Instituto Tecnológico Autónomo de México

Un algoritmo con fundamentos teóricos para resolver Sudokus

Tesis

para obtener el título de

Licenciado en Matemáticas Aplicadas

presenta

Esteban Martínez Licón

Asesor

Dr. Andreas Wachtel

Ciudad de México 2022

Agradecimientos

Cuando terminé prepa no encontré palabras para darles las gracias a mis papás por todo lo que habían hecho por mí. Hoy me considero más letrado que hace 5 años pero todavía soy incapaz de darles las gracias con palabras. Sospecho que, sin importar mi nivel académico, nunca voy a encontrar en mi vocabulario algo digno de un agradecimiento hacia ustedes.

Gracias Bárbara, por manipular conciente e inconcientemente cada una de mis deciciones académicas. Gracias por ponerle la vara muy alta a mis otros modelos a seguir.

Gracias Andreas por la pasión con la que te entregaste a este trabajo.

La primer Macintosh tiene una placa adentro con los nombres de las personas que la hicieron.

Claudia María Licón Ávila

Jorge Bernardo Martínez Aguilar

Resumen

Esta tesis aporta dos contribuciones principales, un algoritmo de código abierto Solveku que es capaz de resolver cualquier sudoku, y una teoría del sudoku que crea un modelo matemático del acertijo.

Solveku está compuesto por seis técnicas que emulan técnicas humanas para resolver sudokus y combinadas son capaces de resolver cualquier sudoku. En la tesis descomponemos cada técnica en un análisis que se divide en tres partes. La primera parte es la explicación que simplemente describe la técnica desde un punto de vista pedagógico con la intención de mostrar al lector la motivación detrás de ésta. La segunda parte es la teoría del sudoku relacionada a la técnica, esta sección contiene el fundamento matemático de la técnica así como una prueba utilizando la teoría para demostrar la validez de la técnica. Al hablar de validez nos referimos al contexto del acertijo, es decir, demostramos que la técnica de ninguna manera nos puede llevar a una solución errónea. Finalmente, la tercera parte del análisis contiene una implementación en Python de la técnica con una breve descripción del código y un análisis de complejidad computacional.

Después de describir todas las técnicas que componen Solveku, empieza la parte estadística de la tesis. En primer lugar, hacemos un estudio de Solveku por sí mismo, es decir aplicamos el algoritmo a múltiples sudokus y vemos qué técnicas son las más utilizadas. Posteriormente, comparamos el comportamiento de Solveku contra los algoritmos más populares para resolver sudokus en internet.

La teoría de sudoku se desarrolla a la par de las técnicas de Solveku y su objetivo principal es demostrar la validez de cada una de las técnicas del algoritmo. Sin embargo, consideramos que nuestra teoría de sudoku es una contribución autónoma puesto que sus beneficios no tienen porqué estar limitados a Solveku. La teoría describe sudoku como un modelo matemático, y ésta puede ser usada para cualquier análisis matemático del acertijo, no limitado al desarrollo de algoritmos.

Finalmente, el algoritmo Solveku tiene varias contribuciones que representan una verdadera innovación en el contexto de sudokus. En primer lugar el algoritmo resuelve sudokus de tamaño n por n, en vez de limitarse al tamaño clásico de 3 por 3, lo cual nos permitió hacer un análisis de complejidad computacional en la implementación de las técnicas de Solveku. Además, limitando el algoritmo al clásico 3 por 3, éste resulta más eficiente que los algoritmos más populares disponibles en internet. Por otro lado, el algoritmo es de código abierto, esto no tiene precedentes ya que los algoritmos que resuelven sudokus emulando técnicas humanas son rentables pues también sirven para generar sudokus. Por último, la mayor contribución de este trabajo se da gracias a la combinación de la teoría y el algoritmo Solveku, ya que esta combinación nos permitió demostrar que Solveku está matemáticamente fundamentado y así probar que es capaz de resolver cualquier sudoku.

Contents

Symbols

The following list collects symbols that are used or defined in this work.

n	perfect square, size of the sudoku puzzle (see page 2).
I	set of indexes (see page 8).
$\mathbb{G}$	grid of a Sudoku (see page 8).
$\mathbb{R}_i$	i^{th} row of the grid. (see page 11).
$\mathbb{C}_j$	j^{th} column of the grid. (see page 11).
$\mathbb{B}_i$	i^{th} box of the grid. (see page 11).
Ω	set of valid symbols. (see page 11).
γ	value function. (see page 12).
Γ	set value function. (see page 12).
$\preceq$	restriction of a function. (see page 13).
$\mathbb{F}_0$	Fixed initial partition (see page 14).
$(n, \mathbb{F}_0, \gamma_0)$	sudoku puzzle. (see page 14).
$\mathbb{N}$	neighbors function. (see page 28).
$\mathbb{AS}$	available set function. (see page 28).
$(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$	k^{th} setting of a puzzle. (see page 32).
$\mathcal{O}$	time complexity's big O (see page 39).
$\perp$	Orthogonal intersection between two brackets (see page 99).
Λ_k	Orthogonal intersection function for a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ (see page 99).

$\mathfrak{R}$	Set of rows in an orthogonal subset (see page 100).
$\mathfrak{C}$	Set of columns in an orthogonal subset (see page 100).
R	Set of all the rows in a grid $\mathbb{G}$ (see page 106).
C	Set of all the columns in a grid $\mathbb{G}$ (see page 106).

Chapter 1

Introduction

In this work, we present *Solveku*, an algorithm capable of solving any sudoku puzzle, more efficiently than common solving algorithms. Furthermore, we introduce a sudoku theory, a body of knowledge based on definitions that allow us to establish a mathematical model for the puzzle.

We deeply explain how Solveku works, and based on our sudoku theory, we will prove that it solves every puzzle. Finally, we will test our algorithm exhaustively to understand how it behaves and compare its performance against the most popular solving algorithms available.

1.1 The puzzle

Sudoku is a combinatorial number-placement puzzle; its solving is completely based on logic. The puzzle is made up of a $n \times n$ grid where $\sqrt{n} \in \mathbb{N}$, which contains inside n different $\sqrt{n} \times \sqrt{n}$ subgrids we call boxes. The objective of the puzzle is to fill the grid with numbers from 1 to n, so that each row, each column, and each box have a single appearance of each number.

For the classic sudoku $n = 9$, therefore, it is a 9×9 grid and should be filled with numbers from 1 to 9. The puzzle begins with a partially completed grid that should have a valid unique solution. The numbers that pre-fill the grid by the puzzle setter are called ***hints***. Typically, there is an inverse relationship between the number of hints and the difficulty of the puzzle, meaning the more hints a sudoku has, the easier it is to solve.

The puzzle has become widely popular, and it is a non-discriminatory puzzle, meaning that the puzzler does not need any kind of requirements or specific knowledge to complete the puzzle, only a pencil and some logic. This work may seem to be out of time, since most of the academic work related to sudoku puzzles was done between 2005 and 2015 for the simple reason that it was then when the puzzle became a global outbreak. As Sivamani once said, “It is never too late to do anything new when it comes to art”.

1.2 The history

1.2.1 The etymology

Even though the origin of the puzzle is not clear, we called the 9×9 grid the classical sudoku because it was the one that gained popularity in the 1980's thanks to the Japanese puzzle publisher *Nikoli*. Nikoli has been publishing puzzles since 1980, they currently hold the Guinness Record for the largest crossword puzzle. Besides that, they claim to have coined the sudoku puzzle, and their founder Maki Kaji is widely known as "The godfather of sudoku" [Cam21].

Kaji hardly invented the sudoku puzzle, but he had a major role in popularizing the game, and naming it [Bel21]. Sudoku is shortened from *Suuji wa dokushin ni kagiru* which means "digits should remain single". In fact, numbers are unrelated to the puzzle since they only play the part as symbols that should not be repeated, there is no condition between them so a sudoku could be played with any different symbols. That is why, in spite of numbers being involved, arithmetic is useless for solving the puzzle.

1.2.2 The outbreak

Even though the origin of the sudoku puzzle is very unclear, it is very likely that Latin Squares were it's first predecessor, since both share the same non-repeating symbol principle. The name "latin square" was inspired by the mathematician Leonard Euler [WW11].

Definition 1.2.1. A ***latin square*** is an $n \times n$ array which is filled with n different symbols, with $n \in \mathbb{N}$. Where each row and column contains all of the n symbols.

The name was given because Euler used Latin characters as symbols to fill the latin square. Euler began the general theory of Latin Squares which nowadays has a lot of applications in statistics and mathematics.

Corollary 1.2.2. *A finished sudoku puzzle is a Latin square.*

In fact all finished sudokus are a subset of all the Latin Squares, this is because the no repetition rule in rows and columns is the same in both, however sudokus add that rule to boxes as well.

The applications of Latin square in statistics were leaded by Sir Ronald Fisher, in fact, in his book *The Design of Experiments*, published in 1935, which is considered a foundational work on experiment design, the whole fifth chapter is dedicated to Latin Squares. [Fis74].

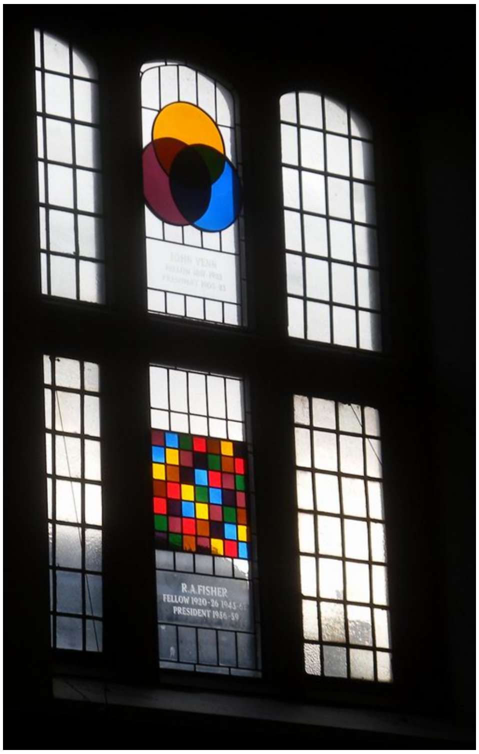


Figure 1.1: Sir Ronald Fisher window in Cambridge, showing a latin square, right below the Venn window [San09](The window was removed due to Fisher's support of Euginic ideas).

Before sudoku's worldwide fame, in the late 1970s, besides Nikoli, Dell Puzzle Magazines was quietly producing a very similar puzzle called *Number Place*.

The global outbreak happened almost 20 years after that, when a judge from New Zealand called Wayne Gould was shopping at a bookstore in Tokyo, after recently moving to Honk Kong. While in the bookstore, Gould found a sudoku puzzle from Nikoli.

Gould became fascinated by the puzzle and spent the next six years of his life developing an algorithm named Pappocom that could create sudoku puzzles [Smi05]. He let hundreds of newspapers around the world run his Pappocom sudoku puzzles free in return for promoting Gould's computer program and books [Sho06]. Since his action popularized the sudoku puzzle, he became one of Time's Magazine 100 World's most influential people of 2006. Gould also has spent some time developing solving techniques for sudokus, one of which we use in Solveku.

After being propagated by several magazines and newspapers, thanks to Gould, this puzzle spread with an incredibly rapid worldwide rise.

Thousands of books and software related to the puzzle have been published. It was featured on game consoles such as the Nintendo DS. A simple search on GitHub, the Internet hosting for software development, yields more than 56 thousand results of repositories related to the puzzle.

In 2008 when Apple, the American multinational technology company, launched its App store, the first application store that allows third parties to develop software for the company's mobile devices. Just two weeks after the launch, about 30 different apps were made that feature the sudoku puzzle.

In June of that same year, 105 witnesses and three months of evidence were wasted because a drug trial costing 1 million dollars was aborted when it emerged that jurors had been playing sudoku since the trial's second week. This was possible due to the fact that the puzzle only requires a paper, a pen, and concentration which may lead to think that a person is taking notes rather than filling a sudoku Grid. Even the author of these pages took advantage of this trick during the 2 hour Friday lectures on Investigation methodology on his senior year of high school.

1.3 Theory introduction

In this chapter we introduce all the foundational concepts of our sudoku theory. Definitions and affirmations are made for an $n \times n$ grid, where n is a perfect square, which means that $\sqrt{n}$ is a natural number. However, the examples will be done on a classic-size sudoku, $n = 9$, just for visual simplicity and because that is the standard size of sudoku puzzles.

For the ordering of the grid, we start counting indexes on zero because in most of the programming languages the start index for arrays is zero, so it simplifies the correspondence with the code implementation. Let $I = \{0, 1 \ldots n - 1\}$ be the set of indexes.

Definition 1.3.1. We define a ***Grid*** $\mathbb{G}$ as the following Cartesian product $\mathbb{G} = I \times I$. The elements of $\mathbb{G}$, (i, j) $i, j \in I$ will be called ***cells***. To refer to a specific cell, we will use the notation $cell_{ij} = (i, j) \in \mathbb{G}$.

Each occurrence of a double stroked letter in this work, such as $\mathbb{G}$, will denote a set of cells, that is, a subset of $\mathbb{G}$. In this way, it will be easier to recognize and differentiate a set of cells from a number or a set of numbers.

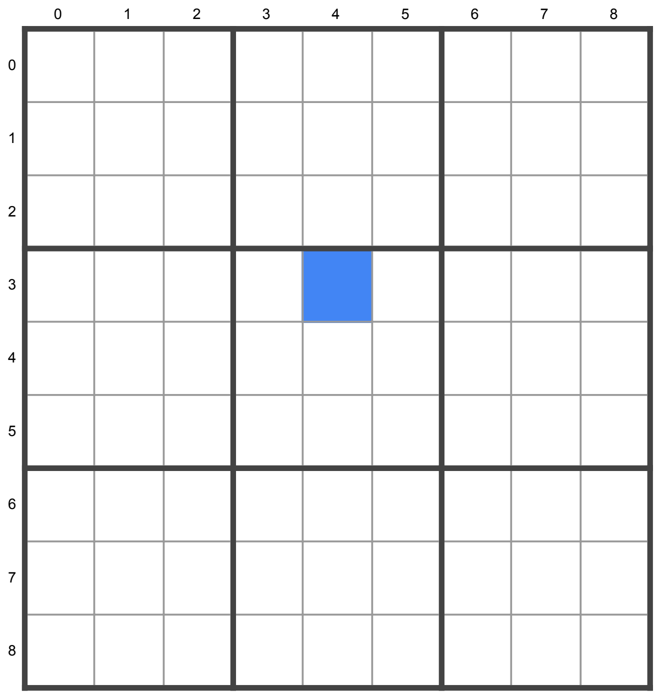


Figure 1.2: A sudoku grid with the $cell_{34}$ highlighted.

Next, we will define the concept of row and column, which are quite essential in the rules of the puzzle. We will also state and index the concept of the box. The boxes are the little $\sqrt{n} \times \sqrt{n}$ subgrids, which arise as the only difference between a sudoku and a Latin square.

Similarly to rows and columns, the boxes will also be indexed by I. Now, to formally identify a box, we will start with a function that separates and groups cells by the box to which they belong. Note that the creation of boxes of equal size n is possible and guaranteed because n is a perfect square. For an $n \times n$ grid, there will be n different boxes. The indexation order will go from top to bottom and from left to right, as shown in Figure 1.3.

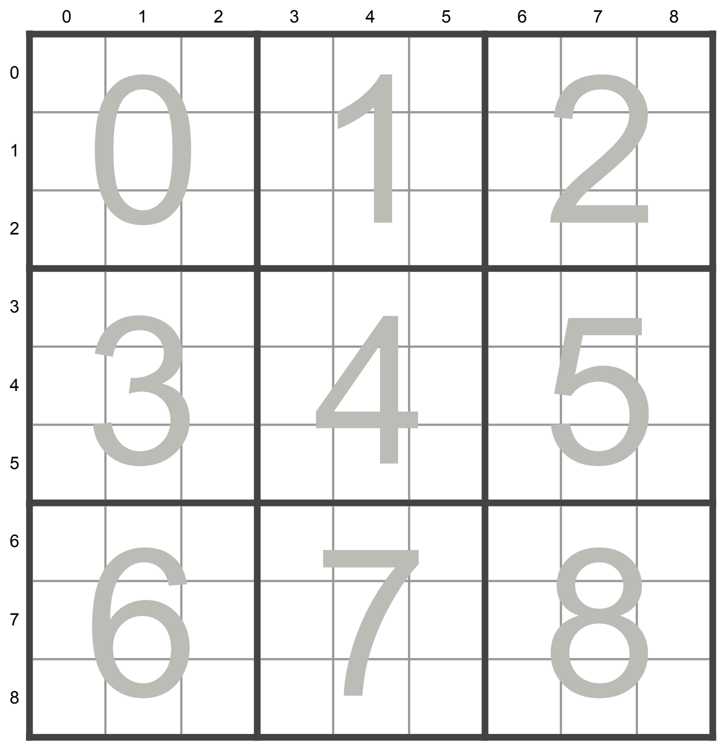


Figure 1.3: A classical sudoku grid with labeled boxes.

Definition 1.3.2. We define *bof* : $\mathbb{G} \to I$ as the function that assigns the corresponding box index to a particular cell:

$$\mathit{bof}(\mathit{cell}_{ij}) = \sqrt{n}\left\lfloor \frac{i}{\sqrt{n}} \right\rfloor + \left\lfloor \frac{j}{\sqrt{n}} \right\rfloor \qquad \text{for all } i, j \in I.$$

Notice that the fact that $\sqrt{n}$ is a natural number implies that $\mathit{bof}(\mathit{cell}_{ij})$ is also a natural number and that $\mathit{bof}(\mathit{cell}_{ij}) \in I$ for all $i, j \in I$.

Let us verify the *bof* function with a few examples.

Example 1.3.3. In a classical sudoku grid ($n = 9$), the first cell of the grid, the cell $cell_{00}$ should be in the first box, that is, the box with index 0.

$$bof(cell_{00}) = \sqrt{n}\left\lfloor\frac{0}{\sqrt{n}}\right\rfloor + \left\lfloor\frac{0}{\sqrt{n}}\right\rfloor = \sqrt{9}\left\lfloor\frac{0}{\sqrt{9}}\right\rfloor + \left\lfloor\frac{0}{\sqrt{9}}\right\rfloor = 3\left\lfloor\frac{0}{3}\right\rfloor + \left\lfloor\frac{0}{3}\right\rfloor = 0.$$

Now, let us say that the $cell_{45}$, again in the original grid, relies right next to the center of the grid, still in the center box, and therefore it belongs to the box number 4, since the indexation begins at 0, and there are three boxes above and one on the left.

$$\begin{aligned} bof(cell_{45}) = \sqrt{n}\left\lfloor\frac{4}{\sqrt{n}}\right\rfloor + \left\lfloor\frac{5}{\sqrt{n}}\right\rfloor = \sqrt{9}\left\lfloor\frac{4}{\sqrt{9}}\right\rfloor + \left\lfloor\frac{5}{\sqrt{9}}\right\rfloor &= 3\left\lfloor\frac{4}{3}\right\rfloor + \left\lfloor\frac{5}{3}\right\rfloor \\ &= 3 + 1 = 4. \end{aligned}$$

Definition 1.3.4. For all $i, j \in I$, we define:

The i^{th} ***Row*** $\mathbb{R}_i \subseteq \mathbb{G}$ as follows: $\mathbb{R}_i = \{cell_{ik} : \forall k \in I\}$.

The j^{th} ***Column*** $\mathbb{C}_j \subseteq \mathbb{G}$ as follows: $\mathbb{C}_j = \{cell_{kj} : \forall k \in I\}$.

The i^{th} ***Box*** $\mathbb{B}_i \subseteq \mathbb{G}$ as follows: $\mathbb{B}_i = \{cell \in \mathbb{G} : bof(cell) = i\}$.

Also, we say that a subset $\mathbb{S}$ of $\mathbb{G}$ is a ***bracket*** if there exists $i \in I$ such that $\mathbb{S} = \mathbb{R}_i$ or $\mathbb{S} = \mathbb{C}_i$ or $\mathbb{S} = \mathbb{B}_i$.

The grouping of rows, columns, and boxes in the concept of brackets will be very useful because there are several properties that apply to all brackets regardless of the type.

Now that we have defined the visual elements of the grid, we can focus on the puzzle itself. The objective of the puzzle is to fill the grid with symbols that follow specific rules. Then, each cell will eventually be related with a unique symbol. Therefore, the next step is to assign a relationship between each *cell* and a symbol.

First, let Ω be the set of symbols that will fill the grid; in fact, Ω can be any set of n different symbols, but for simplicity and because it is a common practice, we define it as the following set: $\Omega = \{1, 2, \ldots, n\}$. Note that unlike I, Ω starts with 1 just like any sudoku puzzle out there.

Now, we address the relationship between $\mathbb{G}$ and Ω. As with any relationship in mathematics, it will be done through a function. The next definition refers to a function that assigns values of Ω to each cell in $\mathbb{G}$. Note that at the beginning of the puzzle there are some values of *cells* that are momentarily unknown. Therefore, we define this relationship for any subset of $\mathbb{G}$.

Definition 1.3.5. We define a ***value function*** as a function that assigns cells in $\mathbb{S} \subset \mathbb{G}$ to a particular value in Ω. We denote it by $\gamma : \mathbb{S} \subseteq \mathbb{G} \to \Omega$.

Given this relationship between a cell and a symbol, it will also be useful to have the equivalent for a set of cells. Meaning, for a set of cells, we next define a function that returns a set of the values associated with these cells.

Definition 1.3.6. Given a value function $\gamma : \mathbb{S} \to \Omega$, we define its ***set associated value function*** Γ as follows, $\Gamma(\mathbb{X}) = \{\omega \in \Omega :$ there exists $cell \in \mathbb{X}$ that $\gamma(cell) = \omega\}$ for every $\mathbb{X} \subset \mathbb{G}$.

The set-associated value function will only be used for brackets, so that we can tell which values are already in a bracket, since, as we will see in the rules, no values should be repeated in brackets.

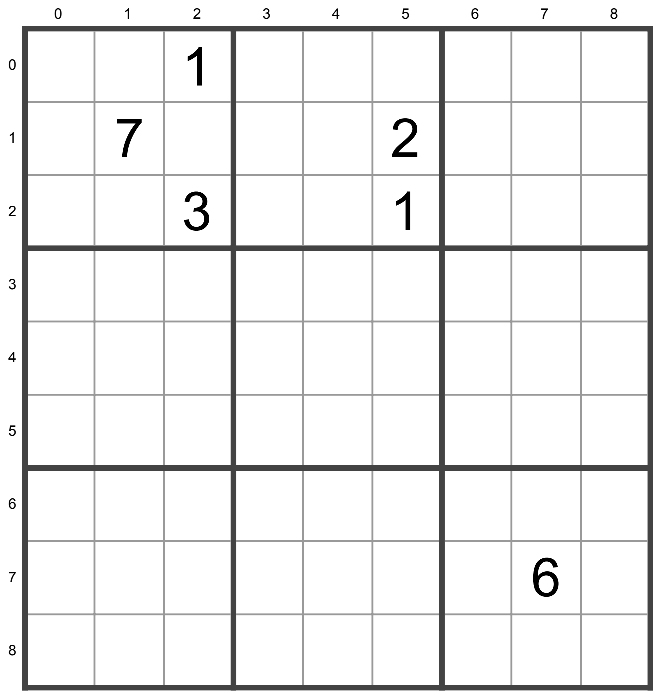


Figure 1.4: Grid with a value function.

Example 1.3.7. In Figure 1.4 we have a grid with a value function:

$$\gamma(cell_{ij}) = \begin{cases} 1 & \text{if } (i,j) \in \{(0,2),(2,5)\} \\ 7 & \text{if } (i,j) = (1,1) \\ 2 & \text{if } (i,j) = (1,5) \\ 3 & \text{if } (i,j) = (2,2) \\ 6 & \text{if } (i,j) = (7,7) \end{cases}$$

Then, for instance, $\Gamma(\mathbb{R}_0) = \{1\}$, $\Gamma(\mathbb{R}_1) = \{2,7\}$, and $\Gamma(\mathbb{B}_0) = \{1,3,7\}$.

Definition 1.3.8. Let $f : X \to Y$ be a function, let $X_0 \subseteq X$ we say that

$f_0 : X_0 \to Y$ is a ***restriction*** of f if the following condition holds true:

- $f(x_0) = f_0(x_0)$ for all $x_0 \in X_0$

We use the notation $f_0 \preceq f$ for stating that f_0 is a restriction of f.

Restriction is a concept that is widely used in linear algebra and specifically in vector spaces. In this work we will restrict its definition to functions which helps us establish relationships between different value functions.

Definition 1.3.9. A ***sudoku puzzle*** consists in the following three parts.

- A perfect square n, such that $\sqrt{n}$ is a natural number.
- An initial condition of the puzzle as follows, a set of cells $\mathbb{F}_0 \subset \mathbb{G}$ that have a fixed associated value of Ω.
- An initial value function $\gamma_0 \colon \mathbb{F}_0 \to \Omega$.

We will use the following notation for a sudoku puzzle: $(n, \mathbb{F}_0, \gamma_0)$.

Example 1.3.10. In Figure 1.5 we have the following initial state: $\mathbb{F}_0 = \{cell_{06}, cell_{13}, cell_{25}, cell_{33}, cell_{78}\}$. Note that all *cells* in $\mathbb{F}_0$ are colored blue.

$$\gamma_0(cell_{ij}) = \begin{cases} 3 & \text{if } (i,j) = (0,6) \\ 5 & \text{if } (i,j) = (1,3) \\ 1 & \text{if } (i,j) = (2,5) \\ 3 & \text{if } (i,j) = (3,3) \\ 4 & \text{if } (i,j) = (7,1) \end{cases}$$

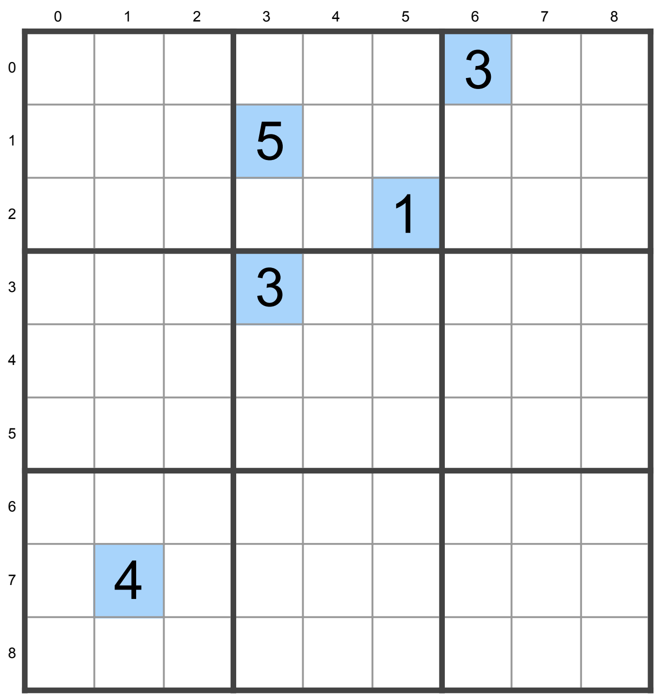


Figure 1.5: Representing the sudoku puzzle in Example 1.3.10.

Definition 1.3.11. Given a sudoku puzzle $(n, \mathbb{F}_0, \gamma_0)$, a ***solution*** is a value function $\gamma \colon \mathbb{G} \to \Omega$ that satisfies the following properties:

- γ_0 is a restriction of γ.
- $|\Gamma(\mathbb{R}_i)| = n \ \forall i \in I$, that is, each row contains every number in Ω.
- $|\Gamma(\mathbb{C}_j)| = n \ \forall j \in I$, that is, each column contains every number in Ω.
- $|\Gamma(\mathbb{B}_i)| = n \ \forall i \in I$, that is, each box contains every number in Ω.

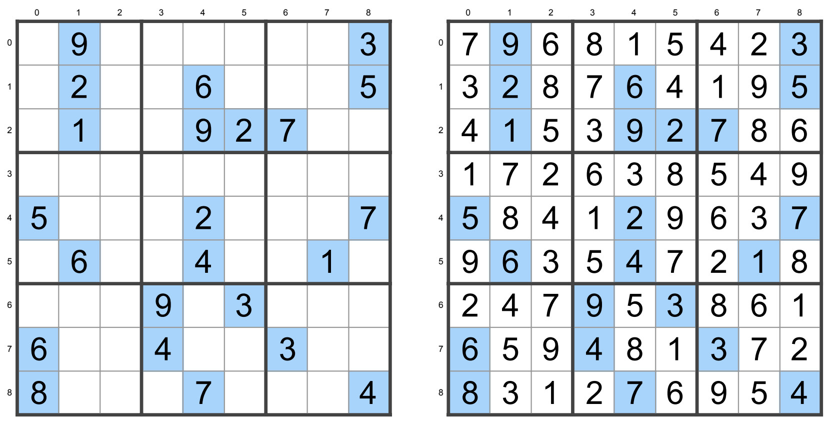


Figure 1.6: A sudoku puzzle in the left, and a solution to that puzzle in the right.

Lemma 1.3.12. *For a bracket $\mathbb{S}$, the following expressions are equivalent:*

- $|\Gamma(\mathbb{S})| = n$.
- *$\forall \omega \in \Omega$ there is exactly only one pair (i, j) such that $\gamma(cell_{ij}) = \omega$ and $cell_{ij} \in \mathbb{S}$.*
- *There is no pair $(i_1, j_1) \neq (i_2, j_2) \in \mathbb{S}$ such that $\gamma(cell_{i_1 j_1}) = \gamma(cell_{i_2 j_2})$*

Proof. The equivalence between the second statement and the third statement is self-evident. We now prove the equivalence between the first and second statements.

We have $|\mathbb{S}| = n$ and $|\Gamma(\mathbb{S})| = n$. Since they are of the same size, by the pigeonhole principle, for all $\omega \in \Omega$ there is exactly only one pair (i, j) such that $\gamma(cell_{ij}) = \omega$. $\square$

Definition 1.3.13. We say that a puzzle $(n, \mathbb{F}_0, \gamma_0)$ is ***well defined*** if there exists a unique solution for that initial state.

	0	1	2	3	4	5	6	7	8
0	4	9		8	2		7	5	1
1	2	1	8	7	5	4	6	9	3
2	7	5		1	9		4	8	2
3	5	3	1	9	4	8	2	7	6
4	8	2	7	3	6	1	9	4	5
5	6	4	9	2	7	5	1	3	8
6	1	7	5	4	3	2	8	6	9
7	3	8	4	6	1	9	5	2	7
8	9	6	2	5	8	7	3	1	4

Figure 1.7: A not well-defined puzzle with two possible solutions.

Example 1.3.14. We see that the Figure 1.7, with the following puzzle $(n, \mathbb{F}_0, \gamma_0)$ is not a well-defined puzzle.

Because both

$$\gamma(cell_{ij}) = \begin{cases} \gamma_0(cell_{ij}) & \text{if } cell_{ij} \in \mathbb{F}_0 \\ 3 & \text{if } (i,j) = (0,2) \\ 6 & \text{if } (i,j) = (0,5) \\ 6 & \text{if } (i,j) = (2,2) \\ 3 & \text{if } (i,j) = (2,5) \end{cases}$$

$$\gamma'(cell_{ij}) = \begin{cases} \gamma_0(cell_{ij}) & \text{if } cell_{ij} \in \mathbb{F}_0 \\ 6 & \text{if } (i,j) = (0,2) \\ 3 & \text{if } (i,j) = (0,5) \\ 3 & \text{if } (i,j) = (2,2) \\ 6 & \text{if } (i,j) = (2,5) \end{cases}$$

are solutions. A good example of a well-defined puzzle is in Figure 1.6, since the solution in its right is a unique solution, but proving the uniqueness of a solution is complicated and will not be done until the end of Chapter 3.

1.4 The mathematics of sudoku

Even though the sudoku puzzle has been judged as not being a mathematical puzzle due to the fact that the numbers just play the role of symbols and there is no arithmetic involved, this could not be more wrong.

Mathematics has not been linked to sudoku as much as it should because there are no arithmetic requirements to solve the puzzle, that much is true. Nevertheless, a theory based on mathematical logic helps answer many of the questions related to the puzzle. Below we list some of the most interesting questions related to sudoku puzzles that were allowed to be solved through a mathematical approach.

We introduce these facts after the introduction of our sudoku theory, to show usefulness outside of our algorithm. Our theory will help us understand some of the concepts used to solve these difficult problems and prove some results.

1.4.1 Minimum number of clues for a sudoku puzzle

One of the most popular math problems related to sudoku is answering the following question: What is the smallest number of clues that can be given such that a sudoku puzzle has a unique completion?

In January 2012, in the book *Taking sudoku Seriously, The Math Behind the World's Most Popular Pencil Puzzle* Jason Rosenhouse and Laura Taalman devoted Section 9.4 to this problem and claimed as *The Rock Star Problem*, saying: "If you can figure it out, you will be a rock star in the universe of people who care about such things. Granted, that is a far smaller universe than the one full of people who care about actual rock stars, but still, it would be great."

It is in fact a very complicated problem, even though no one has ever found a 16 clue sudoku puzzle, which is hardly a proof that it does not exist. Before answering whether there exists a 16 clue sudoku puzzle, there is an interesting result that provides a lower bound for the number of sudoku clues that can create a valid puzzle.

We now prove a theorem that establishes a lower bound of clues for sudoku puzzles of any size. By allowing us to do this proof, our sudoku theory shows its potential. This proof can easily be done with simple words, but we decided to do it with our theory as an introduction to how we are going to prove the results in the following chapters. The following theorem allows to switch symbols in a solution if these symbols were not contained in the puzzle's initial condition.

Theorem 1.4.1. *For a sudoku puzzle $(n, \mathbb{F}_0, \gamma_0)$, where $2 < n$, let γ be a solution, $s_1, s_2 \in \Omega \setminus \Gamma(\mathbb{F}_0)$, $s_1 \neq s_2$ and*

$$\gamma^*(cell_{ij}) = \begin{cases} \gamma(cell_{ij}) & \text{if } \gamma(cell_{ij}) \notin \{s_1, s_2\} \\ s_1 & \text{if } \gamma(cell_{ij}) = s_2 \\ s_2 & \text{if } \gamma(cell_{ij}) = s_1 \end{cases}$$

then γ^* is a solution.

Proof. Assume that γ^* is not a solution for $(n, \mathbb{F}_0, \gamma_0)$, then by definition 1.3.11 we have two possible cases:

Case 1: γ_0 is not a restriction of γ^*.

Then there must exist a $cell_{ij}$ such that $\gamma_0(cell_{ij}) \neq \gamma^*(cell_{ij})$.

Case 1.1: $\gamma^*(cell_{ij}) \notin \{s_1, s_2\}$

Then, by definition of γ^* we get $\gamma^*(cell_{ij}) = \gamma(cell_{ij})$,
subsequently $\gamma(cell_{ij}) \neq \gamma_0(cell_{ij})$, which implies
γ not a solution, and a contradiction.

Case 1.2: $\gamma^*(cell_{ij}) \in \{s_1, s_2\}$

By definition of γ, we get $\gamma(cell_{ij}) \in \{s_1, s_2\}$ as well, but γ_0 is a restriction of γ then $\gamma_0(cell_{ij}) \in \{s_1, s_2\}$, but by hypothesis, there is no such *cell* that $\gamma_0(cell) \in \{s_1, s_2\}$, since $s_1, s_2 \in \Omega \setminus \Gamma(\mathbb{F}_0)$.

However, Γ is the set associated function. So we have a contradiction.

