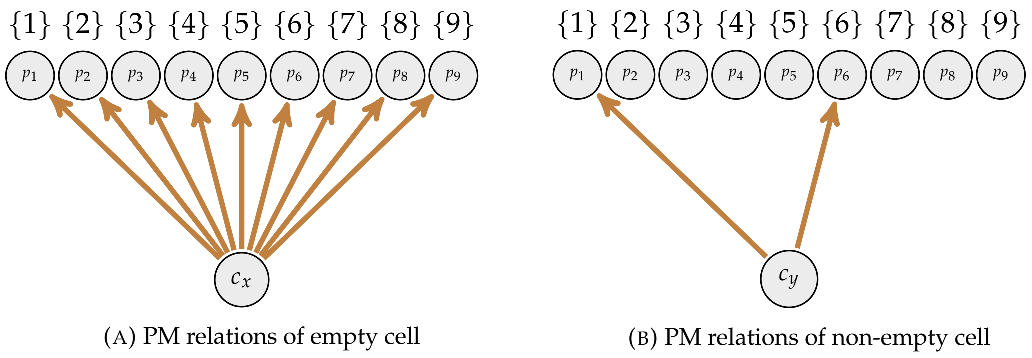

{1} {2} {3} {4} {5} {6} {7} {8} {9}
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9
c_x
(A) PM relations of empty cell
{1} {2} {3} {4} {5} {6} {7} {8} {9}
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9
c_y
(B) PM relations of non-empty cell

FIGURE 2.4: PM accessibility relations. 2.4a corresponds to a cell c_x where all digits are still considered possible. 2.4b corresponds to a cell c_y where only the digits 1 and 6 are being considered possible.

This leaves us with a proper way to discuss Sudoku's Pencil Marks. For example, it is now possible to say "$S, c_{21} \models \Box_P(3 \otimes 5)$", meaning "In our Sudoku Model S cell c_{21} necessarily contains either a 3 or a 5." or "$S, c_{81} \models \Box_H(\Diamond_P 6 \wedge \Diamond_P 8)$", meaning "In our Sudoku Model S, all the cells horizontally reachable from cell c_{81} possibly contain a 6 and possibly contain an 8".

The only thing left is to formally define the goal, which is done in Definition 2.2.4. This means that ultimately all the cells necessarily only contain one of the possible digits, which corresponds directly with the goal of Sudoku as described in 2.1.1.

Definition 2.2.4 (Sudoku Goal).
In our Model S, given its clues, eliminate as much possible pencil marks as possible, in such a way that: $\forall x \in \{1, \ldots, 81\}(S, c_x \models \Box_P d)$, with $d \in D$.

2.3 Formalisation of the Sudoku strategies

I have compiled 13 of the most popular strategies, from previous papers[4,23] and websites[25–28]. While there are many more strategies, I have made a selection based on which I deem to be the most useful and varied strategies, ranging in difficulty from implicit to hard. Many strategies are Tuple-based strategies, meaning that they exist for singles, pairs, triples, quads, etc. As shown in earlier research[23], anything above triples rarely shows up. Because of this, all of the strategies are either about singles, pairs or triples.

The compiled list of strategies can be seen in Table 2.1. Here, the strategies are sorted by their difficulty. In this paper, I assume previous research on sudoku difficulty is accurate. It is subjective whether a strategy is hard to perform, but it gives a good measure to base a categorisation on. Implicit strategies are strategies that are so easy that human players would perform them without thinking about it explicitly. One could even question whether these steps are even strategies at all, but as they are crucial to the strategy-based solving process, I deemed it necessary to include them into the list of strategies. Beginner strategies are your typical Sunday morning puzzler strategies (often well-known), while Intermediate and Advanced strategies are lesser known and more complicated. Strategy difficulty is, though, different than strategy complexity, which will be analysed in Section 2.4.1.

Difficulty	**Strategy**	**Abbreviation(s)**	**Subsection**
Implicit	Cell Based Elimination	CBE	(2.3.1)
Implicit	Pencil Mark Duality	PMD1, PMD2, PMD3	(2.3.2)
Beginner	House Based Elimination	HBE	(2.3.3)
Beginner	Last Remaining Cell	LRC	(2.3.4)
Beginner	Pencil Mark Introduction	PMI	(2.3.5)
Intermediate	Naked Tuples Elimination	N2E, N3E	(2.3.6)
Intermediate	Pointing Tuples Elimination	P2E, P3E	(2.3.7)
Advanced	Hidden Pairs Elimination	H2E	(2.3.8)
Advanced	X-Wing Elimination	XWE	(2.3.9)

TABLE 2.1: Strategies ordered by difficulty

In the following subsections (2.3.1 - 2.3.9) each strategy will be discussed. The way in which they work will be explained and ND-rules will be derived from them, based on earlier definitions (2.1 - 2.2). The strategies can get complicated rather quickly, so some of the more difficult formalizations will be accompanied by explanatory Figures.

Note that these ND-rules stand for the strategies, which are about cells and their digits. When a strategy concerns only one cell, this cell is called c. If it concerns multiple cells, they are referred to as c_1, c_2, c_3 etc. This is just to indicate that the strategy concerns multiple cells. It is left implicitly that $c_1 \neq c_2, c_2 \neq c_3$, etc. The same principle is used for the digits, only then with d instead of c.

2.3.1 Cell Based Elimination (Implicit)

$$\boxed{\begin{array}{l} S, c \vDash \Box_P d_1 \\ \dots \\ S, c \vDash \neg \Diamond_P d_2 \end{array}}$$

FIGURE 2.5: CBE: Cell Based Elimination ND-Rule

According to the rules of sudoku, each cell can only contain one digit (2.1.1). From this, an implicit strategy named Cell Based Elimination (CBE) follows directly: ”If a cell already contains a certain digit, it cannot contain another digit”. From this, a simple ND-rule follows, as shown in Figure 2.5. If d_1 must be the case in cell c, then it is not possible that d_2 in c.

2.3.2 Pencil Mark Duality (Implicit)

$$\boxed{\begin{array}{l} S, c \vDash \Diamond_P d_1 \\ S, c \vDash \neg \Diamond_P d_2 \wedge \dots \wedge \neg \Diamond_P d_9 \\ \dots \\ S, c \vDash \Box_P d_1 \end{array}}$$

(A) PMD1

$$\boxed{\begin{array}{l} S, c \vDash \Diamond_P d_1 \wedge \Diamond_P d_2 \\ S, c \vDash \neg \Diamond_P d_3 \wedge \dots \wedge \neg \Diamond_P d_9 \\ \dots \\ S, c \vDash \Box_P (d_1 \otimes d_2) \end{array}}$$

(B) PMD2

$$\boxed{\begin{array}{l} S, c \vDash \Diamond_P d_1 \wedge \Diamond_P d_2 \wedge \Diamond_P d_3 \\ S, c \vDash \neg \Diamond_P d_4 \wedge \dots \wedge \neg \Diamond_P d_9 \\ \dots \\ S, c \vDash \Box_P (d_1 \otimes d_2 \otimes d_3) \end{array}}$$

(C) PMD3

FIGURE 2.6: PMD: Pencil Mark Duality ND-Rules

A strategy less obvious than Cell Based Elimination is the duality of Pencil Marks. Generally speaking, with a total of n digits, if in a certain cell there are m PM’s possible and $n - m$ PM’s not possible, this means that the digit to fill in must be either one of the m Pencil Marks. One could see this as a matter of rephrasing, or as a way of describing the $\Box_P / \Diamond_P$-duality. In Figure 2.6 the ND-rules for PMD1 through PMD3 are shown. If a certain amount of d_x’s are possibly the case in cell c and all the other d_y’s are not possible in c, then either one of these d_x must be the case. It is evident that PMD1 is an edge case, where if only one PM is possible, this must be the digit to fill in this cell.

2.3.3 House Based Elimination (Beginner)

$$\boxed{\begin{array}{l} S, c_1 \vDash \Box_P d \\ c_1 G c_2 \\ \dots \\ S, c_2 \vDash \neg\Diamond_P d \end{array}}$$

FIGURE 2.7: HBE: House Based Elimination ND-Rule

Also, according to the rules of sudoku, each house can contain all digits exactly once (2.1.1). From this, an easy strategy named House Based Elimination (HBE) follows directly: "If a cell in a house already contains a digit, another cell in the same house cannot contain the same digit". From this, a simple ND-rule follows, as shown in Figure 2.7. If d must be the case in cell c_1 and c_1 is in the same house as c_2, then it is not possible that d in c_2. This is the first rule that contains an accessibility relation, as one of the premisses is that the cells can access each other through a House (Group) relation.

Note the similarity between this rule and Cell Based Elimination (2.3.1), the only difference being that this strategy adds a second cell in the same house. CBE could therefore also have been achieved by making each house relation not only transitive, but also reflexive. Then HBE would have sufficed to also find the same steps as CBE. This way is chosen, as it is deemed more intuitive that CBE would be its own rule.

2.3.4 Last Remaining Cell (Beginner)

$$\boxed{\begin{array}{l} S, c \vDash \Box_G \neg\Diamond_P d \\ S, c \vDash \Diamond_P d \\ \dots \\ S, c \vDash \Box_P d \end{array}}$$

FIGURE 2.8: LRC: Last Remaining Cell ND-Rule

Consequently, if each house must contain all digits exactly once (2.1.1), another easy strategy, named Last Remaining Cell (LRC), can be derived: "If all the other cells in the same house cannot possibly contain a certain digit, then this cell must contain that digit". From this, an ND-rule follows, as show in 2.8. If for every cell reachable from cell c through a house relation d is not possible, yet and for c it is the case that d is possible, then c must contain d. Note that $S, c \vDash \Box_G \dots$ is used as short-hand to state that "For every cell in a certain house reachable from cell c, … must hold".

2.3.5 Pencil Mark Introduction (Beginner)

$$\boxed{\begin{array}{l} S, c \vDash \Box_H \neg \Box_P d \\ S, c \vDash \Box_V \neg \Box_P d \\ S, c \vDash \Box_B \neg \Box_P d \\ \dots \\ S, c \vDash \Diamond_P d \end{array}}$$

FIGURE 2.9: PMI: Pencil Mark Introduction ND-Rule

To be able to place a new Pencil Mark in a cell, one must use the Pencil Mark Introduction (PMI) strategy. If each house can contain all digits only once (2.1.1), it must be so that "If in none of the three houses of a cell another cell must contain a digit, this cell can possibly contain this digit". From this, an ND-rule follows, as shown in 2.9. The rule is that if for every cell reachable from cell c through each of its house relations (horizontals, verticals and blocks) it is not the case that it must contain d, then it is possible for c to contain d.

2.3.6 Naked Tuples Elimination (Intermediate)

$$\boxed{\begin{array}{l} S, c_1 \vDash \Box_P(d_1 \otimes d_2) \\ S, c_2 \vDash \Box_P(d_1 \otimes d_2) \\ c_1 G c_2 \wedge c_1 G c_3 \\ \dots \\ S, c_3 \vDash \neg \Diamond_P d_1 \end{array}}$$

(A) N2E

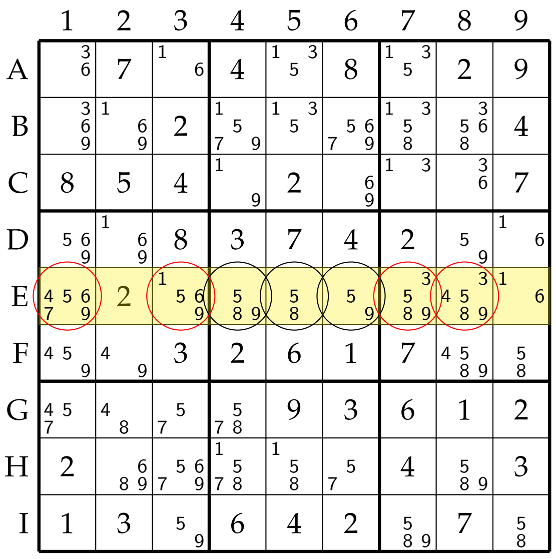
$$\boxed{\begin{array}{l} S, c_1 \vDash \Box_P(d_1 \otimes d_2 \otimes d_3) \vee \Box_P(d_1 \otimes d_2) \\ S, c_2 \vDash \Box_P(d_1 \otimes d_2 \otimes d_3) \vee \Box_P(d_2 \otimes d_3) \\ S, c_3 \vDash \Box_P(d_1 \otimes d_2 \otimes d_3) \vee \Box_P(d_1 \otimes d_3) \\ c_1 G c_2 \wedge c_1 G c_3 \wedge c_1 G c_4 \\ \dots \\ S, c_4 \vDash \neg \Diamond_P d_1 \end{array}}$$

(B) N3E

FIGURE 2.10: NTE: Naked Tuples Elimination ND-Rules

The Naked Pairs Elimination (N2E) strategy works as follows: "When two cells in the same house contain the exact same two pencil marks and only these pencil marks, these pencil marks can be removed from other cells in this house"[27]. The intuition is that when there are just two digits possible in two cells, it is not possible for other cells in this house to contain these digits. The corresponding ND-rule is 2.10a If there are two cells c_1 and c_2 that must contain either d_1 or d_2, a third cell c_3 in the same house cannot possibly contain d_1 (or d_2, for that matter, which is achieved by switching the d_1 and d_2 when using the rule).

