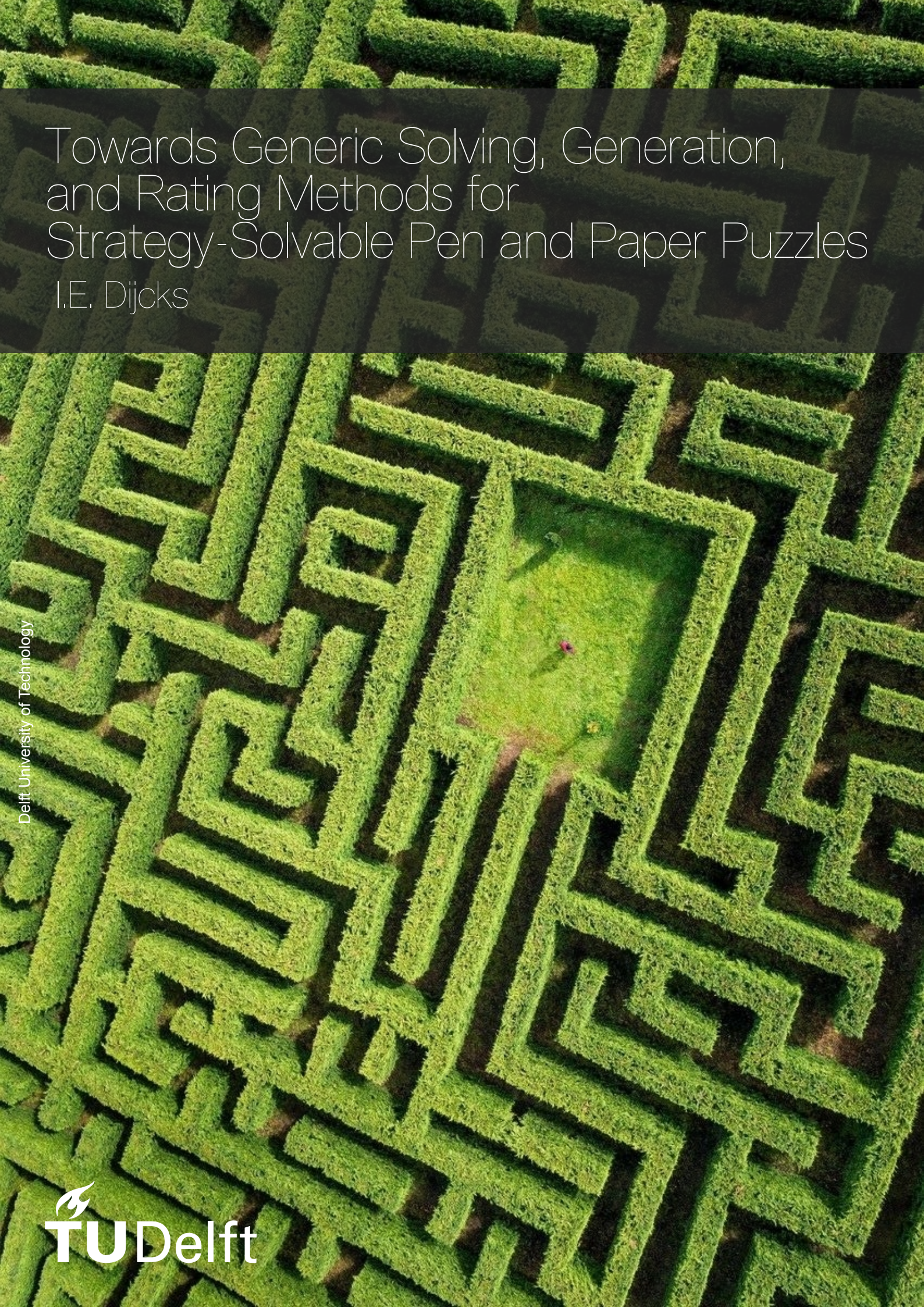
Towards Generic Solving, Generation, and Rating Methods for Strategy-Solvable Pen and Paper Puzzles

I.E. Dijcks



Delft University of Technology

TU Delft

Towards Generic Solving, Generation, and Rating Methods for Strategy-Solvable Pen and Paper Puzzles

by

I.E. Dijcks

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday June 21, 2022 at 13:00.

Student number: 4371151
Project duration: November 2020 – June 2022
Thesis committee: Dr. E. Demirović, TU Delft, supervisor
Prof.dr. A.E. Zaidman, TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.
Accompanying software is available at
`https://github.com/Ishadijcks/puzzle-agnostic-strategy-solver`.

TU Delft

Preface

Before you lies my MSc thesis, a result of seven years of education and growing while studying Computer Science at Delft University of Technology. Everything I have learned distilled into a single work about the wonderful world of puzzles. From solving, to generation, to rating, every aspect is discussed. I hope understanding it will not be a puzzle.

I would like to thank Steenhuis puzzles for their hospitality and willingness to share details about their puzzles. I would also like to thank Emir Demirović for his guidance and support during this project. But also his understanding when progress was not always as smooth as it was supposed to be. Finally I would like to thank all my friends and family for continuously asking me when I would finally graduate. Without such interest and motivation I truly would not have gotten where I am today.

Isha Dijcks
Delft, June 2022

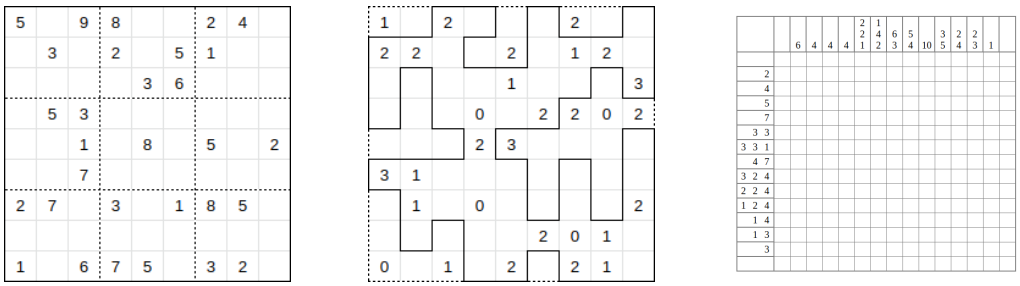
Abstract

Pen and paper puzzles are a fun pastime to test your logical reasoning skills, with Sudoku being the most popular of these puzzles. While the problem of solving these puzzles is usually in NP-Complete, generating them is more difficult, depending on the type of puzzle. When generating puzzles for humans, we have to keep in mind how they solve them, as we cannot expect them to brute-force the solution. Generation methods exist for simple number-based puzzles like the Sudoku, but these methods do not translate to the domain of language-based puzzles. We take the concept of strategy-solving to mimic human solving methods and apply them to the Raatsel, a language-based puzzle, and the Sudoku, to generalize a common methodology for generating, solving and rating these problem instances. A new method will be introduced to generate Raatsels using a reduction from Subgraph Isomorphism. This concept of instance generation with specific properties has many applications outside of puzzles, such as validating correctness and generating training data for machine learning.

Contents

1 Introduction

Pen and paper puzzles are a popular pastime. While there are various types of puzzles, they contain some similarities. In Figure 1.1a we can see a Sudoku, where the goal is to place the numbers 1 to 9 in a grid. Figure 1.1b shows a Slitherlink, where the player has to draw a single continuous loop along the edges of cells. Figure 1.1c contains a Nonogram, where the player has to color cells to create a picture.



(a) Sudoku **(b)** Slitherlink **(c)** Nonogram

Figure 1.1: Various popular pen and paper puzzles.

Even though these puzzles have different rules, there is a similarity in the way they are solved. A player uses the rules of the puzzle to conclude the placement or elimination of symbols. This process is repeated until the puzzle is solved, or the player gets stuck and gives up.

While the decision problem related to solving these puzzles commonly belongs in NP-Complete[6], they are trivially solvable by a modern computer. The instances are often quite small, with the most popular Sudoku format being 9x9. When generating a puzzle like Sudoku, we are usually not interested in the solving difficulty for a computer, but for humans. While a computer can simply brute-force all possibilities, a human does not have this luxury. To reason about puzzles in the context of humans, (e.g. to determine the difficulty and enjoyment of a puzzle), a human-like method to approach the problem of solving puzzles is needed.

1.1. Strategy-solvability

Instead of brute-forcing moves like a computer, humans solve puzzles using *strategies*; a set of "tricks" that are logical conclusions from the rules of the puzzle. Successfully applying a strategy either directly places a symbol or rules out candidates for a cell. A strategy is understood as an if-then rule, e.g. *If a cell can only contain one symbol because all other options are eliminated, the cell must contain that symbol*.