Case 2: There is a bracket $\mathbb{S}$ such that $|\Gamma(\mathbb{S})| \neq 9$:

Then by Lemma 1.3.12, point three there exist two pairs of indexes $(i_1, j_1) \neq (i_2, j_2) \in \mathbb{S}$ with $\gamma^*(cell_{i_1 j_1}) = \gamma^*(cell_{i_2 j_2})$, then by definition of γ^* we have $\gamma(cell_{i_1 j_1}) = \gamma(cell_{i_2 j_2})$ for $(i_1, j_1), (i_2, j_2) \in \mathbb{S}$, therefore, we have the contradiction that γ is not a solution.

So by *reductio ad absurdum* we have that γ^* is a solution. □

Theorem 1.4.2. *For a sudoku puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, where* $2 < n$ *and* $|\mathbb{F}_0| \leq n - 2$ *then the puzzle is not well defined.*

Proof. As the size of $\mathbb{F}_0$ is less or equal than $n - 2$ then there exist two different numbers: $s_1, s_2 \in \Omega$ such that $s_1, s_2 \notin \Gamma(\mathbb{F}_0)$. Then by Theorem 1.4.1 for every solution γ to the puzzle. There exists γ^* a different solution, therefore there is not a unique solution for the puzzle, and the puzzle is not well defined. □

Given the last theorem we now have a lower bound for the number of clues a sudoku must have to be a well defined puzzle. In particular we can claim that if a classical sudoku has 7 hints or less then the puzzle is not well defined. Still with this lower bound there is a big gap between 7, the lower bound, to 17, the currently biggest lower bound. The bound is 17 because there have been found multiple puzzles with this number of clues and a unique solution, for instance the Figure 1.8. The proof can be found in [Che16].

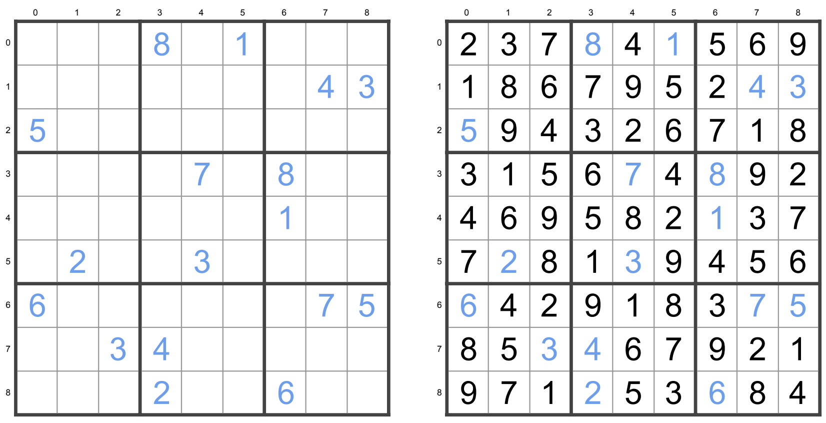


Figure 1.8: A 17 clue puzzle with $\Gamma(\mathbb{F}_0) = 8$ and a unique solution.

Regardless of the previous theorems, the question of the minimum well-defined sudoku puzzle stands, since we still do not know the minimum possible number of clues to have a well-defined puzzle.

Here is where Professor Gary McGuire's work from the School of Mathematical Sciences, University College Dublin, becomes relevant. He and some colleagues did a computer-assisted proof that there is no well-defined 16 clue sudoku puzzle. The proof was done with the help of his *checker* algorithm that was able to find more than one solution to every 16 hint sudoku puzzle [MG13].

1.4.2 Number of sudoku grids

In 2006, Professor Frazer Jarvis addressed the question of the number of sudoku grids in his paper *Mathematics of Sudoku I*. As we stated earlier, sudoku grids are special cases of latin squares, but even for latin squares the calculation is a difficult problem because no general formula is known.

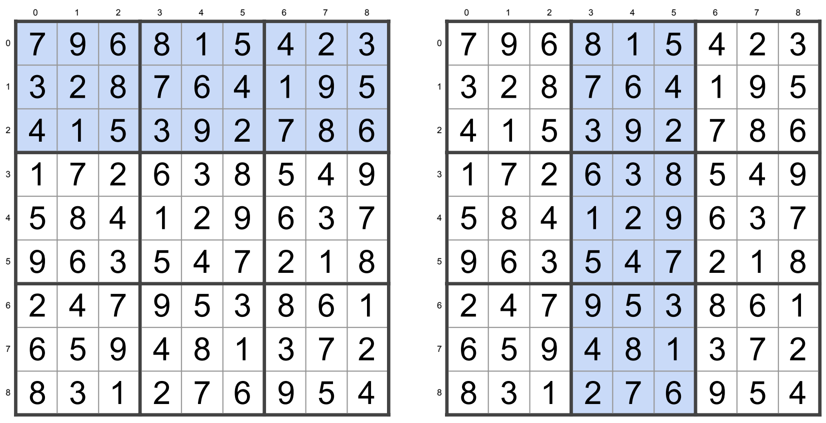
Jarvis used a brute-force calculation in a clever way to conclude, in a feasible computing time, that the total number of well defined Sudoku grids is $6670903752021072936960 \sim 6.671 \times 10^{21}$.

Then in *Mathematics of Sudoku II* Frazer reduces that number by identifying and eliminating (not counting) various symmetries. To understand these, we introduce the following definitions.

Definition 1.4.3. A ***stack*** is a union of $\sqrt{n}$ columns, performed in the following way $\text{Stack}_k = \bigcup_{i=k\sqrt{n}}^{i=(k+1)*\sqrt{n}-1} \mathbb{C}_i$ for all $k \in \{0, \sqrt{n}-1\}$

Very similarly on the horizontal side,

Definition 1.4.4. A ***Band*** is a group of $\sqrt{n}$ rows, done in the following way $\text{Band}_k = \bigcup_{i=k\sqrt{n}}^{i=(k+1)*\sqrt{n}-1} \mathbb{R}_i$ for all $k \in \{0, \sqrt{n}-1\}$



	0	1	2	3	4	5	6	7	8
0	7	9	6	8	1	5	4	2	3
1	3	2	8	7	6	4	1	9	5
2	4	1	5	3	9	2	7	8	6
3	1	7	2	6	3	8	5	4	9
4	5	8	4	1	2	9	6	3	7
5	9	6	3	5	4	7	2	1	8
6	2	4	7	9	5	3	8	6	1
7	6	5	9	4	8	1	3	7	2
8	8	3	1	2	7	6	9	5	4

	0	1	2	3	4	5	6	7	8
0	7	9	6	8	1	5	4	2	3
1	3	2	8	7	6	4	1	9	5
2	4	1	5	3	9	2	7	8	6
3	1	7	2	6	3	8	5	4	9
4	5	8	4	1	2	9	6	3	7
5	9	6	3	5	4	7	2	1	8
6	2	4	7	9	5	3	8	6	1
7	6	5	9	4	8	1	3	7	2
8	8	3	1	2	7	6	9	5	4

Figure 1.9: In the left, a puzzle with the first band highlighted. In the right a puzzle with the second stack highlighted.

Symmetries can have many different meanings, but in this context, Frazer specifies six different operations that create a group of all similar grids. The complete list of operations that Frazer states that can be performed without leaving the group of similar grids is the following:

- Relabeling symbols (permutations of symbols)
- Permutation of stacks
- Permutation of bands
- Permutation of columns in the same stack
- Permutation of rows in the same stack
- Any reflection or rotation

After applying some techniques of combinatorial mathematics to calculate all the possible symmetries that a grid could have, the number of possible sudoku grids has been calculated again. The number of different sudoku grids in different symmetric groups is $5472730538 \sim 5.47 \times 10^9$. This new number is much smaller than the complete set, which allowed symmetries.

Our theory has just been proved useful to explain and prove some of the most interesting mathematical results in the world of sudoku. Now, it is time to use it to start solving puzzles.

1.5 Solving Algorithms

Before taking off with our algorithm, let us review the most popular solving algorithms that already exist. In Chapter 4, we will actually compare our algorithm with some of the examples reviewed here.

First of all, taking advantage of the fact that the most common sudoku grid is a 9×9 grid, there have been many implementations to solve the puzzle using a backtracking algorithm. Such algorithms use brute force to solve the problem, taking into account the small size of the grid, a backtracking algorithm is able to solve most puzzles in a fraction of a second. However, there are some special cases that make this algorithm less feasible, which also occurs when n grows larger.

A popular variant of backtracking is Donald Knuth's Algorithm X. Which uses a technique known as dancing links. This technique uses recursion and backtracking through a data structure called a doubly linked list. In fact, when doing a google search, the first web page that shows up offering a sudoku solver claims to be using this technique. Dancing links are famous for their excellent performance results when solving sudoku puzzles [HL14].

One mathematical approach is to treat the problem as a linear programming problem. In this case, there is no objective function to optimize, but the constraints are taken care of in the feasible region, so the problem is solved using the famous simplex algorithm. Simplex has been proven to work just fine when there is a unique solution, which means when the puzzle is well defined [Van19].

More recently, artificial intelligence approaches have been applied, some specifically through machine learning. The puzzle was recently tried to be tackled with convolutional neural networks with a training of a million puzzles. This approach ended up with an accuracy of 0.86, where accuracy is defined as the number of sudoku puzzles solved divided by the total puzzles used to test the algorithm [Par18].

We also consider our algorithm an artificial intelligence approach since we are emulating human techniques when solving the puzzle. However, when these techniques are not able to reach a solution we end up backtracking, so we can also see our algorithm as an enhanced backtracking. We prefer to label our algorithm as an artificial intelligence algorithm, since backtracking is really a last resource, and as we will see in the performance section of Chapter 4, it is barely used.

But before, in the upcoming chapters 2 and 3, we specify and elaborate the human emulating techniques used in our algorithm, these techniques can be separated into two groups.

The first group will be the attesting techniques. These techniques are called attesting because, given a puzzle $(n, \mathbb{F}_0, \gamma_0)$ with a set of cells with defined values $\mathbb{F}_0$, these techniques will assign values to cells that did not have one (cells not in $\mathbb{F}_0$). The second group, the pruning techniques will be discussed in Chapter 3. They will help the attesting techniques to "know" which values should be associated to certain cells.

Chapter 2

Attesting stages of Solveku

2.0.1 Fundamentals

This subsection provides the fundamental theory for the code implementation of the attesting techniques. The following couple of definitions will be widely used throughout the rest of this chapter.

Definition 2.0.1. We define the ***neighbors*** $\mathbb{N}$ of a *cell*, as a subset of $\mathbb{G}$ that contains all the cells that share a bracket with the selected cell:

$$\mathbb{N}(cell_{ij}) = (\mathbb{R}_i \cup \mathbb{C}_j \cup \mathbb{B}_{bof(cell_{ij})}) \setminus \{cell_{ij}\} \, .$$

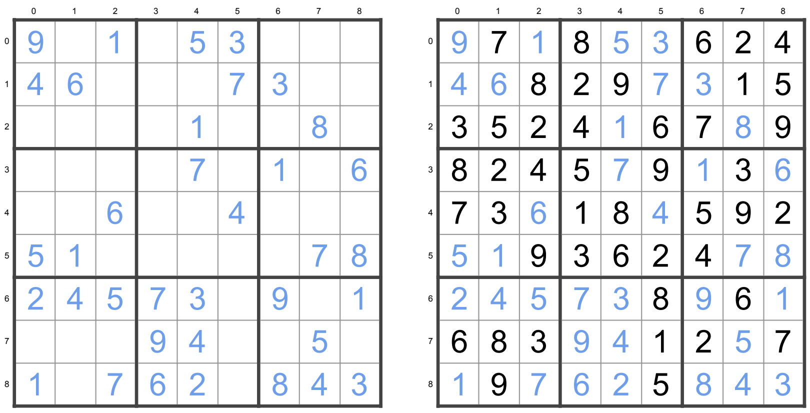
Sometimes we will write N_{ij} instead of $N(cell_{ij})$. Please do not misread the neighbors $\mathbb{N}$ as the set of natural numbers.

In chapter 1 we associated a cell to a value through the value function. Now, we need a function that links the cells that do not have a value with a set of possible values it may take (candidates). This set of candidates will be called the ***available set*** of a cell. The link between the cells and their available sets will be done through an ***available set function***.

Definition 2.0.2. Given a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we define the ***available set function*** $\mathbb{AS}$ as any function that has the following domain and image, $\mathbb{AS}\colon \mathbb{G} \to \mathcal{P}(\Omega)$. The function is ***valid*** if for every solution γ of the puzzle, we have that $\gamma(cell) \in \mathbb{AS}(cell)$ for every *cell*.

When an algorithm is solving the puzzle it can not use the definition to verify if an available set function is valid or not, because, of course, it does not have access to all the solutions of the puzzle. However, if the puzzle has a *cell* with an empty set, it becomes clear that for every solution γ we have that $\gamma(cell) \notin \mathbb{AS}(cell)$.

Subsequently, we illustrate available sets for the puzzle shown in Figure 2.1 (left) and its unique solution shown Figure 2.1 (right).

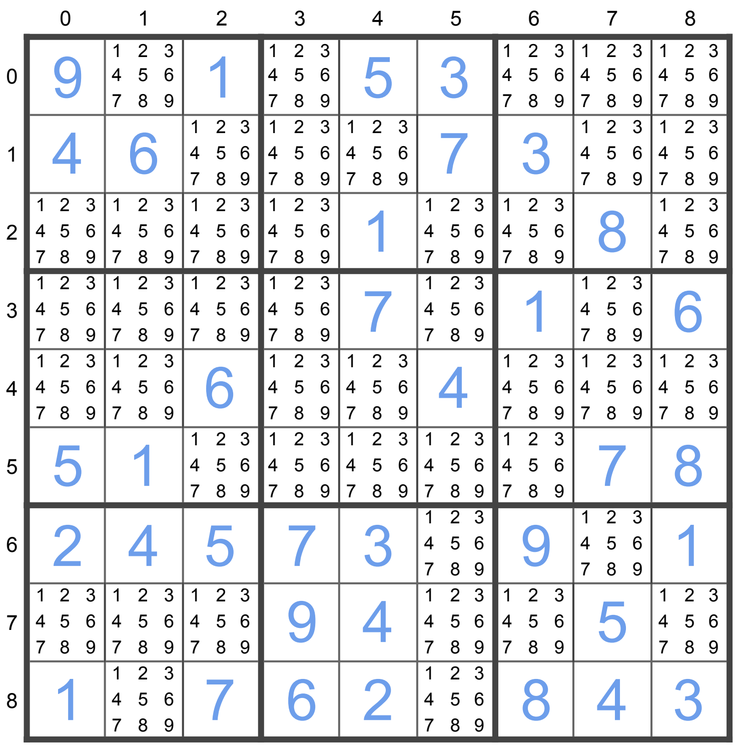


	0	1	2	3	4	5	6	7	8
0	9		1		5	3			
1	4	6				7	3		
2					1			8	
3					7		1		6
4			6			4			
5	5	1						7	8
6	2	4	5	7	3		9		1
7				9	4			5	
8	1		7	6	2		8	4	3

	0	1	2	3	4	5	6	7	8
0	9	7	1	8	5	3	6	2	4
1	4	6	8	2	9	7	3	1	5
2	3	5	2	4	1	6	7	8	9
3	8	2	4	5	7	9	1	3	6
4	7	3	6	1	8	4	5	9	2
5	5	1	9	3	6	2	4	7	8
6	2	4	5	7	3	8	9	6	1
7	6	8	3	9	4	1	2	5	7
8	1	9	7	6	2	5	8	4	3

Figure 2.1: Puzzle and its unique solution γ.

Example 2.0.3. Here are two examples of valid available set functions to the puzzle in Figure 2.1, a trivial one $\mathbb{AS}_1$ and a reduced one $\mathbb{AS}_2$. Each *cell*'s available set is represented with the small numbers inside it. The *cells* that are in the domain of the value function have an available set of one element, which is the value function's result. It is clear that for every *cell* we have that $\gamma(cell) \in \mathbb{AS}_1(cell)$ and $\gamma(cell) \in \mathbb{AS}_2(cell)$.



	0	1	2	3	4	5	6	7	8
0	9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	5	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1	4	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	6
4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
5	5	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	8
6	2	4	5	7	3	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1
7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9
8	1	1 2 3 4 5 6 7 8 9	7	6	2	1 2 3 4 5 6 7 8 9	8	4	3

Figure 2.2: The trivial AS_1 for the puzzle in the Figure 2.1.

	0	1	2	3	4	5	6	7	8
0	**9**	2 3 5 7 8	**1**	1 2 4 6 8 9	**5**	**3**	1 2 3 4 5 6 7 8 9	1 2 4 6 7 9	1 2 4 6 7 9
1	**4**	**6**	2 3 5 7 8	1 2 3 4 6 8 9	2 4 6 8 9	**7**	**3**	1 2 3 5 7 8 9	1 2 5 7 8 9
2	2 3 5 6 7 8	2 3 5 7 8	2 3 5 6 7 8	2 4 6 7 8 9	**1**	1 2 3 4 6 7 8 9	1 2 4 5 6 7 9	**8**	2 4 5 6 7 8 9
3	3 4 7 8 9	1 2 3 4 8 9	2 3 4 7 8 9	1 2 3 4 5 8	**7**	2 3 5 8 9	**1**	1 2 3 4 5 6 9	**6**
4	3 7 8 9	1 2 3 7 8 9	**6**	2 3 5 8 9	1 5 6 7 8 9	**4**	1 2 3 4 5 6 9	2 3 4 5 6 7 9	1 2 3 4 5 9
5	**5**	**1**	2 3 4 7 8 9	1 2 3 8	1 3 5 6 8 9	1 2 3 5 6 9	2 3 4 5 9	**7**	**8**
6	**2**	**4**	**5**	**7**	**3**	1 3 4 6 7 8	**9**	1 2 3 4 5 6 7 8 9	**1**
7	1 3 6 7 8	1 3 6 7 8 9	1 3 6 8	**9**	**4**	1 2 3 5 6 8	2 3 6 7 8	**5**	2 3 4 7 8 9
8	**1**	2 3 6 8 9	**7**	**6**	**2**	1 5 6 7 9	**8**	**4**	**3**

Figure 2.3: The slightly reduced AS_2 for the puzzle in the Figure 2.1.

Example 2.0.4. Now, Figure 2.4 shows an available set function of the puzzle in the Figure 2.1 which is not valid. It is not valid because γ is a solution and $\gamma(cell_{14}) = 9 \notin \mathrm{AS}(cell_{14}) = \{2, 4\}$. Another way to verify that AS is not valid is a further reduction of the available set. The set can be reduced by using the fact that a value can not be repeated in the same column. The presence of $2, 4$ in column 4 results in an empty set for $\mathrm{AS}(cell_{14})$.

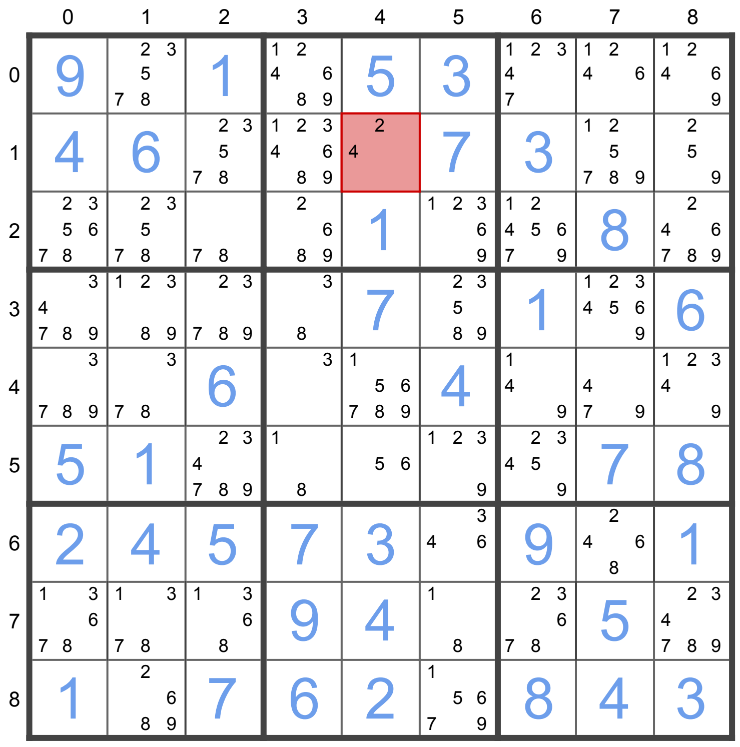


Figure 2.4: Invalid available set function for the puzzle in the Figure 2.1.

When solving a sudoku puzzle $(n, \mathbb{F}_0, \gamma_0)$ we aim to find a unique solution γ. However, it is unusual to determine γ directly from γ_0. Usually, we start to assign a value to cells that did not have one, which adds more constraints to the puzzle and reduces the available set of some cells. Then, these new constraints help us find new values for other cells and so on and so forth. In other words, we constantly update γ_0. Each time we update γ_0, we extend its domain gradually by iterating through different settings. We formalize this process, and the with the following definition.

Definition 2.0.5. Given a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we define $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ as a

setting, if the following conditions are met:

- $\mathbb{F}_0 \subseteq \mathbb{F}_k \subseteq \mathbb{G}$.
- $\gamma_k \colon \mathbb{F}_k \to \Omega$.
- AS_k is a valid available set function.

Additionally, we define the ***initial available set function*** as

$$\mathrm{AS}_0(cell_{ij}) = \begin{cases} \{\gamma_0(cell_{ij})\} & \text{if } cell_{ij} \in \mathbb{F}_0, \\ \Omega \setminus \Gamma_0(\mathbb{N}_{ij}) & \text{if } cell_{ij} \notin \mathbb{F}_0, \end{cases}$$

and the initial setting of the puzzle as $(\mathbb{F}_0, \gamma_0, \mathrm{AS}_0)$.

In the settings we no longer write the n, number that represents the size of the grid, nor the grid $\mathbb{G}$ itself. We stop writing them because both of them stay constant for every setting.

Theorem 2.0.6. *Given a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, the initial setting* $(\mathbb{F}_0, \gamma_0, \mathrm{AS}_0)$ *is indeed a setting.*

Proof. We have that $\mathbb{F}_0 \subseteq \mathbb{F}_0$, and by Definition 1.3.9 of γ_0, we get $\gamma_0 \colon \mathbb{F}_0 \to \Omega$, and $\gamma_0 \preceq \gamma_0$.

Then, what is missing to prove is that AS_0 is a valid available set function, that is, for every solution γ of $(n, \mathbb{F}_0, \gamma_0)$, and $cell \in \mathbb{G}$, $\gamma(cell) \in \mathrm{AS}_0(cell)$, if and only if, $\gamma(cell) \in \Omega \setminus \Gamma_0(\mathbb{N}(cell))$. Thus, we have to prove that $\gamma(cell) \notin \Gamma_0(\mathbb{N}(cell))$. Let us recall that $cell \notin \mathbb{N}(cell)$. Assuming $\gamma(cell) \in \Gamma_0(\mathbb{N}(cell))$, we have that there exists $cell'$ such that $cell' \in \mathbb{N}(cell) \cap \mathbb{F}_0$ and $\gamma(cell) = \gamma_0(cell')$. But $\gamma_0 \preceq \gamma$, then $\gamma(cell) = \gamma(cell')$. But $cell$ and $cell'$ are neighbors, so we have the contradiction γ is not a solution. Therefore, $\gamma(cell) \notin \Gamma_0(\mathbb{N}(cell))$, so $(\mathbb{F}_0, \gamma_0, \mathrm{AS}_0)$ is a setting. □

The available set and setting definitions will be useful when explaining the algorithm. The algorithm is made up by techniques we call *stages*.

Each stage of the algorithm can be thought as a function. A stage receives a puzzle setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$, and returns a new setting $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathbb{AS}_{k+1})$. The idea is to apply stages to the puzzle until a solution is found. Next we explore each stage deeply. After the exploration of all the stages we will state the complete algorithm. The algorithm is just the order and number of occurrences of which each stage is applied to the puzzle, concluded by backtracking if necessary.

The exploration of each stage contains three different parts

- Explanation,
- Sudoku theory,
- Code implementation.

The explanation part gives an insight about the idea that inspired the technique and how it works. The sudoku theory gives step by step guide of the stage and a rigorous demonstration without loss of generality, which proves that the result of a stage $((\mathbb{F}_{k+1}, \gamma_{k+1}, \mathbb{AS}_{k+1}))$ is actually a setting. When proving that the result of the stage is a setting, we are proving that no solutions are lost when applying this technique. The proof will be done by contradiction in every case, because we believe is the easiest way for most of the stages, we want to keep a consistency across all the stages.

Finally, we will have an explained Python code implementation, and a computational complexity analysis. The complete implementation is not shown, in this section we will just give an insight about the way we decided to implement the algorithm. However, the implementation shows a way to find these scenarios where the techniques can be applied and how to apply them also. Going through the code implementation is not necessary for the reader to understand the logic of the algorithm, but we consider it necessary if the reader wants to understand the complexity statements.

All the stages will be explained and described for a generalized $n \times n$

sudoku grid, using examples of only classical 9×9 grids.

The stages of the algorithm will be introduced in a specific order. This order was chosen to follow, what we think, is the order of techniques most people follow when solving a puzzle. The initial stages emulate the most basic techniques with very simple logic required, while later stages require a deeper level of abstraction.

Before going into the stages we show some fundamental code, that is, code that is a necessary base for understanding the implementation of every stage. Additionally we use $\mathcal{O}$ to calculate computational complexity. In each stage's code implementation we will analyze its time complexity using $\mathcal{O}$.

2.1 Fundamental code

For now, we introduce the three main classes, `SudokuCell`, `BracketContainer` and `SudokuGrid` with reduced functionality. We will later extend the functionality of these classes. Basic object oriented programming is required to understand this fundamental code. The class ***SudokuCell*** starts with the following three attributes:

- `value`, integer in Ω that represents the result of the value function applied to that cell in the grid.
- `av_set`, set that represents the available set of the cell.
- `fixed`, boolean that states if the cell is a hint or not.

Please note that if a cell has a `value` $= 0$ it means that the cell is not in $\mathbb{F}_0$. We use 0 because $0 \notin \Omega$. The default constructor will assign the attributes value to 0 and fixed to false. The SudokuCell also contains the following methods `has_value()`, boolean method that checks if the cell has an associated value.

```python
class SudokuCell:
    n = 9
    def __init__(self, value=0, fixed=False):
        self.value = value
        self.fixed = fixed
        self.av_set = None
        if not fixed:
            self.av_set = set()

    def has_value(self):
        return self.value != 0
```

Sudoku Cell class

The SudokuGrid class is initiated with the following attributes:

- `n`, which is the n that determines the size of the Grid ($n \times n$).
- `I` and `Omega` which stand for I and Ω respectively.
- `grid`, which is a matrix of SudokuCell. This data structure represents the actual puzzle grid.

Also, SudokuGrid is initiated with the following methods:

- `box_of()` calculates the box of a *cell* given by row and column.
- The constructor of the class which initiates the attributes and creates the grid.

The constructor receives a matrix of integers `number_grid` in $\{0\} \cup \Omega$, that represents the puzzle $(n, \mathbb{F}_0, \gamma_0)$. The entries of the matrix different from 0, represent the cells in $\mathbb{F}_0$.

```python
class SudokuGrid:
    n = 9
    I = [i for i in range(n)]
    Omega = [i for i in range(1,n+1)]

    def box_of(self, row, col):
        return 3 * int(row / 3) + int(col / 3)


    def __init__(self, number_grid):
        self.grid = [[SudokuCell(num, True) if num != 0 else SudokuCell() for
 num in row] for row in number_grid]
        for row in self.I:
            for col in self.I:
                cell = self.grid[row][col]
                if not cell.has_value():
                    for val in self.Omega:
                        cell.av_set.add(val)
```

Sudoku Grid class

For the Bracket container class, we will not be showing the implementation since it is a bit lengthy, but we will show how use it, and explain the general idea behind its implementation. BracketContainer class receives a `SudokuGrid` and groups the cells and its values by brackets. This is useful because the constraints of the puzzle are based on the bracket. The `BracketContainer` class will contain each type of bracket and their image grouped by type in arrays. We use the word image in a mathematical context, where the image of a function is the set of all output values it may produce.

For instance, `BracketContainer.row[i]` represents $\mathbb{R}_i$ as an array of cells, and `BracketContainer.rowImage[i]` represents $\Gamma(\mathbb{R}_i)$ for $i \in I$ as a set of numbers. For all brackets and images we have them separated by type (row, col or box) and also grouped together in an attribute called `all` and `all_images` of size $3n$. We also need the grouped because sometimes we will need to do operations to all the brackets regardless of their type. The brackets are stored together in the `all` attribute in the following order: row, column, box. And each of them is stored in ascending order. You can take a look at Figure 2.5 to understand the properties of the `BracketContainer` class.

Note that the brackets and their images are not stored twice. The way this works is that each type of bracket is pointing to a slice of `BracketContainer.all`, and each type of image is pointing to a slice of `BracketContainer.all_images`. For instance, in a classical grid, `col[2]` points to the same memory location as `all[9+2]`. So any changes done to `col[2]` will also be reflected in `all[9+2]`.

Now, assuming we have the implementation of `BracketContainer`

Now we add one `BracketContainer` object to the `SudokuGrid` class constructor.

```
class SudokuGrid:
    """..."""
    def __init__(self, number_grid):
        """..."""
        self.brackets = BracketContainer(self)
```

Bracket added to SudokuGrid

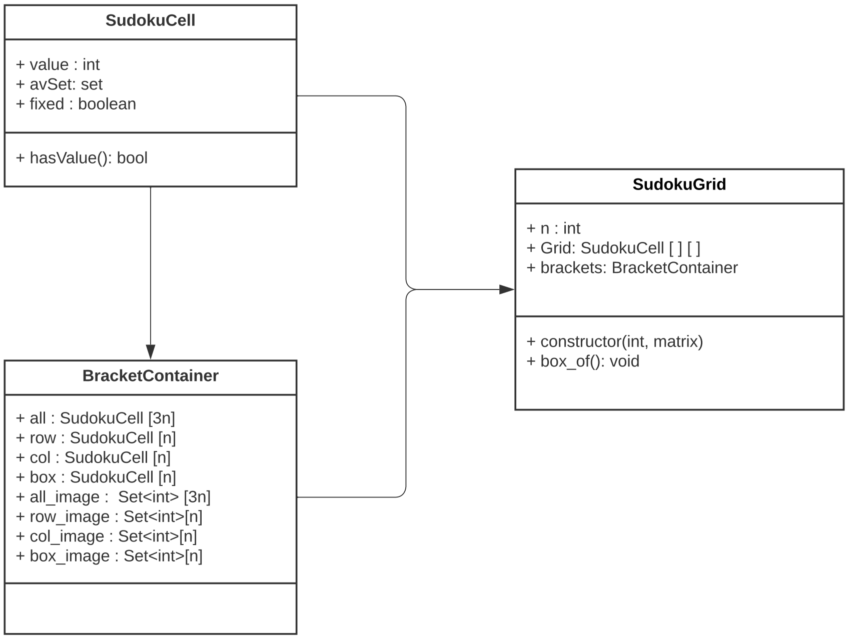


Figure 2.5: UML diagram of SudokuCell and SudokuGrid.

Finally, we take advantage of the `Bracket.row_image`, `Bracket.col_image` and `Bracket.box_image` to define the available sets of all the cells $\notin \mathbb{F}_0$ through a new method: `define_available_sets()`. For a candidate ω we have by Definition 2.0.5 of the initial state of available set that $\omega \in \mathrm{AS}_0(cell_{ij})$ if only if $\omega \notin \Gamma(\mathbb{N}_{ij})$ if only if $\omega \notin \Gamma(\mathbb{R}_i)$ and $\omega \notin \Gamma(\mathbb{C}_j)$

and $\omega \notin \Gamma(\mathbb{B}_{bof(i,j)})$. We use this last property to implement the initial state of the initial available sets in the code.

```python
class SudokuGrid:
    """..."""
    def define_initial_available_sets(self):
        for row in self.I:
            for col in self.I:
                cell = self.grid[row][col]
                if not cell.has_value():
                    for val in self.Omega:
                        if val in self.brackets.rowImage[row] or val in self.
 brackets.colImage[col] or val in self.brackets.boxImage[self.box_of(row, col
 )]:
                            cell.av_set.remove(val)

    def __init__(self, number_grid):
        """..."""
        self.define_available_sets()
```

Available Set definition

Before going through the techniques, we provide the reader with some fundamental theory for understanding the time complexity of the algorithm.

We use computational complexity to describe the behavior of the algorithms associated to the techniques of Solveku. Computational complexity is a simplified way of describing the amount of computing resources (time and space) that a particular algorithm consumes. In this work, we will only focus on time complexity, since the space complexity analysis would require advanced computer science knowledge which lies outside of the scope of this work.

We will over simplify the notion of time complexity by only providing the necessary theory required to understand the time complexity explored in this work. However, there are great resources that explain extensively the notion of time and space complexity [Pap14]. For measuring the time complexity of an algorithm we need a way of "counting" operations, we simplify the definition of operation to assignation and condition checking. Assuming these operations are equally expensive we just need to add them up and have a total number of operations.

Things get interesting when the number of operations of an algorithm depends on a variable value, in our case, all the stages depend on the value n, which is the size of the puzzle. Although we just simplified operations, accurately counting them can be a very difficult task, specially in times when the problem to solve is not deterministic, sudoku puzzles are not deterministic since the initial puzzle changes. Therefore, instead of counting all the operations, the convention is to provide a function that works as an upper bound and works for every possible case.

Definition 2.1.1. We say that an algorithm depending on n has a ***big O*** complexity of $\mathcal{O}(h(n))$ if there exists a constant $C \in \mathbb{R}$ such that the number of operations of the algorithm is smaller than $C \cdot h(n)$ for every $n \in \mathbb{N}$, where $h\colon \mathbb{N} \to \mathbb{R}$. Just for this case $\mathbb{N}$ is the set of natural numbers, and $\mathbb{R}$ is the set of real numbers.

Example 2.1.2. Let us calculate the time complexity of the function `complexity_example`. Assuming getting a random number is inexpensive, in line 2 we are just doing $3n$ assignations for every element of the array. Then, for every element we will be checking if its value is under 0.5 in line 4, these are $3n$ condition checks, which represent another $3n$ operations. Finally, in the worst case, every element will be under 0.5 and we will assign 0 to every element of the array, which are another $3n$ operations. Therefore, we have that the number of operations will be, in the worst case $3n + 3n + 3n = 9n$. Let $C = 10$, and $h(n) = n$ we have that the big O complexity of `complexity_example` is $\mathcal{O}(n)$ since $9n < 10n$ for every n in the natural numbers.

```
1 def complexity_example(n):
2     arr = [random.randint(0, 1) for i in range(3 * n)]
3     for i in range(3 * n):
4         if arr[i] < 0.5:
5             arr[i] = 0
```

Time complexity example.

In the literature, big O is simplified to just take in count the most significant elements in the function, we describe this simplification in the following ***addition axiom***.

Axiom 2.1.3. *For a function $h\colon \mathbb{N} \to \mathbb{R}$ such that $h(n) = h_1(n) + h_2(n) + \ldots + h_r(n)$, for r a finite natural number, and*

$$\lim_{n\to\infty} \frac{h_i(n)}{h_1(n)} = 0 \textit{ for } i \in \{2, \ldots, r\}\,.$$

We have that $\mathcal{O}(h(n)) = \mathcal{O}(h_1(n))$. Note that we chose h_1 without loss of generality since the order in $h_1(n) + h_2(n) + \ldots + h_r(n)$ is unimportant.

This simplification takes place because time complexity is usually calculated for asymptotic cases. Then, n grows really large and r is fixed, the contribution of all the h_i is negligible.