This strategy is not as easily generalizable as other strategies, as the Naked Triples Elimination (N3E) deviates slightly from its Pairs variant. It is described as follows: "When three cells in the same house each contain a different subset of these three pencil marks, it is not possible for other cells in this house to contain these numbers"[27]. This means that it is either possible that these three cells contain either the exact same three pencil marks or only a couple of these pencil marks. An explanatory example of this can be seen in Figure 2.11.



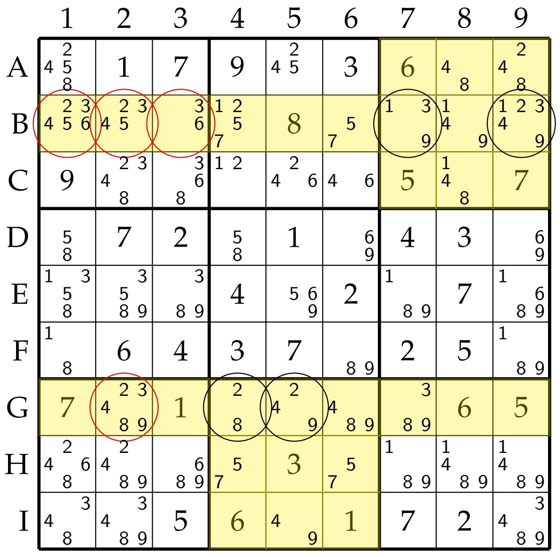
In Row E, in the centre Block, are the cells c_{E4}, c_{E5} and c_{E6} containing the PM's $5 \wedge 8 \wedge 9$, $5 \wedge 8$ and $5 \wedge 9$ respectively. Together, those three cells contain three different subsets of $5 \wedge 8 \wedge 9$. All premises for a Naked Triple Elimination are found! This allows us to remove those numbers from the rest of the cells in the house this Triple is aligned on, Row E. This means that in c_{E1} PM 5 can be removed, in c_{E3} PM 5 can be removed, in c_{E7} PM's $5 \wedge 8 \wedge 9$ can be removed and in c_{E8} PM $5 \wedge 8 \wedge 9$ can be removed.

FIGURE 2.11: Explained example of Naked Triple Elimination[27]

The corresponding ND-rule is 2.10b. If there are three cells c_1, c_2 and c_3 that must contain either d_1, d_2 or d_3 or must contain a different pair of d_1, d_2 or d_3, a fourth cell c_4 in the same house cannot possibly contain d_1 (or d_2/d_3, for that matter, which is achieved by switching the d_1, d_2 and d_3 when using the rule). Note that this rule explicitly leaves out subsets of size one ($S, c_x \vDash \Box_P d_1$), as this would mean that such a cell c_x must contain that digit, which leaves the two other considered cells with only two optional digits, so this would then be a case for Naked Pairs Elimination.

2.3.7 Pointing Tuples Elimination (Intermediate)

Where the Naked Tuples Elimination does not generalise well, the Pointing Tuples Elimination (PTE) scales pretty easily. This rule is defined as follows: "If any one pencil mark occurs twice or three times in just one block, then we can remove that pencil mark from the intersection of a line"[26]. This means that if two or three cells in the same row/column and same block contain the same PM, this PM can be removed from all other cells in this row/column. An explanatory example of this can be seen in Figure 2.12. Note that this example contains only two horizontal P2E's, but it should be evident, as Sudokus can be rotated, that this rule works on verticals too.



Here are two Pointing Pairs at the same time. In the upper right Block only two cells, c_{B7} and c_{B9} contain PM 3. These cells are both in the same Row, Row B. All premises for a Pointing Pair Elimination are found! This allows us to remove the 3 from all other cells in Row B. This means that in c_{B1}, c_{B2} and c_{B3} PM 3 can be removed. The same rule applies in Row G, with only two cells, c_{G4} and c_{G5} in the lower Block containing PM 2. This means that in c_{G2} PM 2 can be removed.

FIGURE 2.12: Explained example of Pointing Pairs Eliminations[26]

From this, two ND-rules are derived, as shown in 2.13. As far as P2E goes (2.13a), if two cells c_1 and c_2 in the same box and line can possibly contain d and if all other cells in this box cannot possibly contain d, then for all cells on that same line it is also not possible that d. In this case, P3E (2.13b) has almost exactly the same rule, with the only difference with P2E being that an additional cell c_3 on the same line also has to potentially contain d. Note that a Triple is the largest Tuple possible for this rule, as there can only be three cells in the same line and the same block, as the intersection of blocks and lines consists of a small line of three cells.

$$\boxed{\begin{array}{l} S, c_1, c_2 \vDash \Diamond_P d \\ S, c_3, \ldots, c_9 \vDash \neg\Diamond_P d \\ c_1 B c_2 \wedge \ldots \wedge c_1 B c_9 \\ c_1 L c_2 \wedge c_1 L c_{10} \\ \ldots \\ S, c_{10} \vDash \neg\Diamond_P d \end{array}}$$

(A) P2E

$$\boxed{\begin{array}{l} S, c_1, c_2, c_3 \vDash \Diamond_P d \\ S, c_4, \ldots, c_9 \vDash \neg\Diamond_P d \\ c_1 B c_2 \wedge \ldots \wedge c_1 B c_9 \\ c_1 L c_2 \wedge c_1 L c_3 \wedge c_1 L c_{10} \\ \ldots \\ S, c_{10} \vDash \neg\Diamond_P d \end{array}}$$

(B) P3E

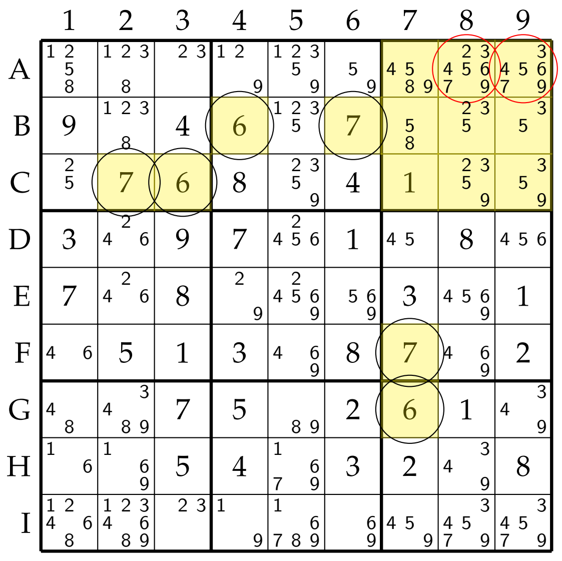
FIGURE 2.13: PTE: Pointing Tuples Elimination ND-Rules

2.3.8 Hidden Pair Elimination (Advanced)

$$\boxed{\begin{array}{l} S, c_1 \vDash \Diamond_P d_1 \land \Diamond_P d_2 \land \Diamond_P d_3 \\ S, c_2 \vDash \Diamond_P d_1 \land \Diamond_P d_2 \\ S, c_3, \ldots, c_9 \vDash \neg\Diamond_P d_1 \land \neg\Diamond_P d_2 \\ c_1 G c_2 \land \ldots \land c_1 G c_9 \\ \ldots \\ S, c_1 \vDash \neg\Diamond_P d_3 \end{array}}$$

FIGURE 2.14: H2E: Hidden Pairs Elimination ND-Rule

As its name implies, Hidden Pair Elimination (H2E) is all about finding a pair of hidden candidates. It is defined as follows: ”If two cells in the same house contain two of the same pencil marks, and it is known that all other cells in this house cannot contain these digits, this must be a pair and thus every other pencil mark in these two worlds can be removed”[25]. The fact that this concerns candidates who are often very well hidden between other candidates is what makes this strategy so difficult. For such a long and maybe convoluted definition, an explanatory example is in place. This can be seen in Figure 2.15.



In this Sudoku there are Hidden Pairs in the Block in the upper right corner. Of this Block, only cells c_{A8} and c_{A9} can possibly contain the PM’s $6 \land 7$. This is because all the other possibilities in this Block are restricted by the cells c_{B4}, c_{B6}, c_{C2}, c_{C3}, c_{F7} and c_{G6}, containing either a 6 or 7. All premises for a Hidden Pairs Elimination are found! This allows us to remove all other PM’s from cells c_{A8} and c_{A9}. This means that in c_{A8} PM’s $2 \land 3 \land 4 \land 5 \land 9$ can be removed and in c_{A9} PM’s $3 \land 4 \land 5 \land 9$ can be removed.

FIGURE 2.15: Explained example of Hidden Pairs Elimination[25]

From the given definition, the corresponding ND-Rule is defined in 2.14. If two cells c_1 and c_2 in the same house can possibly contain two digits, d_1 and d_2, and if these digits are not possible in any other cell in the house, if one of the cells could possibly contain another digit d_3, this cell cannot possibly contain d_3 any more.

While Hidden Triple Elimination (H3E) exists as well, it scales extremely poorly from H2E because it has side clauses similar to N3E (2.10b) and previous research has deemed it as rather useless[23], which is why it is chosen not to formally define this strategy.

2.3.9 X-Wing Elimination (Advanced)

$$
\boxed{\begin{array}{l}
S, c_1, c_2 \vDash \Diamond_P d \\
S, c_3, \ldots, c_9 \vDash \neg \Diamond_P d \\
c_1 H c_2 \wedge \ldots \wedge c_1 H c_9 \\
S, c_{10}, c_{11} \vDash \Diamond_P d \\
S, c_{12}, \ldots, c_{18} \vDash \neg \Diamond_P d \\
c_{10} H c_{11} \wedge \ldots \wedge c_{10} H c_{18} \\
S, c_{19} \vDash \Diamond_P d \\
c_1 V c_{19} \vee c_{10} V_{19} \\
\ldots \\
S, c_{19} \vDash \neg \Diamond_P d
\end{array}}
$$

(A) Horizontal XWE

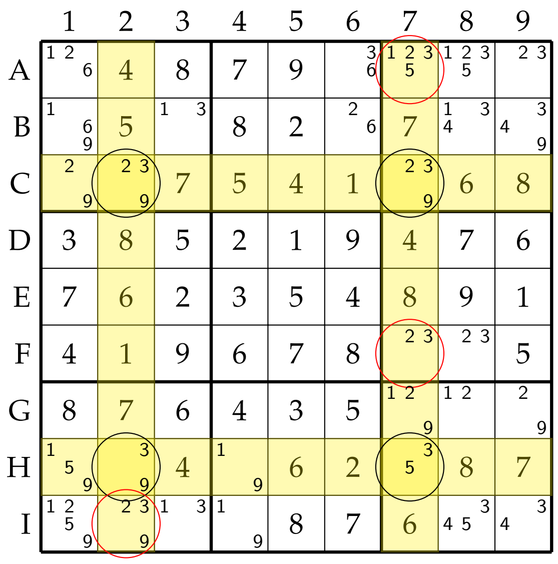
$$
\boxed{\begin{array}{l}
S, c_1, c_2 \vDash \Diamond_P d \\
S, c_3, \ldots, c_9 \vDash \neg \Diamond_P d \\
c_1 V c_2 \wedge \ldots \wedge c_1 V c_9 \\
S, c_{10}, c_{11} \vDash \Diamond_P d \\
S, c_{12}, \ldots, c_{18} \vDash \neg \Diamond_P d \\
c_{10} V c_{11} \wedge \ldots \wedge c_{10} V c_{18} \\
S, c_{19} \vDash \Diamond_P d \\
c_1 H c_{19} \vee c_{10} H_{19} \\
\ldots \\
S, c_{19} \vDash \neg \Diamond_P d
\end{array}}
$$

(B) Vertical XWE

FIGURE 2.16: XWE: X-Wing Elimination ND-Rules

The most difficult strategy concludes this list of strategies. X-Wing Elimination (XWE) is defined as follows: "When there are only two possible cells for a certain candidate in each of two different rows and these candidates also lie in the same columns, then all other candidates for this value in the columns can be eliminated. The reverse is also true for two columns with two common rows"[28]. This definition shows that XWE can be done both horizontally and vertically, which is why this strategy is split into Horizontal XWE (HXWE) and Vertical XWE (VXWE). This is a very complicated strategy to grasp without a visual example. To clear things up, such an explanatory example is shown in Figure 2.17.