If repeatedly applying a set of strategies leads to a solved puzzle instance, we call that instance *strategy-solvable* with respect to those strategies[3]. We do not consider strategies that involve brute-forcing or guessing and backtracking. This goes against the goal of simulating human solving methods.

1.2. Raatsel

The concept of strategy-solvability has already been applied to the Sudoku[14]. To the best of our knowledge, there are no applications to a language-based puzzle at the time of writing.

One example of a language-based puzzle is the Raatsel. It consists of 30 words and 7 categories which have to be placed in a hexagonal grid such that each word belongs to their neighbouring categories and vice versa. Chapter 4 describes the Raatsel in more depth and contains an example Raatsel in Figure 4.2.

While the Raatsel is primarily a language puzzle, it contains a logic component for eliminating the possible positions in which words and categories can be placed. This makes it an interesting target of study; it has enough overlap with logic puzzles to analyze, while being different enough that existing methods can not be directly applied.

Creating a Raatsel by hand is time-consuming. The example Raatsel in Figure 4.2 took approximately 4 hours to create manually. While this time will decrease with experience, it is slow in comparison with automatically generated Sudokus, which can be created in milliseconds[9]. Automating the generation of Raatsels will improve their commercial viability.

1.3. Puzzle Generation

There are various existing methods for generating puzzle instances. For Sudoku the input is a set of desired clue positions, and the output a valid Sudoku instance with a single solution. Clues within these positions are changed until a desired instance is found. In a partially solved Sudoku, changing the value of a clue is a cheap operation, it is unlikely to propagate further to neighbouring cells. Because of the language aspect of Raatsel, changing a single word has more consequences for the neighbouring words and categories. It will be shown that existing methods are not applicable to the Raatsel, and a new generation method will be introduced.

1.4. Contributions

In this thesis we will explore the generation, solving, and difficulty rating of logic puzzles, with the end goal of generating fun and human-solvable puzzles. We aim to answer the question: **Can the concept of strategy-solvability be used to solve, generate and rate human-solvable generic pen and paper puzzles?** with help of the following sub-questions:

- Is strategy-solving a valid way to simulate the human solving process?
- Can a language-based puzzle be solved using strategies?
- Can strategy-solvability be applied to generate a language-based puzzle?
- Can rating methods utilizing strategy-solvability provide new information compared to existing rating methods?

The Raatsel puzzle has not been studied before, and it has to be formally introduced before a generation method can be designed for it.

The main contributions of this thesis are to introduce Raatsel generation as an NP-Complete decision problem, to develop a method to generate and solve Raatsels using strategies, and to provide a general framework to determine the difficulty of any logic puzzle. We will discuss how these findings can be applied to other fields of study, for example Machine Learning and Software validation, where instance generation is an important aspect.

1.5. Outline

The thesis is organized as follows. Chapter 2 will discuss the previous work done on logic puzzles. Chapter 3 introduces notation and terminology related to logic puzzles. Chapter 4 explains Raatsel and introduces it as a decision problem. Chapter 5 proposes a puzzle-agnostic solving framework and applies this to Raatsel. Chapter 6 evaluates existing generation methods and proposes a new method to generate Raatsel instances. Chapter 7 provides a method to calculate the difficulty for any type of logic puzzle based on the solving framework from Chapter 5. Chapter 8 describes the experiments performed to answer our research questions and evaluates their outcome. Finally, Chapter 9 discusses the consequences of this research while providing possible continuations and how they affect other areas of research.

2

Previous work

This chapter will discuss the state-of-the-art in the world of logic puzzles, from generation to the solving experience.

2.1. Classification of Logic Puzzles

Before we can compare solving, generation, and rating methods across puzzles, it would be beneficial to have a classification of logic puzzles. Such a classification makes it easier to reason about why certain concepts do or do not translate between different puzzles.

Japanese publisher *Nikoli Puzzles* is the world's leading producer of pen and paper puzzles[13]. They classify puzzles into three main categories: 数字 (numbers), 言葉 (language), and 絵 (picture).

This classification places Slitherlink and Sudoku in the same *numbers* category, even though they have little overlap. Sudoku is about placing numbers in cells, while Slitherlink is about drawing a continuous loop along the edges of cells. For a reminder on Sudoku and Slitherlink, see Figure 1.1.

Hufkens and Browne[5] present a more rigorous classification which can be seen in Figure 2.1. They describe a taxonomy which is able to classify 100 different puzzles. Instead of the content of the puzzle, it takes into account the goal of the puzzle, and what the player is writing or drawing to solve it. With this taxonomy, we can note that Slitherlink belongs to the Logic/Static/Abstract/Path/Loop class, and Sudoku to the Logic/Static/Abstract/Position/Symbols/Ordered class.

2.2. Solving Puzzles

The process of solving puzzles has been studied thoroughly, with Sudokus getting the most scientific attention. There are three main methods to solve logic puzzles: *Brute-forcing*, *Constraint Programming*, and *Strategy-solving*.

Brute-forcing is the naive concept of repeatedly trying solutions until one is valid. In the context of puzzles it repeatedly guesses a symbol for a cell, and backtracks if any rules are violated. Eventually a brute-forcer will always find a solution if it exists, or run out of options if no solution exists.

Constraint Programming is a powerful paradigm for solving combinatorial search problems[16]. The idea is to model the rules of the puzzle as constraints over decision variables. It utilises practices such as backtracking and constraint propagation to perform an exhaustive search on the search space. Since logic puzzles are combinatorial decision problems in essence, it would seem fitting to apply this method when solving logic puzzles. An advantage of this method is that it can easily be determined if a puzzle instance has a proper solution. If a solution can not be found, or more than one exists, the instance was not proper.

Strategy-solving is an approach to simulate human solving behaviour. A *strategy* is a logical consequence of the rules of a puzzle; an if-then rule that allows the player to deduce or eliminate a symbol. If a combination of strategies can be applied to bring a puzzle instance from a partially solved to a solved state, we call that instance *strategy-solvable* with respect to those strategies[3]. Strategies do not involve brute-forcing or guessing and backtracking. This violates the purpose of simulating human solving methods.

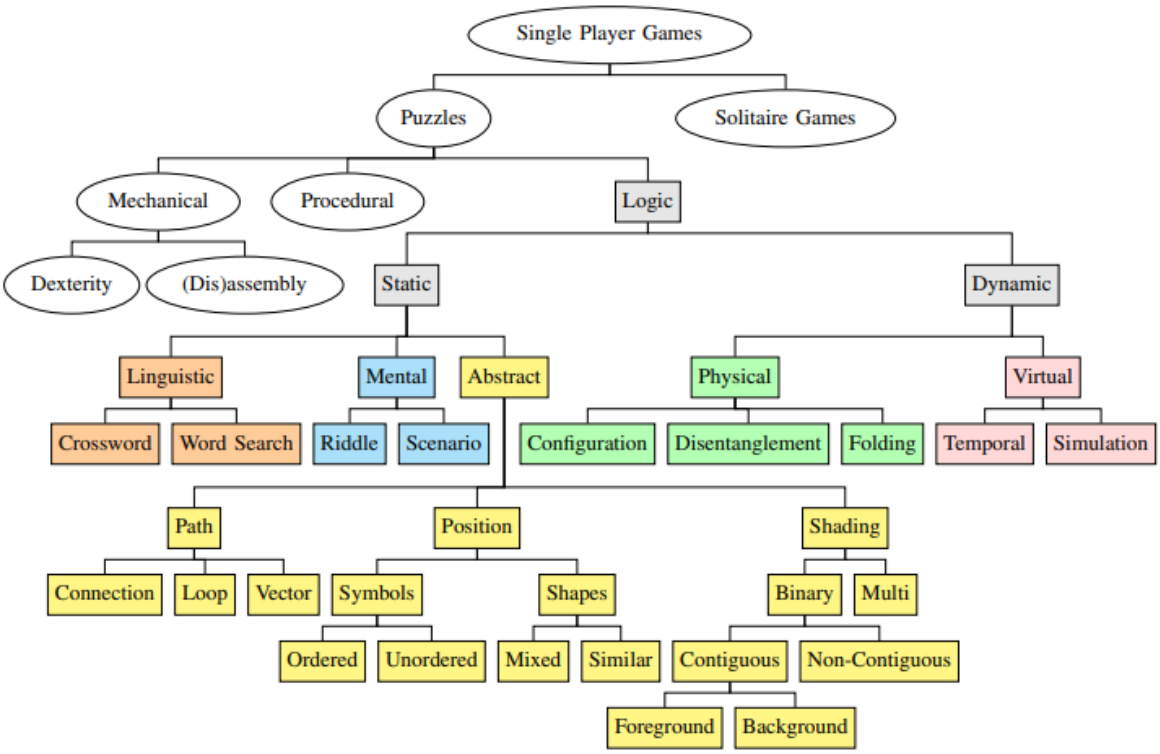


Figure 2.1: The taxonomy of logic puzzles by Hufkens and Browne[5].