Example 2.1.4. Let us prove that for $h(n) = 7(2^n) + n^2 + 4n$ we have that $\mathcal{O}(h(n)) = \mathcal{O}(2^n)$. We define $h_1(n) = 7(2^n)$, $h_2(n) = n^2$ and $h_3(n) = 4n$. We then have that

$$\lim_{n\to\infty} \frac{n^2}{7(2^n)} = 0 \qquad \text{and} \qquad \lim_{n\to\infty} \frac{4n}{7(2^n)} = 0\,.$$

By the addition axiom we get $\mathcal{O}(h(n)) = \mathcal{O}(7(2^n))$. Finally, let $C = 8$, and $g(n) = 2^n$ we get that $h(n) < C \cdot g(n)$ for every n in the natural numbers. Then, by definition of big O complexity, we have that $\mathcal{O}(h(n)) = \mathcal{O}(2^n)$.

2.2 Stage One, Singletons

The first stage of this algorithm consists in singleton elimination. In mathematics, a singleton, also known as unit set, which is a set with exactly one element, a concept widely used in probability theory. Here we will take advantage of this concept to recognize when a cell should be assigned to a specific value.

Definition 2.2.1. Given a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we say that the pair cell and value $(cell^*, \omega)$ is a ***singleton*** when $cell^* \notin \mathbb{F}_k, \mathbb{AS}_k(cell^*) = \{\omega\}$.

Then next step, of course, will just be assigning ω as the new value of $cell^*$, this process is called ***singleton elimination*** and will be fully described in the sudoku theory section.

Example 2.2.2. In Figure 2.6 we show an example of a singleton in a setting. We have that $(cell_{20}, 2)$ is a singleton. Additionally, we highlighted the value 2 from available sets of some neighbors once we assign the value 2 to $cell_{20}$.

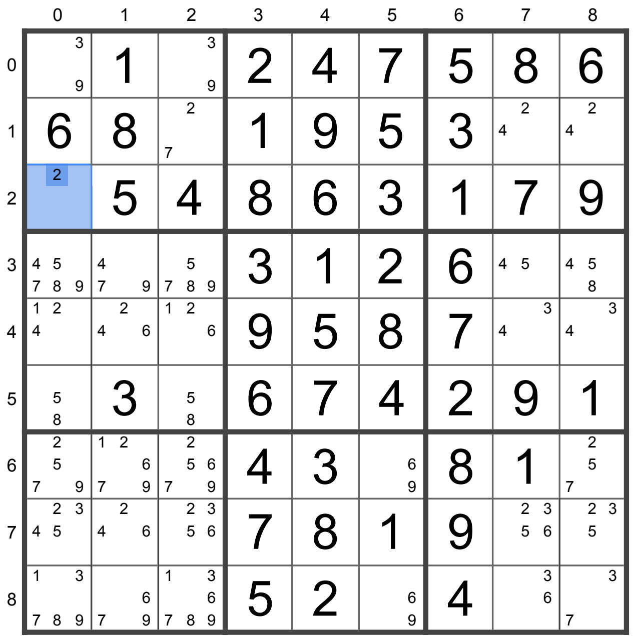


Figure 2.6: The $cell_{20}$ is a singleton.

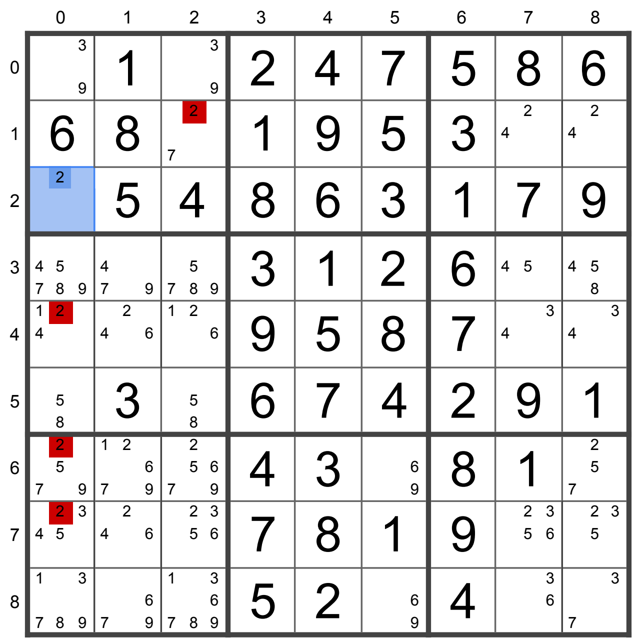


Figure 2.7: All the neighbors (in red) affected by the singleton $cell_{02}$.

2.2.1 Sudoku theory

Here we state the formal theory of the just described singleton elimination process.

Given a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we define a function f_1 that receives a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a singleton $(cell^*, \omega)$ and returns a new setting $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$.

Since, a setting consists of three different parts, f_1 can be split into three different functions f_1^1, f_1^2, f_1^3, such that,

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) &= (f_1^1(\mathbb{F}_k, cell^*), f_1^2(\mathbb{F}_k, \gamma_k, cell^*, \omega), f_1^3(\mathrm{AS}_k, cell^*, \omega)) \\ &= f_1((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (cell^*, \omega)).\end{aligned}$$

Given a singleton cell^*, we define:

$$\mathbb{F}_{k+1} = f_1^1(\mathbb{F}_k, \mathit{cell}^*) = \mathbb{F}_k \cup \{\mathit{cell}^*\},$$

for every $\mathit{cell} \in \mathbb{G}$:

$$\gamma_{k+1}(\mathit{cell}) = f_1^2(\mathbb{F}_k, \gamma_k, \mathit{cell}^*, \omega)\big|_{\mathit{cell}} = \begin{cases} \gamma_k(\mathit{cell}) & \text{if } \mathit{cell} \in \mathbb{F}_k \\ \omega & \text{if } \mathit{cell} = \mathit{cell}^* \end{cases}$$

$$\mathrm{AS}_{k+1}(\mathit{cell}) = f_1^3(\mathrm{AS}_k, \mathit{cell}^*, \omega)\big|_{\mathit{cell}} = \begin{cases} \mathrm{AS}_k(\mathit{cell}) & \text{if } \mathit{cell} \notin \mathbb{N}(\mathit{cell}^*) \\ \mathrm{AS}_k(\mathit{cell}) \setminus \{\omega\} & \text{if } \mathit{cell} \in \mathbb{N}(\mathit{cell}^*) \end{cases}$$

The following theorem states that the definition of f_1 above preserves all solutions by proving that the result of f_1 is indeed a setting.

Theorem 2.2.3. *Given a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of puzzle* $(n, \mathbb{F}_0, \gamma_0)$ *and a singleton* $(\mathit{cell}^*, \omega)$*,* $f_1((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathit{cell}^*, \omega)) = (\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$ *is a setting.*

Proof. By definition of f_1^1 and f_1^2, it is clear that $\mathbb{F}_k \subseteq \mathbb{F}_{k+1}$ and $\gamma_k \preceq \gamma_{k+1}$. Now we only have left to prove that AS_k is a valid available set function. We aim to prove this by contradiction. Hence, AS_{k+1} is not a valid available set function, then, there must exist a solution γ and cell', such that $\gamma(\mathit{cell}') \notin \mathrm{AS}_{k+1}(\mathit{cell}')$. We distinguish two different cases.

Case 1, $\mathit{cell}' \notin \mathbb{N}(\mathit{cell}^*)$:

Then, by definition of AS_{k+1} by f_1^3, we have $\mathrm{AS}_{k+1}(\mathit{cell}') = \mathrm{AS}_k(\mathit{cell}')$. Therefore, we conclude $\gamma(\mathit{cell}') \notin \mathrm{AS}_k(\mathit{cell}')$, but this contradicts that AS_k is a valid available set function.

Case 2, $\mathit{cell}' \in \mathbb{N}(\mathit{cell}^*)$:

Then, as AS_k is a valid available set function we get $\gamma(\mathit{cell}') \in \mathrm{AS}_k(\mathit{cell}')$, so by definition of f_1^3 we have $\gamma(\mathit{cell}') \in \mathrm{AS}_k(\mathit{cell}') \setminus$

$\mathrm{AS}_{k+1}(cell') = \{\omega\}$ and $\gamma(cell') = \omega$. Nevertheless, as γ is a solution and $cell^*$ is a singleton, $\gamma(cell^*) = \omega$. Then, we get $\gamma(cell^*) = \gamma(cell')$, which contradicts that γ is a solution, because $cell'$ and $cell^*$ are neighbors.

By *reducto ad absurdum*, AS_{k+1} is a valid available set function, and therefore, $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$ is a setting. □

2.2.2 Implementation

First, we create two new methods in the SudokuGrid class: `update_neighbors_available_set` and `update_cell` which will be used for assigning a value to a cell and handling all the corresponding aftermath. The `update_cell` function is called when a cell becomes valued with ω, it sets the value attribute of the cell to ω, and adds ω to the images of the corresponding brackets. Then, the function `update_neighbors_available_set` removes ω from the available sets of all the neighbors that do not have an associated value. Note that `update_neighbors_available_set` is triggered by `update_cell`. In fact, these two functions will be useful through the whole algorithm, whenever we discover the value of a cell.

```python
class SudokuGrid:
    """..."""

    def update_neighbors_available_set(self, row, col, num):
        for it in self.I:
            self.brackets.row[row][it].av_set_remove(num)
            self.brackets.col[col][it].av_set_remove(num)
            self.brackets.box[self.box_of(row, col)][it].av_set_remove(num)

    def update_cell(self, row, col, num):
        self.grid[row][col].value = num
        self.grid[row][col].av_set = None
        if num in self.brackets.rowImage[row] or num in self.brackets.colImage[
    col] or num in self.brackets.boxImage[
                self.box_of(row, col)]:
                raise Exception("Group constraint violated")
        self.brackets.rowImage[row].add(num)
        self.brackets.colImage[col].add(num)
        self.brackets.boxImage[self.box_of(row, col)].add(num)
        self.update_neighbors_available_set(row, col, num)
```

Cell update

Now, we execute stage 1, we iterate through every cell in the Grid, and use ***update_cell*** whenever we find a singleton. Here, ω is the only number in the *cell*'s available set.

```
class SudokuGrid:
    """..."""
    def stage_one(self):
        for row in self.I:
            for col in self.I:
                cell = self.grid[row][col]
                if not cell.has_value() and len(cell.av_set) == 1:
                    for num in cell.av_set:
                        self.update_cell(row, col, num)
```

Stage 1

An important thing to notice is that as the algorithm keeps moving through the grid, some cells might be new singletons. These new singletons were created because the algorithm affects the available set of some cells. This is illustrated in Figure 2.8. There is no logic difference between the implementation and the theoretical method. As $cell_{10}$ is a singleton, when updating its neighbors, $cell_{22}$ gets updated and becomes a singleton. Now, as (1,0) is checked before (2,2), $cell_{22}$ is treated as a singleton in the same iteration. However, if the rows 1 and 2 were exchanged in the same puzzle, then the singleton $cell_{20}$ would cause a singleton in $cell_{12}$ and this new singleton would have to be treated in the next iteration.

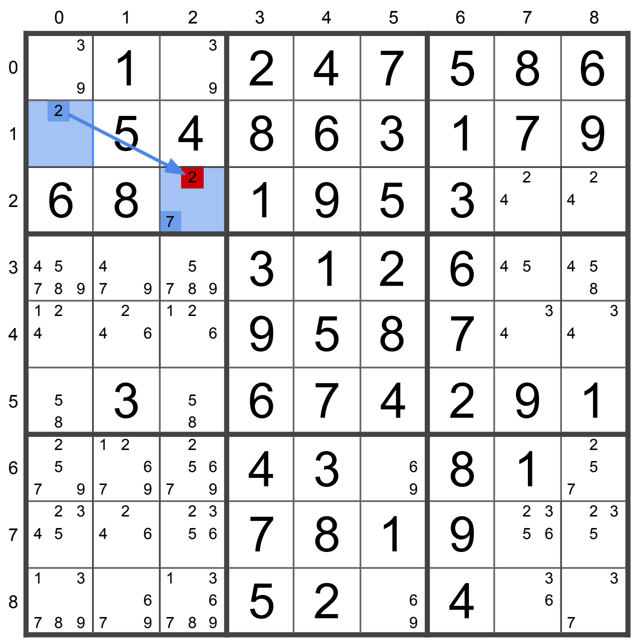


Figure 2.8: Example of two singletons in same iteration.

2.2.3 Complexity

Since the grid has n^2 cells, we first have $\mathcal{O}(n^2)$ as we are iterating through the whole grid. However, every time we find a singleton we update.

Once we have found a cell that is a singleton we assign to it the only value in its available set and change the available sets of the neighbors. The complexity for changing the available sets of the neighbors of the cell cell is $\mathcal{O}(3n)$ because it goes through a whole row, column and box. Then, the whole stage complexity is $\mathcal{O}(n^2)\mathcal{O}(3n) = \mathcal{O}(n^2 \cdot 3n) = \mathcal{O}(n^3)$.

2.3 Stage Two, Hermit

The objective of this stage is to assign a value to a cell when it is the only one in a bracket that has a specific candidate. The technique is directly related with the definition of the solution, Definition 1.3.11, page 15. For a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and a solution γ, for every bracket $\mathbb{S}$, we have that $|\Gamma(\mathbb{S})| = n$. Thus, all the brackets should contain every number in Ω. That is, for every bracket $\mathbb{S}$ and value $\omega \in \Omega$, there should exist a $cell \in \mathbb{S}$ such that $\gamma(cell) = \omega$. This statement motivates the following definition.

Definition 2.3.1. For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, a bracket $\mathbb{S}$, and a candidate $\omega \in \Omega$, we say that a $(\mathbb{S}, cell^*, \omega)$ is a ***hermit***, if $cell^*$ is the only cell in $\mathbb{S}$ such that $\omega \in \mathbb{AS}_k cell^*$.

A hermit is only useful when $2 \leq |\mathbb{AS}(cell^*)|$, otherwise it is a simple singleton. When a hermit $(\mathbb{S}, cell^*, \omega)$ is found, all the remaining values should be deleted from $\mathbb{AS}(cell^*)$, leaving $\mathbb{AS}(cell^*) = \{\omega\}$. Then, since ω will be the value of $cell^*$, ω can be removed from all the neighbors' available sets, except the ones in $\mathbb{S}$. The neighbors in $\mathbb{S}$ can be ignored to avoid redundancy, since they are the ones that caused the hermit, so they cannot have ω in their available set. We call this process ***hermit elimination***.

Example 2.3.2. For the setting in Figure 2.9 we have that $(\mathbb{B}_0, cell_{01}, 6)$ is a hermit because $cell_{01}$ is the only cell in $\mathbb{B}_0$ that has a 6 in its available set. Therefore, 6 will become the value of $cell_{01}$, and 6 can be eliminated from all the neighbors of $cell_{01}$ that are not in the box $\mathbb{B}_0$.

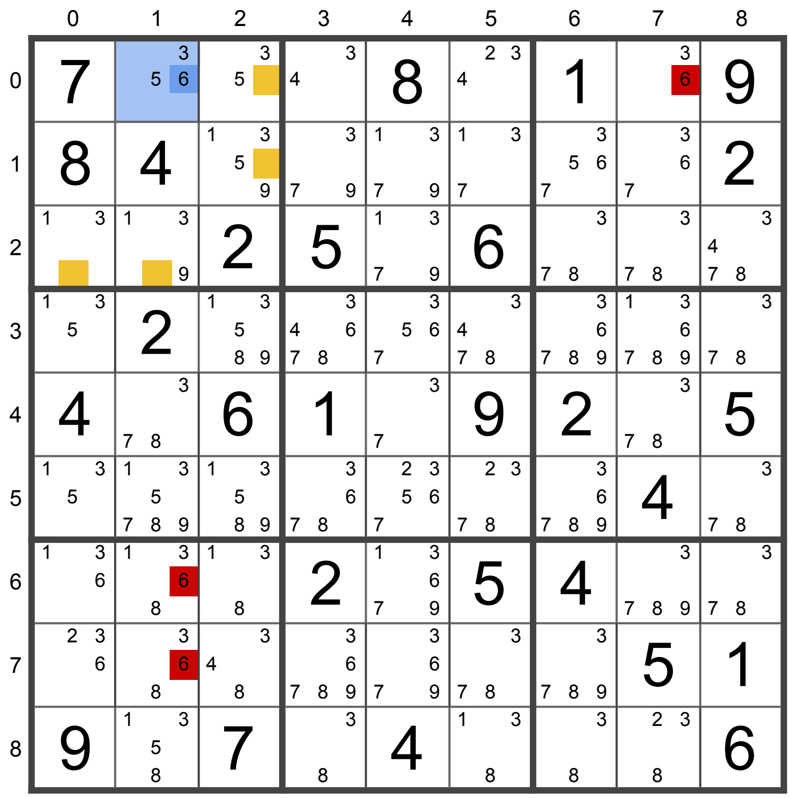


Figure 2.9: Hermit example $cell_{01}$.

A cell can be a hermit with two different brackets and the same value and that is not a problem. Let us assume $cell^*$ is a hermit on $(\omega, \mathbb{S}_1)$ and on $(\omega, \mathbb{S}_2)$. Then the only consequence will be that less cells will have ω removed from their available sets since they did not have it in the first place. It is impossible for a *cell* to be a hermit with two different values ω_1, ω_2. This property will be proven in the formal theory subsection.

Example 2.3.3. Figure 2.10 shows that $cell_{34}$ is a hermit on $(5, \mathbb{R}_3)$ and a hermit on $(5, \mathbb{B}_4)$. Therefore, the only *cells* that can be affected are in $\mathbb{N}(cell_{34}) \setminus (\mathbb{R}_3 \cup \mathbb{B}_4) = \{cell_{04}, cell_{24}, cell_{74}, cell_{85}\} \subseteq \mathbb{C}_4$. In this case, the only affected *cells* are $cell_{74}, cell_{84}$.

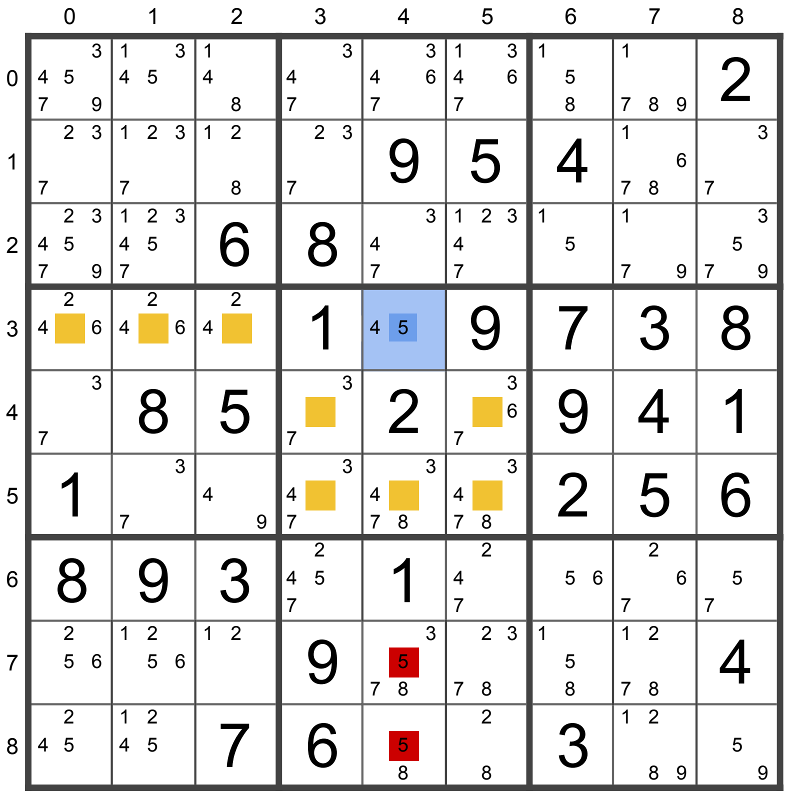


Figure 2.10: Hermit example $cell_{34}$.

The stage two of the algorithm consists in recognizing all the hermits in the grid. We iterate throughout every bracket, and for each candidate ω we count in how many *cells*' available sets it is contained. If we find a candidate ω that is contained only by $cell^* \in \mathbb{S}$ on the whole bracket $\mathbb{S}$, then $(\mathbb{S}, cell^*, \omega)$ is a hermit and we do a hermit elimination.

2.3.1 Sudoku theory

In this section we state here the formal theory of the candidate elimination process that is triggered when a hermit is found.

Now, just like in stage 1, we define a function f_2 that will receive a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a hermit $(\mathbb{S}, cell^*, \omega)$, and will return another setting.

Accordingly, we define,

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) &= f_2((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}, cell^*, \omega)) \\ &= (f_2^1(\mathbb{F}_k, cell^*), f_2^2(\mathbb{F}_k, \gamma_k, cell^*, \omega), f_2^3(\mathrm{AS}_k, cell^*, (\omega, \mathbb{S}))\,,\end{aligned}$$

where

$$\mathbb{F}_{k+1} = f_2^1(\mathbb{F}_k, cell^*) = \mathbb{F}_k \cup \{cell^*\}\ ,$$

and for every $cell \in \mathbb{G}$:

$$\gamma_{k+1}(cell) = f_2^2(\mathbb{F}_k, \gamma_k, cell^*, \omega)\big|_{cell} = \begin{cases} \gamma_k(cell) & \text{if } cell \in \mathbb{F}_k \\ \omega & \text{if } cell = cell^* \end{cases}$$

$$\begin{aligned}\mathrm{AS}_{k+1}(cell) &= f_2^3(\mathrm{AS}_k, (\mathbb{S}, cell^*, \omega))\big|_{cell} \\ &= \begin{cases} \{\omega\} & \text{if } cell = cell^* \\ \mathrm{AS}_k(cell) & \text{if } cell \in \mathbb{G} \setminus (\mathbb{N}(cell^*) \cup \{cell^*\}) \\ \mathrm{AS}_k(cell) & \text{if } cell \in \mathbb{S} \setminus \{cell^*\} \\ \mathrm{AS}_k(cell) \setminus \{\omega\} & \text{if } cell \in \mathbb{N}(cell) \setminus \mathbb{S} \end{cases}\end{aligned}$$

Now we prove that $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$ is indeed a setting, which means, no solutions are lost during the hermit elimination.

Theorem 2.3.4. *Given a setting* $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, and a hermit* $(\mathbb{S}, cell^*, \omega)$*,* $f_2((\mathbb{F}_k, \gamma_k, \mathbb{AS}_k), (\mathbb{S}, cell^*, \omega)) = (\mathbb{F}_{k+1}, \gamma_{k+1}, \mathbb{AS}_{k+1})$ *is a setting.*

Proof. It is clear that $\mathbb{F}_k \subseteq \mathbb{F}_{k+1}$ and that $\gamma_k \preceq \gamma_{k+1}$. Then it remains to prove that $\mathbb{AS}_{k+1}$ is a valid available set function. We have to prove that for every $cell \in \mathbb{G}$, and for every solution γ of $(n, \mathbb{F}_0, \gamma_0)$, we have that $\gamma(cell) \in \mathbb{AS}_{k+1}(cell)$.

Let us suppose it is not, then, there must exist a solution γ and $cell' \in \mathbb{G}$, such that $\gamma(cell') \notin \mathbb{AS}_{k+1}(cell')$. Since $\mathbb{AS}_k$ is a valid available set function, we have that for every $cell \in \mathbb{G}$, $\gamma(cell) \in \mathbb{AS}_k(cell)$, so in particular, we have the following property for $cell'$

$$\gamma(cell') \in \mathbb{AS}_k(cell') \setminus \mathbb{AS}_{k+1}(cell') \tag{2.1}$$

Case 1, $cell' = cell^*$:

We have that $\mathbb{AS}_{k+1}(cell^*) = \{\omega\}$. Then, by (2.1) $\gamma(cell') \neq \omega$, which means $\gamma(cell^*) \neq \omega$. Then, as $cell^* \in \mathbb{S}$ and $\gamma(cell^*) \neq \omega$, there exists another $cell_\omega \in \mathbb{S}$ such that $\gamma(cell_\omega) = \omega$. But, since $(\mathbb{S}, cell^*, \omega)$ is a hermit, we have that $\omega \notin \mathbb{AS}_k(cell)$ for each $cell \in \mathbb{S}$ except $cell^*$. Which means that $\omega = \gamma(cell_\omega) \notin \mathbb{AS}_k(cell_\omega)$. This contradicts the hypothesis that $\mathbb{AS}_k$ is a valid available set function.

Case 2, $cell' \in \mathbb{G} \setminus (\mathbb{N}(cell^*)) \cup \{cell^*\})$ or $cell' \in \mathbb{S} \setminus \{cell^*\}$:

By definition of $\mathbb{AS}_{k+1}$, $\mathbb{AS}_{k+1}(cell') = \mathbb{AS}_k(cell')$, then by (2.1) we have that $\gamma(cell') \in \emptyset$, which is not possible.

Case 3, $cell' \in \mathbb{N}(cell^*) \setminus \mathbb{S}$:

By (2.1) we have that $\gamma(cell') \in \mathbb{AS}_k \setminus (\mathbb{AS}_k \setminus \omega))$, which means $\gamma(cell') = \omega$. Now, since $(\mathbb{S}, cell^*, \omega)$ is a hermit, $\gamma(cell^*) = \omega$.

Then, $\gamma(cell^*) = \gamma(cell')$, which is a contradiction because $cell'$ and $cell^*$ are neighbors.

By *reducto ad absurdum* AS_{k+1} is a valid aset function and $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$ is a setting. □

It is now clear that a *cell* cannot be a hermit with two different values ω_1, ω_2. This is impossible because of Case 1 in the proof, since it would imply that every solution satisfies $\gamma(cell) = \omega_1$ and $\gamma(cell) = \omega_2$ which is a contradiction because γ is a function.

2.3.2 Implementation

Since we want check for hermits in every bracket regardless of its type, we take advantage of the `all` attribute in `BracketContainer`.

Then, given a bracket index and the cell's inside that bracket, we need a way to identify the cell's position in the grid. These are just arithmetic functions, whose implementation is unrelated to the algorithm. Therefore, we are going to assume that we have them, `get_row` and `get_col` which give the row and column of a cell based on the index of its bracket, and its index inside that bracket.

Now, we add the method that executes stage two on the `SudokuGrid` class.

For every bracket and every cell we check every candidate in its available set. For each of those candidates we add it to a dictionary called `candidate_map` that maps the candidate to the index of the cell where it was found. Then, if a candidate is found more than once we change its index to -1, since it cannot be a hermit, we use -1 because it is not a valid index for a cell.

By definition, after checking all the cells of a bracket, all the candidates that appear in the map with a positive index are hermits, since they were found only once in a bracket. Subsequently, we do the hermit elimination process, using the `update_cell` function implemented in stage 1.

```
1  class SudokuGrid:
2      def stage_two(self):
3          for bracket_index in range(len(self.brackets.all)):
4              bracket = self.brackets.all[bracket_index]
5              candidate_map = {}
6              for cell_index in SudokuGrid.I:
7                  cell = bracket[cell_index]
8                  if not cell.has_value():
9                      for candidate in cell.av_set:
10                         if candidate in candidate_map:
11                             candidate_map[candidate] = -1
12                         else:
13                             candidate_map[candidate] = cell_index
14             for candidate, cell_index in candidate_map.items():
15                 if 0 <= cell_index:
16                     row = self.brackets.get_row(bracket_index, cell_index)
17                     col = self.brackets.get_col(bracket_index, cell_index)
18                     self.update_cell(row, col, candidate)
```

Stage 2

The inner loop in line 9, in the worst case goes through every candidate ($\mathcal{O}(|\Omega|) = \mathcal{O}(n)$) of every cell ($\mathcal{O}(n)$) of every bracket ($\mathcal{O}(3n)$). This results in a time complexity of $\mathcal{O}(3n \cdot n \cdot n) = \mathcal{O}(3n^3) = \mathcal{O}(n^3)$.
Moreover, line 17 is inside two loops, for each bracket ($\mathcal{O}(3n)$), and for each candidate (($\mathcal{O}(|\Omega|) = \mathcal{O}(n)$). Then, the line itself calls the `update_cell()` which has a complexity of ($\mathcal{O}(3n)$) because it goes through every neighbor of the cell. For this process, we have a time complexity of $\mathcal{O}(3n \cdot n \cdot 3n) = \mathcal{O}(9n^3) = \mathcal{O}(n^3)$.
Finally, the total time complexity of the stage is $\mathcal{O}(n^3) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$.

Stage 1 and stage 2 are the only stages that assign values to cells in the grid. The next stages will only reduce the available set of some cells but will never assign a value to a cell. However as available sets are reduced by the next stages, there is a chance to produce new singletons or hermits which will cause a return to stages one and two.

Chapter 3

Pruning sections of Solveku

In this chapter we will go through three more stages. These stages use more complicated techniques which implies that the theory and the implementation will be less straightforward. Because of this, the code section for stage four and five will need some additional theory from the sudoku theory section.

The main difference between the three techniques in this chapter and the previous ones is that these are just pruning stages. By pruning we mean that the stages' goal is just to reduce the available sets of some cells, they are not going to fix values in cells, like stages 1 and 2 did.

3.1 Stage Three, Bracket Intersection

Stage 3 consists in the pruning of available sets of cells in a specific bracket. This stage differs from the two previous stages because is focused on brackets while Stage 1 and 2 are focused on cells or the relationship between an specific cell and a bracket. The pruning happens when two brackets (root $\mathbb{S}_r$ and target $\mathbb{S}_t$) and a candidate ω hold the following condition. All the cells in the root, that have ω in their available set, belong to the intersection between the root and the target, that is $\mathbb{S}_r \cap \mathbb{S}_t$. This intersection gives the name to the technique ***bracket intersection***. We show state an example to illustrate this scenario.

Example 3.1.1. In the setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of Figure 3.1 we can see that all the cells in the column $\mathbb{C}_1$ that have the candidate 6 in their available set are inside the box $\mathbb{B}_3$ as well. Therefore, $\mathbb{C}_1$ is the root and $\mathbb{B}_3$ is the target. So we can prune 6 from the available set of all the $cells \in \mathbb{B}_3 \setminus \mathbb{C}_1$, in this case, the $cells$ that are pruned are $cell_{32}, cell_{42}, cell_{52}$. We are able to prune these cells because if one of those cells took the value 6, then there would be no cell in $\mathbb{C}_1$ with 6 in its available set.

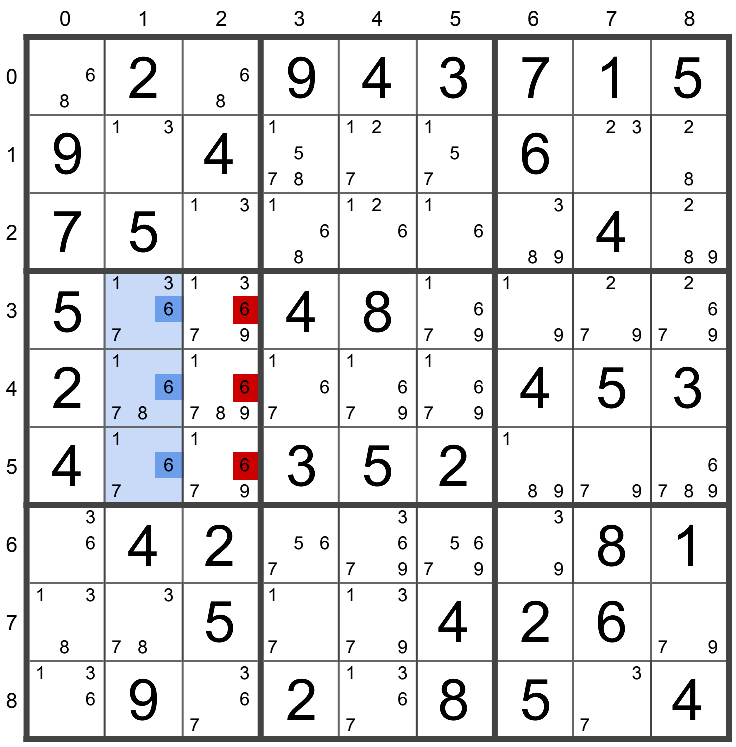


Figure 3.1: Bracket intersection example for $\mathbb{C}_1$ and $\mathbb{B}_3$

Example 3.1.1 shows a column intersection, which is one of the four different possible types of bracket intersections. First, we define each type of bracket intersection separately, so that the concept can be extensively explained.

Definition 3.1.2. For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$.
A ***box-r intersection*** $(\mathbb{B}_{i_1}, \mathbb{R}_{i_2}, \omega)$ happens when all the cells in the box $\mathbb{B}_{i_1}$ that contain ω in their available set are in the row $\mathbb{R}_{i_2}$, for $i_1, i_2 \in I$.

A ***box-c intersection*** $(\mathbb{B}_{i_1}, \mathbb{C}_{i_2}, \omega)$ happens when all the cells in $\mathbb{B}_{i_1}$ that contain ω in their available set are in the column $\mathbb{C}_{i_2}$, for $i_1, i_2 \in I$.

We say that the box $\mathbb{B}_{i_1}$ is the ***root*** and the row $\mathbb{R}_{i_2}$ or the column $\mathbb{C}_{i2}$

are the ***target*** respectively, and the ***intersection value*** is ω.

Example 3.1.3. In Figure 3.2 we show a box-r intersection, where $\mathbb{B}_8$ is the root, and $\mathbb{R}_6$ is the target with value 4.

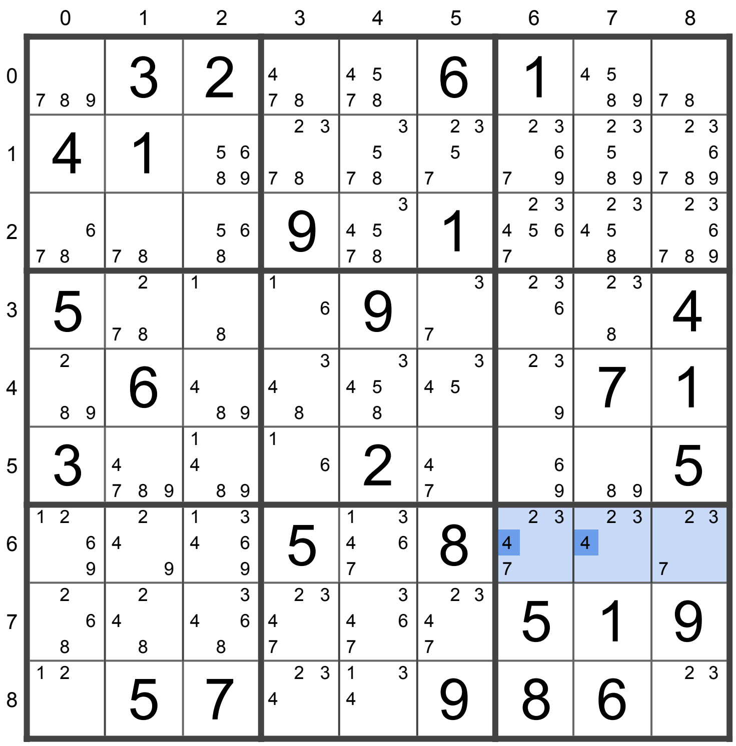


Figure 3.2: Row prune.

Now we define the two remaining types of bracket intersection.

Definition 3.1.4. Given a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, a ***row intersection*** on $(\mathbb{R}_{i_1}, \mathbb{B}_{i_2}, \omega)$, when all the cells in the row $\mathbb{R}_{i_1}$ that contain ω in their available set are in the box $\mathbb{B}_{i_2}$, for $i_1, i_2 \in I$.

A ***column intersection*** on $(\mathbb{C}_{i_1}, \mathbb{B}_{i_2}, \omega)$, when all the cells in the column $\mathbb{C}_{i_1}$ that contain ω in their available set are in the box $\mathbb{B}_{i_2}$, for $i_1, i_2 \in I$.

Example 3.1.5. In Figure 3.3, shows the setting presented in Example 3.1.1. We have a column intersection, where $\mathbb{C}_1$ is the root, and $\mathbb{B}_3$ is a target with value 6.