From the given definition, two corresponding ND-Rules are defined, as shown in 2.16. This is by far the longest Rule of all of the strategies, as many premisses have to be checked. For rule 2.16a to work, there must be two pairs of cells – the first pair being c_1 and c_2 and the second pair being c_{10} and c_{11} – that are the only potential holders of digit d of their horizontal. These horizontal pairs must also be divided into two Verticals, so c_1 and c_{10} must be in the same vertical and c_2 and c_{11} too. Then, if none of the other cells in one of the rows can potentially contain d, it must be that any cell in either one of the Verticals cannot contain d either. This can be understood in a way that d is in either one of the cells of the pairs in the Vertical, so it cannot show up anywhere else in the Vertical. Rule 2.16b works exactly the same, only difference being that all the vertical and horizontal relations are swapped.



In this sudoku there is a Vertical X-Wing Elimination to be found. Row C and Row H both contain a pair of cells, in which there can potentially be a 3 (c_{C2} and c_{C7} form one pair, c_{H2} and c_{H7} form the other). All of the other cells in these Rows cannot contain a 3. These two cells are also in the same Columns, namely Column 2 and Column 7. All premises for a Vertical X-Wing Elimination are found! This allows us to remove all other 3's from the cells in both Columns. This means that in the cells c_{A7}, c_{F7} and c_{I2} PM 3 can be removed.

FIGURE 2.17: Explained example of X-Wing Elimination[4]

For clarity's sake, the different rows are stated in the ND-rule on multiple lines. So, while not recommended, this rule can be shortened by merging its first three lines with its second three lines, losing clarity in the process. So, to conclude, the tricky bit of this strategy is that it is not applicable often[23] and that it needs a lot of premises to eliminate very little information, which makes it a strategy that is only useful on specific occasions.

2.3.10 Guessing

Guessing is defined as "placing a digit in a cell without logical reasoning[29]". This definition is debatable, as there can be some thought about how one would guess and if one could tactically guess. If all strategies fail, an educated guess could be made by filling a cell with an allowed digit and see whether a solution can be generated from this point. Even if that fails, knowledge is gained: this cell digit cannot contain this digit. This is how many backtracking algorithms work and what one would logically do when getting stuck. The use of guessing could therefore be seen as a last resort.

For the purpose of this thesis, though, this definition is used and guessing will not be allowed by the algorithm and therefore by the helper. This is because I am interested in the performance of these strategies, while with enough guessing someone can eventually solve any problem.

2.4 Complexity and Structural Analysis

In this final Section the logical Chapter will be concluded by Analysing the complexity of the gained ND-ruleset (2.4.1) and their structural coherence (2.4.2).

2.4.1 Complexity Analysis

Complexity		**Difficulty**	**Strategy**	**Abbr.**	**Subsection**
Classic	**General**				
1	1	Implicit	Cell Based Elimination	CBE	(2.3.1)
1	1	Easy	House Based Elimination	HBE	(2.3.3)
2	2	Implicit	Conjunction Introduction	$\wedge$I	(2.1.3)
2	2	Medium	Naked Pairs Elimination	N2E	(2.3.6)
6	6	Medium	Naked Triples Elimination	N3E	(2.3.6)
9	n	Implicit	Pencil Mark Duality (Single)	PMD1	(2.3.2)
9	n	Implicit	Pencil Mark Duality (Pair)	PMD2	(2.3.2)
9	n	Implicit	Pencil Mark Duality (Triple)	PMD3	(2.3.2)
9	n	Easy	Last Remaining Cell	LRC	(2.3.4)
9	n	Medium	Pointing Pairs Elimination	P2E	(2.3.7)
9	n	Medium	Pointing Pairs Elimination	P3E	(2.3.7)
19	$2 \cdot n+1$	Hard	Hidden Pairs Elimination	H2E	(2.3.8)
19	$2 \cdot n+1$	Hard	X-Wing Elimination	XWE	(2.3.9)
24	$3 \cdot (n-1)$	Easy	Pencil Mark Introduction	PMI	(2.3.5)

TABLE 2.2: Strategies ordered by analysed complexity

As stated before, strategy difficulty and strategy complexity are not the same. A strategy could be very easy to understand, but complex to perform. While difficulty can be subjective – a human player could have their preferred strategies – complexity can be an objective and thus useful measure. The strategy complexity is defined in Definition 2.4.1. Each strategy's complexity is analysed and shown in 2.2[3].

Definition 2.4.1 (Strategy Complexity).
the complexity of a strategy is *the amount of cells* $\times$ *the amount of PM's to consider*

Comparing the complexity and difficulty in Table 2.2 shows that some easy strategies are indeed much more complex than others. It is noticeable in strategies with Tuples that as the Tuples grow larger, the more complex the strategies become. Also, it is evident that Pencil Mark Introduction is a great example of an easy strategy (to grasp and to perform), but the most complex of all to execute as a player would have to check the contents of 24 cells.

This complexity analysis is a great finding, to be used in the algorithm, as one would logically prefer searching for less complex strategies. Instead of just trying to perform any found strategy, also keeping its complexity into account is therefore a step in making the algorithm more like a human solving process.

[3]Note that its analysed complexity is split into Proper Sudoku (left column) and General Sudoku (right column). This works because all the

2.4.2 Structural Analysis

When analysing all of these rules created in Section 2.3 , I notice that all of the premisses and conclusions are ultimately linked with each other in a coherent structure. This makes sense, as Sudoku is a puzzle of eliminating possibilities and gaining knowledge about what digits must be filled in. Every bit of new knowledge the player gains leads to gaining more new knowledge until the puzzle is solved. Each breakthrough by the use of a rule leads to new potential breakthroughs. But how are these rules related, exactly? Extensive analysis of the form[4] of the premisses and conclusions of each ND-rule, results in the following Strategy Graph, Figure 2.18[5].

Note that Kripke Models and graphs like the Strategy Graph are different structures. From now on, when discussing graphs instead of worlds and accessibility relations, the terms *nodes/vertices* and *edges* are used.

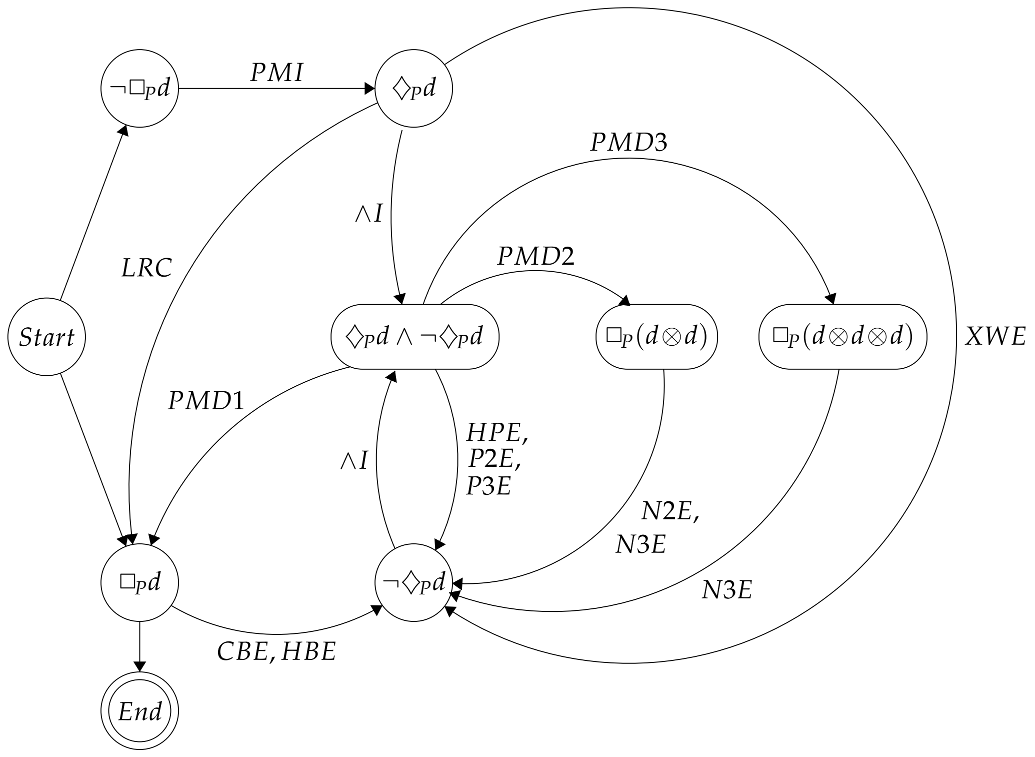


FIGURE 2.18: Strategy Graph of how all strategies relate to each other. Its nodes contain the form of the formulas used by the strategies. Its edges are labelled with strategies.

The workings of this Strategy Graph need some explanation. As shown in the Figure, each Sudoku starts with the digits that are necessary and those that aren't necessary in each cell ($\Box pd$ and $\neg\Box pd$). From these nodes, different strategy edges can be traversed to reach other nodes. PMI, for example, has $\neg\Box pd$ as premise and something in the form of $\Diamond pd$ as conclusion. This graph can be traversed until a solution is found and the End node is reached. This happens when $\Box pd$ holds for every cell, directly corresponding to Sudokus goal as defined in 2.2.4.

[4]Note that because this Graph is about the form of the formulas and not their specific meaning, everything in Figure 2.18 is called d.

[5]Note that Conjunction Introduction, as seen before (2.2) is added to the Strategy Graph as a fourteenth strategy. This is because when a cell is updated with a $\Diamond pd$ or $\neg\Diamond pd$, there needs to be checked if it now contains formulas of the form $\Diamond pd \wedge \neg\Diamond pd$.

It is evident from the Graph that to solve the easiest sudoku no strategies are required, as one can get from the start node to the end node without travelling over any strategy edges. This is because the easiest sudoku to solve is an already complete sudoku! In the edge case where the Sudoku is already entirely filled in at the start, all cells would contain formulas of the form $\Box_p d$ and the End node would be reached immediately. While its shortest path is easy to find, it is not possible to find a longest path in the Graph. This is because the Graph is a cyclic graph; it contains loops. It is not possible to analyse a longest path, but a lot of information on how these strategies relate to each other can be extracted.

This coherence of strategies is interesting for multiple reasons, because it provides us more insights on the order of these strategies. It is evident that most of the strategies (8 out of 14) end up at $\neg\Diamond_p d$, these all eliminate some digit possibility. Sudoku truly is a game of removing possibilities. It can also be seen that some strategies need other strategies to be useful. For example, LRC and XWE can only be performed if PMI can be performed. And the other way around too, as PMD3 is only useful when N3E can be performed as well. The Graph also shows how XWE is only useful in specific situations, but that there are other, easier ways to end up in the $\neg\Diamond_p d$-node. These insights are useful for this research, as well as for puzzlers learning how to solve Sudoku as they could derive from this Graph what would be useful strategies to learn in addition to their current skill set.

This structure is a direct improvement to earlier research, where topological orderings are made based on linearly checking each strategy in a row[15]. Instead of this, a topological ordering can be made based on gained logic based knowledge about the coherence of these rules. For this purpose, the strategy graph will be used as a data structure for our algorithm.