While Strategy-solving is an approach used intuitively[8, 11], the method is first formally described for Sudoku by Nishikawa et al[14]. T. Davis describes over 15 different strategies for Sudoku in *Mathematics of Sudoku*[3]. It is important to note that strategies are not intrinsic to a puzzle. They can be arbitrarily defined and be as complicated or trivial as needed, as long as they do not utilise brute-forcing or backtracking.

When strategy-solving, something interesting happens when a puzzle instance has zero or multiple solutions. In either case, the solver will halt when no more strategies can be applied and the instance is not solved yet. It is however unable to tell if this is because it does not have the right strategies to solve the instance, or whether the instance itself is improper and has no solution. Either way, it can conclude that there is no logical way to solve this puzzle instance with the provided set of strategies, and declare it not strategy-solvable.

2.3. Rating Puzzles

Puzzles can be rated based on human performance[15]. Online solving portals keep metrics on how long their users take to solve puzzle instances. This provides large amounts of data, but can be unreliable as there is no control over the participants. This process cannot be automated, and puzzles would need to be released before they can be rated.

The difficulty of a puzzle instance can also be described as a function of the strategies it uses. A difficulty weight can be given to each strategy, and a solver will repeatedly try to apply them in ascending difficulty[17]. The difficulty is the sum of weights of the strategies used to solve the instance. The weights do not need to be static, their value can diminish based on how often the strategy has been applied already. This intuitively makes sense, applying a strategy for the first time is more difficult than applying it over and over.

Fun is an important concept in creating puzzles. While difficult to formally quantify, a player can easily tell if they are having fun or not. *A Theory of Fun* states that the act of finding solutions to a problem is fun, while repeatedly applying these solutions becomes boring quickly[7]. Applying this to the context of logic puzzles, we find that discovering a strategy is fun, applying different strategies in different context is fun, while applying the same strategy over and over becomes tedious.

To generate fun puzzle instances, we need a way to introduce variety in the solving process by requiring varying sets of strategies to solve.

2.4. Generating Puzzles

A desired property while generating puzzle instances is that they have a *unique* solution. This is a necessary condition for the instance being solvable by logic alone. We differentiate between *absolute* and *relative* uniqueness. Absolute uniqueness means there is only one single valid solution to the puzzle instance. Relative uniqueness uses the unique solution guarantee during solving, so the solver can eliminate positions that lead to multiple solutions. Relative uniqueness can still lead to non-unique valid solutions, based on the order that the puzzle is solved in[2]. Therefore, we will only consider absolute uniqueness when referring to uniqueness.

There are several generation methods for logic puzzles: *local search*, *forward search*[19], *backward search*[20, 1], and *logical modeling*[14]. While most applications deal with Sudoku, they can be applied to any type of puzzle.

Local search consists of generating a random partial solution, and modifying it until it can be solved uniquely. This is trivial to enforce when using a strategy-solver, as it will be unable to solve a puzzle instance with multiple solutions. Without a strategy-solver, the best option is to ban the first solution, and confirm there is no other solution when running the solver again. For a Sudoku, modification can mean swapping, adding, or removing numbers until a desired grid is found. This method provides little control over the difficulty of the resulting puzzle, but it can be used to find partial solutions with a specific clue pattern.

Forward search is the process of generating an initial partial solution randomly, and checking if a solution can be found using a solver. Some of these approaches take strategy-solvability into account, but this is not strictly necessary. The generator of Zama and Sasano[19][1] uses forward search to generate Sudokus. It takes a desired set of clue positions and fills them with random clues. If the selected set of strategies can be applied until the puzzle is solved, it outputs the clues as a valid Sudoku instance. Since the generator is searching randomly, there is no guarantee it will ever terminate for a given set of clue positions, or recognize it will never terminate.

Backward search starts with a completed solution, and repeatedly eliminates clues while keeping the uniqueness constraint. Sudoku Programming with C[20] tests for uniqueness without a strategy-solver and has to brute-force the uniqueness guarantee by banning the original solution. Another variation within backward search is to define the inverse of a strategy. Boothby et al. use these inverse strategies for the Sudoku to generate instances of a desired difficulty[1]. The difficulty of the resulting instance can be varied by applying easier or harder inverse strategies. Note that when removing a clue with a difficult strategy, there is no guarantee it can *only* be placed back with that same strategy. It could be the case it can be placed with an easy strategy, and the difficulty of the instance is overestimated.

Logical modeling involves modeling the generation problem into equivalent constraints with a CSP, and applying a generic solver to solve them. Nishikawa and Toda[14] use this method successfully to generate Sudokus solvable with the three most basic strategies: *Naked Singles*, *Hidden Singles*, and *Locked Candidates*. Their approach takes in a set of clue positions, and outputs not only a proper Sudoko instance, but a sequence of instances representing the entire solving process. It guarantees that the next instance can only be reached by applying the three strategies encoded into the CSP. To the best of our knowledge, this is the first puzzle generation method that can recognize whether the given set of clue position can generate a proper Sudoku instances. The CSP will eventually terminate if no valid assignment can be found.

[1] https://www.cs.ise.shibaura-it.ac.jp/2016-GI-35/

3

Preliminaries

While most readers are intuitively familiar with pen and paper puzzles, this chapter will define the terms used in the rest of the thesis.

3.1. Terminology

Puzzle

A *puzzle* is a type of decision problem that is described by the triplet (C, G, R):

Given a set of clues C, is it possible to assign a symbol to each cell in grid G, such that each rule $r \in R$ is satisfied?

Note that a grid does not have to be a two-dimensional space, it can be any structure.

Puzzles differ from games as puzzles do not have an adversarial component, and are solvable by a single player. Examples of puzzles are Sudoku, Slitherlink, Raatsel.

Cell

A *cell* represents a decision variable. It is up to the player or solver to assign a *symbol* to the Cell.

Symbol

A symbol is a value that can be assigned to a *cell*. This can be a number in a Sudoku, a word in a Crossword, or an edge in a Slitherlink.

Clue

A *clue* is anything that helps the player or solver solve a *puzzle instance*. This can be a placed *Symbol* as in Sudoku, or a number in a cell in Slitherlink.

Strategies

A *strategy* is an if-then rule that allows deducing the *symbol* which needs to be placed in a *cell*, or the elimination of a symbol in a cell. In a Sudoku, the simplest strategy is the *Naked Single*: ”If a *cell* can only contain one *symbol* because all other options are eliminated, the cell must contain that Symbol”.

Puzzle Instance

A *puzzle instance* is a (partially solved) instance of a *puzzle* that can be solved and *rated*. Solving a puzzle instance is finding the solution to the instance based on the provided *clues* and rules of the *puzzle*. A puzzle instance that has exactly one valid solution is considered to be *proper*.

Humans Versus Computers

A *player* is a human who is attempting to solve a *puzzle instance*, while a *solver* is an automated process. A *Setter* is a human who is manually generating a puzzle.

Rating

Rating is the process of applying a numerical score to a *puzzle instance* to indicate its difficulty. This numerical score can be binned to get a sensible rating such as 1–5 stars or easy/medium/hard.

3.2. Example

With our terminology defined, we can now claim that Figure 1.1a contains a *puzzle instance* of the Sudoku *puzzle*. It consists of a 9 by 9 grid with 81 *cells*. The *player* is given 30 *clues*, so there are 51 *symbols* left to place in the remaining cells. The *symbol* 5 can be placed in the bottom-left group using the *hidden single* strategy.

4 Raatsel

This chapter will explain the rules of the Raatsel and introduce it as a decision problem.

4.1. Puzzle

The standard size Raatsel consists of a symmetric hexagon with 37 cells. 7 are colored, and 30 are white. The cells have to be filled with categories and words respectively. It is up to the player to fill each cell with a category or word. The categories and words are given, but they have to be placed correctly in the hexagon. Every central, colored cell should contain a category, such that the word in each of the six neighboring cell belongs to that category. Each white cell should contain a word, such that it belongs to all neighboring categories. The words along each edge also need to belong to the category for that edge.

A smaller Raatsel and its solution is shown in Figure 4.1. A regular sized Raatsel is shown in Figure 4.2. It is *highly* recommended to try to solve it before continuing reading. The solution can be found in Appendix A.