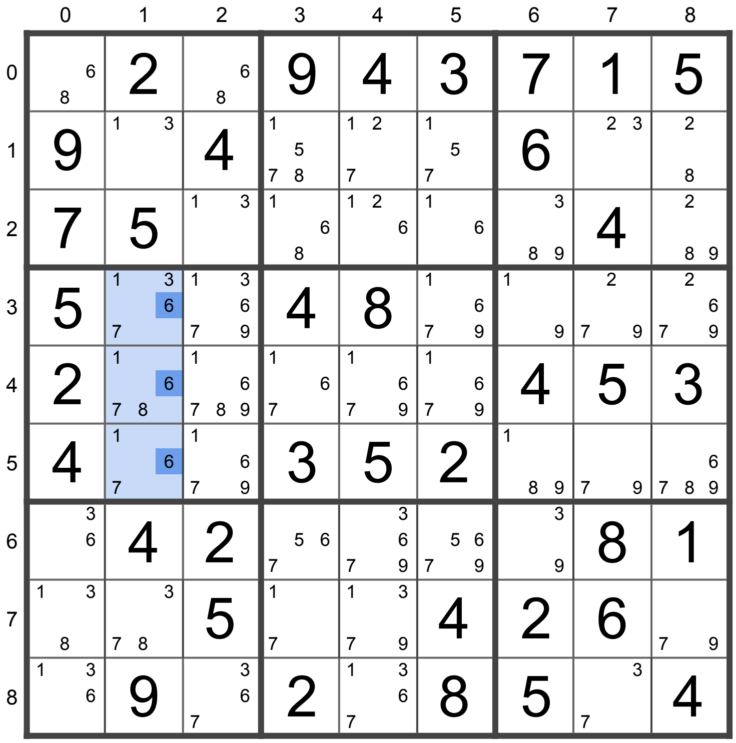


Figure 3.3: Box prune.

We stated that there are four types of bracket intersections, but we did not explain why there are not more. There exist three different types of brackets, which means there are $3^2 = 9$ different types of combinations of source and target. First of all, two different brackets of the same type can never intersect, so we are left with 6 possible combinations. Now, do not consider the intersection between row and column because the intersection is only one cell, which would be a hermit if we ask that this cell is the only one in the source whose available set contains ω. Therefore, we ignore this case because it is already covered by stage 2.

The four types of intersection presented earlier can be grouped into one. We defined them separately for didactic purposes. However, we are now going to group them together into one definition, so that we can then generalize the bracket pruning. This generalization will also include the intersections between rows and columns, however, in the implementation of the algorithm we look exclusively for the four types of intersection defined earlier to avoid redundant computations. This general definition will be very useful for the sudoku theory since it will allow to generalize the theorems as well.

Definition 3.1.6. For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we say that $(\mathbb{S}_r, \mathbb{S}_t, \omega)$ is a ***bracket intersection*** if for all $cell \in \mathbb{S}_r$ we have that if $\omega \in \mathbb{AS}_k(cell)$ then $cell \in \mathbb{S}_t$. We call $\mathbb{S}_r$ root, $\mathbb{S}_t$ target, ω intersection value.

Additionally, the elimination of ω from $\mathbb{AS}(cell)$, for $cell \in \mathbb{S}_t$ such that $cell \notin \mathbb{S}_r$ will be called ***bracket difference prune***.

Example 3.1.7. For the setting in Figure 3.4, we show a bracket difference prune for the bracket intersection $(\mathbb{S}_r = \mathbb{B}_8, \mathbb{S}_t = \mathbb{R}_6, 4)$. The target is $\mathbb{R}_6$ with 4 as an intersection value, then 4 will be removed from the available sets of $cell_{61}, cell_{62}, cell_{64}$.

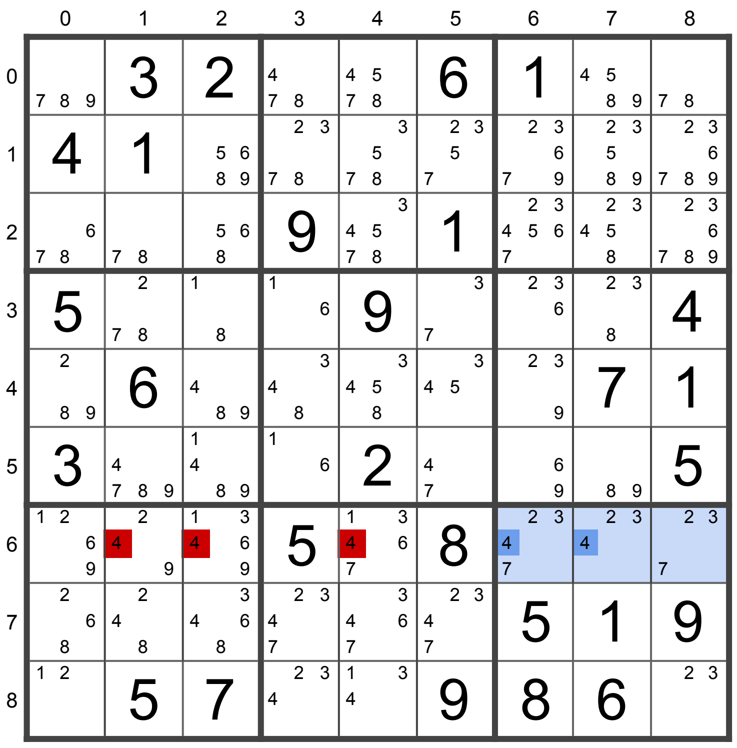


Figure 3.4: Bracket prune for $\mathbb{R}_6$.

We now formalize this pruning process with the sudoku theory.

3.1.1 Sudoku Theory

Given a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and a bracket intersection $(\mathbb{S}_r, \mathbb{S}_t, \omega)$, we are going to show that applying a bracket difference prune does not loose solutions.

Let f_3, be the function that receives a setting and an intersection and produces another setting with a bracket prune. Since the function f_3 does not assign values to cells, it preserves $(\mathbb{F}_k, \gamma_k)$. We define,

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) &= f_3((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}_r, \mathbb{S}_t, \omega)) \\ &= (\mathbb{F}_k, \gamma_k, f_3^3(\mathrm{AS}_k, (\mathbb{S}_r, \mathbb{S}_t, \omega)))\,,\end{aligned}$$

where, for every *cell* $\in \mathbb{G}$:

$$\begin{aligned}\mathrm{AS}_{k+1}(cell) &= f_3^3((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}_r, \mathbb{S}_t, \omega))\big|_{cell} \\ &= \begin{cases} \mathrm{AS}_k & \text{if } cell \in \mathbb{G} \setminus \mathbb{S}_t \\ \mathrm{AS}_k & \text{if } cell \in \mathbb{S}_t \cap \mathbb{S}_r \\ \mathrm{AS}_k \setminus \{\omega\} & \text{if } cell \in \mathbb{S}_t \setminus \mathbb{S}_r \end{cases}\end{aligned}$$

Now, the following theorem ensures no solutions are lost when doing a bracket intersection prune, this is achieved by proving the result of applying f_3 to a setting is still a setting.

Theorem 3.1.8. *Given a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and an intersection $(\mathbb{S}_r, \mathbb{S}_t, \omega)$, the tuple*

$$f_3((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}_r, \mathbb{S}_t, \omega)) = (\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$$

is a setting.

Proof. By definition of f_3 we have that $\mathbb{F}_{k+1} = \mathbb{F}_k$, so $\mathbb{F}_k \subseteq \mathbb{F}_{k+1}$ and $\gamma_{k+1} = \gamma_k$, then $\gamma_k \preceq \gamma_{k+1}$. Therefore, the only thing left to prove is that AS_{k+1} is a valid available set function.

Let us suppose it is not, then there must be a solution γ, and a $cell' \in \mathbb{G}$ such that $\gamma(cell') \notin \mathrm{AS}_{k+1}(cell')$. Hence, we get the following property:

$$\gamma(cell') \in \mathrm{AS}_k(cell') \setminus \mathrm{AS}_{k+1}(cell')\,. \tag{3.1}$$

Case 1, $cell' \in \mathbb{G} \setminus \mathbb{S}_t$ or $cell' \in \mathbb{S}_t \cap \mathbb{S}_r$:

Then by definition of f_3^3 we have that $\mathrm{AS}_{k+1}(cell') = \mathrm{AS}_k(cell')$.

Therefore, by (3.1), we have that $\gamma(cell') \in \emptyset$ which is a contradiction.

Case 2, $cell' \in \mathbb{S}_t \setminus \mathbb{S}_r$:

By (3.1) and by definition of f_3^3, we have $\gamma(cell') \in \{\omega\}$, subsequently $\gamma(cell') = \omega$.

Since $\omega \in \Omega$ and γ is a solution by Definition 1.3.11, page 15 we have that in every bracket and for each value in Ω there should be a unique cell in the bracket with that value. Then, let $cell_r$ be the cell in the root $\mathbb{S}_r$ such that $\gamma(cell_r) = \omega$. This implies that $\omega \in \mathbb{AS}_k(cell_r)$ because $\mathbb{AS}_k$ is valid.

Given that $(\mathbb{S}_r, \mathbb{S}_t, \omega)$ is an intersection, by Definition 3.1.6 we have $cell_r$ must also belong to the target, that is, $cell_r \in \mathbb{S}_t$. But $\gamma(cell_r) = \omega$ and $\gamma(cell') = \omega$. Then, we have two different cells with the same value in the same bracket $\mathbb{S}_t$, which is a contradiction. We are certain they are different cells because $cell_r \in \mathbb{S}_t \cap \mathbb{S}_r$ and $cell' \in \mathbb{S}_t \setminus \mathbb{S}_r$.

Finally by *reducto ad absurdum* we have that $\mathbb{AS}_{k+1}$ is a valid available set function and then $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathbb{AS}_{k+1})$ is a setting. □

3.1.2 Implementation

We first need a method that finds intersections in the grid. We aim to construct a method that works for all four types of intersections, since we do not want to repeat code. For that, we need another function that, given the bracket type of the root and the bracket type of the target, is able to separate cells by the potential target bracket they belong to. This function is going to separate potential targets by indexes, these indexes will go from 1 to $\sqrt{n}$ since for every intersection (four types presented earlier) each source has $\sqrt{n}$ potential target brackets for a specific target type. Because each row and column intersect with $\sqrt{n}$ different boxes, and likewise, each box intersects with $\sqrt{n}$ different rows and $\sqrt{n}$ different columns. The logic of this function is independent from Solveku, so we are going to assume we have it, and the function is called `split_function`.

In other words, the split function will receive the type of source (row, column or box), the type of the target, and an internal index. The internal index represent the index of a cell inside the source bracket, that is, the index of the array representation of a bracket. Finally `split_function` returns the target index, which is a natural number in $\{0, \ldots, \sqrt{n} - 1\}$ Will only have two types of split functions, one that splits the internal indexes by module ψ_m (base $\sqrt{n}$), and one that splits by division ψ_d, which uses arithmetical division and takes the floor of the result

Example 3.1.9. We now illustrate both types of split functions with two different examples. For any puzzle $(n, \mathbb{F}_0, \gamma_0)$. First, let ψ_d be the split function, that separates by division where $\mathbb{C}_3$ is the root, and the target type is box. Since $\mathbb{C}_3$ is the root, its array representation is

$$[cell_{03}, cell_{13}, cell_{23}, cell_{33}, cell_{43}, cell_{53}, cell_{63}, cell_{73}, cell_{83}]$$

The order of this array representation is crucial, because it gives us the internal index of these cells. For an internal index $i \in I$, the division split function looks as follows: For $cell \in \mathbb{C}_3$

$$\psi_d(i) = \left\lfloor \frac{i}{\sqrt{n}} \right\rfloor$$

Notice that if we separate the array representation of $\mathbb{C}_3$ by the result of applying ψ_d to their internal index, we end up with the following sub arrays:

$(\psi_d(i) = 0)$ $[cell_{03}, cell_{13}, cell_{23}] \subseteq \mathbb{B}_1$,

$(\psi_d(i) = 1)$ $[cell_{33}, cell_{43}, cell_{53}] \subseteq \mathbb{B}_4$,

$(\psi_d(i) = 2)$ $[cell_{63}, cell_{73}, cell_{83}] \subseteq \mathbb{B}_7$.

Notice that the cells are separated by the box they belong to, which is exactly the result we were expecting.

Now, let us also give an example of the split by module function. $\mathbb{B}_8$ is the root, and the target type is column. Recall from the definition of `BracketContainer`, page 37, that the array representation of $\mathbb{B}_8$ looks as follows:

$$[cell_{66}, cell_{67}, cell_{68}, cell_{76}, cell_{77}, cell_{78}, cell_{86}, cell_{87}, cell_{88}]$$

Now, we have the following definition of the module split function,

$$\psi_m(i) = i \mod \sqrt{n}$$

Notice that if we separate the array representation of $\mathbb{C}_3$ by the result of applying ψ_d to their internal index, we end up with the following sub arrays:

$(\psi_m(i) = 0)$ $[cell_{66}, cell_{76}, cell_{86}] \subseteq \mathbb{C}_6$,

$(\psi_m(i) = 1)$ $[cell_{67}, cell_{77}, cell_{78}] \subseteq \mathbb{C}_7$,

$(\psi_m(i) = 2)$ $[cell_{86}, cell_{87}, cell_{88}] \subseteq \mathbb{C}_8$.

Notice that the cells are separated by the column they belong to, which is exactly the result we were expecting.

We also need a method, whose implementation is unrelated to our algorithm, that given the source and target index, and the source and target bracket type is able to find the target bracket. That means that this method is able to locate in memory the array that actually holds the cells of the target brackets. We want to find the array of cells that represents the bracket so that we can prune it. Since this function is used exclusively for performance purposes, and is completely independent from Solveku's logic. Understanding this function is not necessary, but if the reader is interested please check this work's repository, this analysis we also assume it is implemented and it is called `find_target`.

The function `prune_bracket` method is the implementation of the bracket difference prune process, defined in Definition 3.1.6. The method goes through all the cells in the target bracket, and deletes the `intersection_value` for the available sets of cells that do not belong to the intersection $\mathbb{S}_r \cap \mathbb{S}_t$that is not in the intersection set.

```python
class SudokuGrid:
    """..."""
    def prune_bracket(self, target_bracket, intersection, intersection_value):
        res = False
        for cell in target_bracket:
            if cell not in intersection:
                if cell.av_set_remove(intersection_value):
                    res = True
        return res
```

Prune bracket.

Finally, the function `prune_intersection` looks for intersections as defined in Definition 3.1.6 and performs the corresponding pruning. For every bracket in `root_brackets`, the algorithm assumes it is a root, and it looks for a candidate such that all cells with an available set containing the candidate are in the same intersection bracket. The dictionary `target_map` maps every candidate with the `target_index` of the first cell containing it. If a candidate was not in the dictionary it is then mapped to its target index and a singleton of the current cell. If two cells with a different `target_index` have the candidate, then that candidate cannot be an intersection value and that candidate is mapped to -1. We use -1 because it cannot be a valid index, since $-1 \notin I$. If the dictionary already had an entry for a candidate with the same `target_index` then the current cell will just be added to the result of `intersection_map`.

After checking all the cells, every candidate that is mapped to a valid index is a bracket intersection value. Then, we look for the object of the target bracket using `find_target_bracket` and then we prune it using `prune_bracket`.

```python
class SudokuGrid:
    """..."""
    @staticmethod
    def root_div(index):
        return int(index/math.sqrt(SudokuGrid.n))

    @staticmethod
    def root_mod(index):
        return int(index%math.sqrt(SudokuGrid.n))

    def prune_intersection(self, root_brackets, split_function, root_type,
     target_type):
        target_map = {}
        intersection_map = {}
        for root_index in self.I:
            target_map.clear()
            intersection_map.clear()
            root = root_brackets[root_index]
            for cell_index in self.I:
                cell = root[cell_index]
                target_index = split_function(cell_index)
                if not cell.has_value():
                    for candidate in cell.av_set:
                        if not candidate in target_map:
                            target_map[candidate] = target_index
                            intersection_map[candidate] = {cell}
                        else:
                            if target_map[candidate] == target_index:
                                intersection_map[candidate].add(cell)
                            else:
                                intersection_map[candidate] = None
                                target_map[candidate] = -1
            for candidate, target_index in target_map.items():
                if target_index in SudokuGrid.I:
                    target_bracket = self.get_target_bracket(root_index,
     target_index, root_type, target_type)
                    self.prune_bracket(target_bracket, intersection_map[
     candidate], candidate)
```

Find intersection.

The `stage_three()` method calls `prune_intersection` with every valid combination of root and target.

```python
class SudokuGrid:
    """..."""
    def stage_three(self):
        row_intersections = self.find_intersection(self.brackets.box, self.
     root_div, box_type, row_type)
        col_intersections = self.find_intersection(self.brackets.box, self.
     root_mod, box_type, col_type)
```

```
        box_r_intersections = self.find_intersection(self.brackets.row, self.
    root_div, row_type, box_type)
        box_c_intersections = self.find_intersection(self.brackets.col, self.
    root_div, col_type, box_type)
```

Stage 3

The time complexity of the method `prune_intersection` can be split in two parts. First, in line 22 of the inside the method `prune_intersection` is inside three four loops, the one in line 14 and line 18 ($\mathcal{O}(|I|) = \mathcal{O}(n)$), and the four loop in line 22 ($\mathcal{O}(|\Omega|) = \mathcal{O}(n)$). These result in a $\mathcal{O}(n) \cdot \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^3)$ time complexity.

Besides, in line 32, we are inside of the for loop of line 14 $\mathcal{O}(n)$, and line 32 itself goes through every candidate in the `target_map`($\mathcal{O}(n)$), it prunes the valid ones. Then, in line 35, `prune_bracket` method has a $\mathcal{O}(n)$ time complexity, since it goes throughout every cell in the target bracket. Which results in a complexity of $\mathcal{O}(n) \cdot \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^3)$.

Then, for the complete function `prune_intersection` we have a time complexity of $\mathcal{O}(n^3) + \mathcal{O}(n^3) = 2\mathcal{O}(n^3) = \mathcal{O}(n^3)$. Finally, for the time complexity of the stage 3, we have that the `stage_three` method calls `prune_intersection` four times. So we have a total time complexity of $4 \cdot (\mathcal{O}(n^3)) = \mathcal{O}(n^3)$.

3.2 Stage Four, Bracket subset

In this stage, we look for a bracket with specific conditions that enables us to prune the available sets of cells inside this bracket. In contrast to stage 3, we now look for properties within the cells of the bracket. That is, at this stage, all the information required to prune a bracket will be in the bracket itself, rather than in a relationship between brackets. More specifically, we look for a subset of a bracket with certain conditions.

Now, before we define the first type of subset we are looking for, let us show, with an example, the simplest version of these subsets and their characteristics. This example should give the reader a general idea before the formal definition.

Example 3.2.1. For the setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ in Figure 3.5, $cell_{10}$ and $cell_{20}$ are in the same column $\mathbb{C}_0$, and have the same available set $\{3, 7\}$. Therefore, since $\mathbb{AS}_k$ is a valid available set function, for any solution γ, we have that $\{\gamma(cell_{10}), \gamma(cell_{20})\} = \{3, 7\}$. This implies that there cannot be another $cell \in \mathbb{C}_0$ such that $cell \neq cell_{10}, cell \neq cell_{20}$ and $\gamma(cell) \in \{3, 7\}$. Otherwise, a value would be repeated within the bracket $\mathbb{C}_0$. Therefore, we can remove values 3 and 7 from the available sets of the remaining cells in $\mathbb{C}_0$, in this case $cell_{30}$, $cell_{40}$ and $cell_{50}$ are the only cells to be pruned. We call this scenario ***naked pair***, in agreement with [Ber07] and [Stu08].

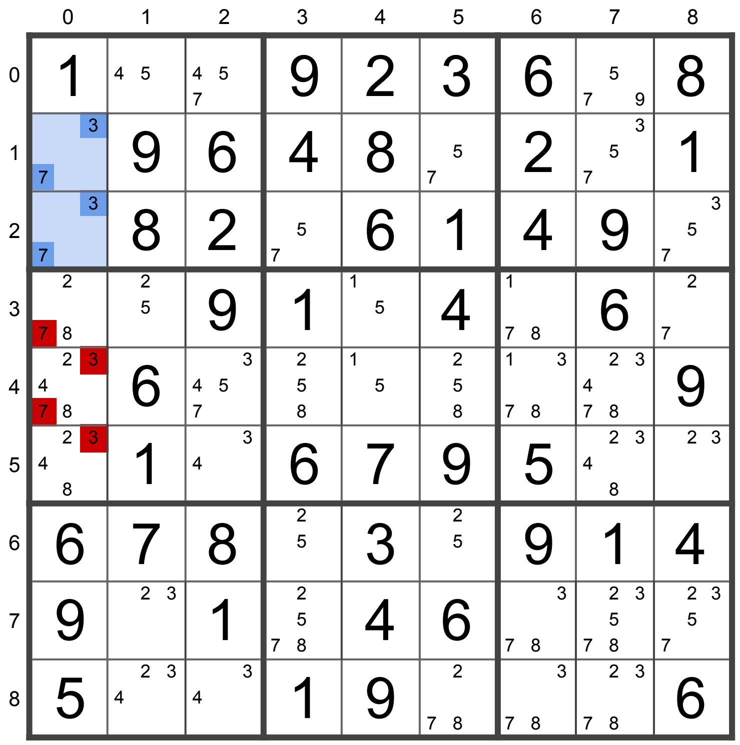


Figure 3.5: Example of a naked pair.

We are going to generalize the idea of the Example 3.2.1. We will see that a naked pair is a naked subset of size 2 of a bracket. In general, this type of subsets can have size m where $m \in \{2, \dots, n-1\}$.

Definition 3.2.2. For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a bracket $\mathbb{S}$, we have that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a ***naked subset*** if the following conditions are met:

- $\mathbb{V} \subseteq \mathbb{S}$ subset of cells and $\mathcal{V} \subseteq \Omega$ subset of values.
- $|\mathbb{V}| = |\mathcal{V}|$, same number of *cells* and values.
- $\mathrm{AS}_k(cell) \subseteq \mathcal{V}$ for every $cell \in \mathbb{V}$.

Once a naked subset is found, then, for every value $\omega \in \mathcal{V}$, and for every $cell \in \mathbb{S} \setminus \mathbb{V}$, ω can be eliminated from the *cell*'s available set. We call this process ***naked subset prune***.

Example 3.2.3. For the setting in Figure 3.6, we will prove that $(\mathbb{B}_1, \{cell_{03}, cell_{04}, cell_{05}\}, \{4,7,9\})$ is a naked subset. This is done by checking that all the conditions of Definition 3.2.2 are met:

- $\{cell_{03}, cell_{04}, cell_{05}\} \subseteq \mathbb{B}_1$, and $\{4,7,9\} \subseteq \Omega = \{1,\ldots,9\}$.
- $|\{4,7,9\}| = 3 = |\{cell_{03}, cell_{04}, cell_{05}\}|$.
- $\text{AS}(cell_{03}) = \{7,9\} \subseteq \{4,7,9\}$, $\text{AS}(cell_{04}) = \{4,9\} \subseteq \{4,7,9\}$, $\text{AS}(cell_{05}) = \{4,7,9\} \subseteq \{4,7,9\}$.

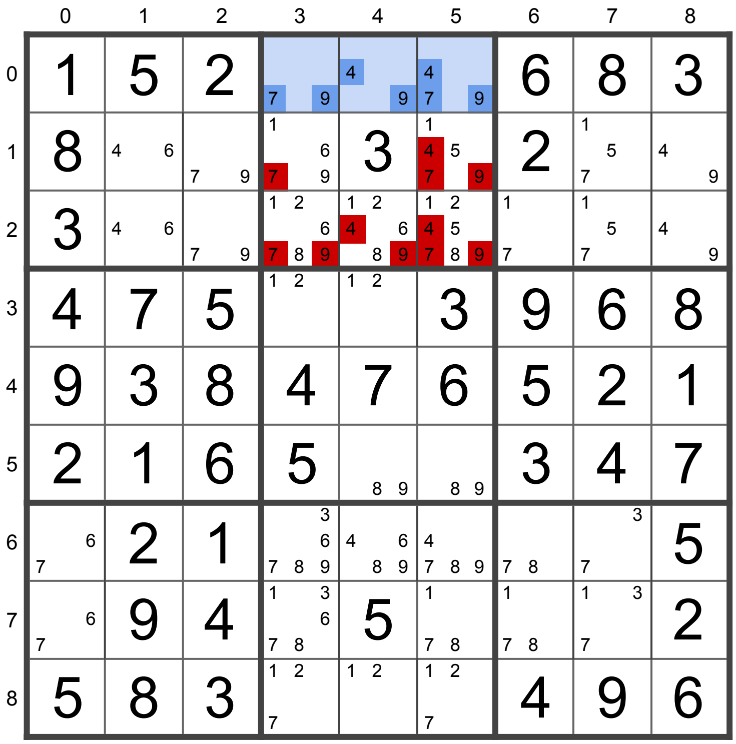


Figure 3.6: Example of a naked subset of size 3.

So, we have that $(\mathbb{B}_1, \{cell_{03}, cell_{04}, cell_{05}\}, \{4, 7, 9\})$ is a naked subset.

Now, we are going to show a different type of bracket subset. This new type of subset will also have conditions for pruning the bracket's available sets. Just like with naked subsets, we first illustrate with a basic example and after that we state the formal definition.

Example 3.2.4. For the setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ in Figure 3.7 $cell_{32}$ and $cell_{42}$ are the only cells in $\mathbb{B}_3$ that have candidates 2 and 4. Since they are the only cells that can take one of the values $\{2, 4\}$ and for every solution γ, there should be $cell, cell' \in \mathbb{B}_3$ such that $\gamma(cell) = 2$ and $\gamma(cell') = 4$. Then, we know that for every solution γ, $\gamma(cell_{32}) \in \{2, 4\}$ and $\gamma(cell_{42}) \in \{2, 4\}$. Therefore, we must have $\mathrm{AS}_k(cell_{32}) = \{2, 4\}$ and $\mathrm{AS}_k(cell_{42}) = \{2, 4\}$. That is, we can delete the other candidates in $cell_{32}$ and $cell_{42}$, since these cells cannot take a value different from 2 or 4. Because if they do, then no other cell on $\mathbb{B}_3$ will be able to take the value 2 or the value 4. We call this scenario ***hidden pair***, in agreement with [Ber07] and [Stu08].

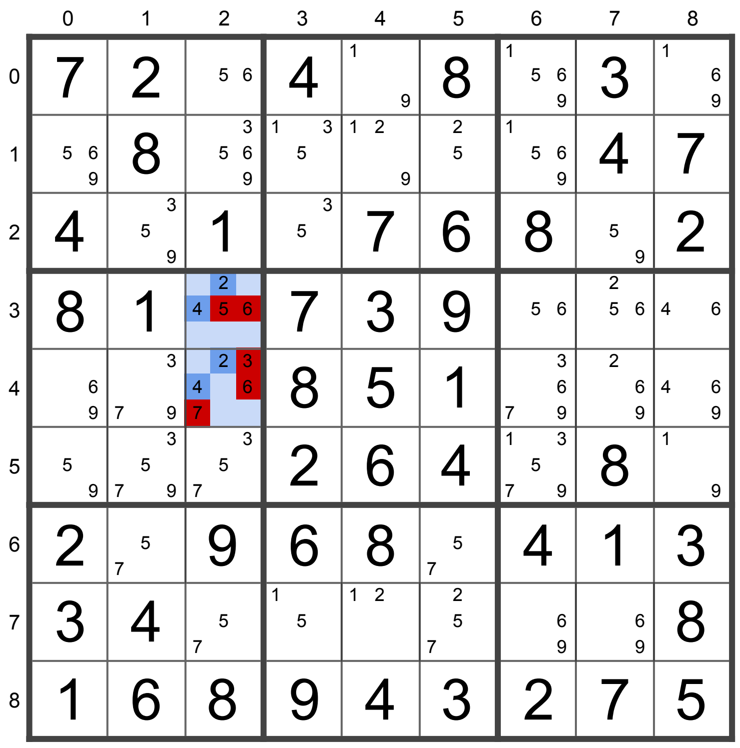


Figure 3.7: Example of a hidden pair.

Now, we generalize the idea of hidden pairs from size 2 to size m where $m \in \{2, \ldots, n-1\}$.

Definition 3.2.5. For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and a bracket $\mathbb{S}$, we have $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is a ***hidden subset*** if the following conditions are met:

- $\mathbb{U} \subseteq \mathbb{S}$ subset of cells and $\mathcal{U} \subseteq \Omega$ subset of values.
- $|\mathbb{U}| = |\mathcal{U}|$.
- $\mathrm{AS}_k(\mathit{cell}) \subseteq \Omega \setminus \mathcal{U}$ for every $\mathit{cell} \in \mathbb{S} \setminus \mathbb{U}$.

After finding a hidden subset, for every $\omega \in \Omega \setminus \mathcal{U}$, and for every $\mathit{cell} \in \mathbb{U}$, we see that ω can be eliminated from $\mathrm{AS}(cell)$, and we call this process ***hidden subset prune***.

Now we show an example of a bigger hidden subset.

Example 3.2.6. For the setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ in Figure 3.8 we prove that $(\mathbb{B}_4, \{cell_{33}, cell_{35}, cell_{53}, cell_{55}\}, \{1, 2, 5, 7\})$ is a hidden subset of size 4. Let us check that every condition of Definition 3.2.5 is met:

- $\mathbb{U} = \{cell_{33}, cell_{35}, cell_{53}, cell_{55}\} \subseteq \mathbb{B}_4$, and $\mathcal{U} = \{1, 2, 5, 7\} \subseteq \Omega$.
- $|\{1, 2, 5, 7\}| = 4 = |\{cell_{33}, cell_{35}, cell_{53}, cell_{55}\}|$.
- Since $\mathcal{U} = \{1, 2, 5, 7\}$, we then have $\Omega \setminus \mathcal{U} = \{3, 4, 6, 8, 9\}$. Now,
  $\mathrm{AS}_k(cell_{34}) = \{3, 4, 8, 9\} \subseteq \Omega \setminus \mathcal{U}$,
  $\mathrm{AS}_k(cell_{43}) = \{4, 8, 9\} \subseteq \Omega \setminus \mathcal{U}$,
  $\mathrm{AS}_k(cell_{44}) = \{3, 4, 6, 8, 9\} \subseteq \Omega \setminus \mathcal{U}$,
  $\mathrm{AS}_k(cell_{45}) = \{4, 6, 8, 9\} \subseteq \Omega \setminus \mathcal{U}$,
  $\mathrm{AS}_k(cell_{54}) = \{3, 4, 6, 8, 9\} \subseteq \Omega \setminus \mathcal{U}$.

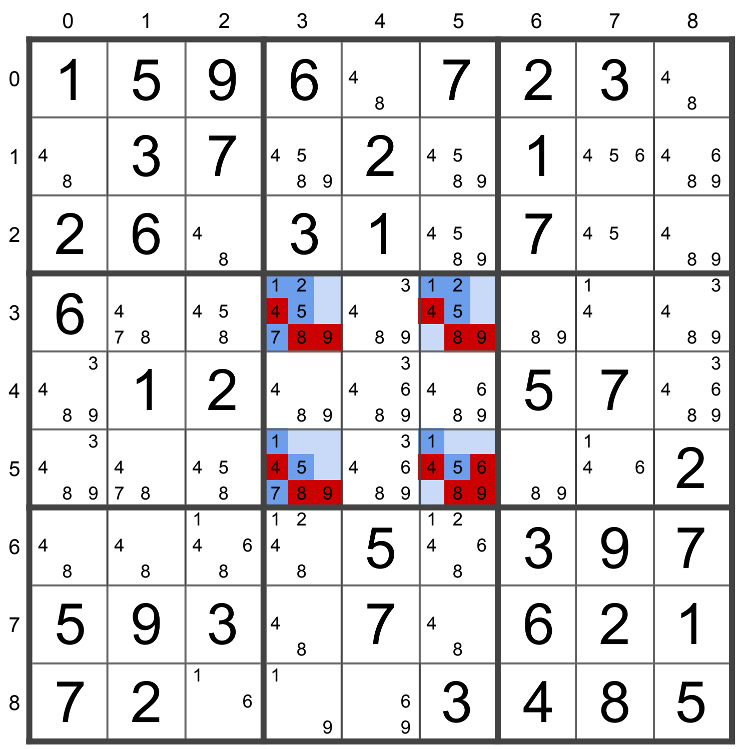


Figure 3.8: Example of a hidden subset of size 4.

3.2.1 Sudoku theory

Now, we state a theorem that shows that for every naked subset, there exists a complementary hidden subset in the same bracket, and the cells of the naked and its complementary hidden subset form a partition of the "parent" bracket. However, prior to that, we present the following lemma that will be used in the proof of the naked and hidden relationship theorem.

Lemma 3.2.7. *Let X, Y be two sets such that $X \subseteq Y$, then $Y \backslash (Y \backslash X) = X$.*

Proof. Let $Z = Y \setminus X$, then $Z \subseteq Y$, $Z \cup X = Y$, and $X \cap Z = \emptyset$. Therefore,

$Y \setminus (Y \setminus X) = Y \setminus Z = (X \cup Z) \setminus Z = X \cup \emptyset = X.$ □

With that in mind, we now state the theorem mentioned earlier. The theorem declares that a naked subset exists in a bracket if and only if the remaining cells form a hidden subset.

Theorem 3.2.8. *For a setting* $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*,* $\mathbb{S}$ *a bracket,* $\mathbb{V} \subseteq \mathbb{S}$ *and* $\mathcal{V} \subseteq \Omega$*, then* $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ *is a naked subset if and only if* $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ *is a hidden subset.*

Then, $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ *and* $(\mathbb{S}, \mathbb{S}\setminus\mathbb{V}, \Omega\setminus\mathcal{V})$ *are called* **complementary subsets***.*

Proof. We will use the following two equations throughout the proof. Since $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset, we have $|\mathbb{V}| = |\mathcal{V}|$. Let us define $m = |\mathbb{V}|$. Also, since $\mathbb{V} \subseteq \mathbb{S}$, we have

$$|\mathbb{S} \setminus \mathbb{V}| = |\mathbb{S}| - |\mathbb{V}|. \tag{3.2}$$

Additionally, since $\mathcal{V} \subseteq \Omega$, we have

$$|\Omega \setminus \mathcal{V}| = |\Omega| - |\mathcal{V}|. \tag{3.3}$$

First, we assume that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset and we are going to prove that $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ is a hidden subset, that is, we check that the three conditions of Definition 3.2.5 hold for $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$:

C1, $\mathbb{S} \setminus \mathbb{V} \subseteq \mathbb{S}$ and $\Omega \setminus \mathcal{V} \subseteq \Omega$:
It is clear that $\mathbb{S} \setminus \mathbb{V} \subseteq \mathbb{S}$ and $\Omega \setminus \mathcal{V} \subseteq \Omega$.

C2, $|\mathbb{S} \setminus \mathbb{V}| = |\Omega \setminus \mathcal{V}|$:
Since $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset, we have $|\mathbb{V}| = |\mathcal{V}|$, then $\mathcal{V} = m$. By (3.2) and (3.3), we have $|\mathbb{S}\setminus\mathbb{V}| = |\mathbb{S}| - |\mathbb{V}| = n - m = |\Omega| - |\mathcal{V}| = |\Omega \setminus \mathcal{V}|$.