Basing a data structure on logical coherence seems like a good plan for an efficient algorithm, because unnecessary steps can be filtered out. There is no need to check if the puzzle is finished after each step, only if you are in the $\Box_p d$-node. This corresponds to human behaviour as well, as trivially humans do not ponder whether they are done after each performed strategy, only when they feel they have filled in the last cell. There is also no need to check each strategy for each cell, as they can be checked for only the updated cell. Improvements like this make it very useful to use such a logical data structure when creating a good solving algorithm, because these side clauses do not need to be incorporated into the search algorithm itself. In other words, an intelligent data structure makes a simple search to be effective enough.

It is a very flexible data structure, as it gives an excellent overview on the strategies. If one would want to solve a sudoku without the use of a certain strategy, all that needs to be done is to remove its strategy edge from the Graph. This works the other way around too as when new strategies are needed, these can be added to the structure. Also, the beauty of this is that because all the rules work for a General Sudoku, this structure does too. The solving steps for a larger Sudoku would remain the same. Even in Sudoku variants with extra houses, such as Diagonals, this structure would still work. If sudoku variants contain extra rules, their corresponding strategies could be added to the Graph! Therefore, this structure works for all Sudokus and various variants. Even so, due to the logical nature of the rules, such a structure could also be deducted for other logical puzzles that require the same kind of logical eliminative thinking. Therefore, this logical analysis could be done not only for all Sudoku variants, but also for multiple other logical puzzles, like Nonograms[30] and Logic Grids[19]. More on this flexibility will be discussed in Chapter 4.

Chapter 3

Algorithmic Implementation

In this Chapter I will introduce a Sudoku solving system. This system heuristically searches for a next best strategy, based on the complexity deduced and stated in Table 2.2, in combination with the graph consisting of all the strategy ND-rules (Graph 2.18). Because of this, the system is called the '*Helping Heuristic Alethic Natural Deduction*'-system, or '*Helping HAND*', for short.

This Chapter consists of three Sections in which I will talk about the code of the weighted graph search algorithm (3.1), testing the algorithm (3.2), and the visual application (3.3).

Both the search algorithm and the visual application are programmed in the programming language C#. While I made it a goal to code as efficiently as possible, the program still counts over 1500 lines of code. As a lot of this code is used to make the application look visually appealing and is therefore not semantically relevant, I did not include the entire program in the Appendix. The full program is available on Github (see link in Chapter 4). In Appendix A, the code for each of the strategy methods can be found.

3.1 Weighted Graph Search Algorithm

The algorithm behind the *Helping HAND*-system is a weighted graph search based algorithm. To explain how the program works, I wrote the algorithm in pseudocode as well, which can be found in Algorithm 1. The algorithm takes four inputs: a Sudoku, a Graph and two of its vertices. The Graph is created by adding weights based on the found Strategy Complexity of Table 2.2 to the corresponding edges of Strategy Graph 2.18. The starting vertices v_0 and v_1 correspond to the starting nodes in Strategy Graph 2.18 that hold $\neg\Box_p d$ and $\Box_p d$ respectively.

The algorithm works in two parts. In the first part, it assigns a starting vertex for each cell in the sudoku (see lines 4 - 8 in Algorithm 1), based on whether the cell holds a given or not. This can be seen as the initialization part of the algorithm. In the second part –the iteration part– it will keep considering new pairs of cells and vertices and their outgoing edges, which are new strategies to perform, until no more steps can be taken.

This algorithm is heavily based on Depth-first Search[31] in the sense that it uses a Stack as data structure to keep track of the vertices. Consequently, it searches for a strategy to perform and it will continue with the updated cell in the next step. Because the graph it searches through is weighted with strategy complexity, it will pick the least complex strategy in every step it takes. The graph's weighted edges are sorted beforehand, so the first strategy in the list of outcomes (see line 13, 14 and 18 in Algorithm 1) will always be the least complex at that point in the graph.

Algorithm 1 Sudoku Solver Search Pseudocode

```
1: input: Sudoku $S$, graph $G$, starting vertices $v_0$ and $v_1$
2: procedure SUDOKUSOLVESEARCH($S, G, v_0, v_1$)
3:     let $St$ be a Stack                                  ▷ of vertex,cell-tuples $\langle v, c\rangle$
4:     for all $c \in S$ do                                 ▷ push each cell to the stack
5:         if $c$.boxlist is not empty then
6:             $St$.push($\langle v_0, c\rangle$)
7:         else
8:             $St$.push($\langle v_1, c\rangle$)
9:     while $St$ is not empty do
10:        $\langle v, c\rangle \leftarrow St$.pop()             ▷ consider next vertex,cell-tuple
11:        for all directed edges $e$ in $G$ from $v$ to $w$ do
12:            if $e$.IsAllowed then
13:                let $L$ be a List                        ▷ of strategy outcomes ($\langle c_u, \varphi\rangle$-tuples)
14:                $L \leftarrow e$.strategy($c$)            ▷ try strategy and return outcomes
15:                if $L$.Length $> 1$ then                  ▷ more than one update possible
16:                    $St$.push($\langle v, c\rangle$)       ▷ save this strategy for later
17:                if $L$ is not empty then
18:                    apply first strategy of $L$ to cell $c_u$   ▷ Update Cell in Sudoku
19:                    $St$.push($\langle w, c_u\rangle$)
```

3.2 Testing

The algorithm is tested for multiple reasons, of which the first is simple; testing is necessary to see if the algorithm works as intended. The second reason is more complex, as it is interesting to find out which strategies are needed to solve Sudokus of different difficulties. The latter can then be used in the Visual Application, as *Helping HAND* can use this data to offer its user advice on the strategies. For testing purposes, a dataset is created. This dataset is explained (3.2.1) and analysed (3.2.2).

3.2.1 Dataset Choices

While multiple extensive Sudoku datasets exist[21,22], none of these include grading on its puzzle difficulty. This is why I decided to create a dataset on my own. Multiple websites exist that generate free-to-use graded sudokus. I chose to use a simple website, which generates multiple graded sudokus and their solution. This website can be found in the Links section in Chapter 4.

Sudokus are often found online as visual two-dimensional puzzles in PDF-format, as are the sudokus from this website, but the algorithm does not have a way of visually recognising and importing these kinds of puzzles. As such, a non-visual dataset is required. For this, I converted each sudoku into a string of digits, with each digit in the string corresponding to the given in each cell. For empty cells the corresponding digit is 0. Each sudoku is then tagged with its difficulty and given a unique index. This dataset can be found in Appendix B.

The dataset consists of ten Easy sudokus, ten Medium sudokus and ten Hard sudokus, as graded by the website. I chose for a total of 30 sudokus, as this test is intended to show what kind of sudokus can be solved. With this purpose in mind, the dataset does not need to be large at all, it merely needs to reflect the range in difficulty between different sudokus.

3.2.2 Data Analysis

Four subsets of strategies are tested against the dataset of three graded categories. These subsets consists of all strategies up to the Implicit (Implicit), up to Beginner (≤Beginner), up to Intermediate (≤Intermediate) and up to Advanced (≤Advanced) strategies. The results are shown in Figure 3.1.

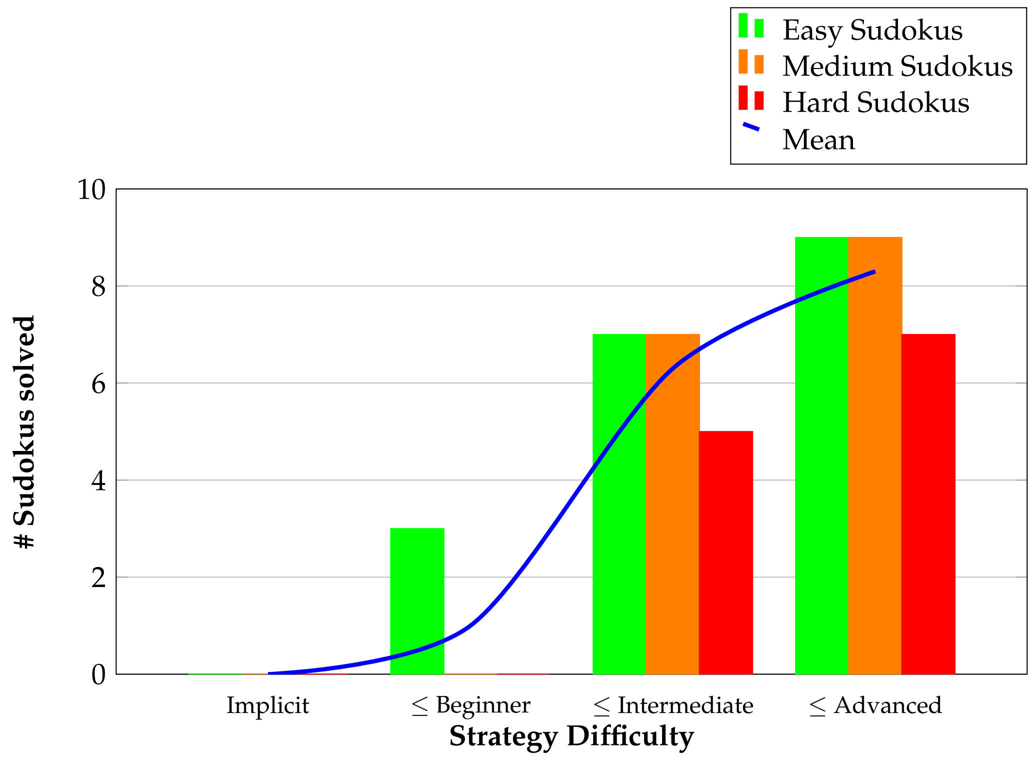


FIGURE 3.1: Grouped Bar Chart of amount of graded Sudokus solved by different subsets of Strategies.

First of all, Figure 3.1 confirms that the algorithm can solve sudokus! This confirms that the C# implementation of the Weighted Graph Search and the C# implementation of all the strategies (Appendix A) work as intended. The Figure also gives us a good grasp of what kind of sudokus can be solved by which strategies.

As discussed earlier on, Implicit strategies merely have an assisting purpose to other strategies. It is therefore unsurprising to see in Figure 3.1 that the Algorithm cannot solve any sudokus by using just the Implicit strategies. By using the Implicit and Beginner strategies, the algorithm can solve some Easy sudokus, but no others. Adding the use of Intermediate strategies results in a spike in amount of Easy, Medium and Hard sudokus solved. The addition of the two Advanced strategies slightly increases the amount of sudokus solved even further.

The clear incline of the mean of amount of Sudokus solved (blue line in the Figure) is exactly the desired outcome. This means that as more advanced strategies are used, more sudokus can be solved, which implies the categorisation of strategy difficulty is pretty good. It is also apparent that not all of the sudokus can be solved by the solver. This means that more strategies would be needed to accomplish this. It can be said that this selection performs really well, but not perfectly. More on this will be discussed in Chapter 4.

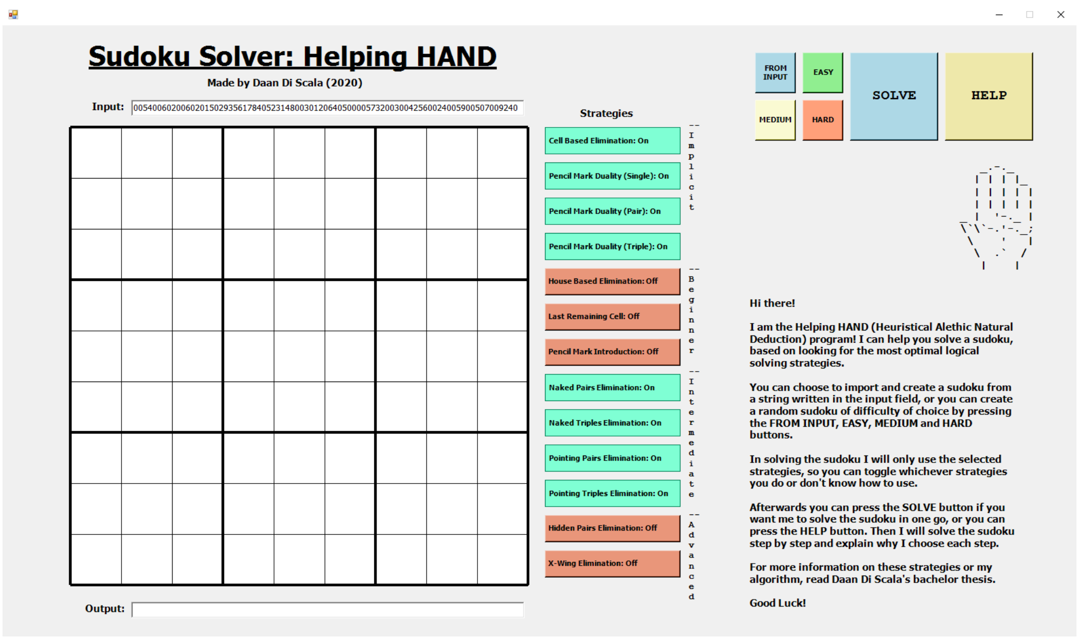
A selection is made of the sudokus that can be solved. This selection consists of Easy sudokus that can be solved by ≤Beginner strategies, Medium sudokus that can be solved by ≤Intermediate strategies and Hard sudokus that can be solved by ≤Advanced difficulty strategies. This selection of sudokus will be used by the Visual Application, which makes it so the solver can give advice based on the data. More on this advice is discussed in the next section.