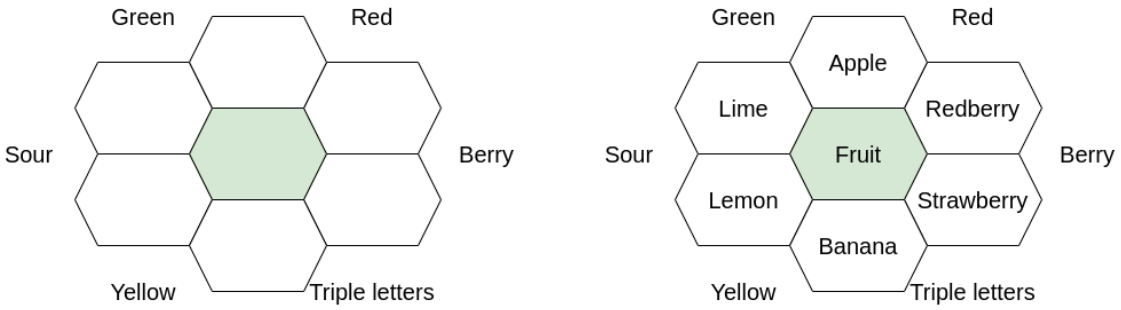


Figure 4.1: A small Raatsel and its solution.

Categories Fruit

Words Apple Banana Lemon Lime Redberry Strawberry

The Raatsel was first developed by Steenhuis Puzzels[1] in The Netherlands. It featured in *NRC Handelsblad*, a national daily newspaper. From correspondence with Steenhuis Puzzels, it was learned there have been a total of around 20 Raatsels published. At the time of writing, Raatsels are no longer created because it became too time-consuming to generate them manually.

4.2. Formalization

As input, the Raatsel takes a set of words $W = \{w_0, w_1, \ldots, w_{29}\}$, a set of fixed edges $E = \{e_0, e_1, \ldots, e_5\}$, and a set of categories $C = \{c_0, c_1, \ldots, c_6\}$. Since edges and categories have a similar role, the set

[1] https://steenhuispuzzels.nl/

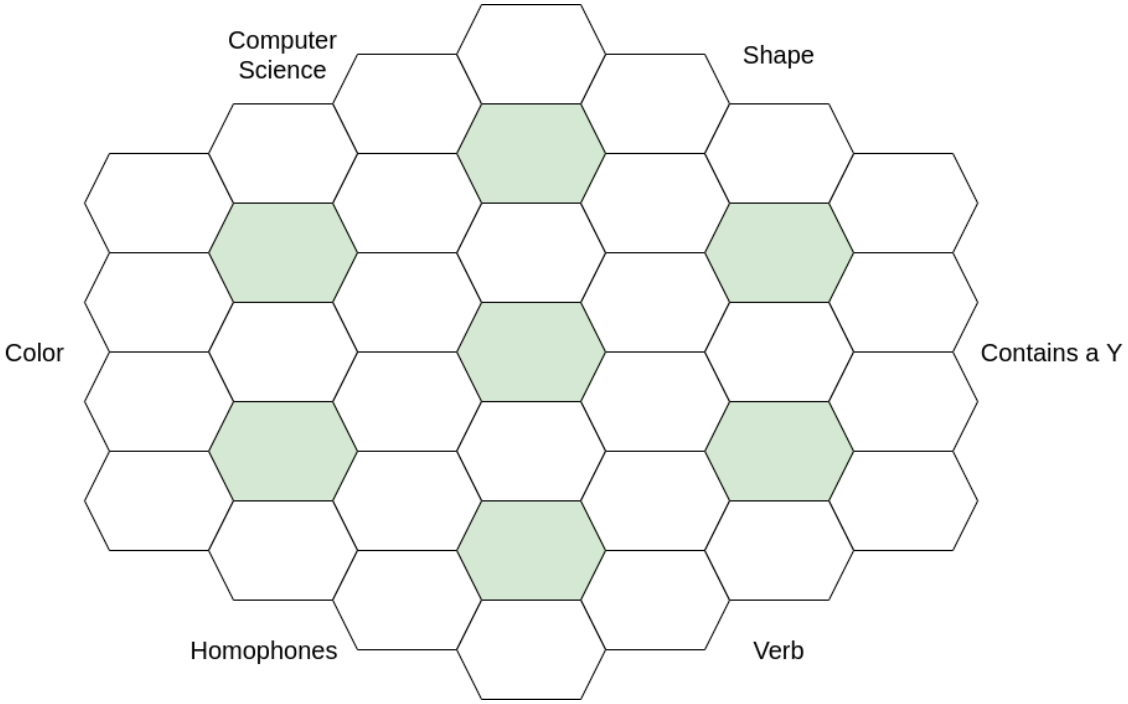


Figure 4.2: An example Raatsel. The solution can be found in Appendix A.

Categories

3 Characters	Animal	Plant	Pole	Snow	Space	Structure

Words

A*	Bay	Bulbasaur	Cold	Complexity	Crescent
Dance	DFS	Duck	Fish	Flea	Flower
Galanthus Nivalis	Ice	Iglo	Leopard	Lynx	May
North	Observatory	Pentagon	Pyramid	Red	RGB
Rose	Sun	Totem	Utility	Violet	Westminster Abbey

groups is introduced to combine them: $G = C \cup E$. The order of the words and groups can be seen in Figure 4.3.

To abstract the language aspect of the puzzle, truth matrix M is introduced to determines which words belong to which categories. For any word $w \in W$ and group $g \in G$, $M[w, g] = 1$ if and only if w is said to *belong* to g. (e.g. the word Apple belongs to the category Fruit)

The function $N(w)$ returns the neighbouring groups of word w. $N(w_{24}) = \{c_0, c_6\}$ for example, and $N(w_0) = \{c_0, e_0, e_5\}$.

The function $O(g)$ returns the neighbouring words of group g. $O(e_0) = \{w_0, w_6, w_2, w_3\}$ for example, and $O(c_0) = \{w_0, w_1, w_{17}, w_{18}, w_{23}, w_{24}\}$.

The decision problem for solving Raatsel then becomes:

Given a set of words W, a fixed set of edges E, a set of categories C, and a truth matrix M, does there exist an assignment of words and categories such that

$$\forall_{n \in N(w)} M[w, n] = 1 \qquad \forall w \in W$$
$$\forall_{n \in O(g)} M[n, g] = 1 \qquad \forall g \in G$$

4.2.1. Taxonomy

While the Raatsel might look like a linguistic puzzle at first glance, The functional taxonomy of logic puzzles described by Hufkens and Browne [5] would place it in the Logic/Static/Abstract/Symbols/Ordered class. While there is language involved, it is only needed to construct the matrix M. If M is provided to the solver, the puzzle is not solved yet. All the words and categories still need to be placed correctly. Note that this is the same class as the Sudoku.

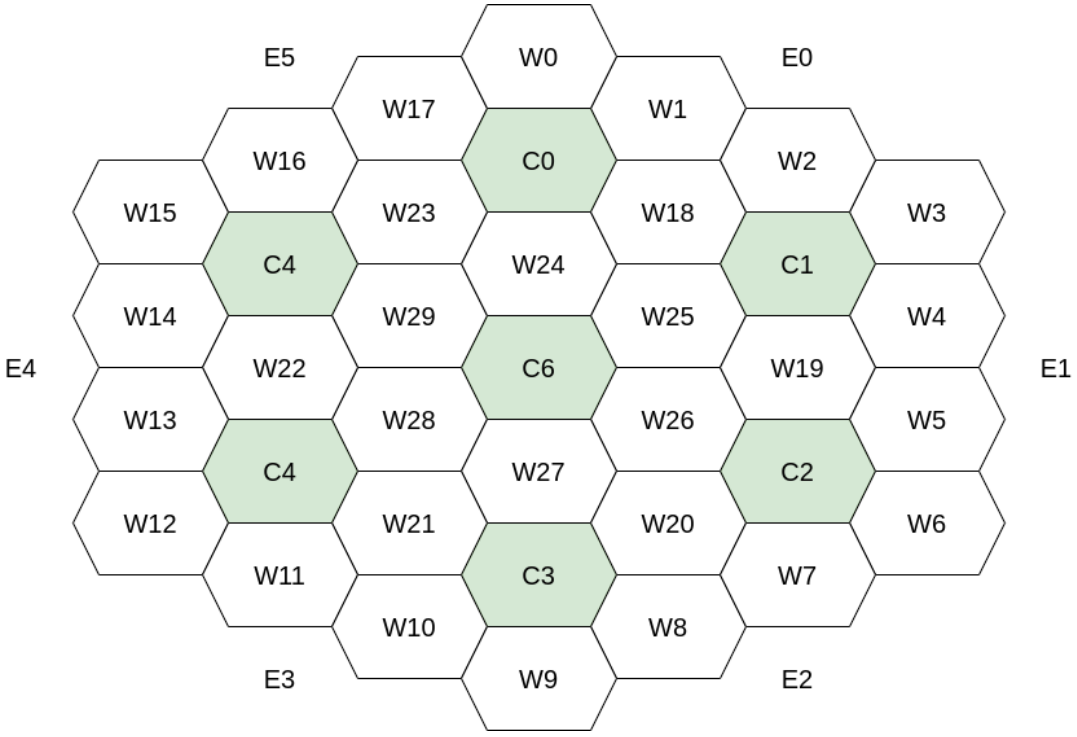
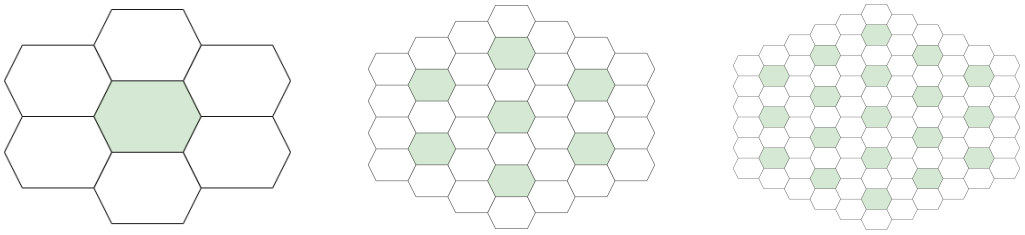


Figure 4.3: An indexed Raatsel showing the order of words, edges and categories.