C3, $\mathrm{AS}_k(cell) \subseteq \Omega \setminus (\Omega \setminus \mathcal{V})$ for every $cell \in \mathbb{S} \setminus (\mathbb{S} \setminus \mathbb{V})$:
First, since $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset, according to the third condition of the definition, we have that: $\mathrm{AS}(cell) \subseteq \mathcal{V}$ for every $cell \in \mathbb{V}$, then, applying Lemma 3.2.7 to $(\mathcal{V}, \Omega)$ and $(\mathbb{V}, \mathbb{S})$ we get $\mathrm{AS}(cell) \subseteq \Omega \setminus (\Omega \setminus \mathcal{V})$ for every $cell \in \mathbb{S} \setminus (\mathbb{S} \setminus \mathbb{V})$.

Therefore, we see that $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ is a hidden subset.

Now, for the second part of the proof, we assume that $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ is a hidden subset, and we will prove that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset.

C1, $\mathbb{V} \subseteq \mathbb{S}$ and $\mathcal{V} \subseteq \Omega$:
It is clear that $\mathbb{V} \subseteq \mathbb{S}$ and $\mathcal{V} \subseteq \Omega$ by hypothesis.

C2, $|\mathbb{V}| = |\mathcal{V}|$:
By hypothesis, since $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ is a hidden subset we have $|\mathbb{S} \setminus \mathbb{V}| = |\Omega \setminus \mathcal{V}|$. Then, by (3.2) and (3.3), we have $|\mathbb{S}| - |\mathbb{V}| = |\Omega| - |\mathcal{V}|$, therefore, $n - |\mathbb{V}| = n - |\mathcal{V}|$, finally $|\mathbb{V}| = |\mathcal{V}|$.

C3, $\mathrm{AS}(cell) \subseteq \mathcal{V}$ for every $cell \in \mathbb{V}$:
We know that $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ is a hidden subset, and by the third condition we get $\mathrm{AS}(cell) \subseteq \Omega \setminus (\Omega \setminus \mathcal{V})$ for every $cell \in \mathbb{S} \setminus (\mathbb{S} \setminus \mathbb{V})$. Now, applying Lemma 3.2.7 for $(\mathcal{V}, \Omega)$ and $(\mathbb{V}, \mathbb{S})$ we get $\mathrm{AS}(cell) \subseteq \mathcal{V}$ for every $cell \in \mathbb{V}$.

Therefore, $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset.

Finally, we conclude that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset if and only if $(\mathbb{S}, \mathbb{S} \setminus \mathbb{V}, \Omega \setminus \mathcal{V})$ is a hidden subset. □

We are now going to formalize the process of the naked subset prune and the hidden subset prune. Then, we will prove that applying them does not lose solutions. We start with the formalization of the naked subset prune. Given a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a naked subset $(\mathbb{S}, \mathbb{V}, \mathcal{V})$, let f_{N4} be the function that receives a setting and a naked subset and produces a new setting with a naked subset prune. We define

$$(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) = f_{N4}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}, \mathbb{V}, \mathcal{V})) \\ = (\mathbb{F}_k, \gamma_k, f^3_{N4}(\mathrm{AS}_k, \mathbb{S}, \mathbb{V}, \mathcal{V})),$$

we define $f^3_{N4}\big|_{cell}$ for every $cell \in \mathbb{G}$

$$\mathrm{AS}_{k+1}(cell) = f^3_{N4}(\mathrm{AS}_k, \mathbb{S}, \mathbb{V}, \mathcal{V})\big|_{cell} = \begin{cases} \mathrm{AS}_k(cell) & \text{if } cell \in \mathbb{G} \setminus \mathbb{S} \\ \mathrm{AS}_k(cell) & \text{if } cell \in \mathbb{V} \\ \mathrm{AS}_k(cell) \setminus \mathcal{V} & \text{if } cell \in \mathbb{S} \setminus \mathbb{V} \end{cases} \tag{3.4}$$

Now, we prove that no solution is lost when applying the just-described naked subset prune to a setting.

Theorem 3.2.9. *For a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, and a naked subset* $(\mathbb{S}, \mathbb{V}, \mathcal{V})$*,*

$$f_{N4}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}, \mathbb{V}, \mathcal{V})) = (\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) \text{ is a setting.}$$

Proof. By definition of f_{N4} we have that $\mathbb{F}_{k+1} = \mathbb{F}_k$, so $\mathbb{F}_k \subseteq \mathbb{F}_{k+1}$ and $\gamma_{k+1} = \gamma_k$, then $\gamma_k \preceq \gamma_{k+1}$. Therefore, the only thing left to prove is that AS_{k+1} is a valid available set function. Aiming for a contradiction, let us suppose it is not. Then there must exist a solution γ, and a $cell' \in \mathbb{G}$ such that $\gamma(cell') \notin \mathrm{AS}_{k+1}(cell')$.

We have that AS_k is a valid available set function, therefore $\gamma(cell') \in \mathrm{AS}_k(cell')$, so we can conclude the following relation:

$$\gamma(cell') \in \mathrm{AS}_k(cell') \setminus \mathrm{AS}_{k+1}(cell'). \tag{3.5}$$

Case 1, $cell' \in \mathbb{G} \setminus \mathbb{S}$ or $cell' \in \mathbb{V}$:

Then, by definition of f^3_{N4}, we have $\mathrm{AS}_{k+1} = \mathrm{AS}_k$, then by (3.5) we have $\gamma(cell') \in \emptyset$, which is a contradiction.

Case 2, $cell' \in \mathbb{S} \setminus \mathbb{V}$:

Then, by definition of f^3_{N4}, we have that $\mathrm{AS}_{k+1} = \mathrm{AS}_k \setminus \mathcal{V}$, then by (3.5) we have that $\gamma(cell') \in \mathrm{AS}_k \setminus (\mathrm{AS}_k \setminus \mathcal{V})$. By definition of set difference $\mathrm{AS}_k \setminus (\mathrm{AS}_k \setminus \mathcal{V}) = \mathrm{AS}_k \setminus (\mathrm{AS}_k \setminus (\mathrm{AS}_k \cap \mathcal{V}))$. Now, we can apply Lemma 3.2.7, and we get $\gamma(cell') \in (\mathrm{AS}_k \cap \mathcal{V})$, in particular $\gamma(cell') \in \mathcal{V}$.
Let $\gamma(cell') = v \in \mathcal{V}$.

So we have that $cell'$ is not in $\mathbb{V}$ because we are in Case 2, but its value is in $\mathcal{V}$.

Given the fact that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset, we know from the third condition of the Definition 3.2.2 of naked subset that

$$\mathrm{AS}_k(cell) \subseteq \mathcal{V} \text{ for every } cell \in \mathbb{V} \tag{3.6}$$

And since AS_k is a valid available set function for every $cell \in \mathbb{V}$ we have $\gamma(cell) \in \mathcal{V}$. But, for every $cell \in \mathbb{V}$, different than $cell'$, since $cell' \in \mathbb{S} \setminus \mathbb{V}$ we see that $cell'$ and $cell$ are neighbors, and two neighbors cannot have the same value. Therefore, for every $cell \in \mathbb{V}$, we must have $\gamma(cell) \in \mathcal{V} \setminus \{v\}$.

Then we get m *cells* in $\mathbb{V}$, and $m - 1$ values in $\mathcal{V} \setminus \{v\}$, and every *cell* should take a value in $\mathcal{V} \setminus \{v\}$ without repeating because they are in the same bracket $\mathbb{S}$. This is clearly not possible because we have fewer values than *cells*.

We can then conclude that there exists a $cell^* \in \mathbb{V}$ such that $\gamma(cell^*) \notin \mathcal{V}$. Also, $cell^* \in \mathbb{V}$, then (3.6) implies that $\mathrm{AS}_k(cell^*) \subseteq \mathcal{V}$. Therefore, $\gamma(cell^*) \notin \mathrm{AS}_k(cell^*)$ which is a contradiction because AS_k is a valid available set function.

Then, by *reducto ad absurdum*, $\mathbb{AS}_{k+1}$ is a valid available set function, subsequently $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathbb{AS}_{k+1})$ is a setting. □

Now, we formally define the hidden subset prune.
Given a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a hidden subset $(\mathbb{S}, \mathbb{U}, \mathcal{U})$, let f_{H4} be the function that receives a setting and an intersection and produces a new setting with a hidden subset prune. We define

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathbb{AS}_{k+1}) &= f_{H4}((\mathbb{F}_k, \gamma_k, \mathbb{AS}_k), (\mathbb{S}, \mathbb{U}, \mathcal{U})) \\ &= (\mathbb{F}_k, \gamma_k, f^3_{H4}(\mathbb{AS}_k, \mathbb{U}, \mathcal{U}))\end{aligned}$$

where, for every $cell \in \mathbb{G}$:

$$\mathbb{AS}_{k+1}(cell) = f^3_{H4}(\mathbb{AS}_k, \mathbb{U}, \mathcal{U})\big|_{cell} = \begin{cases} \mathbb{AS}_k(cell) & \text{if } cell \in \mathbb{G} \setminus \mathbb{U} \\ \mathbb{AS}_k(cell) \cap \mathcal{U} & \text{if } cell \in \mathbb{U} \end{cases} \tag{3.7}$$

Now, we are going to prove that no solutions are lost when applying a hidden subset prune to a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$. We take advantage of Theorem 3.2.9, which states that a naked subset prune does not lose solutions.

First, we will prove that the prune performed by a hidden subset prune f_{H4} is exactly the same as the prune performed by its complementary naked subset prune f_{N4}. Then, since we have proven that every naked subset prune does not lose solutions, we can conclude that every hidden subset prune also does not lose solutions.

The following theorem proves that complementary prunes are the same.

Theorem 3.2.10. *Given a setting* $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$ *and a naked subset* $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ *with its complementary hidden subset* $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ *defined in Theorem 3.2.8. Let* $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_N)$ *be the result of applying the naked* *subset prune:*

$$(\mathbb{F}_k, \gamma_k, \mathrm{AS}_N) := f_{N4}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}, \mathbb{V}, \mathcal{V})),$$

and let $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_H)$ *be the result of applying the hidden subset prune:*

$$(\mathbb{F}_k, \gamma_k, \mathrm{AS}_H) := f_{H4}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{S}, \mathbb{V}, \mathcal{V})).$$

Then $\mathrm{AS}_N = \mathrm{AS}_H$*.*

Proof. Since $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ and $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ are complementary subsets, we have the following equations

$$\mathbb{U} = \mathbb{S} \setminus \mathbb{V} \tag{3.8}$$

and

$$\mathcal{U} = \Omega \setminus \mathcal{V} \tag{3.9}$$

Now, let us prove that $\mathrm{AS}_N(\mathit{cell}) = \mathrm{AS}_H(\mathit{cell})$ for every $\mathit{cell} \in \mathbb{G}$. Please recall the formal definition of naked subset prune f_{N4} (3.4), page 79, and the definition of hidden subset prune f_{H4} (3.7), page 81.

Case 1, $\mathit{cell} \in \mathbb{G} \setminus \mathbb{S}$:

By definition of f^3_{N4}, we have $\mathrm{AS}_N(\mathit{cell}) = \mathrm{AS}_k(\mathit{cell})$. By definition of f^3_{H4}, since $\mathit{cell} \in \mathbb{G} \setminus \mathbb{S}$, then $\mathit{cell} \in \mathbb{G} \setminus \mathbb{U}$ because $\mathbb{U} \subseteq \mathbb{S}$, then we have $\mathrm{AS}_H(\mathit{cell}) = \mathrm{AS}_k(\mathit{cell})$. Therefore, $\mathrm{AS}_N(\mathit{cell}) = \mathrm{AS}_H(\mathit{cell})$.

Case 2, $\mathit{cell} \in \mathbb{V}$:

By definition of f^3_{N4}, we have $\mathrm{AS}_N(\mathit{cell}) = \mathrm{AS}_k(\mathit{cell})$.
Now, for f^3_{H4}, and $\mathit{cell} \in \mathbb{V}$, we also get $\mathit{cell} \in \mathbb{S} \setminus \mathbb{U}$, by equation (3.8) and Lemma 3.2.7 we have $\mathbb{V} = \mathbb{S} \setminus (\mathbb{S} \setminus \mathbb{V}) = \mathbb{S} \setminus \mathbb{U} \subseteq \mathbb{G} \setminus \mathbb{U}$. Therefore, the definition of f^3_{H4} gives $\mathrm{AS}_H(\mathit{cell}) = \mathrm{AS}_k(\mathit{cell})$ Therefore, $\mathrm{AS}_N(\mathit{cell}) = \mathrm{AS}_H(\mathit{cell})$.

Case 3, $\mathit{cell} \in \mathbb{S} \setminus \mathbb{V}$:

By definition of f^3_{N4}, we have $\mathrm{AS}_N(\mathit{cell}) = \mathrm{AS}_k(\mathit{cell}) \setminus \mathcal{V}$.

Now, for f^3_{H4}, since $cell \in \mathbb{S} \setminus \mathbb{V}$, it means $cell \in \mathbb{U}$, due to (3.8). Then, by the definition of f^3_{H4}, we have $\text{AS}_H(cell) = \text{AS}_k(cell) \cap \mathcal{U}$. Furthermore, we have $\mathcal{U} = \Omega \setminus \mathcal{V}$ because of (3.9). Then, $\text{AS}_H(cell) = \text{AS}_k(cell) \cap (\Omega \setminus \mathcal{V}) = \text{AS}_k(cell) \setminus \mathcal{V}$. Therefore, we can conclude that $\text{AS}_N(cell) = \text{AS}_H(cell)$.

In summary, we have $\text{AS}_N(cell) = \text{AS}_H(cell)$ for every $cell \in \mathbb{G}$. □

Now that we have proven that the prune of complementary subsets is the same, we are going to prove that no solution is lost when applying a hidden subset prune to a setting.

Corollary 3.2.11. *For a setting* $(\mathbb{F}_k, \gamma_k, \text{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$ *and a hidden subset* $(\mathbb{S}, \mathbb{U}, \mathcal{U})$*,*

$$f_{H4}((\mathbb{F}_k, \gamma_k, \text{AS}_k), (\mathbb{S}, \mathbb{U}, \mathcal{U})) = (\mathbb{F}_{k+1}, \gamma_{k+1}, \text{AS}_{k+1}) \textit{ is a setting.}$$

Proof. Using Theorem 3.2.8, page 77, we know that there exists a complementary naked subset $(\mathbb{S}, \mathbb{S} \setminus \mathbb{U}, \Omega \setminus \mathcal{U})$. Now, applying the last Theorem 3.2.10, we get the following:

$$f_{N4}((\mathbb{F}_k, \gamma_k, \text{AS}_k), (\mathbb{S}, \mathbb{S} \setminus \mathbb{U}, \Omega \setminus \mathcal{U})) = \\ f_{H4}((\mathbb{F}_k, \gamma_k, \text{AS}_k), (\mathbb{S}, \mathbb{U}, \mathcal{U})) = (\mathbb{F}_{k+1}, \gamma_{k+1}, \text{AS}_{k+1}).$$

By Theorem 3.2.9 page 79, we know that every naked subset prune produces a setting. That is, $f_{N4}((\mathbb{F}_k, \gamma_k, \text{AS}_k), (\mathbb{S}, \mathbb{S} \setminus \mathbb{U}, \Omega \setminus \mathcal{U}))$ is a setting. Therefore, $(\mathbb{F}_{k+1}, \gamma_{k+1}, \text{AS}_{k+1})$ is a setting. □

3.2.2 Implementation

The implementation for finding subsets and pruning them is not straightforward; hence we will divide this subsection into two: Theory and code.

In the theoretic part, we introduce a concept that is a generalization of both types of subsets (naked and hidden). Having a concept that groups both types of subsets helps us simplify the implementation by reusing code. In the code part we will apply the theory to search for subsets and calculate the time complexity.

Theory

First, we will introduce two general concepts that will eventually include naked and hidden subsets.

Definition 3.2.12. Given two sets X and Y and a function $\phi \colon X \to \mathcal{P}(Y)$, we define (X, Y, ϕ) as a ***cover*** if the following conditions are met:

- $1 \leq |X| = |Y|$.
- $\phi(x) \subseteq Y$ for every $x \in X$.

We refer to ϕ as ***cover function*** and X, Y as ***cover domain*** and ***cover image***, respectively. We say that m is the size of a cover if $m = |X| = |Y|$

We introduce covers because we want to group both naked and hidden subsets into one concept. Since both types of subsets are subsets of a bracket, we want the tuple $(\mathbb{S}, \Omega, \mathrm{AS}_k)$ to be a cover, where $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ is a setting of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and $\mathbb{S}$ is a bracket. This is not the case because the domain of AS_k is the whole grid $\mathbb{G}$, and not the bracket $\mathbb{S}$. However, with the following restriction we will be able to use only the part of AS_k that we are interested in.

Definition 3.2.13. Given a function $\phi \colon X \to Y$, and a subset $\tilde{X} \subseteq X$, we define $\phi_{\tilde{X}} \colon \tilde{X} \to Y$ as follows: $\phi_{\tilde{X}}(x) = \phi(x)$ for every $x \in \tilde{X}$.

We say that $\phi_{\tilde{X}}$ is a ***restricted function***, and, particularly, $\phi_{\tilde{X}}$ should be read as ϕ restricted to $\tilde{X}$.

It is important to notice that restrictions to functions can be recursively applied, meaning, we can restrict a function that was previously restricted whenever the new domain is a subset of the previous domain. For this scenario, we will use the following notation: For a function $\phi\colon X \to Y$, and $X_2 \subseteq X_1 \subseteq X$ we denote

$$\phi_{(X_1)(X_2)} := (\phi_{(X_1)})_{(X_2)} = \phi_{X_2}$$

Now, we prepare a theorem about equivalent restricted functions.

Corollary 3.2.14. *For a function $\phi : X \to Y$, and $X_2 \subseteq X_1 \subseteq X$, we have that $\phi_{(X2)} = \phi_{(X1)(X2)}$.*

A special case of $\phi_{(X_1)}$ is $\mathrm{AS}_{k(\mathbb{S})}$, which will be used frequently in the following pages.

Lemma 3.2.15. *For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and a bracket $\mathbb{S}$, we have that $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$ is a cover.*

Proof. Since $\mathbb{S}$ is a bracket, we have $|\mathbb{S}| = |\Omega| = n$, and since AS_k is a valid available set function, we have $\mathrm{AS}_{k(\mathbb{S})}(cell) \subseteq \Omega$. □

Finally, we define the concept that we were longing to state, because eventually this concept will include both naked and hidden subsets.

Definition 3.2.16. Given a cover (X, Y, ϕ), we define $(X', Y', \phi_{(X')})$ as ***subcover*** if the following conditions are met:

- $X' \subseteq X$ and $Y' \subseteq Y$.
- $(X', Y', \phi_{(X')})$ is a cover.

Now, we state a theorem that relates naked subsets and subcovers.

Theorem 3.2.17. *For a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, then* $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ *is a naked subset if and only if* $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ *is a subcover of* $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$*.*

Proof. In this case, we are going to prove that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover of $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$.

We assume that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset, and we will prove that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover of $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$. By Definition 3.2.2, page 71 of a naked subset, we have $\mathbb{V} \subseteq \mathbb{S}$, $\mathcal{V} \subseteq \Omega$. Then, the only thing left to prove is that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a cover. First, let $m = |\mathbb{V}|$, by Definition 3.2.2 of a naked subset, we have $|\mathbb{V}| = |\mathcal{V}| = m$ and $\mathrm{AS}_k(cell) \subseteq \mathcal{V}$ for every $cell \in \mathbb{V}$.

Also by Definition 3.2.13 of restricted function $\mathrm{AS}_{k(\mathbb{V})}$, we have that for every $cell \in \mathbb{V}$, $\mathrm{AS}_{k(\mathbb{V})}(cell) = \mathrm{AS}_k(cell)$, and we get that for every $cell \in \mathbb{V}$ that $\mathrm{AS}_{k(\mathbb{V})}(cell) \subseteq \mathcal{V}$.

Now, by Corollary 3.2.14 we have that $\mathrm{AS}_{k(\mathbb{V})} = \mathrm{AS}_{k(\mathbb{S})(\mathbb{V})}$. Therefore, all conditions of Definition 3.2.16 are satisfied and we conclude that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover of $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$.

Now, on the other hand, we assume that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover of $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$, and we have to show that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset, that is, we check the three conditions in Definition 3.2.2:

C1, $\mathbb{V} \subseteq \mathbb{S}$ and $\mathcal{V} \subseteq \Omega$:
By Definition 3.2.16 of the subcover, we get $\mathbb{V} \subseteq \mathbb{S}$, and $\mathcal{V} \subseteq \Omega$.

C2, $|\mathbb{V}| = |\mathcal{V}|$:
Since $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover, it is also a cover; therefore, we have $|\mathbb{V}| = |\mathcal{V}|$.

C3, For every $cell \in \mathbb{V}$ we need $\mathrm{AS}_k(cell) \subseteq \mathcal{V}$:

By the definition of cover, for every *cell* $\in \mathbb{V}$ we get $\mathrm{AS}_{k(\mathbb{V})}(cell) \subseteq \mathcal{V}$, and by the definition of a restricted function, for every *cell* $\in \mathbb{V}$ we have $\mathrm{AS}_{k(\mathbb{V})}(cell) = \mathrm{AS}_k(cell)$, which implies $\mathrm{AS}_k(cell) \subseteq \mathcal{V}$.

Then we finally see that $(\mathbb{S}, \mathbb{V}, \mathcal{V})$ is a naked subset if and only if $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover of $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$ □

Recall that we introduced subcovers to include naked and hidden subsets, but so far we have only proved that naked subsets are subcovers. To include hidden subsets as subcovers, we will have to look at things from a different perspective.

Previously, we have used a function AS that relates each *cell* to a set of candidates, and when this relationship satisfies certain conditions, we know that we have found a naked subset.

For hidden subsets, we will want to see the relationship between cells and candidates from the opposite point of view. Instead of having a function that relates a *cell* with its set of candidates, we want a function that relates each candidate to the set of *cells* whose available sets contain the candidate. However, since we are looking for the cover of a bracket, we are going to limit the image of this function to a bracket that contains the cells we are interested in. Consequently, we will take the preimage of an available set function that is restricted to a specific bracket and candidate ω. We will loosely use the attribute *inverse* instead of the preimage. We use the notation $\mathrm{AS}^{-1}_{k(\mathbb{S})}$, which is a shorthand notation of $(\mathrm{AS}_{k(\mathbb{S})})^{-1}$. That is, first we restrict the function and then take the preimage.

Definition 3.2.18. For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a bracket $\mathbb{S}$ we define:

$$\mathrm{AS}^{-1}_{k(\mathbb{S})}(\omega) = \{cell_1, \ldots, cell_p\}$$

where $\omega \in \Omega$ and $\{cell_1, \ldots, cell_p\} \subseteq \mathbb{S}$ such that $\omega \in \mathrm{AS}_k(cell_1), \ldots, \omega \in \mathrm{AS}_k(cell_p)$.

Now, we include hidden subsets as covers by defining the cover of a bracket with this inverse relationship as the cover function.

Lemma 3.2.19. *For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and a bracket $\mathbb{S}$, the tuple $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$ is a cover.*

Proof. Since $\mathbb{S}$ is a bracket, $|\mathbb{S}| = |\Omega| = 9$. By the definition of the inverse available set function we have $\mathrm{AS}^{-1}_{k(\mathbb{S})}(\omega) \subseteq \mathbb{S}$ for each $\omega \in \Omega$. □

Now, before proving that hidden subsets are also subcovers, let us state an equivalence that will be useful for establishing the relation between hidden subsets and subcovers.

Theorem 3.2.20. *For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, a bracket $\mathbb{S}$, and $\mathbb{U} \subseteq \mathbb{S}$, $\mathcal{U} \subseteq \Omega$ we see that $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is a hidden subset if and only if $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a subcover of $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$.*

Note that the order of $\mathbb{S}$ and Ω changed compared to Theorem 3.2.17.

Proof. First, we assume $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is a hidden subset, we are going to prove that $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a subcover of $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$. Therefore, we have to prove the following pair of conditions:

C1, $\mathbb{U} \subseteq \mathbb{S}, \mathcal{U} \subseteq \Omega$:
By hypothesis we have that this condition is true.

C2, $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a cover:
Using the definition of hidden subsets, we have $|\mathbb{U}| = |\mathcal{U}|$. We only have left to prove that $(\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})}(\omega') \subseteq \mathbb{U}$ for every $\omega' \in \mathcal{U}$. By Definition 3.2.18, we have the following self evident equivalence:

$$\text{There exists } \omega' \in \mathcal{U} \text{ such that } (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})}(\omega') \cap (\mathbb{S} \setminus \mathbb{U}) \neq \emptyset,$$
$$\text{if and only if}$$
$$\text{there exists a } \mathit{cell}' \in \mathbb{S} \setminus \mathbb{U} \text{ such that } \mathrm{AS}_k(\mathit{cell}') \cap \mathcal{U} \neq \emptyset. \quad (3.10)$$

Let us suppose C2 is false, then we have that it exists $\omega' \in \mathcal{U}$ such that $(\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\Omega \setminus \mathcal{U})}(\omega') \cap \mathbb{S} \setminus \mathbb{U} \neq \emptyset$ and by (3.10) we have a contradiction since there exists $\mathit{cell}' \in \mathbb{S} \setminus \mathbb{U}$ such that $\mathrm{AS}_k(\mathit{cell}') \cap \mathcal{U} \neq \emptyset$ which means that $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is not a hidden subset. Then $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a cover.

Therefore $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a subcover of $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$.

Now, on the other hand, we assume that $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a subcover of $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$, and we show that $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is a hidden subset.

According to the definition of hidden subsets Definition 3.2.5, page 74, we have to verify the following conditions:

C1, $\mathbb{U} \subseteq \mathbb{S}$ and $\mathcal{U} \subseteq \Omega$:
By hypothesis we have that this condition is true.

C2, $|\mathbb{U}| = |\mathcal{U}|$:
By definition of subcover we have that $|\mathbb{U}| = |\mathcal{U}|$.

C3, $\mathrm{AS}_k(\mathit{cell}) \subseteq \Omega \setminus \mathcal{U}$ for every $\mathit{cell} \in \mathbb{S} \setminus \mathbb{U}$:
Let us suppose C3 is false then there exists $\mathit{cell}' \in \mathbb{S} \setminus \mathbb{U}$ such that $\mathrm{AS}_k(\mathit{cell}') \cap \mathcal{U} \neq \emptyset$. Now, applying (3.10) we have that it exists $\omega' \in \mathcal{U}$ such that $(\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})}(\omega') \cap (\mathbb{S} \setminus \mathbb{U}) \neq \emptyset$, which is a contradiction since $(\mathcal{U}, \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$ is a subcover of $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$.

Since all conditions are met, we see that $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is a hidden subset. Finally, $(\mathbb{S}, \mathbb{U}, \mathcal{U})$ is a hidden subset if and only if $(\Omega \setminus \mathcal{U}, \mathbb{S} \setminus \mathbb{U}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\Omega \setminus \mathcal{U})})$ is a subcover of $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$. □

We have now included both types of subsets as subcovers. Now, the following concept will be used when searching for subcovers, in the context of sudokus.

Definition 3.2.21. We say that a subcover $(X', Y', \phi_{(X')})$ of (X, Y, ϕ) is ***isolated*** if $(X \setminus X', Y \setminus Y', \phi_{X \setminus X'})$ is a subcover.

The concept of isolation is useful because, in the context of sudokus, if the subcover representation of a subset is isolated, it means that there is no extra information given by the subset. Meaning that if a subcover is isolated, the subset (hidden or naked) represented by that subcover is not able to do any subset pruning. Therefore, we have to search exclusively for non-isolated covers.

Example 3.2.22. For the setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ in Figure 3.9. Let $\mathbb{V} = \{cell_{81}, cell_{82}\}$, and $\mathcal{V} = \{1, 5\}$ we find that $(\mathbb{B}_6, \mathbb{V}, \mathcal{V})$ is a naked subset. Then, we know, by Theorem 3.2.17 that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is a subcover of $(\mathbb{B}_6, \Omega, \mathrm{AS}_{k(\mathbb{B}_6)})$. Now, let us verify that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is isolated. We need to check that $(\mathbb{B}_6 \setminus \mathbb{V}, \Omega \setminus \mathcal{V}, \mathrm{AS}_{k(\mathbb{B}_6 \setminus \mathbb{V})})$ is a subcover. That is, we are going to verify that for each $cell \in \mathbb{B}_6 \setminus \mathbb{V}$, we have that $\mathrm{AS}_{k(\mathbb{B}_6 \setminus \mathbb{V})}(cell) \subseteq \Omega \setminus \mathcal{V}$. The set $\mathbb{S} \setminus \mathbb{V}$ is equal to $\{cell_{60}, cell_{61}, cell_{62}, cell_{70}, cell_{71}, cell_{72}, cell_{80}\}$.

- $\mathrm{AS}_{k(\mathbb{S})}(cell_{60}) = \{7\} \subseteq \Omega \setminus \mathcal{V}$,
- $\mathrm{AS}_{k(\mathbb{S})}(cell_{61}) = \{3\} \subseteq \Omega \setminus \mathcal{V}$,
- $\mathrm{AS}_{k(\mathbb{S})}(cell_{62}) = \{2, 8\} \subseteq \Omega \setminus \mathcal{V}$,
- $\mathrm{AS}_{k(\mathbb{S})}(cell_{70}) = \{9\} \subseteq \Omega \setminus \mathcal{V}$,
- $\mathrm{AS}_{k(\mathbb{S})}(cell_{71}) = \{2, 8, 6\} \subseteq \Omega \setminus \mathcal{V}$,
- $\mathrm{AS}_{k(\mathbb{S})}(cell_{72}) = \{2, 8, 6\} \subseteq \Omega \setminus \mathcal{V}$,
- $\mathrm{AS}_{k(\mathbb{S})}(cell_{80}) = \{4\} \subseteq \Omega \setminus \mathcal{V}$.

Therefore, we can conclude that $(\mathbb{V}, \mathcal{V}, \mathrm{AS}_{k(\mathbb{V})})$ is isolated. Notice that, as we stated, there are no values to prune with a naked subset prune.

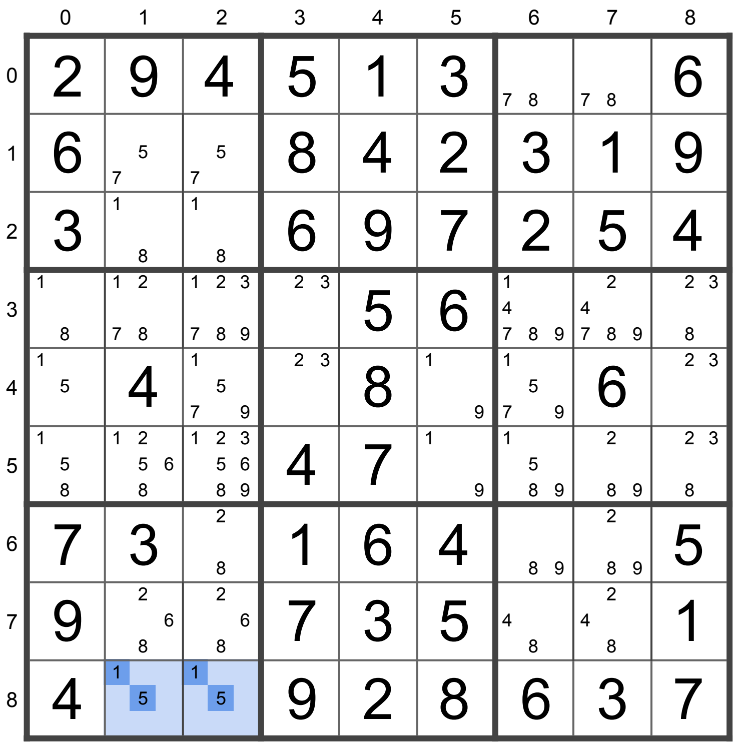


Figure 3.9: Example of an isolated subcover.

It is interesting to note that when a subset is found, after the pruning is done, the cover corresponding to the resultant subset will always be isolated.

In this context, when the cover representation of a subset is isolated, then the subset is both a naked and a hidden subset, which means that the complementary set is a naked and a hidden subset as well.

Code

We will implement a way to look for every naked subset and hidden subset in the grid and then make the corresponding prune. First of all, we are going to create a class `Cover` for any kind of covers, this class will have a functionality that looks for subcovers. Looking for subcovers of any size is not trivial. Therefore, we will make this functionality dependent on the size of the subcover.

Once the cover class is defined, we assign two covers to every bracket $\mathbb{S}$. The first represents the cover $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$ for naked subsets. The second uses the inverse available set function $(\Omega, \mathbb{S}, (\mathrm{AS}^{-1}_{k(\mathbb{S})})_{(\mathcal{U})})$, which represents hidden subsets that we denote as inverse covers. Finally, we will look for subcovers in all the covers we created; if a subcover is found in the standard covers, we have found a naked subset, then if a subcover is found in the inverse covers, we have found a hidden subset. Therefore, we will add two methods, one for naked subsets and one for hidden subsets; these will look for subcovers in the corresponding type of bracket cover, and if one is found, they will extract the cells and candidates involved. Then, we prune the cells and candidates accordingly.

The algorithm in the class `Cover` that finds a subcover is recursive and complicated to understand, so, we are going to show a pseudocode of a simpler version of the algorithm to explain how it works and to calculate its time complexity. After that we will only show the code added to the `SudokuGrid` class, assuming that we got the `Cover` class working correctly. If the reader is interested in the code of the `Cover` class and how we implemented the search for subcovers we have the code in the repository of this work.

```
**Input:** A setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$, and a bracket $\mathbb{S}$.
if We are looking for naked subsets then
    $X := \mathbb{S}$ is a set of cells ;
    $Y := \Omega$ is a set of candidates ;
    $\phi\colon X \to \mathcal{P}(Y)$ is $\mathbb{AS}_{k(\mathbb{S})}$
else
    $X := \Omega$ is a set of candidates ;
    $Y := \mathbb{S}$ is a set of cells ;
    $\phi\colon X \to \mathcal{P}(Y)$ is $(\mathbb{AS}^{-1}_{k(\mathbb{S})})$ ;
end
foreach $X'$ subset of size $m$ of $X$ do
    $Y' = \bigcup_{x' \in X'} \phi(x')$ ;
    if $|Y'| = m$ then
        We have a subcover ;
        if $\bigcup_{x \in X \setminus X'} \phi(x) \cap Y' \neq \emptyset$ then
            The subcover is not isolated ;
            return $(X', Y')$ ;
        end
    end
end
return $(\emptyset, \emptyset)$ ;
```

Algorithm 1: Find a non-isolated subcover

When $|Y'| = m$ is satisfied, we have found a subcover, because by the definition of Y' for every $x' \in X'$ we get $\phi(x') \subseteq Y'$, and of course $|X'| = |Y'|$. When we ask if X' is a subset of X of size m, we are checking if the subcover that we have just found is not isolated, see Definition 3.2.21 of an isolated subcover.

For the time complexity of the algorithm, iterating through every subset of size m from a parent set of size n can be seen as making m nested loops, where each for loop selects one element of the parent set, if we arbitrarily order the parent set. Since we do not want to repeat elements, the first for loop starts with the first element, but the second starts one after the first, and so on and so forth. To iterate every subset of size m we end up with the following time complexity $\mathcal{O}(n \cdot (n-1) \cdot \ldots \cdot (n-(m-1)))$, which is equal to $\mathcal{O}(n^m + \alpha_{m-1}n^{m-1} + \ldots + \alpha_1 n + \alpha_0)$ where α_i is a natural number for every $i \in \{0, \ldots, m-1\}$. Therefore, we end up with a time complexity of $\mathcal{O}(n^m)$.