3.3 Visual Application

In this final section, the Algorithmic Chapter is concluded by discussing the visual design choices and workings of the Visual Application.

I've created a Visual Application, *Helping HAND*, as shown in Figure 3.2. For readability purposes, certain parts from the program are highlighted in the Figures 3.3, 3.4, 3.5 and 3.6. The interface contains a visual Sudoku grid, an input and output field for Sudokus in string form above and below the grid, multiple strategy selection buttons on the right of the grid, sudoku puzzle selection buttons and a Solve and Help button in the upper right corner and informatory labels.

In short, the program works as follows: A user can either input a sudoku in string form or select a sudoku of difficulty of choice. Then the user can either decide to let *Helping HAND* solve the sudoku for them in one go or make *Helping HAND* help the user step by step.



Sudoku Solver: Helping HAND

Made by Daan Di Scala (2020)

Input: 005400602006020150293561784052314800301206405000057320030042560024005900507009240

Cell Based Elimination: On
Pencil Mark Duality (Single): On
Pencil Mark Duality (Pair): On
Pencil Mark Duality (Triple): On
House Based Elimination: Off
Last Remaining Cell: Off
Pencil Mark Introduction: Off
Naked Pairs Elimination: On
Naked Triples Elimination: On
Pointing Pairs Elimination: On
Pointing Triples Elimination: On
Hidden Pairs Elimination: Off
X-Wing Elimination: Off

Output:

Hi there!

I am the Helping HAND (Heuristical Alethic Natural Deduction) program! I can help you solve a sudoku, based on looking for the most optimal logical solving strategies.

You can choose to import and create a sudoku from a string written in the input field, or you can create a random sudoku of difficulty of choice by pressing the FROM INPUT, EASY, MEDIUM and HARD buttons.

In solving the sudoku I will only use the selected strategies, so you can toggle whichever strategies you do or don't know how to use.

Afterwards you can press the SOLVE button if you want me to solve the sudoku in one go, or you can press the HELP button. Then I will solve the sudoku step by step and explain why I choose each step.

For more information on these strategies or my algorithm, read Daan Di Scala's bachelor thesis.

Good Luck!

FIGURE 3.2: The Visual Application *Helping HAND*'s interface

Figure 3.3 shows four buttons that let the user choose what kind of Sudoku to show. When the FROM INPUT button is pressed, *Helping HAND* will convert the Sudoku in string form into a visual Sudoku, which will be shown on screen. When either the EASY, MEDIUM or HARD button is pressed, *Helping HAND* will select one of the graded sudokus from the dataset (Appendix B). This way *Helping HAND* does not have to generate any sudokus with validated grading itself, because at this point in the research, this is not a relevant goal. See Chapter 4 for potential future researches.

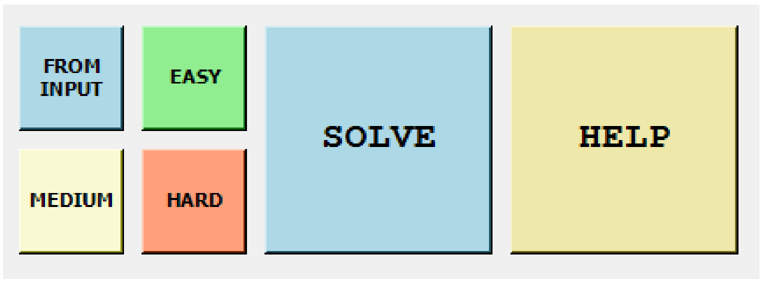
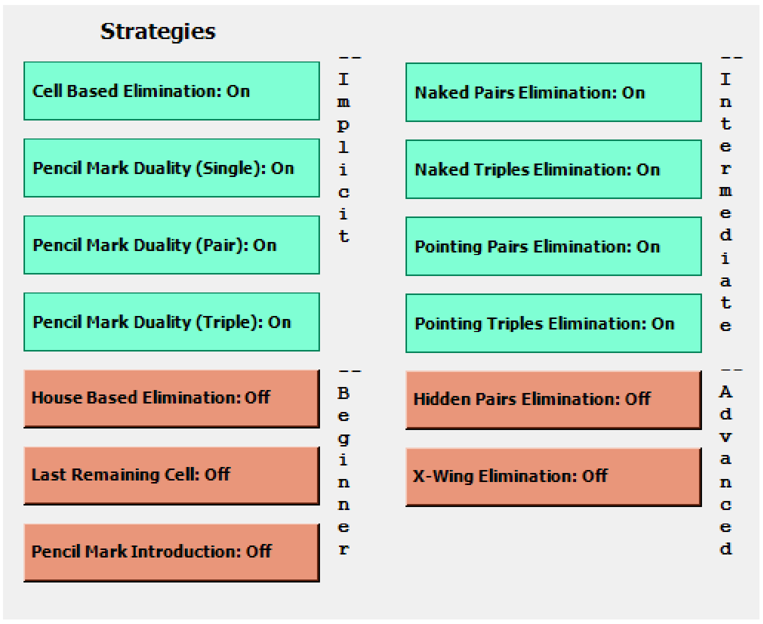


FIGURE 3.3: The FROM INPUT, EASY, MEDIUM and HARD buttons are for sudoku selection purposes. The SOLVE and HELP buttons will let *Helping HAND* solve the selected Sudoku, either in one go or explaining along the way.

Before solving a sudoku, a user can toggle the Strategy Buttons as shown in Figure 3.4. *Helping HAND* will only use the selected strategies in its search algorithm when solving a sudoku, so this way a user can choose whichever strategies they want the program to use, as these can be the strategies the user know how to use.


Strategies

Cell Based Elimination: On
Pencil Mark Duality (Single): On
Pencil Mark Duality (Pair): On
Pencil Mark Duality (Triple): On
-- Implicit

House Based Elimination: Off
Last Remaining Cell: Off
Pencil Mark Introduction: Off
-- Beginner

Naked Pairs Elimination: On
Naked Triples Elimination: On
Pointing Pairs Elimination: On
Pointing Triples Elimination: On
-- Intermediate

Hidden Pairs Elimination: Off
X-Wing Elimination: Off
-- Advanced

FIGURE 3.4: An example of toggled Strategy buttons, with the Beginner and Advanced strategies turned off. When solving, *helping HAND* will now only use the Implicit and Intermediate strategies.

When the SOLVE button is pressed, as shown in Figure 3.3, *Helping HAND* will run the search Algorithm (Algorithm 1) on the Sudoku shown on its screen and will visually solve each step on the sudoku, as seen in Figure 3.5. It will solve the sudoku as much as possible, with the strategies given.

When the HELP button is pressed, as shown in Figure 3.3, *Helping HAND* will first try to apply the search Algorithm without showing it to the user. It then reports whether or not the sudoku can be solved with the given strategies. If the sudoku cannot be solved, *Helping HAND* will give advice on how to improve its solving by selecting more strategies, based on the data analysis in 3.2.2.

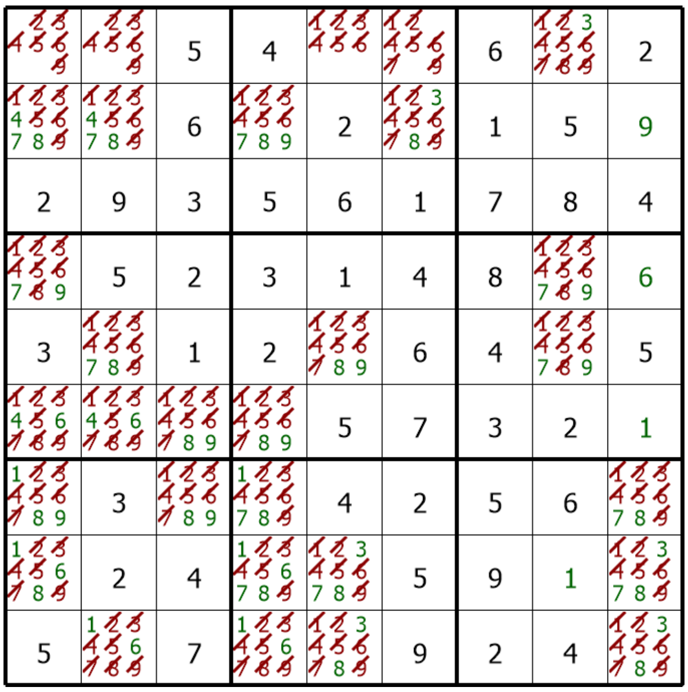
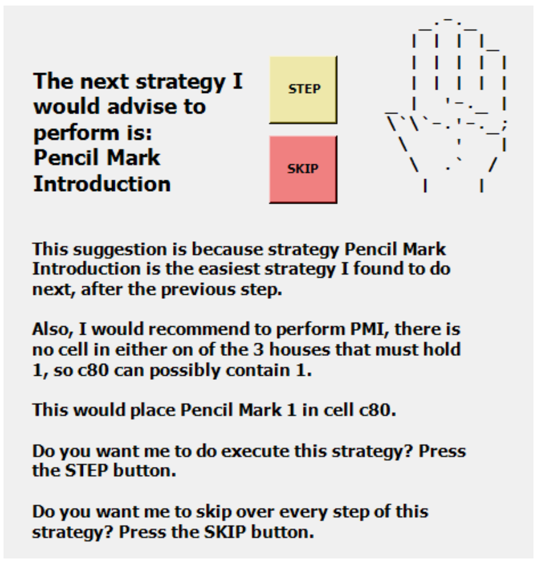


FIGURE 3.5: A sudoku in the middle of being solved by *Helping HAND*. Givens are black digits while cells filled by the algorithm are green digits. PM's considered possible are small green digits and PM's considered impossible are red small strike-through digits.

If the selected sudoku can be solved when HELP is pressed, *Helping HAND* will explain for each step why the next strategy is chosen and how it can be applied. Such an explanation is shown in Figure 3.6. A STEP button appears, which the user can press to continue with the next step. A SKIP button also appears, which the user can press to skip any of the same strategies for the rest of the solving process. This way, the *Helping HAND* does its job as a transparent logical helper algorithm. More on this will be discussed in Chapter 4.



The next strategy I would advise to perform is: Pencil Mark Introduction

This suggestion is because strategy Pencil Mark Introduction is the easiest strategy I found to do next, after the previous step.

Also, I would recommend to perform PMI, there is no cell in either on of the 3 houses that must hold 1, so c80 can possibly contain 1.

This would place Pencil Mark 1 in cell c80.

Do you want me to do execute this strategy? Press the STEP button.

Do you want me to skip over every step of this strategy? Press the SKIP button.

FIGURE 3.6: Advice given by *Helping HAND* during one of its solving steps.

Chapter 4

Discussion

In this final Chapter I will discuss the strengths and shortcomings of the logical analysis and the algorithmic implementation. Furthermore, I will explore suggestions for further research and elaborate on the implications of the results for the AI community, possibly even for our daily life.

General Discussion: The Logic

Regarding the Alethic formalisation of the sudoku, there are no concerns. The entire logic framework is built with the Classic Sudoku in mind, but it needs little adjustment to work for General Sudokus as well. The formalisation of the sudoku puzzle itself works great, but the Natural Deduction rules get more convoluted the more complicated the strategies get. This problem occurs because ND does not have a way to concisely talk about things such as amounts of cells. For example, $\Box_G$ is used when referring to all reachable cells in a house, but referring to things like "two cells and none of the other in a row" (See Rule 2.13a), for each of the cells there needs to exist a separate pattern match, which makes the process of the rules cumbersome. Additional strategies should not become much more complex, as many more Advanced strategies exist and the Alethic formalisation might fall short.