4.2.2. Size

While all published Raatsels have been of size 2x2 like Figure 4.2, larger versions can still be considered. Because the hexagonal shape is maintained, the number of edges will stay fixed at 6 as the size grows. The number of the cells grows $|C| = 3N(3N-1)$. The number of categories grows $|G| = 3N(N-1)+1$. This growth can be seen in Figure 4.4.



(a) 1x1 **(b)** 2x2 **(c)** 3x3

Figure 4.4: Various Raatsels and their sizes.

5

Solving

In this chapter we propose a puzzle-agnostic framework that allows solving any type of logic puzzle, and implement a Raatsel solver within this framework.

Section 2.2 described the advantages and disadvantages of the three main solving methods: *brute-forcing*, *constraint programming*, and *strategy-solving*. The strategy-solver provides the most (human-relevant) information about a puzzle instance, and is the only solving method further considered. Evidence for this conclusion can be found in Section 8.1.

5.1. Generic Puzzle Solving

A strategy-solver is a program that takes a puzzle state P, and brings it to a next state by repeatedly applying strategies. Either until it reaches a solved state or until applying strategies no longer has an effect. It is then able to conclude whether the puzzle was solvable with respect to the given strategies. The order in which strategies are applied does not matter for the conclusion. In practice, applying strategies from the lowest difficulty to the highest is more efficient, as easier strategies usually require fewer computations. A high-level overview of such a solver can be found in Algorithm 1.

To aid our research, the solver should have the following features.

- Given a puzzle instance and a set of strategies, calculate whether it is strategy-solvable.
- Calculate all reachable states from a given state and a strategy.
- Represent a state as a string, and reconstruct the state from that string.
- Other puzzles (and their strategies) can be added by implementing a common interface.

A Python implementation of this solver is available online[1].

Algorithm 1: Solve a puzzle instance given a set of strategies.

input : The puzzle instance P, strategies S
output : Whether P is strategy-solvable with respect to S
while P *is not solved* **do**
  foreach $s \in S$ **do**
    $P_{new} \leftarrow apply\_strategy(P, s)$;
  end
  if $P == P_{new}$ **then**
    return $false$;
  end
end
return $true$;

[1] https://github.com/Ishadijcks/puzzle-agnostic-strategy-solver

5.2. Solving Raatsel

A set of strategies is introduced that is capable of solving Raatsels. More strategies might exist, but this set is sufficient to solve all known Raatsel instances. As there is such a small set of available Raatsels to experiment with, it was unfeasible to give the strategies an absolute difficulty rating.

Candidate Elimination

This strategy does not deduce the assignment of a word or categories, but it removes word or category candidates if they have already been placed elsewhere in the grid. This information can then later be used by other strategies to deduce assignments.

Word Naked Single

When considering a word cell c and its neighbouring groups $N(c)$, if there is only a single word w that belongs to all those groups, the assignment of w to c can be concluded.

Applied to the example Raatsel in Figure 4.2, it can be concluded that the top-left cell must contain the word *RGB*, as it is the only word belonging to both *Color* and *Computer Science*.

Category Naked Single

When considering a category cell c and its neighbouring words $O(c)$, if there is only a single category g that all words belong to, the assignment of g to c can be concluded.

Neighbour Elimination

Similar to *Candidate Elimination*, this strategy removes candidates for cells if they cannot be assigned due to a neighbour not having a relation with them.

Applied to the example Raatsel in Figure 4.2, it can be concluded that the top-left category cannot be *Plant*, as the already assigned word *RGB* does not belong to it.

Word Hidden Single

When considering a word w, if there is only one word cell that has it as a candidate left, the assignment of w to that cell can be concluded.

Category Hidden Single

When considering a category g, if there is only one category cell that has it as a candidate left, the assignment of g to that cell can be concluded.

Category Exhaustion

When considering a category for a category cell along the edge, if this category has not enough words belonging to it which also belong to the edges, this category can be eliminated as a candidate for this cell.

Applied to the example Raatsel in Figure 4.2, it can be concluded that the top-left category can not be *Snow*, as that category does not have two words which can belong to *Color* and *Computer Science*.

6

Generation

In this chapter we will discuss why the previously mentioned generation methods do not work for the Raatsel. Instead, we propose a new generation method specifically developed for the Raatsel.

6.1. Existing methods

In Section 2.4, we have seen the advantages and disadvantages of the four main generation methods: *local search*, *forward search*, *backward search*, and *logical modeling*. The first three of these methods are based on the same assumption, that it is *cheap to change a single clue*.

Local search is entirely based on this, it requires continuous modification of clues until we find a satisfying arrangement. With Raatsel, modification of a clue nearly always requires modification to another clue. If a word is replaced with another, it can be assumed to match its two or three neighbouring categories. It is unlikely that this randomly chosen word fits with the strict relations to its neighbours, and it is likely a category would need to be changed to fit. If a category is changed, it is unlikely all neighbouring words still fit to that category. Because the changing of each clue propagates to a change in the neighbouring clues indefinitely, local search is not an effective generation method.

Forward search could be applied in theory. Starting with an initial subset of clues, looking for new clues that fit within the hexagonal grid. Since natural language is involved, the chances of finding these new clues however, become incredibly slim. There needs to be a high overlap between words and categories. Forward search can however be applied to an abstraction of Raatsel, further discussed in Section 6.2.1

Backward search is based on the assumption that a solved grid is cheap to generate, clues are removed until a satisfactory amount is left. With Raatsel however, all clues are given, the player only needs to place them in the hexagonal grid. This means that a solved grid is transformable to a puzzle instance by removing all clues from the grid. Backward search is not possible, precisely because generating a solved grid is the true generation problem.

Logical modeling is not the right approach for the same reason that local search is not. It generates the entire solving path from an initial clue configuration. In a Raatsel, the clue configuration is always the same, therefore adjusting within this configuration amounts to local search.

A major guarantee of logical modeling is that it will either find a puzzle instance, or (eventually) recognize unsatisfiability for a given clue configuration. We will see that our proposed method also provides this guarantee.

6.2. Raatsel generation

Since existing generation methods can not be applied to Raatsel, we should look for properties unique to Raatsel to exploit. One observation is that, with the introduction of the truth matrix M, the linguistic aspect can be completely abstracted from the logic. As M contains the relations between words and categories, it fully determines whether this instance can be logically solved. The actual language associated with these words and categories becomes irrelevant. Note that the linguistic aspect is still important for generating the puzzle; some configurations of M might be impossible to represent by the

words and categories in the natural language, which needs to be taken into account. To represent the natural language, a *language graph* is created. This graph represents relations between words and categories with directed edges from words to categories.

With these fundamentals, the generation process can be divided into three steps.

- Find a suitable M, the structure of the Raatsel.
- Generate words and categories to create a language graph.
- Map the relations described in M to the language graph.

6.2.1. Generating M

The truth matrix M will be randomly generated from the initial, minimal truth matrix which satisfies all neighbouring groups for each word. Random edges will be repeatedly selected to be added to M. If the randomly selected edge does not lead to an unsolvable puzzle, we add it to M. Every edge added this way introduces a *false friend*; a relation which is not utilized in the final answer. This process will be repeated until the desired number of false friends is found. The pseudocode for generating such a truth matrix can be found in Algorithm 2.