However, for each of the subsets we have two if conditions; both of them calculate the union of sets. First, for $\bigcup_{x' \in X'} \phi(x')$, for every $x' \in X'$, we know that $|\phi(x')| \leq |Y| = n$. And since $|X'| = m$, in the worst case, we are calculating a union of m sets of size $|n|$, which means that we have a time complexity of $\mathcal{O}(mn)$. Next, for $\bigcup_{x \in X \setminus X'} \phi(x) \cap Y'$ we know that $|X \setminus X'| < |X| = n$ and $|\phi(x)| \leq n$ for every $x \in X \setminus X'$. Therefore, we have a time complexity of $\mathcal{O}(n^2)$. Finally, for the whole Find subcover algorithm, we have a time complexity of $\mathcal{O}(n^m) \cdot \mathcal{O}(mn) \cdot \mathcal{O}(n^2) = \mathcal{O}(m \cdot n^{m+3})$:

$$\mathcal{O}(\textit{Find subcover}) = \mathcal{O}(n^{m+3}). \tag{3.11}$$

Now that we have the algorithm for finding a subcover and its complexity, we can move on assuming we have a `Cover` class which has a `find_subcover` method. To create a `Cover` object it is necessary to provide two parallel arrays, the first one representing the cover's domain and the second one its image.

For each bracket $\mathbb{S}$, we will add the cover $(\mathbb{S}, \Omega, \mathrm{AS}_{k(\mathbb{S})})$ by adding the method `define_covers` to the `Bracket` class. And we do the same thing for the inverse cover $(\Omega, \mathbb{S}, \mathrm{AS}^{-1}_{k(\mathbb{S})})$ by adding `define_inverse_covers` to `Bracket`. The method `inverse_av_set` is just an implementation of $\mathrm{AS}^{-1}_{k(\mathbb{S})}$ for a specific bracket $\mathbb{S}$.

These methods are called from a function in `SudokuGrid` that creates the covers for that grid.

```
class Bracket:
    """..."""
    def define_covers(self):
        self.covers = [Cover(SudokuGrid.I, [cell.av_set for cell in bracket])
     for bracket in self.all]

    def inverse_av_set(self, bracket):
         raw_av_sets = [{i for i in SudokuGrid.I if bracket[i].av_set is not
     None and omega in bracket[i].av_set} for omega in
         SudokuGrid.Omega]
         return [av_set if 1 <= len(av_set) else None for av_set in raw_av_sets]

    def define_inverse_covers(self):
        self.inverse_covers = [Cover(SudokuGrid.Omega, self.inverse_av_set(
     bracket)) for bracket in self.all]


class SudokuGrid:
    """..."""
    def define_bracket_covers(self):
        self.brackets.define_covers()
        self.brackets.define_inverse_covers()
```

Now, using the cover class in the `SudokuGrid` class, we define the method `find_subset` that receives an array of `Cover` elements and looks through the array for a subcover of size `m`. If it finds a subcover, the method returns it, as well as the index of the subcover in the `covers` parameter. If not, it returns `None`.

```
class SudokuGrid:
    """..."""
    def find_subset(self, covers, m):
        for index in range(len(covers)):
            cover = covers[index]
            subcover = cover.find_subcover(m)
            if subcover is not None:
                return [index, subcover]
        return None
```

The `prune_cells` method receives a list of *cells* and a list of candidates, and then removes all candidates from the available sets of cells in the list.

Finally, the methods `find_naked_subset` and `find_hidden_subset` use the method `find_subset` to look for a subcover of size `m`. When a subcover is found, they convert the subcover back to *cells* and candidates, which is followed by the corresponding `prune_cells` call. Because in the case of naked subsets the cover domain is made up of *cells* and the image of candidates. In the case of hidden subsets, the cover domain consists of candidates and the image consists of *cells*.

```python
class SudokuGrid:
    """..."""
    def find_naked_subset(self, m):
        subset_result = self.find_subset(self.brackets.covers, m)
        if subset_result is not None:
            index, subcover = subset_result
            naked_indexes, naked_values = subcover
            bracket = self.brackets.all[index]
            self.prune_cells([bracket[i] for i in SudokuGrid.I if i not in
     naked_indexes], naked_values)

    def find_hidden_subset(self, m):
        subset_result = self.find_subset(self.brackets.inverse_covers, m)
        if subset_result is not None:
            index, subcover = subset_result
            hidden_values, hidden_indexes = subcover
            bracket = self.brackets.all[index]
            self.prune_cells([bracket[i] for i in hidden_indexes],
                                [omega for omega in SudokuGrid.Omega if omega not
     in hidden_values])

    def stage_four(self,m):
        self.define_bracket_covers(self)
        self.find_naked_subset(m)
        self.find_hidden_subset(m)
```

The time complexity of `find_naked_subset` and `find_hidden_subset` is $\mathcal{O}(3 \cdot n) \cdot \mathcal{O}(\textit{Find subcover})$. That is,

$$\mathcal{O}(\texttt{find\_naked\_subset}) = \mathcal{O}(3 \cdot n) \cdot \mathcal{O}(m \cdot n^{m+3}) = \mathcal{O}(m \cdot n^{m+4})$$

$$\text{and } \mathcal{O}(\texttt{find\_hidden\_subset}) = \mathcal{O}(m \cdot n^{m+4}) \text{ as well.}$$

We can conclude that

$$\mathcal{O}(\texttt{stage\_four}) = 2\mathcal{O}(m \cdot n^{m+4}) = \mathcal{O}(m \cdot n^{m+4}).$$

3.3 Stage Five, Orthogonal subsets

In this stage, for a fixed candidate ω, we look for a subset of the entire grid, made up by rows and columns that have cells with ω as a candidate. This technique is similar to the one implemented in stage four, the main difference being that it needs multiple brackets and that it is only able to prune one candidate for each subset. By a candidate, we mean one specific number; however, this candidate can be pruned from different cells. Let us start with an example to illustrate the technique we want to apply.

Example 3.3.1. For the setting shown in Figure 3.10 we make the following assertion:

There are only two cells in rows $\mathbb{R}_2$ and $\mathbb{R}_4$ that have the candidate 4 and, in both cases, they lie in columns $\mathbb{C}_1, \mathbb{C}_4$. (3.12)

Therefore, we can conclude the following two assertions about where candidate 4 is located, for every solution γ and its set associated function Γ, we have that $4 \in \Gamma(\mathbb{R}_2)$, then:

$$\gamma(cell_{21}) = 4 \text{ or } \gamma(cell_{24}) = 4, \tag{3.13}$$

and since $4 \in \Gamma(\mathbb{R}_4)$, we have:

$$\gamma(cell_{41}) = 4 \text{ or } \gamma(cell_{44}) = 4. \tag{3.14}$$

The interesting thing is that (3.13) and (3.14) imply the following two "orthogonal" conditions (3.15) and (3.16):

$$\gamma(cell_{21}) = 4 \text{ or } \gamma(cell_{41}) = 4, \tag{3.15}$$

because otherwise $\gamma(cell_{21}) \neq 4$ and $\gamma(cell_{41}) \neq 4$ give $\gamma(cell_{44}) = 4$ by (3.14) and $\gamma(cell_{24}) = 4$ by (3.13), which is a contradiction, since they are in column $\mathbb{C}_4$. Additionally, we have the following.

$$\gamma(cell_{24}) = 4 \text{ or } \gamma(cell_{44}) = 4, \tag{3.16}$$

because otherwise $\gamma(cell_{24}) \neq 4$ and $\gamma(cell_{44}) \neq 4$ give $\gamma(cell_{41}) = 4$ by (3.14) and $\gamma(cell_{21}) = 4$ by (3.13), which is a contradiction, since they are in column $\mathbb{C}_1$.

Therefore, thanks to (3.15) and (3.16) we can remove 4 from the available sets of all cells in $\mathbb{C}_1$ and $\mathbb{C}_4$, except those in $\mathbb{R}_2, \mathbb{R}_4$. As highlighted in Figure 3.10.

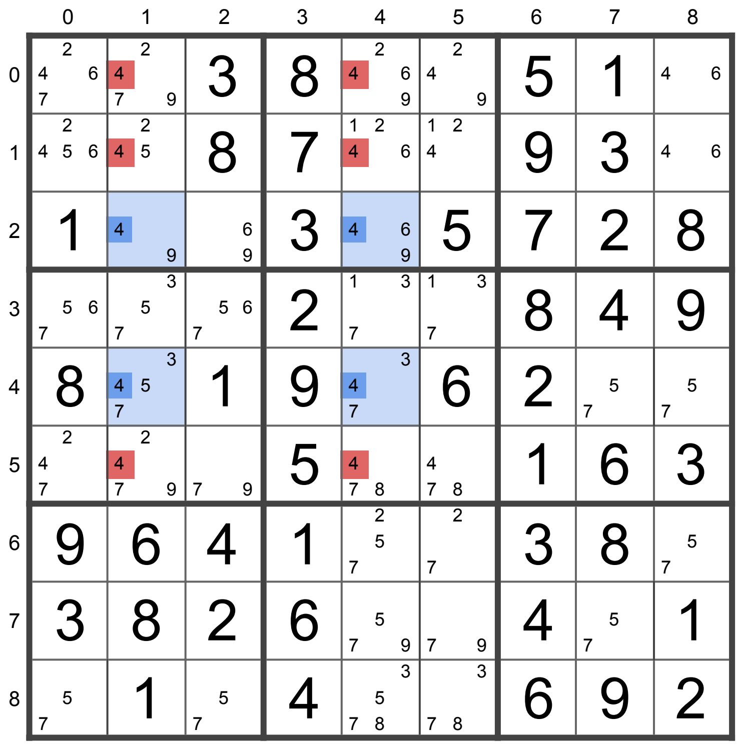


Figure 3.10: Example of an orthogonal pair.

We are going to generalize the technique used for the pair of rows and columns in the previous example to m rows and columns for $m \in \{2, \ldots, n\}$. Since this technique focuses only on rows and columns, we are going to formalize the relationship between rows and columns with the concept of orthogonality.

Definition 3.3.2. Two brackets $\mathbb{S}_1, \mathbb{S}_2$ of a grid $\mathbb{G}$ are defined as ***orthogonal*** if $|\mathbb{S}_1 \cap \mathbb{S}_2| = 1$. And we define an ***orthogonal intersection*** as the only element $cell = \mathbb{S}_1 \perp \mathbb{S}_2 \in \mathbb{S}_1 \cap \mathbb{S}_2$.

It is clear that a pair of brackets is only orthogonal if one of them is a row and the other is a column. The next definition introduces a function that, for a specific bracket and value, returns all orthogonal brackets that contain the value in their available set. This function will be useful for the definition of the new type of subset that is the protagonist of this stage.

Definition 3.3.3. For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and a given bracket $\mathbb{S}$, and value $\omega \in \Omega$. we define the ***orthogonal intersection function*** $\Lambda_k \colon \mathcal{P}(\mathbb{G}) \times \Omega \to \mathcal{P}(\mathbb{G})$ as follows:

$$\Lambda_k(\mathbb{S}, \omega) = \{\mathbb{T} \colon \omega \in \mathbb{AS}_k(cell) \text{ and } |\mathbb{S} \cap \mathbb{T}| = 1 \text{ with } cell = \mathbb{S} \perp \mathbb{T}\},$$

where every $\mathbb{T}$ is a bracket orthogonal to $\mathbb{S}$.

It should be clear that if $\mathbb{S}$ is a row, then $\Lambda_k(\mathbb{S}, \omega)$ is a set of columns and if $\mathbb{S}$ is a column, then $\Lambda_k(\mathbb{S}, \omega)$ is a set of rows. Let us illustrate the orthogonal intersection function using the setting of the last example.

Example 3.3.4. For the $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ in Figure 3.10, we have the following equations:

$$\Lambda(\mathbb{R}_2, 4) = \{\mathbb{C}_1, \mathbb{C}_4\} \text{ and } \Lambda(\mathbb{R}_4, 4) = \{\mathbb{C}_1, \mathbb{C}_4\}$$

Finally, we are going to formalize and generalize the concept illustrated in Example 3.3.1 by observing that $\omega = 4$ appears only twice in rows 2 and 4, and additionally these appearances are in the same two columns 1 and 4. Note that we cannot start the arguments with columns 1 and 4 because there are more than two rows that contain the candidate 4. We will see in the next definition, that we refer to this case as a row-sourced orthogonal subset $(\{(\mathbb{R}_2, \mathbb{R}_4)\}, \{\mathbb{C}_1, \mathbb{C}_4\}, 4)$.

Definition 3.3.5. For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and $m \leq n$ we find that $(\mathfrak{R} = \{\mathbb{R}_{i_1}, \ldots, \mathbb{R}_{i_m}\}, \mathfrak{C} = \{\mathbb{C}_{j_1}, \ldots, \mathbb{C}_{j_m}\}, \omega)$ is an ***orthogonal subset*** if one of the following conditions is met:

- If for every $\mathbb{R} \in \mathfrak{R}$, we get that $\Lambda_k(\mathbb{R}, \omega) \subseteq \mathfrak{C}$, then the subset is ***row-sourced***.
- If for every $\mathbb{C} \in \mathfrak{C}$, we get that $\Lambda_k(\mathbb{C}, \omega) \subseteq \mathfrak{R}$, then the subset is ***column-sourced***.

The set of brackets that is not the source is called ***orthogonal target***.

Let us show a couple of examples, one for a row-sourced orthogonal subset, and one for column-sourced one.

Example 3.3.6. Let $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ be the setting in Figure 3.11. We have that $(\mathfrak{R} = \{\mathbb{R}_1, \mathbb{R}_2, \mathbb{R}_3\}, \mathfrak{C} = \{\mathbb{C}_0, \mathbb{C}_4, \mathbb{C}_6\}, 8)$ is a column-sourced orthogonal subset.

Clearly we have that $|\mathfrak{R}| = |\mathfrak{C}| = 3$, so $m = 3$. Now, let us show it is indeed column-sourced:

$$\Lambda_k(\mathbb{C}_0, 8) = \{\mathbb{R}_1, \mathbb{R}_2, \mathbb{R}_3\} \subseteq \mathfrak{R},$$
$$\Lambda_k(\mathbb{C}_4, 8) = \{\mathbb{R}_1, \mathbb{R}_2, \mathbb{R}_3\} \subseteq \mathfrak{R},$$
$$\Lambda_k(\mathbb{C}_6, 8) = \{\mathbb{R}_1, \mathbb{R}_2, \mathbb{R}_3\} \subseteq \mathfrak{R}.$$

Therefore, we can conclude that $(\mathfrak{R} = \{\mathbb{R}_1, \mathbb{R}_2, \mathbb{R}_3\}, \mathfrak{C} = \{\mathbb{C}_0, \mathbb{C}_4, \mathbb{C}_6\}, 8)$ is a column-sourced orthogonal subset. It is not row-sourced, because in at least one row there are more than 3 columns containing the candidate 8.

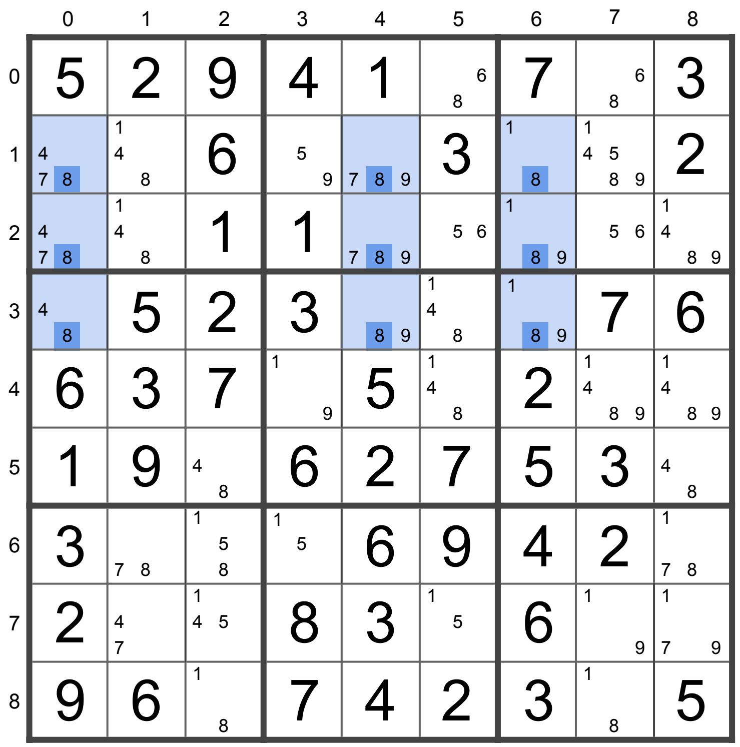


Figure 3.11: Example of a column-sourced orthogonal subset.

Example 3.3.7. Let $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ be the setting in Figure 3.12. We see that $(\mathfrak{R} = \{\mathbb{R}_0, \mathbb{R}_3, \mathbb{R}_5, \mathbb{R}_8\}, \mathfrak{C} = \{\mathbb{C}_0, \mathbb{C}_4, \mathbb{C}_7, \mathbb{C}_8\}, 2)$ is a row-sourced orthogonal subset.

Clearly we have that $|\mathfrak{R}| = |\mathfrak{C}| = 4$, so $m = 4$. Now, let us show that it is indeed row-sourced:

$$\Lambda_k(\mathbb{R}_0, 2) = \{\mathbb{C}_0, \mathbb{C}_7, \mathbb{C}_8\} \subseteq \mathfrak{C}\,,$$
$$\Lambda_k(\mathbb{R}_3, 2) = \{\mathbb{C}_4, \mathbb{C}_8\} \subseteq \mathfrak{C}\,,$$
$$\Lambda_k(\mathbb{R}_5, 2) = \{\mathbb{C}_4, \mathbb{C}_7, \mathbb{C}_8\} \subseteq \mathfrak{C}\,,$$
$$\Lambda_k(\mathbb{R}_8, 2) = \{\mathbb{C}_0, \mathbb{C}_7, \mathbb{C}_8\} \subseteq \mathfrak{C}\,.$$

Therefore, we can conclude that $(\mathfrak{R}, \mathfrak{C}, 2)$ is a row-sourced orthogonal subset.

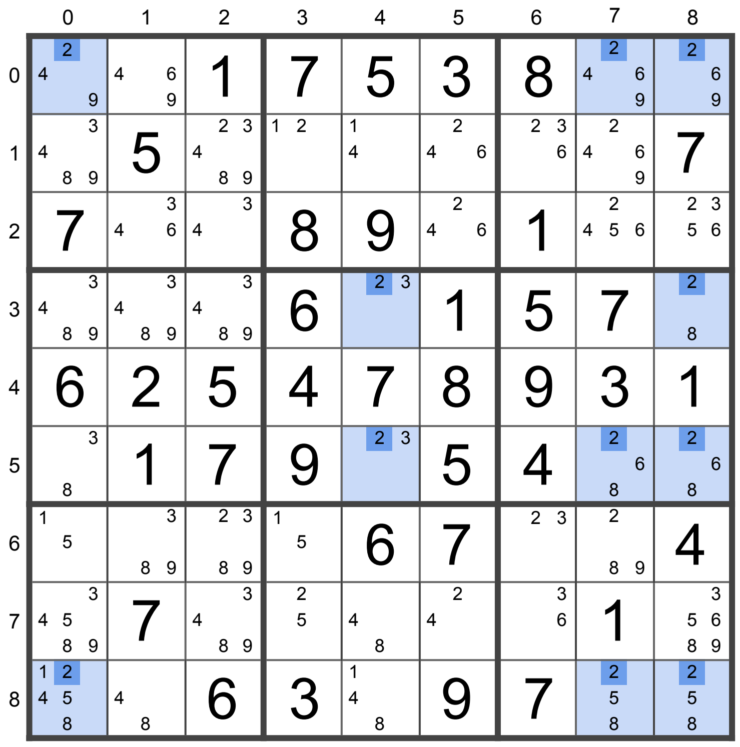


Figure 3.12: Example of row-sourced orthogonal subset.

Now, the pruning that takes place when an orthogonal subset is found is described below. The formal definition of this pruning will be done in the sudoku theory section, as was done for every stage.

For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$ and an orthogonal subset $(\mathfrak{R}, \mathfrak{C}, \omega)$, ***orthogonal pruning*** consists of deleting ω from the available sets of all the cells in brackets in the target, except the ones that belong to a bracket in the source.

Note that a subset can be row-sourced and column-sourced at the same time. If that is the case, then the orthogonal pruning will do nothing since we will have the properties: For every $\mathbb{R} \in \mathfrak{R}$, $\Lambda_k(\mathbb{R}) \subseteq \mathfrak{C}$ and for every $\mathbb{C} \in \mathfrak{C}$, $\Lambda(\mathbb{C}) \subseteq \mathfrak{R}$. Therefore, there are no cells with candidate ω to be pruned.

Let us show an example of the way the orthogonal subset pruning is applied. We will use the same settings as we showed in Example 3.3.6 and 3.3.7.

Example 3.3.8. For the setting in Figure 3.13 and $(\mathfrak{R} = \{\mathbb{R}_1, \mathbb{R}_2, \mathbb{R}_3\}, \mathfrak{C} = \{\mathbb{C}_0, \mathbb{C}_4, \mathbb{C}_6\}, 8)$ a column-sourced orthogonal subset. The value 8 should be pruned from the following *cells*:

$$\begin{aligned} cell &\in \mathbb{R}_1 \setminus (\mathbb{C}_0 \cup \mathbb{C}_4 \cup \mathbb{C}_6), \\ cell &\in \mathbb{R}_2 \setminus (\mathbb{C}_0 \cup \mathbb{C}_4 \cup \mathbb{C}_6), \\ cell &\in \mathbb{R}_3 \setminus (\mathbb{C}_0 \cup \mathbb{C}_4 \cup \mathbb{C}_6). \end{aligned}$$

In this specific case the cells in those sets that have 8 in their available sets are the following: $cell_{11}, cell_{21}, cell_{35}, cell_{17}, cell_{28}$. Therefore, those are the cells that are pruned.

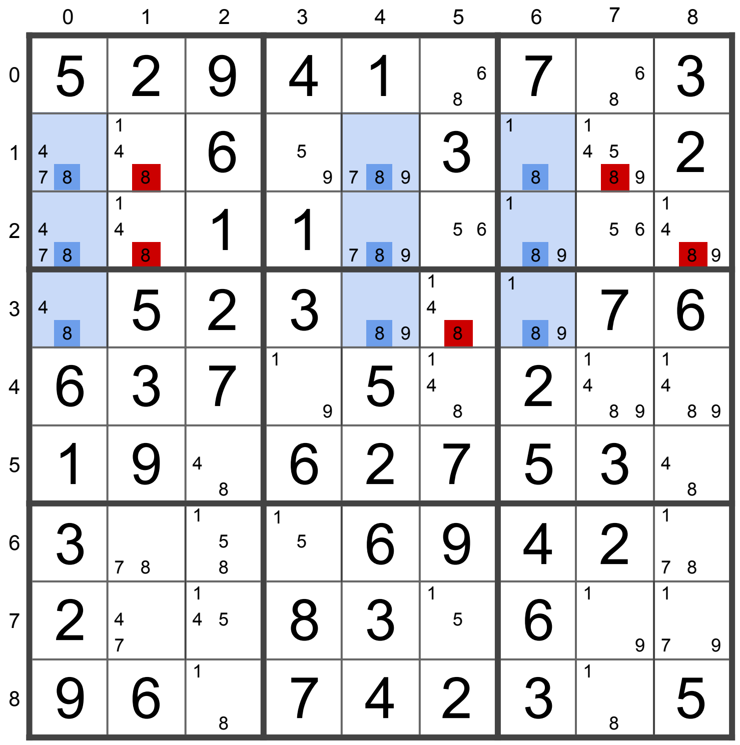


Figure 3.13: Example of a column-sourced orthogonal prune.

Example 3.3.9. Given the setting in Figure 3.14, and the row-sourced orthogonal subset $(\mathfrak{R} = \{\mathbb{R}_0, \mathbb{R}_3, \mathbb{R}_5, \mathbb{R}_8\}, \mathfrak{C} = \{\mathbb{C}_0, \mathbb{C}_4, \mathbb{C}_7, \mathbb{C}_8\}, 2)$, the value 2 can be pruned from the following *cells*:

$$cell \in (\mathbb{C}_0 \cup \mathbb{C}_4 \cup \mathbb{C}_7 \cup \mathbb{C}_8) \setminus (\mathbb{R}_0 \cup \mathbb{R}_4 \cup \mathbb{R}_6 \cup \mathbb{R}_8).$$

In this specific case, the cells in these sets that have 2 in their available sets are the following: $cell_{17}, cell_{27}, cell_{28}, cell_{67}$. Therefore, these are the cells that are pruned.

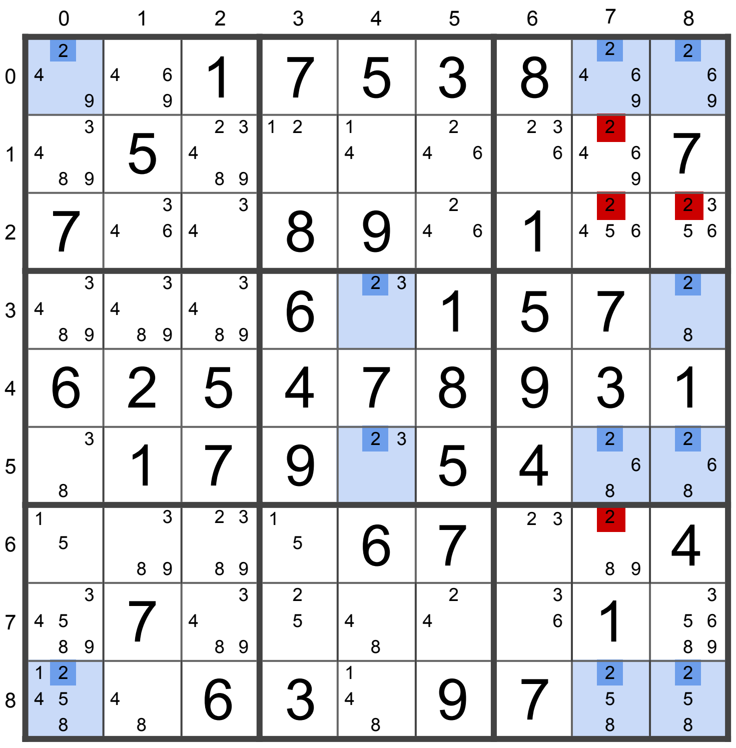


Figure 3.14: Example of a row-sourced orthogonal prune.

3.3.1 Sudoku Theory

First, we provide an important property of the orthogonal intersection function that will be useful in revealing a complementary relationship between row-sourced and column-sourced orthogonal subsets.

Lemma 3.3.10. *For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, the orthogonal intersection function is commutative, that is, for any brackets $\mathbb{S}_1, \mathbb{S}_2$ the following property holds:*

$$\mathbb{S}_2 \in \Lambda_k(\mathbb{S}_1, \omega) \text{ if and only if } \mathbb{S}_1 \in \Lambda_k(\mathbb{S}_2, \omega).$$

Proof. Since $\mathbb{S}_1 \cap \mathbb{S}_2 = \mathbb{S}_2 \cap \mathbb{S}_1$, we then have $\mathbb{S}_1 \perp \mathbb{S}_2 = \mathbb{S}_2 \perp \mathbb{S}_2$, and $|\mathbb{S}_1 \cap \mathbb{S}_2| = 1$ if and only if $|\mathbb{S}_2 \cap \mathbb{S}_1| = 1$.
Now, $\mathbb{S}_2 \in \Lambda_k(\mathbb{S}_1, \omega)$ if and only if $\omega \in \mathbb{AS}_k(\mathbb{S}_2 \perp \mathbb{S}_1)$, if and only if $\omega \in \mathbb{AS}_k(\mathbb{S}_2 \perp \mathbb{S}_1)$, if and only if $\mathbb{S}_1 \in \Lambda_k(\mathbb{S}_2, \omega)$. □

To this end, we are going to state a theorem that establishes a complementarity relationship between row and column-sourced subsets, just like we did for naked and hidden subsets. We will prove that $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset if and only if the remaining rows and columns form a column-sourced orthogonal subset for the same value ω.

Theorem 3.3.11. *For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, let R be the set of all rows and C be the set of all columns of $\mathbb{G}$. Then $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset if and only if $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is a column-sourced orthogonal subset.*

These subsets are called **complementary subsets**.

Proof. First, we assume that $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset, and we are going to prove that $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is a column-sourced orthogonal subset. Since $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset, we can define m in the following way:

$$m = |\mathfrak{R}| = |\mathfrak{C}|. \tag{3.17}$$

We need to check two conditions:

C1, $|R \setminus \mathfrak{R}| = |C \setminus \mathfrak{C}| \leq n$:
We know that any grid has n rows and n columns, therefore $|R| = n$ and $|C| = n$. We now have $|R| = |C|$, which implies $|R| - m = |C| - m$. Using (3.17) we get $|R| - |\mathfrak{R}| = |C| - |\mathfrak{C}|$. Finally, since $\mathfrak{R} \subseteq R$ and $\mathfrak{C} \subseteq C$ we can conclude that $|R \setminus \mathfrak{R}| = |C \setminus \mathfrak{C}| \leq n$.

C2, $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is column-sourced:
Let us suppose that $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is not column-sourced. Then, there exist a row $\mathbb{R}'$ and a column $\mathbb{C}' \in C \setminus \mathfrak{C}$ such that $\mathbb{R}' \in \Lambda(\mathbb{C}', \omega)$ and $\mathbb{R}' \notin (R \setminus \mathfrak{R})$. Since R is the set of all rows, we have $\mathbb{R}' \in R$ and $\mathbb{R}' \notin R \setminus \mathfrak{R}$, therefore $R' \in \mathfrak{R}$.

Now, since we have $\mathbb{R}' \in \Lambda(\mathbb{C}', \omega)$, applying the Lemma 3.3.10 of commutativity, page 105, we get $\mathbb{C}' \in \Lambda(\mathbb{R}', \omega)$.

But since $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal, $\mathbb{C}' \in \Lambda(\mathbb{R}', \omega)$ implies $\mathbb{C}' \in \mathfrak{C}$, which is a contradiction to the hypothesis $\mathbb{C}' \in C \setminus \mathfrak{C}$. Therefore, $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is a column-sourced orthogonal subset.

Now, we assume that $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is a column-sourced orthogonal subset and prove that $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset. Since $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is an orthogonal subset, we can define m as follows:

$$m = n - |R \setminus \mathfrak{R}| = n - |C \setminus \mathfrak{C}|,$$

and it is clear that $0 \leq m \leq n$. Again, we check two conditions:

C1, We have $|R \setminus \mathfrak{R}| = |C \setminus \mathfrak{C}|$. Now, since $\mathfrak{R} \subseteq R$ and $\mathfrak{C} \subseteq C$, we can conclude that $|R| - |\mathfrak{R}| = |C| - |\mathfrak{C}|$. Subsequently, we know that any grid has n rows and n columns; therefore $|R| = n$ and $|C| = n$. Finally, since $m = n - |R \setminus \mathfrak{R}| = n - |C \setminus \mathfrak{C}|$, we have $m = n - (n - |\mathfrak{R}|) = n - (n - |\mathfrak{C}|)$, so we can conclude $m = |\mathfrak{R}| = |\mathfrak{C}|$.

C2, $(\mathfrak{R}, \mathfrak{C}, \omega)$ is row-sourced:
Let us suppose that $(\mathfrak{R}, \mathfrak{C}, \omega)$ is not row-sourced. Then, there exist a column $\mathbb{C}'$ and a row $\mathbb{R}' \in \mathfrak{R}$ such that $\mathbb{C}' \in \Lambda(\mathbb{R}', \omega)$ and $\mathbb{C}' \notin \mathfrak{C}$. Since C is the set of all columns, we have $\mathbb{C}' \in C$ and $\mathbb{C}' \in C \setminus \mathfrak{C}$.

Now, since we have $\mathbb{C}' \in \Lambda_k(\mathbb{R}', \omega)$, applying Lemma 3.3.10, page 105, we get $\mathbb{R}' \in \Lambda_k(\mathbb{C}', \omega)$.

Since $(R \setminus \mathfrak{R}, C \setminus \mathfrak{C}, \omega)$ is a column-sourced orthogonal subset, and $\mathbb{R}' \in \Lambda_k(\mathbb{C}', \omega)$ for $\mathbb{C}' \in C \setminus \mathfrak{C}$. This implies $\mathbb{R}' \in R \setminus \mathfrak{R}$, which is a contradiction to hypothesis $\mathbb{R}' \in \mathfrak{R}$. Therefore, $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset.

□

Next, we formalize the process of orthogonal pruning. We define it for both cases, row-sourced and column-sourced, and then we prove that no solution is lost.

Given a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ and an orthogonal subset $(\mathfrak{R}, \mathfrak{C}, \omega)$, we define the following sets of cells: $\mathbb{RR} = \bigcup_{\mathbb{R} \in \mathfrak{R}} \mathbb{R}$, and $\mathbb{CC} = \bigcup_{\mathbb{C} \in \mathfrak{C}} \mathbb{C}$.

Assuming $(\mathfrak{R}, \mathfrak{C}, \omega)$ is row-sourced, we define the function that performs the orthogonal prune as follows:

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) &= f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega) \\ &= (\mathbb{F}_k, \gamma_k, f^3_{R5}(\mathrm{AS}_k, (\mathfrak{R}, \mathfrak{C}, \omega)))\end{aligned}$$

where for every *cell* $\in \mathbb{G}$,

$$\begin{aligned}\mathrm{AS}_{k+1}(cell) &= f^3_{R5}(\mathrm{AS}_k, (\mathfrak{R}, \mathfrak{C}, \omega))\big|_{cell} \\ &= \begin{cases} \mathrm{AS}_k(cell) & \text{if } cell \in \mathbb{G} \setminus \mathbb{CC} \\ \mathrm{AS}_k(cell) & \text{if } cell \in \mathbb{CC} \cap \mathbb{RR} \\ \mathrm{AS}_k(cell) \setminus \{\omega\} & \text{if } cell \in \mathbb{CC} \setminus \mathbb{RR} \end{cases}\end{aligned}$$

We now state a lemma that will be used in the proof that shows no solution is lost when applying a row-sourced orthogonal prune.

Lemma 3.3.12. *For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, a row-sourced orthogonal subset $(\mathfrak{R}, \mathfrak{C}, \omega)$, and a cell* such that $\gamma_k(cell^*) = \omega$.*

Let , then:

$$\mathbb{C}^* \in \mathfrak{C} \implies \mathbb{R}^* \in \mathfrak{R}.$$

Where $\mathbb{R}^*$ is the row of $cell^*$ and $\mathbb{C}^*$ its column

Proof. Since $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ is a setting, there exists a solution γ, such that $\gamma_k \preceq \gamma$. Let $\mathfrak{R} = \{\mathbb{R}_{i_1}, \dots, \mathbb{R}_{i_m}\}$, then, since each row is a bracket, each row in $\mathfrak{R}$ should have a *cell* such that $\gamma(cell) = \omega$. So there exist $j_1, \dots, j_m$ such that

$$\gamma(cell_{i_1 j_1}) = \omega, \quad \dots, \quad \gamma(cell_{i_m, j_m}) = \omega.$$

Since γ is a solution, it is clear that:

$$j_1, j_2, \dots j_m \text{ are all different,} \tag{3.18}$$

otherwise, two different cells in the same column have the same value ω. Furthermore, since AS_k is a valid available set function, we obtain $\gamma(cell_{i_1 j_1}) \in \mathrm{AS}_k(cell_{i_1 j_1}), \dots, \gamma(cell_{i_m j_m}) \in \mathrm{AS}_k(cell_{i_m j_m})$, or equivalently

$$\omega \in \mathrm{AS}_k(cell_{i_1 j_1}), \quad \dots, \quad \omega \in \mathrm{AS}_k(cell_{i_m, j_m}). \tag{3.19}$$

Now, since $(\mathfrak{R}, \mathfrak{C}, \omega)$ is row-sourced we have that $\Lambda_k(\mathbb{R}, \omega) \subseteq \mathfrak{C}$ for every $\mathbb{R} \in \mathfrak{R}$. Therefore, using (3.19) we have that $\mathbb{C}_{j_1} \in \Lambda_k(\mathbb{R}_{i_1}, \omega) \subseteq \mathfrak{C}, \dots, \mathbb{C}_{j_m} \in \Lambda_k(\mathbb{R}_{i_m}, \omega) \subseteq \mathfrak{C}$, this implies, $\{\mathbb{C}_{j_1}, \dots \mathbb{C}_{j_m}\} \subseteq \mathfrak{C}$. Using (3.18) and that $|\mathfrak{C}| = m$, we get:

$$\{\mathbb{C}_{j_1}, \dots \mathbb{C}_{j_m}\} = \mathfrak{C}. \tag{3.20}$$

Because, by definition of orthogonal subset we know that $|\mathfrak{C}| = m$. Finally, by hypothesis $\mathbb{C}^* \in \mathfrak{C}$, and without loss of generality $\mathbb{C}^* = \mathbb{C}_{j_1}$. Also, since there can only be one cell in each column with a value ω, and $\gamma(cell^*) = \gamma(cell_{i_1 j_1})$, we get that $cell^* = cell_{i_1 j_1}$. That is, $\mathbb{R}^* = \mathbb{R}_{i_1}$, therefore, $\mathbb{R}^* \in \mathfrak{R}$. □

Now, we prove that no solution is lost when applying a row-sourced orthogonal prune. We will prove this by showing that the result of doing an orthogonal prune to a setting is a setting. Of course, for the proof of the following theorem we will take advantage of Lemma 3.3.12.