General discussion: The Strategies

When choosing 13 of the most popular strategies, I assumed their difficulty level was correct. From the data it shows that these strategies are indeed categorised correctly and also sufficient to solve many sudokus of various difficulties. The test results also show that I added the right strategies. All strategy subsets were needed to solve the different sudokus. This shows that besides a good test, and apt logical formalisation of the rules, I also chose the right amount and the right difficulty of strategies. Yet, as concluded earlier, it would be a welcome addition to add more strategies to solve more or even all sudokus.

A lot has already been discussed in the Structural Analysis (2.4.2) about the logical coherence of the strategies. As concluded there, the Strategy Graph is a flexible data structure, which gives a great overview of the logical coherence. Because of its flexibility, this structure would also work for other Sudoku variants. While this structure works specifically for sudokus, I believe it says something in general about how people solve logic puzzles. I believe that such a logical structure in terms of Alethic Logic can be made for other (eliminative) logic puzzles, because many other logical puzzles contain similar solving steps, which can be analysed in similar ways.

To assist human puzzlers, the algorithm needed solve puzzles like a human. I think I succeeded in this by implementing the human strategies. It is not without reason that the largest part of this thesis is spent on the logical analysis. This extensive research on sound logic behind the strategies makes for a simple yet sound algorithm. Thus, prioritising the strategies really paid off for creating the algorithm.

General Discussion: The Algorithm

The logical structure makes that the Algorithm, albeit based on a very simple way of searching, is actually a very clever way of solving Sudokus. This because, to reiterate what has been concluded in Chapter 2, an intelligent data structure makes a simple search to be effective enough. The generalisability of the algorithm is therefore great as well, as no adjustments to the algorithm are needed to add more strategies to the weighted graph. This means that the Algorithm in its current state would work for other sudoku variants as well, as well as other logic puzzles.

One thing that could be looked at is the weight analysis for the weighted graph. I have now merely based the weights on what is logically the best step in terms of strategy complexity, but factors such as strategy frequency could potentially included in the weights. A more extensive research could be conducted on how often these strategies actually occur when solving different types of sudoku.

Something else that could be researched in regards of the algorithm is its choice of programming language. Because of the algorithm and the visual solver I chose to write the code in the imperative language C#, but it would be interesting to see the strategies implemented in a declarative language such as Haskell, because that would suit the logical nature of the rules better. Some strategies where rather difficult to implement in C#, like N3E (2.10b), because it matches on so many different specific patterns of propositions.

General Discussion: The Visual Application

While the Visual Application *Helping HAND* works successfully, some additions could be made which would take too much time or did not fit the purpose of this thesis research. For instance, as the logic behind the strategies could work for general sudokus and therefore the algorithm too, *Helping HAND* could be expanded to include solving larger Sudokus.

Another expansion that would improve the system would be to include interactivity with the sudoku grid, by letting a user fill in cells on their own. This would turn the Visual Application into more of a helper playing alongside the user and thus a slightly better training program. Though, having the system be able to pick up from any point instead of just the start would need a minor adjustment in the algorithm, as the starting vertices would also have to hold filled PM's into account.

A final possible addition would be error detection. This would involve detecting whether a user makes any mistakes and offering them advice to go a few steps back and retry. This could be done by keeping track whether a potential solution still exists.

With these additions in mind, I think the biggest improvement on the Solver –and therefore the most interesting possible follow up research– could be to let the program figure out which strategies are used by human puzzlers. As of right now, users have to select their strategies in the program themselves. Instead of this, it would be interesting to have the program learn what kind of strategies are known by the user and which strategies it can assist with. To achieve this, *Helping HAND* could be used as a tool to let people solve Sudokus in a controlled environment. When test subjects get stuck they can ask *Helping HAND* for a next step/strategy. This way one would be able to map the usefulness of the strategies, which yields a way more objective way of grading. Such a database could then be used to learn what kind of strategy is useful, by the use of some kind of Machine Learning. The result of this would be a program in which users can solve sudokus and the program could find out their patterns and frequencies on its own, being able to assist where necessary.

Conclusion

The question remains whether the *Helping HAND* system can be classified as Explainable Artificial Intelligence. I think the system in its current state is too heavily based on user input to be able to do so. The system does however involve a form of solving like human players, as it performs human strategies. If some form of learning was added as proposed earlier, one could let a system learn which human strategies are used instead of user selecting them themselves. I believe that this would be the crucial step to the system becoming XAI.

In conclusion, –to answer the research question– it is possible to build a transparent logical algorithm, which solves sudoku's based on human strategies. This can be done by basing a logical framework on how we consider sudoku strategies in terms of necessity and possibility and extracting a structure of how these strategies relate to each other logically. Then creating a search algorithm based on that structure, which can therefore explain each step taken, which means it can help people based on this Heuristic Alethic Natural Deduction.

I believe this research proves to be a valuable next step in the field of AI, but also in other fields, as logic is all around us. While this research explicitly focusses on sudoku solving, many more logical challenges are to be found. The same logical and algorithmic approach is useful to support many more important decision making tasks. Because no matter what kind of work is done, if just with puzzles or complicated logic based jobs, people could always use a helping hand.

Links

Github with full C# code of the Helping HAND-system can be found at:
https://github.com/DiScala/BachelorThesis

Website used to generate graded sudokus can be found at:
https://sudoku.cba.si/en/

Figures

All figures have been made in LaTeX with the use of the Tikz package, except for:

University Utrecht Logo (Page i)
https://www.uu.nl/en/organisation/corporate-identity/downloads/logo

Screenshots from *Helping HAND* (Figure 3.2)
https://github.com/DiScala/BachelorThesis

Bibliography

[1] L. Aaronson. Sudoku Science. *IEEE Spectrum*, pages 16–17, February 2006.

[2] O. Abdel-Raouf, I. El-Henawy, M. Abdel-Baset, et al. A novel hybrid flower pollination algorithm with chaotic harmony search for solving sudoku puzzles. *International Journal of Modern Education and Computer Science*, 6(3):38, 2014.

[3] T. Cazenave and I. Labo. A search based sudoku solver. *Labo IA Dept. Informatique Universite Paris*, 8:93526, 2006.

[4] T. Davis. The Mathematics of Sudoku, 2006.

[5] M. de Rijke, H. Wansing, et al. Proofs and expressiveness in alethic modal logic. *A companion to philosophical logic*, pages 422–441, 2002.

[6] B. Felgenhauer and F. Jarvis. Enumerating possible sudoku grids. *Preprint available at http://www. afjarvis. staff. shef. ac. uk/sudoku/sudoku. pdf*, 2005.

[7] F. B. Fitch. Natural Deduction Rules for Obligation. *American philosophical quarterly*, 3(1):27–38, 1966.

[8] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018.

[9] M. Kotseva. Visual sudoku solver. 4th year project report, School of Informatics, University of Edinburgh, November 2018.

[10] S. A. Kripke. A completeness theorem in modal logic. *The journal of symbolic logic*, 24(1):1–14, 1959.

[11] R. Lewis. Metaheuristics can solve sudoku puzzles. *Journal of heuristics*, 13(4):387–401, 2007.

[12] A. K. Maji and R. K. Pal. Sudoku solver using minigrid based backtracking. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 36–44. IEEE, 2014.

[13] H. Malhotra, H. Jain, and P. K. Gupta. Automated puzzle solver using image processing. 2017.

[14] T. Mantere and J. Koljonen. Solving, rating and generating sudoku puzzles with ga. In *2007 IEEE congress on evolutionary computation*, pages 1382–1389. IEEE, 2007.

[15] D. Martin, E. Cross, and M. Alexander. Cracking the sudoku: A deterministic approach. *UMAPJournal*, page 381, 2007.

[16] R. Mastop. Modal logic for artificial intelligence, 2012.

[17] Mathematicians Solve Minimum Sudoku Problem. https://www.technologyreview.com/s/426554/, January 2012. Accessed: 10-02-2020.

[18] H. L. Mattias Harrysson. Solving sudoku efficiently with dancing links. Degree project in computer science, KTH Royal Institute of Technology, 2014.

[19] A. Mitra and C. Baral. Learning to automatically solve logic grid puzzles. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1023–1033, 2015.

[20] M. Nicolau and C. Ryan. Solving sudoku with the gauge system. In *European Conference on Genetic Programming*, pages 213–224. Springer, 2006.

[21] K. Park. 1 million sudoku games, December 2016.

[22] R. Rao. 9 million sudoku puzzles and solutions, November 2019.

[23] A. Stuart. The Relative Incidence of Sudoku Strategies. https://www.sudokuwiki.org/The_Relative_Incidence_of_Sudoku_Strategies, December 2013. Accessed: 30-03-2020.

[24] Sudoku Terminology. https://www.sudokumix.net/sudoku-terminology/. Accessed: 10-02-2020.

[25] Sudokuwiki: Hidden candidates. https://www.sudokuwiki.org/Hidden_Candidates. Accessed: 30-03-2020.

[26] Sudokuwiki: Intersection removal. https://www.sudokuwiki.org/Intersection_Removal. Accessed: 04-04-2020.

[27] Sudokuwiki: Naked candidates. https://www.sudokuwiki.org/Naked_Candidates. Accessed: 30-03-2020.

[28] Sudokuwiki: X-wing strategy. https://www.sudokuwiki.org/X_Wing_Strategy. Accessed: 30-03-2020.

[29] Sudopedia Terminology. http://sudopedia.enjoysudoku.com/Terminology.html/. Accessed: 10-02-2020.

[30] D.-J. Sun, K.-c. Wu, I. Wu, S.-J. Yen, K.-Y. Kao, et al. Nonogram tournaments in taai 2011. *ICGA JOURNAL*, 35(2):120–123, 2012.

[31] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[32] T. Weber. A SAT-based Sudoku solver. In G. Sutcliffe and A. Voronkov, editors, *LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings*, pages 11–15, December 2005.

[33] What Actually Happens At A SWC. https://sudoku.com/how-to-play/what-actually-happens-at-a-sudoku-world-championship/. Accessed: 12-03-2020.

[34] World Sudoku Championships. http://www.worldpuzzle.org/championships/wsc/. Accessed: 16-02-2020.

[35] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.

[36] You may be good at Sudoku, but do you know what the word literally means? https://www.dictionary.com/e/sudoku/. Accessed: 12-03-2020.