Algorithm 2: Randomly generate the shape of a strategy-solvable truth matrix.

input : The desired number of false friends f_d
output : Truth matrix M
$I \leftarrow$ minimal edges required to be solvable;
M is a $(30, 13)$ matrix with all zeroes;
$E \leftarrow$ all edges in $M \setminus I$;
foreach $(w, g) \in I$ **do**
  $M[w, g] = 1$;
end

$f_p \leftarrow 0$ // The current number of false friends introduced;
while $f_p < f_d$ *and* $|E| > 0$ **do**
  select $e = (w, g)$ randomly from E;
  $E \leftarrow E \setminus e$;
  $M[w, g] = 1$;
  if $strategy\_solvable(M)$ **then**
    $f_d \leftarrow f_d + 1$;
  else
    $M[w, g] = 0$;
  end
end
if $f_p < f_d$ **then**
  Could not generate a matrix with the desired f_d;
else
  return M;
end

6.2.2. Generating Words and Categories

We have our truth matrix M, defining which words should belong to each category. In its essence, M is a graph between words and categories. A similar graph can be constructed for natural language.

Natural language is a difficult subject for computers. One effort to capture the relation between the meaning of words is Princeton's WordNet[18]. It is a tree of super-subordinate (Is A) relations. It contains synonyms, but more importantly homonyms, which can be used for interesting word associations. For example, it states that an *Apple* is an *Edible Fruit*, and *Editble Fruit* is *Fruit*. These super-subordinate relationships are transitive, so an *Apple* is also *Fruit*. Any words at the leafs of this language tree are called *instances*, and are what we will use for the Raatsel. The root note of WordNet

is the category *entity*. The entire path from leaf to root will be considered a category for the Raatsel generation.

Assume we have a directed language graph G(V, E), consisting of all known relations between words and categories in the language of choice. A directed edge is drawn from a word to a category, if and only if that word belongs to that category. An example of such a language graph can be seen in Figure 6.1.

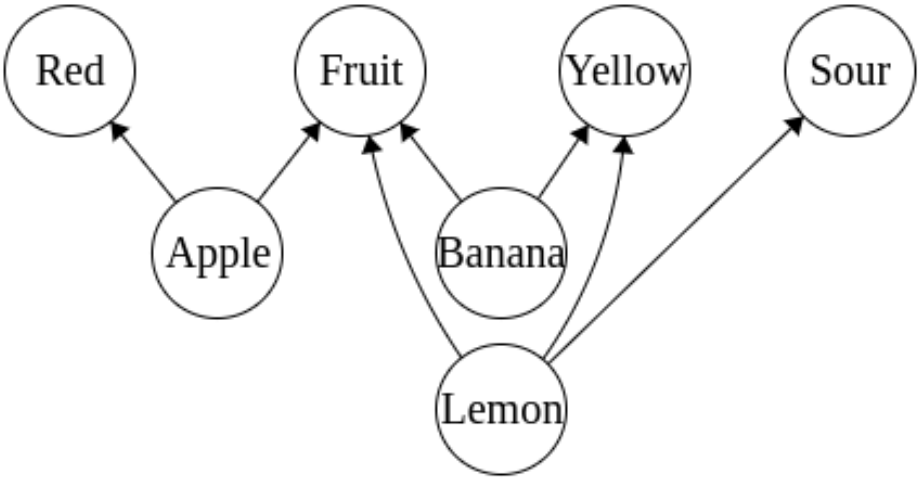


Figure 6.1: An example language graph consisting of 3 words and 4 categories.

To create a language graph, a random search is performed from a pivot word by visiting related parents, children and homonyms. Homonyms, and children of the pivot word are placed in a queue, while all parents are added as categories of this pivot. This can be repeated for every word in the queue until a desired number of words is found or the queue is empty. Because the entirety of WordNet is a single tree, most of the high level relations are undescriptive. Categories such as *entity* and *physical entity* can be banned, and will not be considered. An initial starting word is needed and the search process will take place around it, giving a theme to the generated puzzle. The pseudocode can be found in Algorithm 3. Afterwards the lowest common category between each pair of words is calculated, to make the language graph denser.

After the language graph is created, meta categories are calculated. These categories include for example *Starts with a T*, *Contains a Y*, or *3 characters*. These categories are added to the language graph to build the final space to search for M.

6.2.3. Mapping M to the Language Graph

To find a valid Raatsel that satisfies M, we need to find its representation within the language graph. The problem of finding a pattern graph in a larger target graph is called Subgraph Isomorphism[4]. This process can not introduce new edges between words and categories that are not specified in M, otherwise it might introduce multiple solutions. Therefore, non-adjacent vertices must be mapped to non-adjacent vertices, which is the problem of *induced* subgraph isomorphism. An example of subgraph isomorphism is shown in Figure 6.2. Any solution found by a subgraph isomorphism solver is therefore a valid, proper Raatsel instance.

Algorithm 3: Generate a language graph based around a starting word.

input : The starting word s, banned categories B, the amount of words needed n
output : Directed language graph

$Q \leftarrow \{s\}$;
$G \leftarrow (V \leftarrow \{s\}, E \leftarrow \emptyset)$;
$A \leftarrow \emptyset$ // The already visited words and categories;
while $|V| < n$ *and* $|Q| > 0$ **do**
  select and remove w from Q;
  $Q \leftarrow Q \cup (homonyms(w) \setminus A)$;
  $C \leftarrow parents(w) \setminus B$;
  foreach $c \in C$ **do**
    $I \leftarrow instances(c) \setminus A$;
    $Q \leftarrow Q \cup I$;
    foreach $i \in I$ **do**
      $E \leftarrow E \cup i$;
      $V \leftarrow V \cup (i, c)$;
    end
    $A \leftarrow A \cup I$;
  end
  $A \leftarrow A \cup C$;
end
return G;

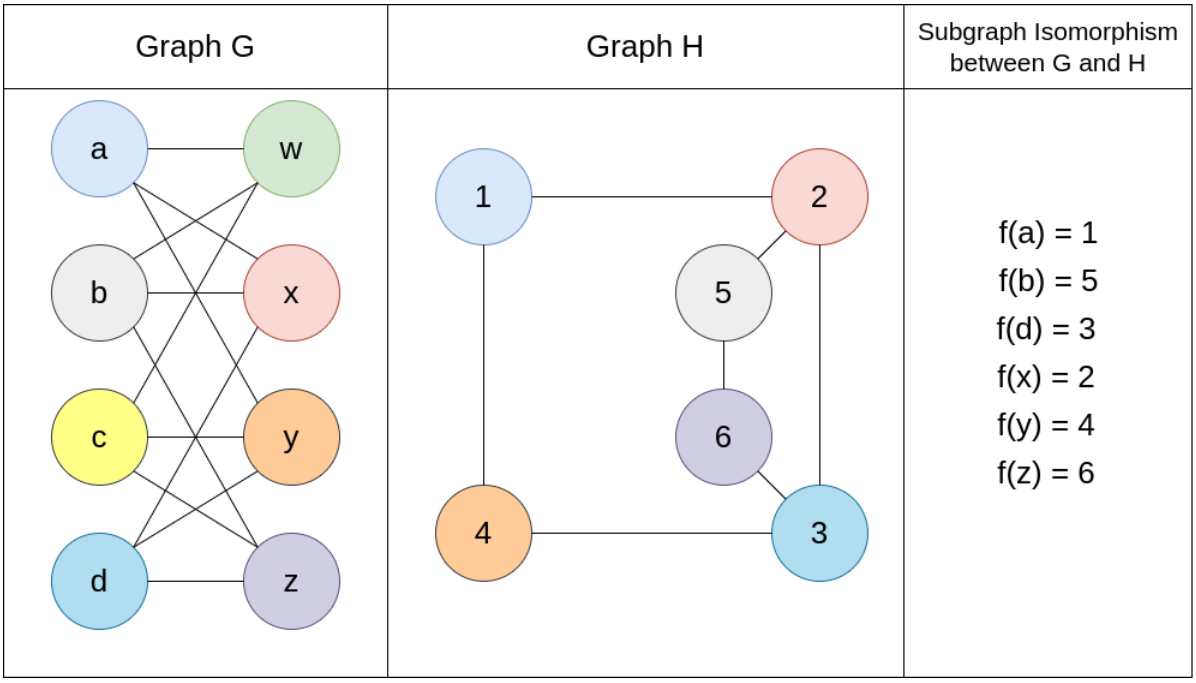


Figure 6.2: An example of subgraph isomorphism.

7

Rating

In this chapter we will discuss a generic puzzle-agnostic framework for calculating the difficulty of a given puzzle instance. When designing a rating method, the following principles need to be taken into account.

Principle 1 *The more steps required for solving, the higher the difficulty should be.*

Principle 2 *Having a higher number of paths to reach the solution should result in a lower overall difficulty.*

7.1. Existing Methods

The current method to rate puzzle instances, is to greedily apply strategies in ascending difficulties, summing their difficulty rating to get a difficulty score[17]. While this method is simple and effective, it can give an inaccurate score on certain puzzle instances. Sometimes applying a (relatively) difficult strategy trivializes the final steps, resulting in a lower overall score than when strategies of medium difficulty were repeatedly applied. This inaccuracy can be observed by slightly randomizing the order of applying strategies resulting in a lower difficulty score.