Theorem 3.3.13. *For a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, and a row-sourced orthogonal subset* $(\mathfrak{R}, \mathfrak{C}, \omega)$*, we have that:*

$$(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) = f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega) \text{ is a setting}$$

Proof. Since $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ is a setting and $\mathbb{F}_{k+1} = \mathbb{F}_k$ and $\gamma_{k+1} = \gamma_k$ it is clear that $\mathbb{F}_0 \subseteq \mathbb{F}_{k+1}$ and $\gamma_{k+1} \preceq \gamma_0$. Therefore, we only have to prove that AS_{k+1} is a valid available set function. As usual, let us suppose it is not, we will try to reach a contradiction. Then, there exists a $cell' \in \mathbb{G}$ and a solution γ such that $\gamma(cell') \notin \mathrm{AS}_{k+1}(cell')$. Since AS_k is a valid available set function, we have that $\gamma(cell') \in \mathrm{AS}_k(cell')$, therefore we can conclude

$$\gamma(cell') \in \mathrm{AS}_k(cell') \setminus \mathrm{AS}_{k+1}(cell') \tag{3.21}$$

Case 1, $cell' \in \mathbb{G} \setminus \mathbb{CC}$ or $cell' \in \mathbb{CC} \cap \mathbb{RR}$:
Then, by definition of f^3_{R5}, we know that $\mathrm{AS}_{k+1}(cell') = \mathrm{AS}_k(cell')$, then applying (3.21) we have that $\gamma(cell') \in \emptyset$, which is a contradiction.

Case 2, $cell' \in \mathbb{CC} \setminus \mathbb{RR}$:
By definition f^3_{R5}, and applying (3.21) we have that $\gamma(cell') \in \mathrm{AS}_k \setminus (\mathrm{AS}_k \setminus \{\omega\})$. If $\omega \notin \mathrm{AS}_k(cell')$, this means that $\gamma(cell') \in \emptyset$ which is a contradiction, then, assuming $\omega \in \mathrm{AS}_k(cell')$ we get:

$$\gamma(cell') = \omega. \tag{3.22}$$

Let $\mathbb{R}'$ be the row of $cell'$ and $\mathbb{C}'$ its column. By hypothesis of Case 2, $cell' \in \mathbb{CC}$ which implies that $\mathbb{C}' \in \mathfrak{C}$, then we can apply Lemma

3.3.12, to *cell*$'$. We then have that $\mathbb{R}' \in \mathfrak{R}$, that is *cell*$' \in \mathbb{RR}$ which is a contradiction to the Case 2.

In conclusion, by *reducto ad absurdum*, AS_{k+1} is a valid available set function, and $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$ is a setting. □

Now, let us formally define the column-sourced orthogonal prune. Suppose that $(\mathfrak{R}', \mathfrak{C}', \omega)$ is column-sourced. We define the function that performs the column orthogonal prune as follows: Let $\mathbb{RR}' = \bigcup_{\mathbb{R} \in \mathfrak{R}'} \mathbb{R}$, and $\mathbb{CC}' = \bigcup_{\mathbb{C} \in \mathfrak{C}'} \mathbb{C}$.

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) = f_{C5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}', \mathfrak{C}', \omega) \\ = (\mathbb{F}_k, \gamma_k, f^3_{C5}(\mathrm{AS}_k, (\mathfrak{R}', \mathfrak{C}', \omega)))\end{aligned}$$

where, for every *cell* $\in \mathbb{G}$,

$$\begin{aligned}\mathrm{AS}_{k+1}(\mathit{cell}) &= f^3_{C5}(\mathrm{AS}_k, (\mathfrak{R}', \mathfrak{C}', \omega))\big|_{\mathit{cell}} \\ &= \begin{cases} \mathrm{AS}_k(\mathit{cell}) & \text{if } \mathit{cell} \in \mathbb{G} \setminus \mathbb{RR}' \\ \mathrm{AS}_k(\mathit{cell}) & \text{if } \mathit{cell} \in \mathbb{RR}' \cap \mathbb{CC}' \\ \mathrm{AS}_k(\mathit{cell}) \setminus \{\omega\} & \text{if } \mathit{cell} \in \mathbb{RR}' \setminus \mathbb{CC}' \end{cases}\end{aligned}$$

Now, we are going to prove that the prune done by a column-sourced orthogonal subset is the same as the one done by its complimentary row-sourced orthogonal subset, see Theorem 3.3.11. This will allow us to eventually prove that a column-sourced orthogonal prune does not loose solutions.

Theorem 3.3.14. *For a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$. *Let* $(\mathfrak{R}, \mathfrak{C}, \omega)$ *be a row-sourced orthogonal subset, and* $(\mathfrak{R}', \mathfrak{C}', \omega)$ *be its complementary column-sourced orthogonal subset. Then,*

$$f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega) = f_{C5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}', \mathfrak{C}', \omega).$$

Proof. Let $(\mathbb{F}_R, \gamma_R, \mathrm{AS}_R) = f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega)$, and $(\mathbb{F}_C, \gamma_C, \mathrm{AS}_C) = f_{C5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}', \mathfrak{C}', \omega)$. We have $\mathbb{F}_R = \mathbb{F}_k$ and $\mathbb{F}_C = \mathbb{F}_k$, then $\mathbb{F}_R = \mathbb{F}_C$. Also, $\gamma_R = \gamma_k$ and $\gamma_C = \gamma_k$, so $\gamma_R = \gamma_C$. The only thing left to prove is that $\mathrm{AS}_R = \mathrm{AS}_C$.

Since $(\mathfrak{R}, \mathfrak{C}, \omega)$ and $(\mathfrak{R}', \mathfrak{C}', \omega)$ are complementary we have that $R = \mathfrak{R} \cup \mathfrak{R}'$ (all rows) and $\mathfrak{R} \cap \mathfrak{R}' = \emptyset$. Similarly, we have $C = \mathfrak{C} \cup \mathfrak{C}'$ (all columns) and $\mathfrak{C} \cap \mathfrak{C}' = \emptyset$.

Now, for rows: $\mathbb{RR} = \bigcup_{\mathbb{R} \in \mathfrak{R}} \mathbb{R}$ and $\mathbb{RR}' = \bigcup_{\mathbb{R} \in \mathfrak{R}'} \mathbb{R}$, and for columns: $\mathbb{CC} = \bigcup_{\mathbb{C} \in \mathfrak{C}} \mathbb{C}$, and $\mathbb{CC}' = \bigcup_{\mathbb{C} \in \mathfrak{C}'} \mathbb{C}$, we can conclude the following two set equalities:

$$\mathbb{G} = \mathbb{RR} \cup \mathbb{RR}' \text{ and } \mathbb{RR} \cap \mathbb{RR}' = \emptyset, \tag{3.23}$$

$$\mathbb{G} = \mathbb{CC} \cup \mathbb{CC}' \text{ and } \mathbb{CC} \cap \mathbb{CC}' = \emptyset. \tag{3.24}$$

Next, we prove that $\mathrm{AS}_R = \mathrm{AS}_C$ by checking that $\mathrm{AS}_R(\mathit{cell}) = \mathrm{AS}_C(\mathit{cell})$ for every $\mathit{cell} \in \mathbb{G}$. Please refer to the definitions of the row-sourced orthogonal prune f_{R5} on page 108, and column-sourced orthogonal prune f_{C5} on page 111.

We can observe that in the row-sourced prune the only cells that have its available set affected are the ones in $\mathbb{CC} \setminus \mathbb{RR}$, meanwhile the affected cells in the column-sourced prune are the ones in $\mathbb{RR}' \setminus \mathbb{CC}'$. And these set of *cells* are affected in the same way, by removing $\{\omega\}$, therefore, we just have to prove that $\mathbb{CC} \setminus \mathbb{RR} = \mathbb{RR}' \setminus \mathbb{CC}'$. Observe that $\mathbb{CC} \setminus \mathbb{RR}$ is equal to $(\mathbb{G} \setminus \mathbb{CC}') \setminus \mathbb{RR}$ by (3.24), which is equal to $(\mathbb{G} \setminus \mathbb{CC}') \setminus (\mathbb{G} \setminus \mathbb{RR}')$. Now, since $\mathbb{RR}' \subseteq \mathbb{G}$, we have that $(\mathbb{G} \setminus \mathbb{CC}') \setminus (\mathbb{G} \setminus \mathbb{RR}') = (\mathbb{G} \setminus \mathbb{CC}') \cap \mathbb{RR}'$ which is equal to $(\mathbb{G} \cap \mathbb{RR}') \setminus \mathbb{CC}'$, which of course is equal to $\mathbb{RR}' \setminus \mathbb{CC}'$.

Therefore, we can conclude that

$$f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega) = f_{C5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}', \mathfrak{C}', \omega)$$

because they remove ω from the available sets of the same *cells* and leave the other *cells* unaffected. $\square$

Now, we prove that the column-source orthogonal prune does not loose solutions, again we prove this by showing that when a column-source orthogonal prune is applied to a setting, the result is a setting, as well.

Corollary 3.3.15. *Given a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, and a column-sourced orthogonal subset* $(\mathfrak{R}', \mathfrak{C}', \omega)$*, the result of* f_{C5} *is a setting.*

Proof. By Theorem 3.3.11, page 106 for $(\mathfrak{R}', \mathfrak{C}', \omega)$ there exists a row-sourced complementary orthogonal subset $(\mathfrak{R}, \mathfrak{C}, \omega)$. Then, by Theorem 3.3.14, the pruning of the complementary subsets is the same, that is $f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega) = f_{C5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}', \mathfrak{C}', \omega)$. Finally, by Theorem 3.3.13, 110 we have that $f_{R5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}, \mathfrak{C}, \omega)$ is a setting. So, we can conclude that $f_{C5}((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), \mathfrak{R}', \mathfrak{C}', \omega)$ is a setting. $\square$

3.3.2 Implementation

For implementing this stage we use the same strategy that we used for stage four and prove that an orthogonal subset is a subcover of a specific type of cover. Then, we define the covers and look for subcovers. Again, we divide the implementation into theory and code.

Theory

We recall the Definition 3.3.3 of the orthogonal intersection function, page 99. For a fixed $\omega \in \Omega$ we denote $\Lambda_k(\mathbb{S}, \omega)$ by $\Lambda_k^{\omega}(\mathbb{S})$ where $\mathbb{S}$ is a bracket.

Before we show that orthogonal subsets are subcovers we will use the same concept we used in stage four for restricting functions.

We recall the Definition 3.2.12 of a set cover, page 84. Again, we use the symbols $R = \{\mathbb{R}_0, \ldots, \mathbb{R}_{n-1}\}$ and $C = \{\mathbb{C}_0, \ldots, \mathbb{C}_{n-1}\}$ to denote all rows and columns, respectively. For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ and a fixed a value ω, we have that for any row $\mathbb{R}$, $\Lambda_k^\omega(\mathbb{R}) \subseteq C$, and similarly for any column $\mathbb{C}$, $\Lambda_k^\omega(\mathbb{C}) \subseteq R$. Then, we can conclude the following lemma.

Lemma 3.3.16. *For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$, and a value ω, we have that $(R, C, \Lambda^\omega_{k(R)})$ and $(C, R, \Lambda^\omega_{k(C)})$ are covers.*

Now, we prove that an orthogonal subset is a subcover of the covers mentioned above.

Theorem 3.3.17. *For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we have that $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset if and only if $(\mathfrak{R}, \mathfrak{C}, \Lambda^\omega_{k(\mathfrak{R})})$ is a subcover of $(R, C, \Lambda^\omega_{k(R)})$.*

Proof. By Corollary 3.2.14 we know that $\Lambda^\omega_{k(\mathfrak{R})} = \Lambda^\omega_{k(R)(\mathfrak{R})}$ so the subcover function matches with the one in the definition of subcovers. First, we assume $(\mathfrak{R}, \mathfrak{C}, \omega)$ is a row-sourced orthogonal subset, and we prove that $(\mathfrak{R}, \mathfrak{C}, \Lambda^\omega_{k(\mathfrak{R})})$ is a subcover of $(R, C, \Lambda^\omega_{k(R)})$. By Definition 3.2.16 of subcovers, page 85, we have to verify two conditions:

C1, $\mathfrak{R} \subseteq R$, $\mathfrak{C} \subseteq C$:
By definition $\mathfrak{R}$ is made up of rows, and $\mathfrak{C}$ is made up of columns, and we have $\mathfrak{R} \subseteq R$, $\mathfrak{C} \subseteq C$.

C2, $(\mathfrak{R}, \mathfrak{C}, \Lambda^\omega_{k(\mathfrak{R})})$ is a cover:
By definition of an orthogonal subset $|\mathfrak{R}| = |\mathfrak{C}|$ and since it is row-sourced we get $\Lambda^\omega_{k(\mathfrak{R})}(\mathbb{R}) \subseteq \mathfrak{C}$ for every $\mathbb{R} \in \mathfrak{R}$, which proves the second condition and concludes the first part.

Now, we assume that $(\mathfrak{R}, \mathfrak{C}, \Lambda^\omega_{k(\mathfrak{R})})$ is a subcover of $(R, C, \Lambda^\omega_{k(R)})$ and we prove that $(\mathfrak{R}, \mathfrak{C}, \omega)$ is an orthogonal subset:

C1, $|\mathfrak{R}| = |\mathfrak{C}|$:

By definition, every subcover is also a cover, then, we have that $|\mathfrak{R}| = |\mathfrak{C}|$, and by definition of subcovers, $\mathfrak{R} \subseteq R$ and $\mathfrak{C} \subseteq C$.

C2, $(\mathfrak{R}, \mathfrak{C}, \omega)$ is row-sourced:

We prove that it $(\mathfrak{R}, \mathfrak{C}, \omega)$ is row-sourced, that is, for all $\mathbb{R} \in \mathfrak{R}$ $\Lambda^{\omega}_{k(R)}(\mathbb{R}) \subseteq \mathfrak{C}$. This holds because that is the exact property of the second condition of the Definition 3.2.12 of covers, page 84.

□

Corollary 3.3.18. *Given a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*, the tuple* $(\mathfrak{R}, \mathfrak{C}, \omega)$ *is a column-sourced orthogonal subset if and only if* $(\mathfrak{C}, \mathfrak{R}, \Lambda^{\omega}_{k(\mathfrak{C})})$ *is a subcover of* $(C, R, \Lambda^{\omega}_{k(C)})$.

The proof of Theorem 3.3.17 gives this result.

Now, the concept of isolation Definition 3.2.21, will be useful for orthogonal subsets as well. For instance, we have that a subcover of $(R, C, \Lambda^{\omega}_{k(R)})$ that represents a row-sourced orthogonal subset $(\mathfrak{R}, \mathfrak{C}, \Lambda^{\omega}_{k(\mathfrak{R})})$ is isolated, if $(\mathfrak{C}, \mathfrak{R}, \Lambda^{\omega}_{k(\mathfrak{C})})$ is a subcover of $(C, R, \Lambda^{\omega}_{k(C)})$, which implies we do not have any cells to prune. This happens when ω is not in the available set of any of the cells that could be pruned. We can conclude that if the subcover that represents an orthogonal subset is isolated, then the orthogonal subset is row-sourced and column-sourced at the same time, therefore, there are no cells to be pruned.

Code

First of all, we add the new covers $(R, C, \Lambda^{\omega}_{k(R)})$ and $(C, R, \Lambda^{\omega}_{k(C)})$ as `Cover` instances as attributes to the `Bracket` class, for each $\omega \in \Omega$ and for each row and column. Then, we will take advantage of the method `find_subset` of the `Cover` class that returns a non-isolated subcover, whose implementation we discussed in stage 4, where its pseudo code can be found. Also, we will add a method to the `SudokuGrid` class to check if any subcover has been found in the new cover attributes.

First, we add the covers to the bracket class, the function `orthogonal_bracket_function` is the implementation of the orthogonal intersection function Λ as defined in Definition 3.3.3, page 99. The value `None` is only returned when there exists a `cell` in the given `bracket` such that `cell.value == omega`.

Then, the function `define_orthogonal_covers` defines all the orthogonal covers $(R, C, \Lambda^{\omega}_{k(R)})$ and $(C, R, \Lambda^{\omega}_{k(C)})$, using the `orthogonal_bracket_function`, for every $\omega \in \Omega$ a row-sourced and a column-sourced cover is made.

The attribute `orthogonal_row_cover` keeps track of covers with rows as domain and columns as counter domain, while `orthogonal_col_cover` handles covers with columns as domain and rows as counter domain.

```python
class Bracket:
    """..."""
    def orthogonal_bracket_function(self, bracket, omega):
        res = {cell_index for cell_index in SudokuGrid.I if bracket[cell_index].av_set is not None and omega in bracket[cell_index].av_set}
        return res if res else None

    def define_orthogonal_covers(self):
        self.orthogonal_row_cover = {
            omega: Cover(SudokuGrid.I, [self.orthogonal_bracket_function(bracket, omega) for bracket in self.row])
            for omega in SudokuGrid.Omega}
        self.orthogonal_col_cover = {
            omega: Cover(SudokuGrid.I, [self.orthogonal_bracket_function(bracket, omega) for bracket in self.col])
            for omega in SudokuGrid.Omega}
```

Now, in the `SudokuGrid` class the covers are defined for `SudokuGrid.brackets` using the method `SudokuGrid.define_orthogonal_covers()`.

The method `find_orthogonal_subset`, receives a parameter `source_covers` that is an array of either row or column-sourced potential orthogonal subsets, and the parameter `target_brackets` is an array with the brackets either rows or columns, the opposite type than the source of the subset. The method looks for a `subcover`, and if it finds one, it uses the parameter `target_brackets` to find the brackets where the cells should be pruned. Finally the method `prune_orthogonal` is called, it goes through the cells in the target brackets, except for the ones that belong to the intersection of the subset. That is why the cells in the `source_indexes` are skipped.

The `stage_five` function just calls `find_orthogonal_subset` for both types of covers, row and column-sourced and the specific size of the subcover we are looking for, that is, `m`.

```python
class SudokuGrid:
    """..."""
    def define_orthogonal_covers(self):
        self.brackets.define_orthogonal_covers()

    def prune_orthogonal(self, target_brackets, source_indexes, omega):
        for bracket in target_brackets:
            for index in SudokuGrid.I:
                if index not in source_indexes:
                    bracket[index].av_set_remove(omega)

    def find_orthogonal_subset(self, source_covers, target_brackets, m):
        for omega, cover in source_covers.items():
            subcover = cover.find_subcover(m)
            if subcover is not None:
                source_indexes, target_indexes = subcover
                target_brackets = [target_brackets[index] for index in
 target_indexes]
                self.prune_orthogonal(target_brackets, source_indexes, omega)

    def stage_five(self, m):
        self.find_orthogonal_subset(self.brackets.orthogonal_row_cover, self.
 brackets.col, m)
        self.find_orthogonal_subset(self.brackets.orthogonal_col_cover, self.
 brackets.row,m)
```

For the time complexity of `stage_five` we recall from stage four, that the time complexity of the algorithm *Find-subcover* is $\mathcal{O}(mn^{m+3})$.
Stage five calls `find_orthogonal_subset` $\mathcal{O}(mn^{m+3})$ on row-sourced covers and column-sourced covers. We have one row-sourced and one column-

sourced cover for each $\omega \in \Omega$, that is, $2n$ covers in total. Therefore, we have that the total complexity of stage 5 is $\mathcal{O}(\texttt{stage\_five}) = \mathcal{O}(2n)\mathcal{O}(mn^{m+3})) = \mathcal{O}(mn^{m+4})$.

Chapter 4

The Solveku Algorithm

By now, we have described all stages that make up our Solveku algorithm. Now, we will merge them and describe how they work together. If all the combined stages are not able to find a solution, we are going to complement them with a backtracking technique.

We first describe how the backtracking works when the stages are not able to find a solution. Then, we describe how we put all the stages together along with the backtracking to form the complete Solveku Algorithm. We will also have an implementation subsection, but instead of having the code for the whole algorithm, we will simply describe some of the most important details about the implementation of the algorithm as a whole. If the reader skipped the implementation part for all stages, it would not make sense to go through this implementation subsection. Additionally, implementation details are not required to understand the overall logic of the algorithm.

We created a website where our algorithm can be tested by introducing any sudoku puzzle `https://www.solveku.com`. The puzzle will be solved using Solveku and the output includes an step by step guide of the solution.

4.1 Backtracking

4.1.1 Plain backtracking

If the stages of the algorithms are able to find a solution, then that solution must be unique because we have proved for every stage that no solutions are lost with every step done by a stage. Subsequently, every time we reach a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ by applying a stage, we know for sure that all the solutions to the puzzle $(n, \mathbb{F}_0, \gamma_0)$ are exactly the same as all the solutions of this setting. We may have more than one solution because we do not know that the puzzle is well defined, which means that there can be multiple solutions for $(n, \mathbb{F}_0, \gamma_0)$, which implies multiple solutions for $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$. But if, after applying all stages, we are stuck in $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$, since we know that no solutions have been lost, we need a technique to proceed and find a solution.

Here, backtracking comes to the rescue. By backtracking, we mean choosing one cell and one candidate of this cell, assuming that it is the value of the cell, and proceeding with the stages. Notice that by arbitrarily placing a value in a cell, we are reducing available sets, from the cell itself and possibly from its neighbors, with no guarantee that the new state is a setting. That is, when we reduce the available sets of cells by backtracking, we may be losing solutions and, in the worst case, we might be losing all the possible solutions, reaching a setting of the puzzle that has no solutions. We will come back to this problem after describing the backtracking process.

Backtracking is a brute-force technique that easily (but not quickly) finds every solution of a puzzle by trying every possible combination of cell and candidate for every cell that does not have an assigned value. However, this is not efficient or even feasible in some scenarios. Nevertheless, in this work, we backtrack with the objective of finding just one solution because we assume that we are working with well-defined puzzles.

Searching for just one solution is much more interesting from an algorithmic point of view. If we assume that we are given a well-defined puzzle, then reaching one solution will be enough, since the solution is unique. Now, the interesting question is which cell and candidate should we choose to backtrack so that we can reach the solution as quickly as possible. Focusing on well-defined puzzles does not limit the algorithm; it just means that it will focus on finding one solution even if many are available. That is the question that we will address in the next subsection.

4.1.2 Informed backtracking

We are backtracking to find one solution. Therefore, we would like to choose a candidate that will take us to a solution as fast as possible. The best way to do so is to assign to a cell the candidate which will eventually be its value. Of course, we do not know what value the cell is going to take, so we really have no way to assert the best cell and candidate for backtracking.

As we discussed earlier, we may choose a wrong candidate and cell to backtrack, and end in an unsolvable puzzle. However, this is not always a bad scenario, because if we assign arbitrarily a value to a cell and then conclude that the puzzle becomes unsolvable, we know for sure that that cell cannot take that specific value, and we can remove it from the cell's available set. That is, we will rather realize quickly that we misplaced a candidate than keep going and keep wasting resources with an unsolvable puzzle.

When choosing a candidate and a cell to backtrack, we will choose the one that affects the most cells possible by reducing their available sets. In this way, we expect to either quickly reach a solution or realize that the new puzzle is unsolvable.

Here, we have the heuristics in the picture. A heuristic is an approach that employs a practical method that is not guaranteed to be optimal, but is effective in many cases. We are in the middle of a search problem, since we are looking for a solution. We will use a heuristic known as the Minimum Remaining Values (MRV) heuristic [Mal20]. The idea of this heuristic is that by making an assignment we will reduce the search space as much as possible, which will eventually mean that we are reducing the number of backtracks that we might need to do later.

In this context, our search space is the set of all the candidates for all cells that do not have an assigned value. We can only backtrack by assigning candidates to one cell. Hence, if one backtrack allows us to delete a lot of candidates from the grid, then we are preventing ourselves from more potential future backtracks.

Once we do a backtrack, we apply all stages to see if we can find the solution. In our algorithm, assigning a cell to a value has a domino effect, because when we assign a value to a cell we delete that value from the available set of all the neighbors of that cell. That is, when we assign a value to a cell, we will most likely be affecting it and many of its neighbors, we say many and not all, because some neighbors might not have the assigned value in their available sets.

We tried one backtrack variation that focuses on brackets that are almost complete, and starts by finishing those. This heuristic turned out to have poor performance in some cases, especially when the available sets of the cells that did not have a value in the chosen bracket contain every valid candidate. By valid we mean the candidates remaining in the chosen bracket. It makes sense that MRV has a better performance, since it chooses a cell based on every related factor that determines that cell's available set, instead of focusing only on one bracket and loosing generality. We also tried a more conservative approach, choosing candidates that do not affect many cells, with the hypothesis that the chance of doing a failed backtrack was less; the results for this approach were terrible, almost five times the number of backtracks than our MRV heuristic. However, this conservative heuristic is useful with not well-defined puzzles, that is, it is useful when looking for multiple solutions.

4.2 Sudoku theory

Technically, we are going to take a slightly different approach from the plain MRV heuristic. By reducing the available set of a cell that has only two candidates, we are effectively assigning a value to it and triggering all the processes just described. Therefore, we prefer to prune a cell with only two candidates than a cell with more candidates because that means that by pruning it, we are triggering a chain reaction.

Furthermore, in general, we rather choose a cell that has as few candidates as possible, it does not necessarily have to be just two candidates. That is, for every $m_1 < m_2 \leq n$ we prefer to prune a cell with m_1 candidates than a cell with m_2 candidates. Because the cell with m_1 is closer to having a value assigned.

As we discussed earlier, we have to choose one cell and one candidate ω to backtrack on. To make this choice in a deterministic way, we need to rank each pair of $(cell, \omega)$ by some rating. We now introduce two definitions, first, a rate that represents how many cells are affected by a backtrack, and then a way to order these ratings. We will rate every available candidate for each cell in the grid that does not have an assigned value. Then we are going to choose the candidate with the highest rate. Instead of rating with a number, we will rate with a collection of numbers. Each number in the rating represents how many cells are affected by the backtrack, grouped by the number of total candidates of those cells. We later define an order of these cells.

Definition 4.2.1. For a setting $(\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, and $cell^* \in \mathbb{G}$ such that $2 \leq |\mathbb{AS}(cell^*)|$ and for every candidate $\omega \in \mathbb{AS}_k(cell^*)$. Let $\mathbb{A}_{cell^*}$ be the set of cells affected directly by a backtrack, which are neighbors of $cell^*$ and $cell^*$ itself, that is

$$\mathbb{A}_{cell^*} = \mathbb{N}(cell^*) \cup \{cell^*\} .$$

Now, let r_i be the number of affected cells that have ω as a candidate and whose available set size is equal to i. For $i \in \{2, \dots, n\}$, we define:

$$r_i(cell^*, \omega) = \Big| \{cell : cell \in \mathbb{A}_{cell^*}, \omega \in \mathbb{AS}_k(cell) \text{ and } |\mathbb{AS}_k(cell)| = i\} \Big|.$$

Finally, we define ***backtrack rate function*** $\mathbb{BR}$ as a vector of values of r_i:

$$\mathbb{BR}(cell, \omega) = (r_2(cell, \omega), r_3(cell, \omega), \dots, r_n(cell, \omega)).$$

Note that since $2 \leq |\mathbb{AS}(cell^*)|$ then $\mathbb{BR}(cell^*, \omega) \neq \mathbf{0}$. Given that $\mathbb{BR}$ is a vector of numbers, we have to define a way to compare different rates, because we will backtrack with the cell and candidate pair that has the highest rate.

Definition 4.2.2. For two different backtracking rates $r = (r_2, \dots, r_n)$, $r' = (r'_2, \dots, r'_n)$, we say that $r < r'$ if there exists an index $l \in \{2, \dots, n\}$ such that for every index $i \in \{2, l-1\}$ we have $r_i = r'_i$ and $r_l < r'_l$.

In plain words, the definition only states that given two rates r and r', to determine which is smaller, we have to find the first entrance of the vector that is different for the rates, the rate whose value at that entrance is smaller is the smaller rate.

We want to choose the couple $cell^*$ and the candidate ω^* that has the largest $\mathbb{BR}$. In other words, we are looking for a couple that has a rate r^*, such that $r^* < r$ is not true for every other rate r in the grid.

Now, we have to handle what happens when we backtrack with the wrong candidate, which means that we chose to backtrack with a candidate in a cell that was not its value. Suppose that we choose to backtrack with $(cell^*, \omega^*)$, but then realize that, in fact, $\gamma(cell^*) \neq \omega^*$, where γ is a solution we are looking for. Although this is not the scenario we want, when it happens, we are not left empty handed, because at least now we know for a fact that $\gamma(cell^*) \neq \omega^*$. This means that we can now remove ω^* from $\mathrm{AS}(cell^*)$. This can happen only during the backtrack, because we are no longer certain that the result is a setting. Let us now formally define this scenario.

Definition 4.2.3. Given a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we define $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$ as a ***semi-setting***, if the following conditions are met:

- $\mathbb{F}_0 \subseteq \mathbb{F}'_k \subseteq \mathbb{G}$.
- $\gamma'_k : \mathbb{F}'_k \to \Omega$.
- AS'_k is a an available set function.

A semi-setting is a loose setting because it requires fewer conditions on the available set function than the standard setting definition. The difference is that standard settings asks for a valid available set function, while a semi-setting any available set function is permitted. It is clear that every setting is a semi-setting. Recall that we proved that when applying any of the stages(1-5) no solutions are lost. We now state a definition and a lemma to generalize this result to semi-settings.

Definition 4.2.4. For a natural number r, we say that a $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$ ***precedes*** another semi-setting $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$ if $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$ can be reached from $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$ by applying stages 1 to 5.
That is,

$$(\mathbb{F}'_{k+i+1}, \gamma'_{k+i+1}, \mathrm{AS}'_{k+i+1}) = g_i((\mathbb{F}'_{k+i}, \gamma'_{k+i}, \mathrm{AS}'_{k+i}))$$

where $g_i \in \{f_1, f_2, f_3, f_{N4}, f_{H4}, f_{R5}, f_{C5}\}$ for $i \in \{0, \dots, r-1\}$.

And now we state in the following lemma that no solutions are lost.

Lemma 4.2.5. *For r a natural number and two semi-settings $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$, $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$ such that $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$ precedes $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$. If $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$ is a setting, then $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$ is a setting.*

Proof. Since $(\mathbb{F}'_k, \gamma'_k, \mathrm{AS}'_k)$ is a setting and all stages preserve solutions, this means that $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ is a setting. Applying the same property r times, we can conclude that $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$ is a setting. □

Now we properly define the backtrack that will be used in our Solveku Algorithm.

Definition 4.2.6. For a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, we define ***Solveku backtrack*** as follows. Let $(cell^*, \omega^*)$ be the best BR rated pair. The ***Solveku backtrack*** returns a semi-setting $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$, where: $\mathbb{F}'_{k+1} = \mathbb{F}_k \cup \{cell^*\}$ and for every $cell \in \mathbb{G}$:

$$\gamma'_{k+1}(cell) = \begin{cases} \gamma_k(cell) & \text{if } cell \in \mathbb{F}_k \\ \omega^* & \text{if } cell = cell^* \end{cases}$$

$$\mathrm{AS}'_{k+1}(cell) = \begin{cases} \mathrm{AS}_k(cell) & \text{if } cell \neq cell^* \\ \{\omega^*\} & \text{if } cell = cell^*. \end{cases}$$

We place an apostrophe in the result of the backtrack because it is a semi-setting, and we are unsure if it is a setting. We will next apply the stages of Solveku to the semi-setting obtained by Definition 4.2.6, and analyze if it is a setting or not.

Definition 4.2.7. Given a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ of a puzzle $(n, \mathbb{F}_0, \gamma_0)$, let the semi-setting $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ be the result of performing a Solveku

backtrack on $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ with the pair $(cell^*, \omega^*)$.
We call the tuple $(cell^*, \omega^*, (\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1}))$ a ***failed backtrack*** if there exists a natural number r such that the following conditions hold:

- There exists $cell^o$ such that $\mathrm{AS}'_{k+r}(cell^o) = \emptyset$.
- $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ precedes $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$.

Now, the scenario is called failed backtrack because before the backtrack we had a setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$, and after performing the backtrack we ended up with $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$, which is not a setting, as we prove below.

Lemma 4.2.8. *For a failed backtrack*

$$(cell^*, \omega^*, (\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1}))$$

, the semi-setting $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ *is not a setting.*

Proof. By definition of failed backtrack we have that there exists a natural number r such that $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$ is not a setting and $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ precedes $(\mathbb{F}'_{k+r}, \gamma'_{k+r}, \mathrm{AS}'_{k+r})$. Now, by applying the logically equivalent contrapositive of Lemma 4.2.5 we can conclude that $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ is not a setting. □

With the previous argument, we can conclude that if we recognize a failed backtrack we can apply the following and last setting updating function f_6 to the original setting $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$:

$$\begin{aligned}(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) &= f_6((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (cell^*, \omega^*)) \\ &= (\mathbb{F}_k, \gamma_k, f_6^3(\mathrm{AS}_k, cell^*, \omega^*)\end{aligned}$$

where, for every $cell \in \mathbb{G}$:

$$\mathrm{AS}_{k+1}(\mathit{cell}) = f_6^3(\mathrm{AS}_k, \mathit{cell}^*, \omega^*)\big|_{\mathit{cell}} = \begin{cases} \mathrm{AS}_k(\mathit{cell}) & \text{if } \mathit{cell} \neq \mathit{cell}^* \\ \mathrm{AS}_k(\mathit{cell}) \setminus \{\omega^*\} & \text{if } \mathit{cell} = \mathit{cell}^* \end{cases}$$

Finally we need to prove that the result of f_6 is in fact a setting. Recall that f_6 is only applied after a failed backtrack.