Appendix A

C# Implementation Strategies

```
/*-------------------------------------2.3.1-------------------------------------*/
List<Move> CBE(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  Cell cell = updated_cell;
  int[] candidates = sudoku.PMs;

  if (cell.box_set.Count == 1)  //if c entails box phi
  {
    int phi = cell.box_list[0];
    foreach (int psi in candidates) //then for all other psi
    {
      //psi is not phi and not yet updated
      if (psi != phi && !cell.negdiamond_set.Contains(psi))
      {
        moves.Add(new Move(cell.name, "-d", psi, CBEskip, "Cell␣Based␣Elimination",
        "CBE,␣because␣cell␣c" + cell.name + "␣must␣contain␣digit" + phi +
        ",␣so␣it␣cannot␣contain␣digit" + psi));
      }
      if (moves.Count == 2) break;
    }
  }
  return moves;
}

/*-------------------------------------2.3.2-------------------------------------*/
List<Move> PMD(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  Cell cell = updated_cell;
  //if c contains di phi 1,2,3 times & c contains neg di phi 8,7,6 times & not yet updated
  if (cell.diamond_set.Count==n && cell.negdiamond_set.Count==(9-n) && cell.box_set.Count!=n)
  {
    List<int> PMs = new List<int>();
    for (int i = 0; i < n; i++) //then c entails phi (or phi (or phi))
    {
      PMs.Add(cell.diamond_list[i]);
    }
    moves.Add(new Move(cell.name, "b", PMs, PMDskip, "Pencil␣Mark␣Duality",
    "PMD" + n + ",␣because␣cell␣c" + cell.name + "␣contains␣" + n + "␣PM's␣and␣the␣other␣"
    + (9 - n) + "␣PM's␣are␣not␣possible␣in␣cell␣c" + cell.name +
    ",␣so␣it␣must␣contain␣either␣one␣of␣these␣" + n + "␣PM's"));
  }
  return moves;
}

/*-------------------------------------2.3.3-------------------------------------*/
List<Move> HBE(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  foreach (int i in updated_cell.whichgroup)
  {
    Cell[] group = sudoku.groups[i];
    foreach (Cell cell in group)
    {
      if (cell.box_set.Count == 1) //if c entails box phi
      {
        int phi = cell.box_list[0];
        foreach (Cell neighbor in group) //then for all cells in same house
        {
          if (cell.name != neighbor.name && !neighbor.negdiamond_set.Contains(phi))
          { //if other cell and not yet updated
            moves.Add(new Move(neighbor.name, "-d",phi,HBEskip,"House␣Based␣Elimination",
            "HBE,␣because␣cell␣c" + cell.name + "␣must␣contain␣" + phi + ",␣so␣cell␣c"
            + neighbor.name + "␣cannot␣contain␣" + phi));
          }
        }
      }
    }
    if (moves.Count == 2) break;
  }
  return moves;
}

/*---------------------------------------2.3.4---------------------------------------*/
List<Move> LRC(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  int[] PMs = sudoku.PMs;
  foreach (int option in sudoku.PMs)
  {
    foreach (int i in updated_cell.whichgroup)
    {
      Cell[] group = sudoku.groups[i];
      foreach (Cell cell in group)
      {
        int optionCount = 0;
        foreach (Cell neighbor in group)
        { //if all other cells in same house entail neg di phi
          if (neighbor.name != cell.name && neighbor.negdiamond_set.Contains(option))
            optionCount++;
        }
        if (optionCount == 8 && cell.filled == false) //then cell c entails box phi
        {
          List<int> options = new List<int>();
          options.Add(option);
          moves.Add(new Move(cell.name, "b", options, LRCskip, "Last␣Remaining␣Cell",
          "LRC,␣because␣cell␣c" + cell.name + "␣is␣the␣last␣remaining␣cell␣in␣its␣house␣" +
          "that␣could␣contain␣" + option + ",␣so␣it␣must␣hold" + option));
        }
      }
    }
  }
  return moves;
}

/*---------------------------------------2.3.5---------------------------------------*/
List<Move> PMI(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  int[] options = sudoku.PMs; //for all possible phi
  foreach (int option in options)
  {
    List<Cell> considered_cells_list = new List<Cell>();
    HashSet<Cell> considered_cells_set = new HashSet<Cell>();
    foreach (int i in updated_cell.whichgroup)
    {
      Cell[] group = sudoku.groups[i];
      foreach (Cell cell in group)
      {
        if (!considered_cells_set.Contains(cell))
        {
          considered_cells_list.Add(cell);
          considered_cells_set.Add(cell);
        }
        int optionCount = 0;
        foreach (Cell neighbor in group)
        {
          //if all other cells in same house entail neg box phi
          if (neighbor.name != cell.name && neighbor.negbox_set.Contains(option))
            optionCount++;
        }
        //if not yet updated and c does not entail diamond or is already filled
        if (optionCount == 8 && !cell.diamond_set.Contains(option) &&
          !cell.negdiamond_set.Contains(option) && cell.filled == false)
          cell.possibility_count += 1; //count this house as possible
      }
    }
    foreach (Cell cell in considered_cells_list)
    {
      if (cell.possibility_count == 3) //if for all 3 houses cell c is in
      {
        moves.Add(new Move(cell.name, "d", option, PMIskip, "Pencil␣Mark␣Introduction",
        "PMI,␣because␣there␣is␣no␣cell␣in␣either␣on␣of␣the␣3␣houses␣that␣necessarily␣must␣" +
        "hold␣" + option + ",␣so␣cell␣c" + cell.name + "␣could␣possibly␣contain␣" + option));
      }
      cell.possibility_count = 0; //reset counter
      if (moves.Count == 2) break;
    }
    if (moves.Count == 2) break;
  }
  return moves;
}

/*--------------------------------------2.3.6--------------------------------------*/
List<Move> N2E(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  foreach (int i in updated_cell.whichgroup)
  {
    Cell[] group = sudoku.groups[i];
    foreach (Cell cell in group)
    {
      if (cell.box_set.Count == 2) //if c contains phi or phi
      {
        List<Cell> same_neighbors = new List<Cell>();
        List<Cell> other_neighbors = new List<Cell>();
        foreach (Cell neighbor in group)
        {
        //for all other cells c in same house, keep track which do and don't contain phi or phi
          if (cell.name != neighbor.name && cell.box_set.SetEquals(neighbor.box_set))
            same_neighbors.Add(neighbor);
          else if (cell.name != neighbor.name && neighbor.filled == false)
            other_neighbors.Add(neighbor);
        }
        //for all other cells c in same house, if 1 of them entail phi or phi
        if (same_neighbors.Count == 1)
        {
          foreach (int candidate in cell.box_list)
          {
            foreach (Cell neighbor in other_neighbors) //then for all other c in same house
            {
              if (!neighbor.negdiamond_set.Contains(candidate)) //if not already updated
              {
                moves.Add(new Move(neighbor.name, "-d", candidate, N2Eskip,
                  "Naked␣Pairs␣Elimination","N2E,␣because␣cell␣c" + cell.name + "␣and␣cell␣c"
                  + same_neighbors[0].name +"␣form␣a␣Naked␣Pair␣of␣which␣either␣one␣of␣the␣" +
                  "two␣must␣hold␣digit␣" + candidate +",␣so␣in␣cell␣c" + neighbor.name +
                  "␣there␣cannot␣be␣PM␣" + candidate));
              }
            }
          }
        }
      }
    }
  }
  return moves;
}
```

```
List<Move> N3E(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  foreach (int i in updated_cell.whichgroup)
  {
    Cell[] group = sudoku.groups[i];
    foreach (Cell cell1 in group)
    {
      int phi = 0;
      int psi = 0;
      int chi = 0;
      int[] ps = new int[3];
      if (cell1.box_set.Count == 2) //if first potential triple found
          {
        phi = cell1.box_list[0];
        psi = cell1.box_list[1];
        foreach (Cell cell2 in group)
        {
          chi = 0;
          //if second potential triple found
          if (cell1.name != cell2.name && cell2.box_set.Count < 4)
          {
            //check whether it has 2 or 3 phis
            if (cell2.box_set.Count == 2)
            {
              if (cell2.box_list[0] == psi && cell2.box_list[1] != phi)
                chi = cell2.box_list[1];
              else if (cell2.box_list[1] == psi && cell2.box_list[0] != phi)
                chi = cell2.box_list[0];
              foreach (Cell cell3 in group)
              {
                //if third potential triple found
                if (cell1.name != cell3.name && cell2.name != cell3.name
                      && cell3.box_set.Count < 4)
                {
                  if (cell3.box_set.Count == 2)
                  {
                    if ((cell3.box_list[0] == phi && cell3.box_list[1] == chi) ||
                        (cell3.box_list[1] == phi && cell3.box_list[0] == chi))
                    {
                      ps = new int[3] { phi, psi, chi };//We found a triple!
                      moves = N3Ehelper(moves, group, ps, cell1, cell2, cell3);
                    }
                  }
                  else if (cell3.box_set.Count == 3)
                  {
                    if (cell3.box_set.Contains(phi) && cell3.box_set.Contains(psi)
                          && cell3.box_set.Contains(chi))
                    {
                      ps = new int[3] { phi, psi, chi };//We found a triple!
                      moves = N3Ehelper(moves, group, ps, cell1, cell2, cell3);
                    }
                  }
                }
              }
            }
            else if (cell2.box_set.Count == 3)
            {
              if (cell2.box_set.Contains(phi) && cell2.box_set.Contains(psi))
              {
                foreach (int p in cell2.box_list)
                  if (p != phi && p != psi)
                    chi = p;
                foreach (Cell cell3 in group)
                {
                  //if third potential triple found
                  if (cell1.name != cell3.name && cell2.name != cell3.name &&
                        cell3.box_set.Count < 4)
                  {
                    if (cell3.box_set.Count == 2)
                    {
                      if ((cell3.box_list[0] == phi && cell3.box_list[1] == chi) ||
                      (cell3.box_list[1] == phi && cell3.box_list[0] == chi))
                      {
                        ps = new int[3] { phi, psi, chi };//We found a triple!
                        moves = N3Ehelper(moves, group, ps, cell1, cell2, cell3);
                      }
                    }
                    else if (cell3.box_set.Count == 3)
                    {
                      if (cell3.box_set.Contains(phi) && cell3.box_set.Contains(psi)
                           && cell3.box_set.Contains(chi))
                      {
                        ps = new int[3] { phi, psi, chi };//We found a triple!
                        moves = N3Ehelper(moves, group, ps, cell1, cell2, cell3);
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
      else if (cell1.box_set.Count == 3)
      {
        phi = cell1.box_list[0];
        psi = cell1.box_list[1];
        chi = cell1.box_list[2];
        foreach (Cell cell2 in group)
        {
          //if second potential triple found
          if (cell1.name != cell2.name && cell2.box_set.Count < 4)
          {
            //check whether it has 3 phis
            if (cell2.box_set.Count == 3)
            {
              if (cell2.box_set.Contains(phi) && cell2.box_set.Contains(psi)
                   && cell2.box_set.Contains(chi))
              {
                foreach (Cell cell3 in group)
                {
                  if (cell1.name != cell3.name && cell2.name != cell3.name
                       && cell3.box_set.Count < 4)
                  {
                    if (cell3.box_set.Count == 2)
                    {
                      if ((cell3.box_list[0] == phi && cell3.box_list[1] == chi) ||
                      (cell3.box_list[1] == phi && cell3.box_list[0] == chi))
                      {
                        ps = new int[3] { phi, psi, chi };//We found a triple!
                        moves = N3Ehelper(moves, group, ps, cell1, cell2, cell3);
                      }
                    }
                    else if (cell3.box_set.Count == 3)
                    {
                      if (cell3.box_set.Contains(phi) && cell3.box_set.Contains(psi)
                           && cell3.box_set.Contains(chi))
                      {
                        ps = new int[3] { phi, psi, chi };//We found a triple!
                        moves = N3Ehelper(moves, group, ps, cell1, cell2, cell3);
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
    return moves;
  }
```

```
List<Move> N3Ehelper(List<Move> moves,Cell[] group,int[] ps,Cell cell1,Cell cell2,Cell cell3)
{
  foreach (Cell cell4 in group)
  {
    if (cell4.name != cell1.name && cell4.name != cell2.name && cell4.name != cell3.name)
    {
      foreach (int p in ps)
      {
        if (!cell4.negdiamond_set.Contains(p) && !cell4.filled)
        {
          moves.Add(new Move(cell4.name, "-d", p, N3Eskip, "Naked␣Triples␣Elimination",
              "N3E,␣because␣cells␣c" + cell1.name + ",␣c" + cell2.name + "␣and␣c" +
              cell3.name+"␣form␣a␣Naked␣Triple␣of␣which␣either␣one␣of␣the␣Three␣must␣" +
              "hold␣digit␣"+p+",␣so␣in␣cell␣c" + cell4.name + "␣there␣cannot␣be␣PM␣" + p));
        }
      }
    }
  }
  return moves;
}

/*------------------------------------2.3.7------------------------------------*/
List<Move> PTE(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  foreach (int phi in sudoku.PMs)//for all possible phi
  {
    Cell[] block = sudoku.blocks[updated_cell.whichblock];
    List<Cell> diamonds = new List<Cell>(); //keep track of which cells contain diamond phi
    List<Cell> negdiamonds = new List<Cell>();
    foreach (Cell cell in block)
    {
      if (cell.diamond_set.Contains(phi))
        diamonds.Add(cell);
      if (cell.negdiamond_set.Contains(phi))
        negdiamonds.Add(cell);
    }
    //if 2/3 cells in the same house contain dia phi and the other 7/6 contain neg di phi
    if (diamonds.Count == n && negdiamonds.Count == (9 - n))
    {
      //keep track if these cells are in the same horizontal or vertical
      bool samehorizontal = true;
      bool samevertical = true;
      int horizontal = diamonds[0].whichhorizontal;
      int vertical = diamonds[0].whichvertical;
      int blocki = diamonds[0].whichblock;
      foreach (Cell cell in diamonds)
      {
        if (cell.whichhorizontal != horizontal)
          samehorizontal = false;
        if (cell.whichvertical != vertical)
          samevertical = false;
      }
      //if these cells are in the same horizontal
      if (samehorizontal)
      {
        foreach (Cell cell in sudoku.horizontals[diamonds[0].whichhorizontal])
        { //then for all other c in this horizontal
          if (!cell.negdiamond_set.Contains(phi) && cell.whichblock !=
          blocki && cell.filled == false) //if not yet updated, not og cells & not done
          {
            moves.Add(new Move(cell.name, "-d", phi, PTEskip, "Pointing␣Tuples␣Elimination",
                "P"+n+"E,␣because␣there␣is␣a␣Pointing␣Tuple␣of␣cells␣containing␣PM␣"
                +phi+",␣which␣prevents␣that␣cell␣c"+cell.name+"␣can␣contain␣PM␣"+phi));
          }
        }
      }
      //if these cells are in the same vertical
      if (samevertical)
      { //then for all other c in this vertical
        foreach (Cell cell in sudoku.verticals[diamonds[0].whichvertical])
```

```
            if (!cell.negdiamond_set.Contains(phi)&&cell.whichblock!=blocki&&cell.filled==false)
            { //if not yet updated and not original cells and not done
              moves.Add(new Move(cell.name, "-d", phi, PTEskip, "Pointing␣Tuples␣Elimination",
                  "P"+n+"E,␣because␣there␣is␣a␣Pointing␣Tuple␣of␣cells␣containing␣PM␣"+phi+
                  ",␣which␣prevents␣that␣cell␣c" + cell.name + "␣can␣contain␣PM␣" + phi)); ;
              //cell.NegdiamondAdd(phi); //c entails neg diamond
            }
        }
    }
    if (moves.Count >= 2) break;
  }
  return moves;
}

/*------------------------------------2.3.8------------------------------------*/
List<Move> H2E(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  int[] candidates = sudoku.PMs;
  List<Tuple<Cell, Cell, int>> pair_cells = new List<Tuple<Cell, Cell, int>>();
  foreach (int phi in candidates)
  {
    foreach (int i in updated_cell.whichgroup)
    {
      Cell[] group = sudoku.groups[i];

      //keep track of which cell entails diamond phi
      List<int> phis = new List<int>();
      List<int> negphis = new List<int>();
      List<Cell> phicells = new List<Cell>();

      foreach (Cell cell in group)
      {
        if (cell.diamond_set.Contains(phi))
        {
          phis.Add(phi);
          phicells.Add(cell);
        }
        else if (cell.negdiamond_set.Contains(phi))
        {
          negphis.Add(phi);
        }
      }

      //if there is a pair of cells that entails a certain diamond phi
      if (phis.Count == 2 && negphis.Count == 7)
      {
        pair_cells.Add(new Tuple<Cell, Cell, int>(phicells[0], phicells[1], phi));
      }
    }
  }

  foreach (Tuple<Cell, Cell, int> cellpair in pair_cells)
  {
    foreach (Tuple<Cell, Cell, int> neighbor in pair_cells)
    {
      //If we find a pair of cells which both contain di phi and di psi
      if (cellpair.Item1 == neighbor.Item1 && cellpair.Item2 == neighbor.Item2
            && cellpair.Item3 != neighbor.Item3)
      {
        //we found a hidden pair! //for both of these cells:
        foreach (Cell cell in new Cell[2] { cellpair.Item1, cellpair.Item2 })
        {
          foreach (int chi in cell.diamond_set)
          {
            //if ther was a chi possible in this cell, it is not possible anymore
            if (chi!=cellpair.Item3&&chi!=neighbor.Item3 && !cell.negdiamond_set.Contains(chi))
            {
              moves.Add(new Move(cell.name, "-d", chi, H2Eskip, "Hidden␣Pairs␣Elimination",
              "H2E,␣because␣cells␣c" + cellpair.Item1.name + "␣and␣c" + cellpair.Item2.name
              + "␣form␣a␣Hidden␣pair" +",␣as␣they␣are␣the␣last␣in␣their␣group␣that␣can␣" +
              "contain␣either␣PM␣" + cellpair.Item3 + "␣or␣" + neighbor.Item3 +
              ", so cell c" + cell.name + " cannot contain PM " + chi));
            }
          }
        }
      }
    }
  }
  return moves;
}

/*-------------------------------------2.3.9-------------------------------------*/
List<Move> XWE(Cell updated_cell, int n)
{
  List<Move> moves = new List<Move>();
  moves = XWEhelper(moves, sudoku.horizontals, sudoku.verticals, 1);
  moves = XWEhelper(moves, sudoku.verticals, sudoku.horizontals, 0);
  return moves;
}

List<Move> XWEhelper(List<Move> moves, Cell[][] lines1, Cell[][] lines2, int hv)
{
  int[] candidates = sudoku.PMs;
  foreach (int phi in candidates)
  {
    List<Tuple<Cell, Cell, int>> pair_cells = new List<Tuple<Cell, Cell, int>>();
    foreach (Cell[] line1 in lines1)
    {
      List<int> phis = new List<int>();
      List<int> negphis = new List<int>();
      List<Cell> phicells = new List<Cell>();
      foreach (Cell cell in line1)
      {
        if (cell.diamond_set.Contains(phi))
        {
          phis.Add(phi);
          phicells.Add(cell);
        }
        else if (cell.negdiamond_set.Contains(phi))
        {
          negphis.Add(phi);
        }
      }
      if (phis.Count == 2 && negphis.Count == 7)
      {
        pair_cells.Add(new Tuple<Cell, Cell, int>(phicells[0], phicells[1], phi));
      }
    }
    foreach (Tuple<Cell, Cell, int> cellpair in pair_cells)
    {
      foreach (Tuple<Cell, Cell, int> neighbor in pair_cells)
      {
        if (cellpair!=neighbor&&cellpair.Item1.whichline[hv]==neighbor.Item1.whichline[hv]
            && cellpair.Item2.whichline[hv] == neighbor.Item2.whichline[hv])
        {
          //we hebben een xwing!
          Cell[][] xverticals = CombineArray(new Cell[1][]
          {lines2[cellpair.Item1.whichline[hv]]},new Cell[1][]{lines2[cellpair.Item2.whichline[hv]]});
          foreach (Cell[] xvertical in xverticals)
            foreach (Cell cell in xvertical)
              if (cell != cellpair.Item1 && cell != cellpair.Item2 && cell != neighbor.Item1
                  && cell != neighbor.Item2 && !cell.negdiamond_set.Contains(phi))
                moves.Add(new Move(cell.name, "-d", phi, XWEskip, "X-Wing Elimination",
                    "XWE, because the cells c" + cellpair.Item1.name + ", c" +
                    cellpair.Item2.name + ", c" + neighbor.Item1.name + " and c" +
                    +neighbor.Item2.name + " form an X-Wing. They all hold " +
                    phi + ", therefore cell c" + cell.name + " cannot hold " + phi));
        }
      }
    }
  }
  return moves;
}
```