Because the total difficulty is increased with each strategy, this method satisfies the first principle. It does however not satisfy the second principle, since it only considers a single greedy path to solve the instance.

7.2. Time-Expanded Rating

Instead of greedily applying the easiest strategy, we introduce a new rating method that considers all possible solving paths from start to finish. As input, we take an unsolved puzzle instance P, which is known to be *strategy-solvable* with respect to a set of strategies S.

Assumption 1 *Each strategy $s_i \in S$ has a positive integer difficulty rating s_d that reflects its relative (human) difficulty with respect to the other strategies.*

The origin of the rating is not of importance, it should however accurately reflect the relative difficulty of the strategies.

Assumption 2 *Each strategy $s_i \in S$ must conclude the assignment of exactly one symbol.*

This is not always the case. Some strategies might eliminate possible symbols in a position, which would allow a different strategy to conclude the assignment of a symbol the next iteration. Such strategies will not be considered for this rating method.

Since P is unsolved, there are T symbols left to assign. Knowing that the puzzle can be solved by the strategies in S, and each strategy places exactly one symbol, the solving process must take exactly T steps. As P is valid and has a unique solution, the value of the deduced symbols are irrelevant, all that matters is the order in which they are deduced. The unique solution will be reached from all paths.

At any solving state at time step t, we can try to apply each strategy $s_i \in S$. Each strategy has the possibility to lead to another state at $t+1$. The cost associated with this transition to the next state is a harmonic mean of the s_d of the strategies bringing us to that state. A time-expanded graph of all possible states per step is constructed, with the amount of possible states per time-step t being upper-bounded by $\binom{T}{t}$.

Calculating the path with the lowest cost from P to the final solution, the easiest way to solve P is obtained. This is a more thorough estimate of the difficulty of P.

Pseudocode for this method can be found in Algorithm 4, and Figure 7.1 shows the process for a Sudoku of rank 2.

Algorithm 4: Calculate the difficulty of a puzzle instance.

input : The puzzle instance P, strategies S
output : The numerical difficulty score r
$T \leftarrow$the number of unsolved cells;
$states \leftarrow [[]]$ // 2d array that collects the reachable states per time-step;
$states[0] = [P]$;
$G \leftarrow (V \leftarrow \{P\}, E \leftarrow \emptyset)$;
foreach $t \in 0..T$ **do**
  $previous\_states \leftarrow states[t]$;
  foreach $P_{prev} \in previous\_states$ **do**
    $new\_states \leftarrow []$ // 2d array of states contains a list of all costs with which the states can be reached;
    foreach $s \in S$ **do**
      $reachable\_states \leftarrow apply\_strategy(P_{prev}, s)$;
      foreach $P_{next} \in reachable\_states$ **do**
        $new\_states[P_{next}].append(s_d)$;
      end
    end
    foreach $P_{next} \in new\_states$ **do**
      $d \leftarrow harmonic\_average(new\_states[P_{next}])$;
      $V \leftarrow V \cup state$;
      $E \leftarrow E \cup (P_{prev}, P_{next}, d)$;
    end
  end
end
$P_T \leftarrow states[T][0]$;
return $shortest\_path(G, P, P_T)$;

This method satisfies the first principle as each step increases the total difficulty. The harmonic mean is crucial to the second principle, it ensures that more options to place a number results in a lower overall difficulty.

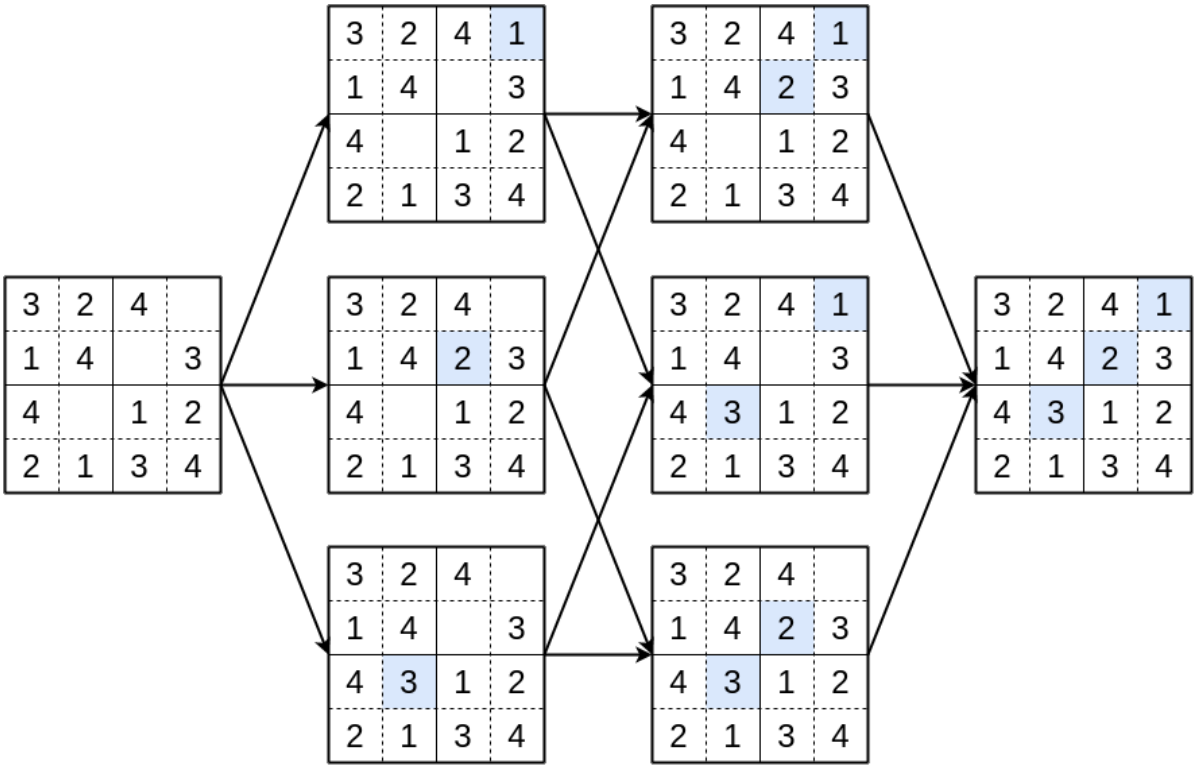


Figure 7.1: A rank 2 Sudoku, $T = 3$.

8

Experiments and Results

In this chapter we describe the setup and implementation details of the experiments performed to solve, generate and rate Raatsels. Afterwards we will discuss the outcomes of these experiments, and whether they confirm the proposed hypotheses.

All experiments are run with the puzzle-agnostic solving framework described in Section 5.1 All results presented are obtained by running our algorithms on an HP ZBook Studio G5 laptop with an Intel i7 processor.

8.1. Comparing Solving Techniques

In Chapter 5 the differences between brute-forcing, constraint programming and strategy-solving were discussed. To prove that strategy-solving resembles human solving behaviour, a correlation between human solving time and strategy-solving time is to be expected. To test this hypothesis, we will apply the three methods to a selection of Sudoku instances. Sudoku of the day[1] is a website that offers free Sudoku instances of varying difficulties. 4 instances of the difficulties *Beginner*, *Medium*, *Fiendish* and *Diabolical* were selected.

- The brute-forcing method tries to guess a valid number from topleft to bottom right inscreasing from 1. If a guess turned out to be incorrect, it will backtrack and try the next number.
- The constraint programming solver uses a Minizinc[12] (Gecode 6.3.0) implementation which can be found in Appendix B.
- The strategy-solver is the Python implementation described in Section 5.1, with access to the strategies *Hidden Single*, *Naked Single*, and *Locked Candidates*.

Results

Figure 8.1 shows the relation between runtime and sudoku instance difficulty for three different solving methods.

It can clearly be seen that the difficulty of an instance has very little influence on the runtime of the CSP; it is nearly constant.

The brute-forcer does appear to take longer with the more difficult instances, but it only performs noticably better on the *beginner* instances. It also has significant variation between runs on the same instances. Rotating the instance can cause the brute-forcer to take 50x as long on some instances, even though it is still logically the same instance. It is especially sensitive to the values of the solution. Since it tries to place numbers from low to high, it takes significantly longer if the first guessed cells contain higher numbers.

The strategy-solver does consistently take longer the more difficult the instances get. A Beginner instance never takes longer than a Medium instance, and a Medium never takes longer than a Fiendish. It was unable to solve the Diabolical instances with its limited set of strategies. This is not a limitation, it actually allows us to conclude that these instances are not strategy-solvable with respect to $\{HiddenSingle, NakedSingle, LockedCandidates\}$.