Theorem 4.2.9. *For a setting* $(\mathbb{F}_k, \gamma_k, \mathrm{AS}_k)$ *of a puzzle* $(n, \mathbb{F}_0, \gamma_0)$*. Assuming* $(\mathit{cell}^*, \omega^*, (\mathbb{F}_k, \gamma_k, \mathrm{AS}_k), (\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1}))$ *is a failed backtrack, then* $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1}) = f_6((\mathbb{F}_k, \gamma_k, \mathrm{AS}_k))$ *is a setting.*

Proof. By Lemma 4.2.8 we have that $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ is not a setting. Let γ be a solution such that $\gamma(\mathit{cell}) \in \mathrm{AS}_k(\mathit{cell})$ for every $\mathit{cell} \in \mathbb{G}$. Now, since $(\mathbb{F}'_{k+1}, \gamma'_{k+1}, \mathrm{AS}'_{k+1})$ is not a setting, then AS'_{k+1} is not a valid available set function. Therefore $\gamma(\mathit{cell}^*) \notin \mathrm{AS}'_{k+1}(\mathit{cell}^*)$ since it is the only difference between AS'_{k+1} and AS_k. Finally, we can conclude $\gamma(\mathit{cell}^*) \neq \omega^*$, therefore $(\mathbb{F}_{k+1}, \gamma_{k+1}, \mathrm{AS}_{k+1})$ is a setting. $\square$

4.3 Structure of the algorithm

We will now put together all of the stages plus the backtrack to form the complete Solveku Algorithm. First of all, as we said at the beginning of the chapter, there are some interesting details in the actual implementation of the algorithm, which explain how the implementation subsection of all stages is placed together in one algorithm. However, these details are not necessary to understand the logic of the algorithm; thus, they are placed in the implementation subsection at the end of this section and will only make sense if the reader has gone through the implementation part of every stage.

The implementation sections of each stage exhibit a way to find a sufficient scenario. By scenario we mean the special case needed to apply the technique of the stage, for instance, a hermit in stage 2, or an internal subset in stage 4. Moreover, the algorithms shown in each stage not only show how to find a scenario, they guarantee that if the scenario exists in the received setting, they will find it. Therefore, in this algorithm description, we take for granted the search of a particular scenario and ask if the scenario exists.

Before proceeding with the algorithm, it is important to mention that Solveku is a recursive algorithm, meaning that it may be called by itself multiple times. This happens because backtracking is a recursive technique. Let us briefly introduce how this recursion works.

The Solveku algorithm receives a semi-setting $(\mathbb{F}'_k, \gamma'_k, \mathbb{AS}'_k)$. First, Solveku will be called with an initial state of a puzzle, if the stages are unable to find the solution by themselves, the algorithm will backtrack with the best rated candidate by $\mathbb{BR}$. Then, Solveku will be called with the semi-setting obtained by the backtrack and start to apply the stages all over again. If Solveku is unable to find solutions to the semi-setting resulted from the backtrack, we have encountered a failed backtrack and we can apply f_6 to the setting we had before the backtrack.

```
**Input:** A (semi-)setting $s_k = (\mathbb{F}_k, \gamma_k, \mathbb{AS}_k)$
while $\mathbb{AS}_k(cell) \neq \emptyset$ for every $cell \in \mathbb{G}$ and $\mathbb{F}_k \neq \mathbb{G}$ do
    if There exists a Singleton $(cell^*)$ on $s_k$ then
        $s_k \leftarrow f_1(s_k, cell^*)$
    else if There exists a Hermit $(\mathbb{S}, \omega)$ on $s_k$ then
        $s_k \leftarrow f_2(s_k, (\mathbb{S}, \omega))$
    else if There exists a Bracket Intersection $(\mathbb{S}_r, \mathbb{S}_t, \omega)$ on $s_k$ then
        $s_k \leftarrow f_3(s_k, (\mathbb{S}_r, \mathbb{S}_t, \omega))$
    else if There exists a Bracket Subset $(\mathbb{W}, \mathcal{W}, \mathbb{S})$ on $s_k$ then
        $s_k \leftarrow f_4(s_k, (\mathbb{W}, \mathcal{W}, \mathbb{S}))$
    else if There exists an Orthogonal Subset $(\mathfrak{R}, \mathfrak{C}, \omega)$ on $s_k$ then
        $s_k \leftarrow f_5(s_k, (\mathfrak{R}, \mathfrak{C}, \omega))$
    else
        Let $(cell^*, \omega^*)$ be the highest $\mathbb{BR}$ rated couple.
        Let $s'_k$ be the result of backtracking on $(cell^*, \omega^*)$
        if Solveku($s'_k$) finds a solution then
            return Solveku($s'_k$)
        else
            We have a failed backtrack
            $s_k \leftarrow f_6(s_k, (cell^*, \omega^*))$
        end
    end
end
if $\mathbb{F}_k = \mathbb{G}$ then
    return $\gamma_k$
else
    $s_k$ does not have solutions.
end
```

Algorithm 2: Solveku Algorithm

Now, let us make a couple of clarifications. First of all, in the last chapter, we did not define f_4, but we use it in Solveku. We used it to simplify the contents of the figure of the algorithm; in fact, the algorithm checks for both types of subsets, naked and hidden, and applies both f_{N4} and f_{H4}.

The same happens for f_5 with row-sourced and column-sourced orthogonal subsets. Finally, our implementation verifies that the subsets found are not isolated, because if we do not, then we may find a subset that does not do any pruning at all, which if not avoided might end up in a infinite while loop.

Taking those considerations into account, we can now be certain that the algorithm never becomes an infinite loop. First, we know that if the algorithm falls in one of the first five conditions, then the function will always prune the candidates from AS_k, and since the grid has a finite number of cells and candidates, we cannot end up in an infinite loop. If we backtrack, we are pruning as well, s'_k has all the candidates except ω^* removed from $\mathrm{AS}_k(cell^*)$. So Solveku is called with fewer candidates, and if the backtrack ends up failing, we remove ω^* from $\mathrm{AS}_k(cell^*)$ so by the same finite candidate argument we can assert that the algorithm never falls into an infinite loop.

Now, the algorithm stated above is a simplification of the actual algorithm; next, we will discuss in more depth the code implementation of the algorithm which has some small discrepancies from the algorithm stated here due to performance reasons.

4.3.1 Implementation

First, let us explain some of the most important discrepancies between the actual code and the algorithm stated above. Recall that stage 1 and 2 are the only ones that fix values to cells, while the latter three just prune the available sets of some cells.

Also, remember that the time complexity of the first three stages is polynomial in n, while the time complexity of stages four and five is exponential in n. We know that an exponential function grows much faster than polynomial functions. For this reason, the first three stages go through the whole grid (stages one and two) and through all the brackets (stage three), even though they find the corresponding special case they are looking for, hoping to find another one. This does not happen in our implementation for stages four and five.

Stages four and five stop when they find a subset that holds the conditions they are looking for, just as we showed in the algorithm. Stages four and five stop because their time complexity is high, so the added time to keep looking for other subsets once they have found one can be considerable, whereas, thanks to the new prune, new singletons or hermits could be found in a polynomial time. Stages four and five depend on an external parameter m that represents the size of the subset they are looking for.

Recall that the time complexity of finding subcovers depends on n as well as on m. Therefore, we want to keep m as low as possible.

It does not make sense to look for naked or hidden subsets of size m with $\lfloor \frac{n}{2} \rfloor \leq m$, because if we have a naked subset of size m with $\lfloor \frac{n}{2} \rfloor \leq m$, we know that there exists a complementary hidden subset of size $n - m \leq m$. Then, it is more convenient to look for the hidden subset of size $n - m$, since the time complexity will be lower than to look for the naked subset of size m. The exact same thing happens when we look for a hidden subset of size m with $\lfloor \frac{n}{2} \rfloor \leq m$. We can conclude that we should only look for subsets with a size smaller than or equal to $\lfloor \frac{n}{2} \rfloor$.

The just-described upper bound can also be applied to orthogonal subsets. We proved the existence of complementary subsets for orthogonal subsets as well. With the same objective, it does not make sense to look for a row-sourced orthogonal subset of size m, with $\lfloor \frac{n}{2} \rfloor < m$, if we can look, with a lower time complexity, for a column-sourced orthogonal subset of size $n - m$.

So, in the way this is actually implemented, we iterate for the valid m values, and for each value we look for subsets of that size before incrementing m, which is the most efficient way to look because the time complexity depends on m.

Finally, in the algorithm, we stated that we recognize that the input has no solution when there exists a *cell* such that $\mathrm{AS}_k(cell) = \emptyset$. In the code we actually implemented other flags to catch earlier if there are possible solutions or not. For instance, we check if the union of all the available sets of all the *cells* in a bracket is not Ω then there can be no unique solution. These are just performance shortcuts that do not have any impact on the logic of the algorithm.

The implementation of our algorithm is open source, so you can check and correct if necessary the implementation of the Solveku algorithm. See `https://github.com/licesma/Solveku`.

4.4 Performance of the algorithm

For testing the algorithm we use a database that contains 4982 puzzles, the database was created by us using different generators and sources [Stu08] [Par16] [Eas22] [Jel06]. We believe that using a larger database is unnecessary since most of the sudoku puzzle databases and resources out there are algorithm-generated, which means that many of their puzzles share lots of patterns that will result in similar solving times. So, adding all of them would just add some noise to our analysis.

By using different generators and sources, we ensure more realistic results because that way we minimize the chances of having better results just because the generator shares features with our algorithm. The database contains only well-defined sudoku puzzles, meaning there is only one solution to each of them.

First, we analyze the behavior of our algorithm with different puzzles, which will help us to understand how our algorithm works. We will check which of the stages are used more, and how many times each of them are used for puzzles. Afterwards, once we understand how Solveku is behaving, we are going to compare its running time with other algorithms and see its efficiency.

To understand the behavior of different algorithms, we will group the puzzles $(n, \mathbb{F}_0, \gamma_0)$ by hint count, that is, they will be grouped by $|\mathbb{F}_0|$. Of course, given that two puzzles have the same number of clues does not mean that the solving times for them will be the same, however, the solving times tend to decrease when the number of hints increases. So, in a general sense, sudoku puzzles with the same number of clues behave similarly.

4.4.1 Behavior of Solveku

We want to see how often each stage is used for reaching a solution of a puzzle. Let us clear something out, we are going to count how many times a stage is applied to a setting if and only if the stage updated the setting either by pruning an available set or by attesting a new value. Also, if a wrong assignation was done while backtracking, then all the stages applied to the puzzles whose origin was this assignation will not be counted. In other words, we will only count the stages that affect the puzzle and that lead directly to a solution.

While the pruning stages (3,4,5) reduce the available set of some cells, the attesting stages (1,2) are the ones that assign values to the puzzle and therefore are necessary to solve it. Thus, it is expected that the attesting stages will be used more than the pruning ones. The graph in Figure 4.1 shows the average of the instances of each stage grouped by the number of hints.

Even though our database has puzzles with up to 50 hints, for this graph, we only use the ones with fewer than 36 because after that most puzzles are solved almost trivially with one stage. In the pruning stages in the graph we have grouped stages three, four and five, backtracking and backtracking prune. We can see that stage one is more used than stage two. This makes sense since stage one is executed first. Actually hermits are a generalization of singletons, so all the singletons could also have been detected by the stage two if stage one was not implemented. On average, our database has about 150 puzzles per hint.

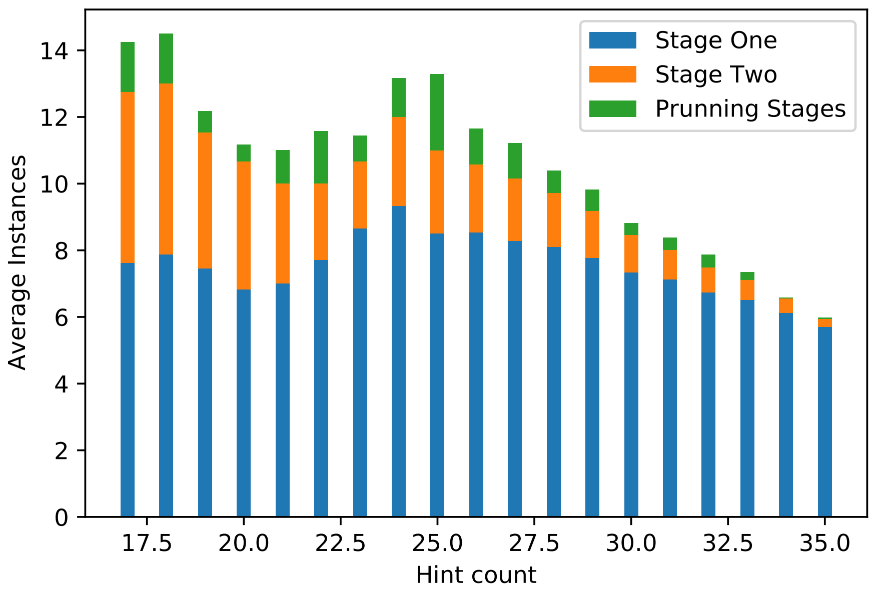


Figure 4.1: Usage of stages of Solveku.

Now, let us focus on the pruning stages and see how their usage is distributed. In the graph in Figure 4.2 we no longer track the attesting stages and we focus on the pruning stages. We refer to failed backtracks as Backtrack Prune since we do a prune of the candidate that we erroneously decided to backtrack with, whereas Backtrack represent the correct backtracks that lead us to a solution. We can see that Stage three and Backtrack Prune are the most broadly used, while Stage five is the least. The fact that Backtrack Prune is more used than some pruning stages does not mean that our heuristic is not good. In fact, we can see that in the worst case (25 hints) its average is approximately one. This means, in the worst case, that we have, on average, one erroneous backtrack per puzzle, which is quite successful.

Also, the fact that Stage 3, 4, and 5 are not used much does not mean they are not useful. Remember that when one of these is used and actually prunes the puzzle, the stages are executed again from the top. This means that one execution of these stages can create a ripple effect, this can result in fewer overall backtracks.

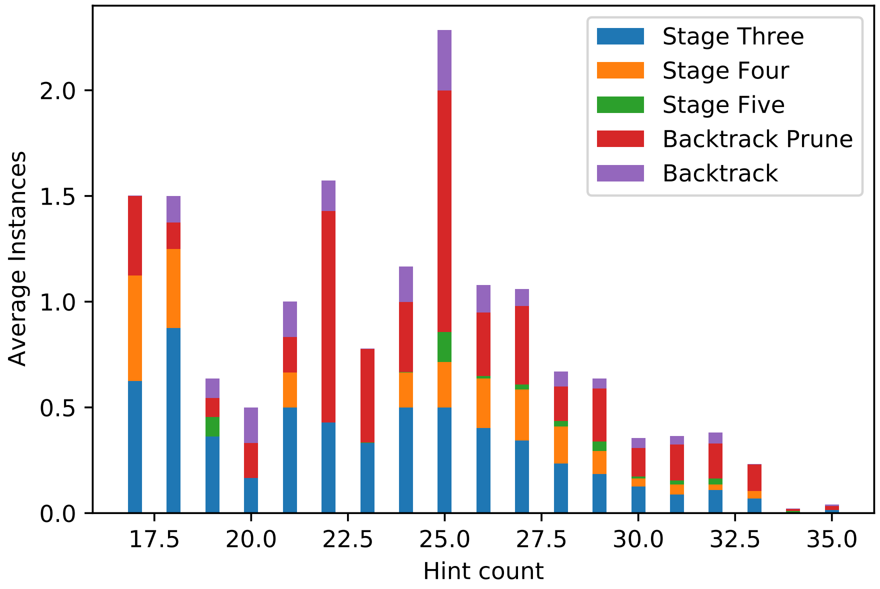


Figure 4.2: Usage of pruning stages of Solveku.

We have now analyzed how often all stages are used to reach a solution. Now, we are going to run the algorithm and see what happens if we do not use a specific stage. This will help us to understand which stages are indispensable for the algorithms and which not so much. This time we are going to test all the puzzles, and group them, again, by number of hints.

In Figure 4.3 we can see the crucial importance of stage one, stage two, and stage three. In the case when stage two is missing and the number of hints is low, we can see we have an outstanding increment on time, and the same effect with less magnitude happens when stage three is missing. Also, with with both stage one and two, we can see an increment of time in the puzzles peaks of the graph. Even though the algorithm without stage two seems very high at the beginning, a 60 ms average is a pretty good time for solving a sudoku puzzle, as we will see in the following subsection. When stage four and five are missing, we actually see, with a small hint count, a little reduction of time, but it is not considerable. This can be explained by the fact that subsets are not very common, but does not mean that the stages are not useful because when they are used the solving time of those puzzles is reduced significantly. However, since they are not used much, the average increments only in some cases. It becomes then a matter of choice whether or not to use them, depending on which aspect we are more willing to sacrifice. After 35 hints we can see that all the algorithms behave approximately the same way, this is due to the fact that most of them are solved trivially by either stage one or stage two.

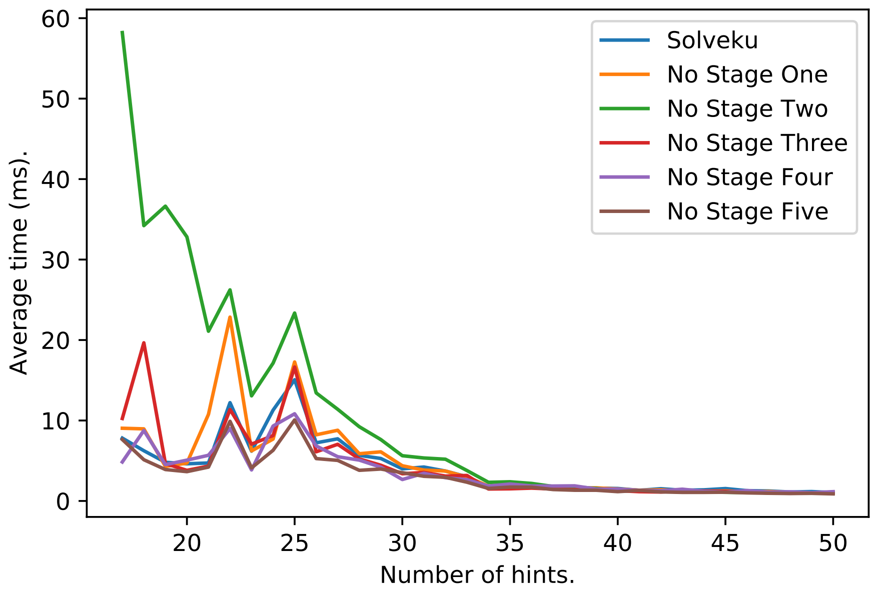


Figure 4.3: Solveku stage deletion comparison.

Let us now compare the Solveku Algorithm with the most popular algorithms for solving sudoku puzzles, first let us give a little context on the algorithms we are comparing.

Plain backtracking is the most popular algorithm for solving a sudoku puzzle because of its straightforward implementation, the one we used to compare was implemented by us as well. The linear programming algorithm uses the python library pulp and treats the Sudoku puzzle as an integer linear programming problem and solves it using a discrete variation of the Simplex Algorithm and other different popular linear programming algorithms. Dancing links is an algorithm created by Donald Knuth which is a variation of backtracking, just as Solveku, which has been widely known for having great results when applied to sudoku puzzles.

There are, of course, other algorithms that use artificial intelligence, like us, to solve Sudoku puzzles, however finding them is not an easy task because of the following reason: The advantage of having an algorithm that uses human-like techniques to solve Sudoku puzzles is that by using it backwards, it allows you to create brand new Sudoku puzzles where you can control which techniques are necessary to solve them, therefore, you can control the difficulty of the generated puzzles. This characteristic makes this type of algorithms profitable because they can produce new puzzles which can then be included in newspapers or magazines. We found one open source artificial intelligence algorithm on Github but this algorithm ended up not solving some puzzles correctly. Therefore, we decided not to include it at all.

In Figure 4.4 we can see that the linear programming algorithm and Dancing Links are not affected by the number of hints in a puzzle. Even though it is not affected by the number of hints, we can conclude that solving a puzzle by linear programming is way worse than using Solveku or Dancing Links. Meanwhile, backtracking grows exponentially when the number of cells without a hint grows, but then, it also becomes the fastest when the puzzles have a lot of hints. However, this does not compensate for its terrible behavior when the hint count is small.

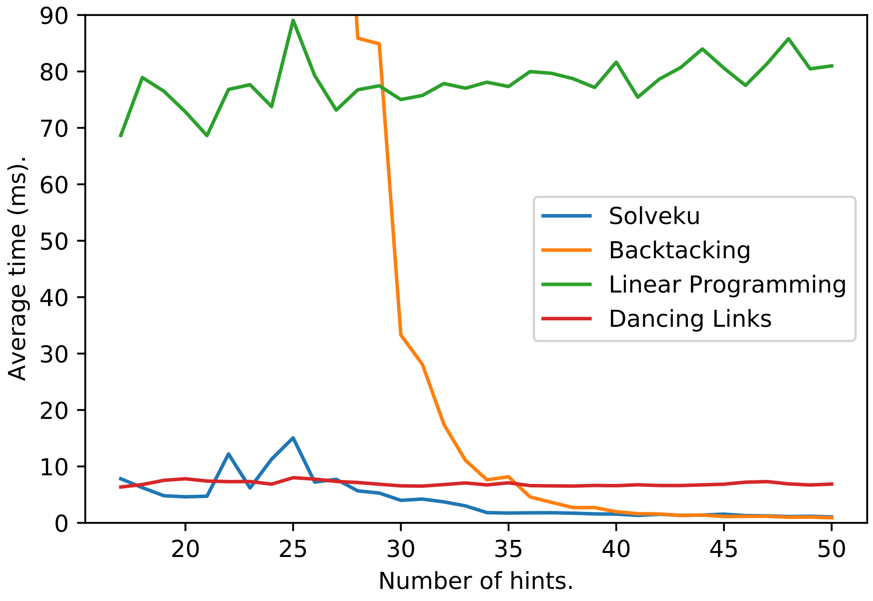


Figure 4.4: Sudoku solving algorithm time comparison.

It seems that the only real competitor for Solveku is Knuth's Dancing Links algorithm. Let us take a closer look, focusing only on this pair of algorithms.

In Figure 4.5 we can see how Dancing Links is amazingly unaffected by the number of hints in a puzzle. Now the differences are from one side, the peaks of Solveku are big on 22 and 25 hints. From the other side, the fact that dancing links is not affected by the number of hints plays badly against it after 30 hints, since Solveku times drops rapidly.

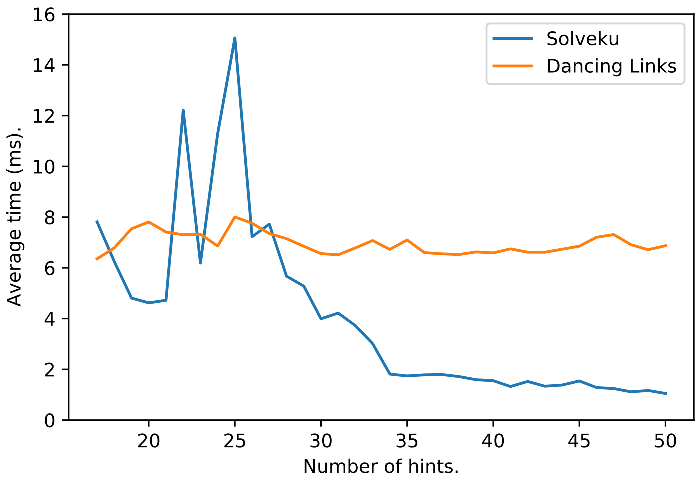


Figure 4.5: Comparison between Solveku and Dancing Links.

Let us now have a look at a box plot of each of the algorithms to better understand if the peaks of Solveku affect its overall performance badly. Box plots are a very popular way to compare two distributions of really big data. These type of plots use the best known statistics so that one can easily compare both distributions.

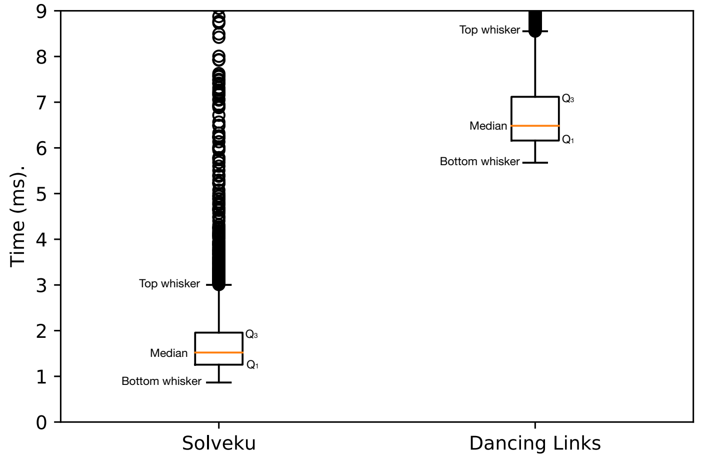


Figure 4.6: Box plot comparison between Solveku and Dancing Links.

We need the definition of percentile to understand the box plot graph. We now give a simplified definition of discrete quantiles which will be a foundation for our definition of percentile.

Definition 4.4.1. For a collection of values $X = (x_1, x_2, \dots, x_\alpha)$, we define the ***accumulation function*** $a\colon X \to [0,1]$ as follows:

$$a(x_i) = \frac{|L_i|}{\alpha}$$

where $L_i = \{x_j \colon x_j \leq x_i \text{ for } j \in \{1, \dots, \alpha\}\}$

Definition 4.4.2. For a collection of values $X = (x_1, x_2, \dots, x_\alpha)$ we define p_k for $k \in \{1, \dots, 100\}$ is the kth ***percentile*** of X as follows:

$$p_k := x_i \text{ such that } \left|\frac{k}{100} - a(x_i)\right| \leq \left|\frac{k}{100} - a(x_j)\right| \text{ for every } j \in \{1, \dots, \alpha\}\,.$$

Note that x_i might not be unique, then, if there exists a sub collection $(x_1, \dots, x_t) \subseteq X$ such that they all hold the above condition then p_k equals the mean of the sub collection.

Note that the accumulation function is different from the cumulative distribution function since X is not required to be a distribution for calculating our accumulation function. A box plot is a standardized way of displaying the distribution of data based on a five number summary. In a box plot, the bottom box line Q_1 represents the 25 percentile p_{25} and the top box line Q_3 is the 75 percentile p_{75}, respectively. The line in the inside of the box represents the median, that is the 50 percentile. The horizontal lines outside the box are called whiskers, the bottom whisker value is $Q_1 - \frac{3}{2} \cdot (Q_3 - Q_1)$ and the top whisker value is $Q_3 + \frac{3}{2} \cdot (Q_3 - Q_1)$. In this box plot we can see that most of the distribution of the Solveku times are smaller than most of the distribution of Dancing Links.

Fig 4.4.1 shows a box plot; the circles outside the whiskers are outliers whose maximum and minimum can be found in the following table (ms is an acronym for milliseconds).

Table 4.1: Solveku and Dancing Links statistic comparison

Metric	**Solveku**	**Dancing Links**
Mean	2.6071 ms	6.8387 ms
Std. Deviation	6.1837 ms	1.1482 ms
Minimum	0.8648 ms	5.6753 ms
25 percentile	1.2548 ms	6.1604 ms
50 percentile	1.5207 ms	6.4834 ms
75 percentile	1.9565 ms	7.1175 ms
Maximum	96.7031 ms	29.2785 ms

As we have seen earlier, on first look Solveku seems a better choice since its mean and every quartile is smaller than Dancing Link's

Nevertheless, Dancing Link's standard deviation is significantly smaller than Solveku's. This means most of Dancing Link's times are clustered around its means and there are only a few outliers.

Now, focusing on the distribution between 25 and 75 percentiles, it lies between $[6.1604 \text{ ms}, 7.1175 \text{ ms}]$. This is an extremely small range, its length is only 0.9571 ms, which is less than a millisecond, recall that a millisecond is one thousandth of a second. However, for Solveku, even though it has a much bigger standard deviation, if we take the half of the distribution with a smaller standard deviation, that is, the distribution between the 25 and 75 percentile, we get the rage $1.2548 \text{ ms}, 1.9565 \text{ ms}$. The length of this range is 0.7017 ms which is even smaller than Dancing Links'. This means, that even though the standard deviation of Solveku is higher than Dancing Link's, the mean's closest half of the distribution is closer in Solveku than in Dancing links, therefore the difference in standard deviations is due to outliers.

We can conclude that Solveku is a faster algorithm to the point that Dancing Links' minimum is bigger than Solveku's 75 percentile, with a 190% difference. Nonetheless, Dancing Links has a smaller standard deviation, due to Solveku's outliers. So Dancing Links will be a better fit if and only if outliers are unacceptable.

Chapter 5

Conclusions

This final chapter is divided into two parts. First, we explain a little bit more about why this work was done and how it was done. Then, we explore what, we think, are the most important contributions of this work and the potential impact of these contributions.

5.1 Why and how

When a person solves a sudoku puzzle they get just a glimpse of its infinite hidden logic. Before this project was conceived, we, avid sudoku solvers, started subtracting some patters out of this hidden mystery by solving sudokus by hand and paper. After some time we decided to not settle with the glimpse, and test how infinite really was this hidden logic.

The decision of creating an algorithm comes from a project in the first year of college that consisted in creating a backtracking sudoku solver. This algorithm was able to solve most algorithms, which was exhilarating, until we found the worst-case puzzle for backtracking. This puzzle is unfeasible for a backtracking algorithms [Ric08], because it is designed to make the standard backtracking algorithms to make as many failed backtracks as possible, taking advantage of the fact that standard backtrack starts assigning candidates in ascending order.

The fact that the worst-case puzzle could be easily solved by hand by any normal person motivated the fundamental question of this work: "What if we provide a little bit of intelligence to our backtracking algorithm?". This little bit of intelligence is now known as Stage 1, and it allowed the algorithm to solve the worst case puzzle in less than a couple of seconds. Eventually, the founding question evolved into "how smart can this thing get?". As a result of this evolution we then wanted to state a finite number of techniques, and create an algorithm that using a combination of these techniques would be able to solve any Sudoku puzzle.

We started by analyzing and decomposing, step by step, our thinking process when solving a puzzle by hand. This analysis originated the first four of the five techniques presented here, meaning they were discovered by ourselves. Therefore, some techniques can be found in other works with different names. For the last technique, we think of it as a tribute to James Gould, who we introduced in Chapter 1 as a fundamental person in the popularization of Sudoku puzzles.

We saw a video in which Gould explained the "x-wing" technique. We were inspired by this technique, and generalized it to create orthogonal subset prune. The "x wing" is a row-based orthogonal subset of size two.

Finally, since these five techniques were not enough to solve every puzzle, we decided to keep backtracking in our algorithm to allow it to be able to solve any puzzle. However, standard backtracking can be pretty inefficient, so we decided to improve it with a new "smart backtracking". For this, we were motivated by the "Search lecture" from the online Harvard's course "Introduction to artificial intelligence", where David Malan describes how can a search algorithm can be optimized using artificial intelligence [Mal20].

5.2 Contributions and impact

This work provides two major contributions. An open source sudoku solving algorithm, which is itself a substantial contribution since it is faster than most of the online algorithms available from free. Furthermore, this work also provides a rigorous sudoku theory which is fundamental for the Solveku Algorithm, besides, it can help anyone, with a math background, formalize ideas and patterns about a sudoku puzzle.

Apart from those two major contributions, one thing that sets this work apart is the fact that the algorithm and its theory were made to solve any $n \times n$ sudoku puzzle, not just the standard 9×9. This is one of the things that could have been done differently, since sudokus larger than 9×9 are not very popular. However, we think that the impact on generalizing the algorithm's size is beyond the popularity of larger puzzles, as we explain below.

All the algorithms that just solve 9×9 sudoku puzzles have a constant computational time complexity. This happens because time complexity relies on variability, and the puzzle has a fixed sized and number of symbols, hence a fixed number of computations will guarantee its solution. Not being able to calculate a computational complexity is a problem because it means we do not have a way of theoretically comparing algorithms; they can only be compared empirically. Generalizing the puzzles to size $n \times n$ allowed us to calculate computational complexities of the stages. Being able to show a time complexity is one of the main motivations to include code on this work, which is another thing that could have been done differently, since it is not the standard in mathematical documents.

Having a time complexity for stages is useful in many ways, first of all, we can compare between stages in a straightforward manner. Furthermore, assuming we are only interested in the 9×9 standard sudoku puzzles, having a theorical result on how an algorithm will behave on bigger puzzles gives us insight on how is it going to behave on 9×9 grid. With this principle, we introduced a way of theoretically comparing sudoku solving algorithms. First, generalize to $n \times n$ puzzles. Then compare time complexities of the generalized versions of the algorithm. Finally, apply the comparisson to the fixed standard size we were interested in. In the words of the philosopher Georg Wilhelm Friedrich Hegel, "Singular substrates or essences can only be known in relation to the general properties that constitute their appearances."

Finally, when we analyzed the behavior of Solveku in Chapter 4, we saw Solveku's hegemony over the rest of algorithms. Moreover, we analyzed some metrics on how the algorithm works, meaning which stages are more useful for Solveku. This analysis is an important contribution because it can help both, persons that solve algorithms by hand and developers of new sudoku solving algorithms. Even though Solveku does not solve a puzzle exactly like a person, we can deduce that if a stage is crucial for Solveku, it would be a good idea to apply it when solving puzzles by hand. Also, future competitor algorithms, can take a look at this analysis and decide which stages should be reused and which should not, according to their needs.

Bibliography

[Bel21] Alex Bello. Maki taji orbituary, 'godfather of sudoku' who paved the way for the worldwide boom in number puzzles, Aug 2021. https://www.theguardian.com/world/2021/aug/20/maki-kaji-obituary.

[Ber07] Denis Berthier. *The Hidden Logic of Sudoku (Second Edition).* Lulu Press, 2007.

[Cam21] Olivia Campbell. Maki kaji: Godfather of sudoku and puzzle enthusiast. https://www.independent.co.uk/news/obituaries/maki-kaji-sudoku-puzzles-obituary-death-b1907694.html, 2021.

[Che16] Haradhan Chel. A novel multistage genetic algorithm approach for solving sudoku puzzle, March 2016. https://www.researchgate.net/publication/311250094.

[Eas22] Easybrain. Sudoku, may 2022. https://sudoku.com/.

[Fis74] Sir Ronald A. Fisher. *The Design of Experiments.* Collier Macmillan, 7th edition, 1974.

[HL14] Mattias Harrysson and Hjalmar Laestander. Solving sudoku efficiently with dancing links, March 2014. https://www.kth.se/social/files/58861771f276547fe1dbf8d1/HLaestanderMHarrysson_dkand14.pdf.

[Jel06] Ernie Jellinek. Sudoklue, January 2006. https://www.learn-sudoku.com/.

[Mal20] David Malan. Cs50's introduction to artificial intelligence with python, April 2020. https://cs50.harvard.edu/ai/2020/weeks/0/.

[MG13] Civario Gilles Mcguire Gary, Tugemann Bastian. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration, August 2013. `https://arxiv.org/abs/1201.0749`.

[Pap14] Christos Papdimitriou. Algorithms, complexity, and the sciences, October 2014. `https://www.pnas.org/doi/10.1073/pnas.1416954111#executive-summary-abstract`.

[Par16] Kyubyong Park. 1 million sudoku games, August 2016. `https://www.kaggle.com/datasets/bryanpark/sudoku`.

[Par18] Kyubyong Park. Can convolutional neural networks crack sudoku puzzles?, August 2018. `https://github.com/Kyubyong/sudoku`.

[Ric08] Alan Rico. Star burst polar graph, March 2008. `https://www.flickr.com/photos/npcomplete/2361922699`.

[San09] Anders Sandberg. Venn and fisher window, January 2009. `https://www.flickr.com/photos/arenamontanus/3212655985`.

[Sho06] Will Shortz. The 2006 time 100, wayne gould, May 2006. `http://content.time.com/time/specials/packages/article/0,28804,1975813_1975838_1976198,00.html`.

[Smi05] David Smith. So you thought sudoku came from the land of the rising sun ..., May 2005. `https://www.theguardian.com/media/2005/may/15/pressandpublishing.usnews`.

[Stu08] Andrew Stuart. Sudoku wiki, April 2008. `https://www.sudokuwiki.org/`.

[Van19] Robert Vanderbei. Sudoku via optimization, October 2019. `https://vanderbei.princeton.edu/tex/talks/INFORMS_19/Sudoku.pdf`.

[WW11] J.C. George W.D. Wallis. *Introduction to combinatorics, Page 212*. CRC Press, 2011.