Appendix B

Dataset Sudoku's

```
#364     , easy , nnyy
760000450000001080100200060204070300070000600008302000000100000000000025430007000
762983451945761283183254967254679318379815642618342579526198734897436125431527896

#13725  , easy , nnny
050010020000942000370000000560000001000805000000090080106000000000003400004000706
459317628681942375372568149568724931917835264243196587126479853795683412834251796

#25791  , easy , nnnn
051000000000809300000000421803200000000000260007400000060004009000070000500080017
351642798274819356986357421893265174415798263627431985768124539139576842542983617

#96157  , easy , nyyy
090600000060000804100000000002030700000007200008000000000008400010370900200080500 43
895624137263715894174389265642538719351972486789461352526843971437196528918257643

#100857, easy , nnny
000000470000040000070010008060205901003000000709006000206100700000603020100800000
851362479692748513374519268468275931523981647719436852236194785987653124145827396

#116130, easy , nyyy
000080000002500003030000120000360000080023001064700000847000000000000206001000000 45
961382754725469831384751269213648597859237416647915328476593182598124673132876945

#117003, easy , nnyy
000007000400200060600000009300007000050280000000090000300040590200200030573800060 40
253967814498231765617548293146372589532894176879615432764159328921483657385726941

#152843, easy , nyyy
000030000400010000008000040000000007600006000001200300526000943007000000600935000 80
951834627436217958728695143349158276587962314612473895265789431874321569193546782

#181679, easy , nnyy
000010093008400000046090005001080060070009050000000000010753000000070020400000000 0
752816493918453627346297815491385762273649158865172349127534986589761234634928571

#223051, easy , nnyy
050600000800000000000009036000000010000010274000080609305300000000000470026670000 040
953642817816735294247198365495381672361927458728564931534216789189473526672859143

______________________________________________________________________________

#1844    , medium, nnnn
050400060081000000200300080000000190000000872003200000000000070005000020109460000 000
953418762681752493274369581748231956516987234329546817192873645835624179467195328

#28507  , medium, nnyy
920000000300000790000008200740000000003016000000000005107020030400670000500000580 0
928476513364512798157938246741859632532164987689327451875291364496783125213645879

#31745  , medium, nnyy
000002003700500060009000804940000000050003000000008000070006800950060070000004501 70
685942713734518962219376854948765321527134689361829547473681295156297438892453176
```

#43387 , medium, nnny
003000080040005060602700000135608000090301000004000000000900100500106000000000208
953264781741895362682713549135628974897341625264579813478932156529186437316457298

#59467 , medium, nnny
000008090000600000300000004800063100230500000704000005050320001000010800006000700
612748593547639218389152674895263147231574986764891325958327461473916852126485739

#66325 , medium, nnyy
200007081000000670500040000001003000002600800008204005470000900000890000005000006
234967581819325674567148392651783249742659813398214765476531928123896457985472136

#78353 , medium, nnyy
000082000007000003080019005001040000203000010000006900005430700000600408100000050
536782149917564283482319675791843562263975814854126937625438791379651428148297356

#79327 , medium, nnyy
000008090900020030804700000000036000009000060045000270060001009000200000280000500
372658194956124738814793652728536941139472865645819273567381429491265387283947516

#111085, medium, nnyy
009000030600000504040080000080000060000060009210000400000004781100070006000095000
759426138628317594341589627983741265475263819216958473592634781134872956867195342

#195964, medium, nnyy
010400200000000000000500437080310000003084060000000004090000000070006590000900703
719463258534872619862591437486315972923784165157629384295137846378246591641958723

#1467 , hard , nnyy
004000000000006052001003700082000400300910000000000520003000286690000000000504000
534279861879146352261853794782635419345912678916487523453791286697328145128564937

#4074 , hard , nnyy
030590100000000780060000040700008500900300000300002000001040096000010004002000001
438597162159426783267831945724168539915374628386952417571243896893615274642789351

#13987 , hard , nnyy
000095600004001000600000080420000005007680000000030009000400208371000000000500000
283749561754861923619352784428197635937685142165234879596473218371928456842516397

#18161 , hard , nnny
000570000920000000000000938014000000050800014000600050200000700800000600000103000
183579426926438175547216938714352869652897314398641257261985743835724691479163582

#51116 , hard , nnyy
900001040000600030080090007006000003000000090050174000000710004004002000067803005
975231846421687539683495127716928453842356791359174682538719264194562378267843915

#87130 , hard , nnnn
089060000300001000600009040500300900000000002410006000000000820904700005005000700
289463157347851296651279348562347981793185462418926573176534829924718635835692714

#88563 , hard , nnnn
000003600071000002040020380000008400030000001086000070720010000010950000003070000
298743615371685942645129387157298436932467851486531279729314568814956723563872194

#98234 , hard , nnnn
007430000100006000000500008000800700000061802001000005009000040600002307010000290
567438129198276453342519678936825714475961832281347965729683541654192387813754296

#114032, hard , nnyy
600000070005900000090204100002000009050600000000000041000060800040028000039400050
684135972215976483793284165472851639951643728368792541527369814146528397839417256

#169585, hard , nnny
320000041600009000910000037004580009000037000000000004000000000030042000000060850
325678941647319285918254637764581329192437568853926174286795413531842796479163852