[1] https://www.sudokuoftheday.com/

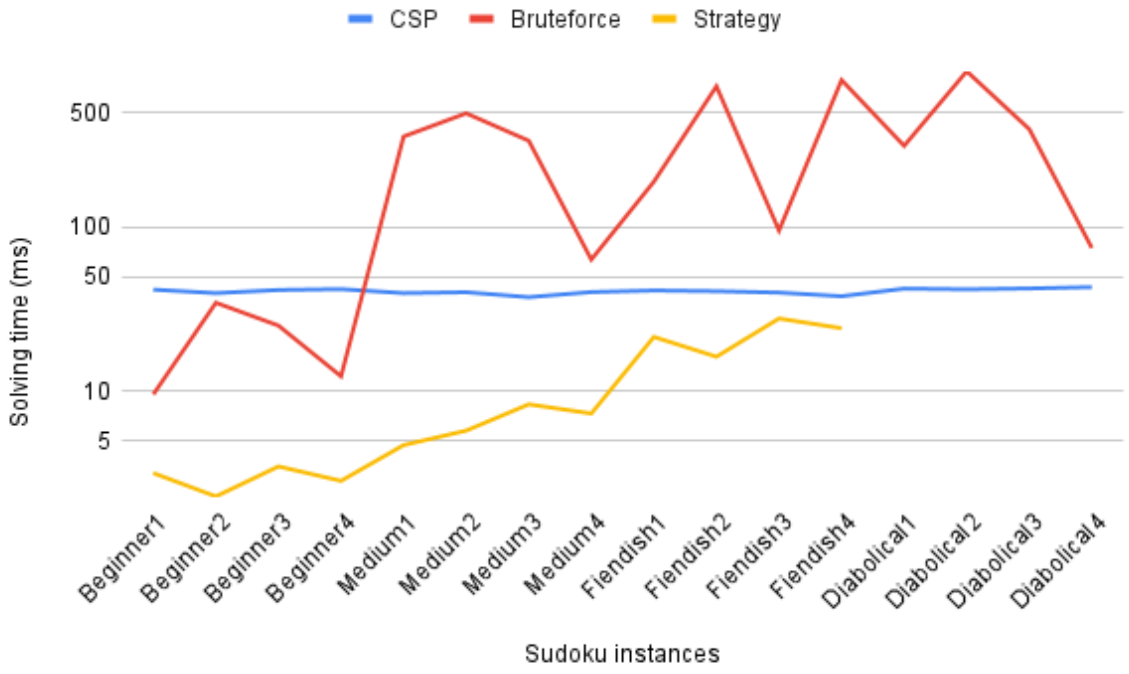


Figure 8.1: Solving time of sudoku instances for different solving methods.

The strategy-solver is the only solving method that shows a clear correlation between solving time for computers and solving difficulty for humans, which confirms our hypothesis.

8.2. Raatsel Generation

To generate Raatsels we need a shape, represented by the truth matrix M, and a collection of words to satisfy this relation.

The algorithm described in Algorithm 2 will be used to generate M. As input we vary the number of desired false friends from 0 to 40. When the amount of false friends increases, it is expected that the success rate of generating M decreased, as the likelihood of the shape becoming unsolvable increases with more relations.

Algorithm 3 will be used to construct the language graph. As base input, the word ”Apple” will be used. Additionally, another experiment will be run with a hand-crafted language graph based on existing Raatsels. Expected is that it will reconstruct one of the existing raatsels.

The Glasgow Subgraph Solver will be used to find the pattern graph. It is a fast implementation with support for induced subgraph isomorphism, and can prove unsatisfiability if no solution can be found[10]. Multiple instances can be retrieved from a single pattern and target graph, but it is highly likely they have significant overlap.

Results

Raatsel generation consists of three parts: Generating the shape M, generating a language graph, and finding M within the language graph.

Firstly, the generation of M goes smoothly when the desired number of false friends is small. In Figure 8.2 it can be seen that the success rate goes down when the amount of false friends grows. A reasonable amount of false friends for a Raatsel is around 10, which has a success rate of 71%. Fewer and the puzzle becomes too easy, while more makes it much harder to find words to fit the shape. This success rate makes generating the shape of Raatsels practical.

Secondly the building of the language graph was less successful. WordNet contains mainly factual hierarchical relations between words. This not only leads to uninteresting puzzles, it is insufficient for

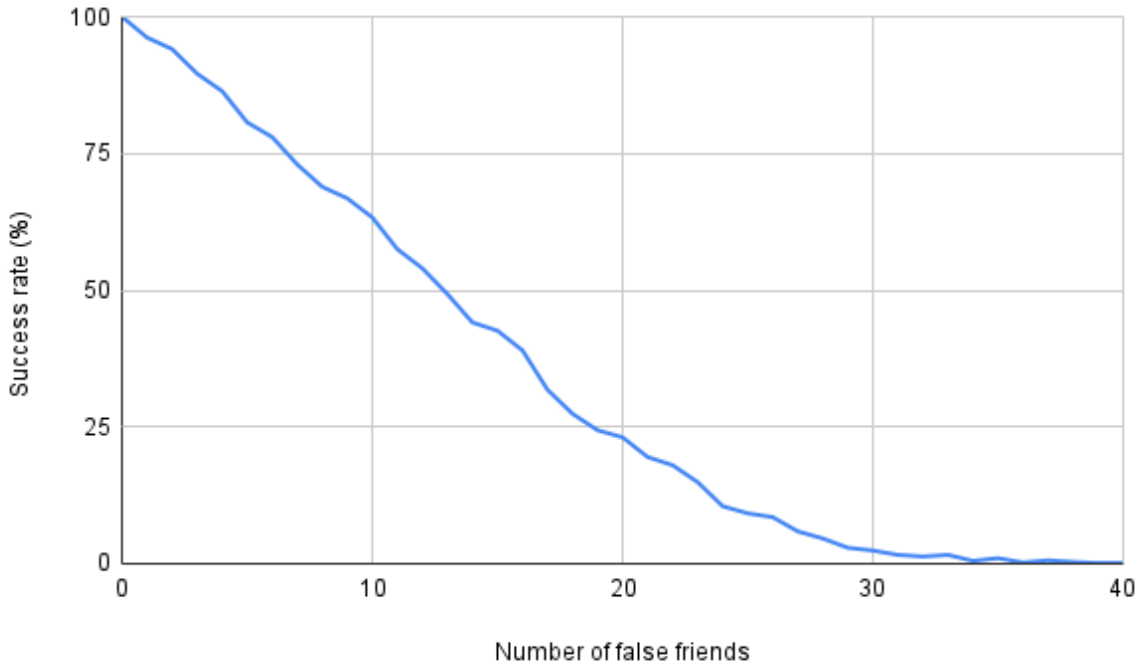


Figure 8.2: The success rate after 1000 trials of generating M for the given number of false friends.

the Raatsel which needs a higher density of overlap between words and categories. It is too sparse as most words only belong to a few direct categories. Conversely, because of the inherent tree structure, words that share a category also share the parents of that category. This introduces many false friends in a way that leads to Raatsels with multiple answers. Figure 8.3 shows an improper Raatsel which can be generated if the requirement of strategy-solvability is dropped. The tree structure of WordNet is clearly reflected in the selected words and categories. From this language graph, uniquely solvable Raatsels could not be generated.

Finally, finding M in the language graph works well, as it is exactly the problem of induced subgraph isomorphism. Even though it could not find uniquely solvable Raatsels in the WordNet language graph, running it on the hand-crafted language graph provided an interesting result. The expectation was that it would recreate one of the Raatsels that was used as input. However, since there was overlap between the Raatsels it generated a new Raatsel which is a combination of the input. This newly created Raatsel can be seen in Figure 8.4, it contains 23 words from the first Raatsel, and 7 from the second Raatsel.

This shows our method for generating Raatsels works, provided the language graph contains sufficient dense relations.

8.3. Rating Sudoku

The rating method described in Section 7.2 has been implemented using the solving framework described in Section 5.1. It will be applied to the same 16 Sudoku instances used in Experiment 8.1. The expectation is that the rating outcomes are lower than their original rating, since our method lowers the difficulty score if there are multiple ways to assign a symbol. The relative order between difficulties should stay consistent however, a *beginner* puzzle should not have a higher rating than a *medium*.

Since a Sudoku usually consists of around 60 steps, at $t = 30$ there is a possibility to have a maximum $\binom{60}{30} \approx 1.18 \times 10^{17}$ different states. There is a possibility this method will not terminate (within reasonable time) for puzzles where the search-space grows large too quickly. Therefore, experiments will be terminated if they exceed 6 hours.

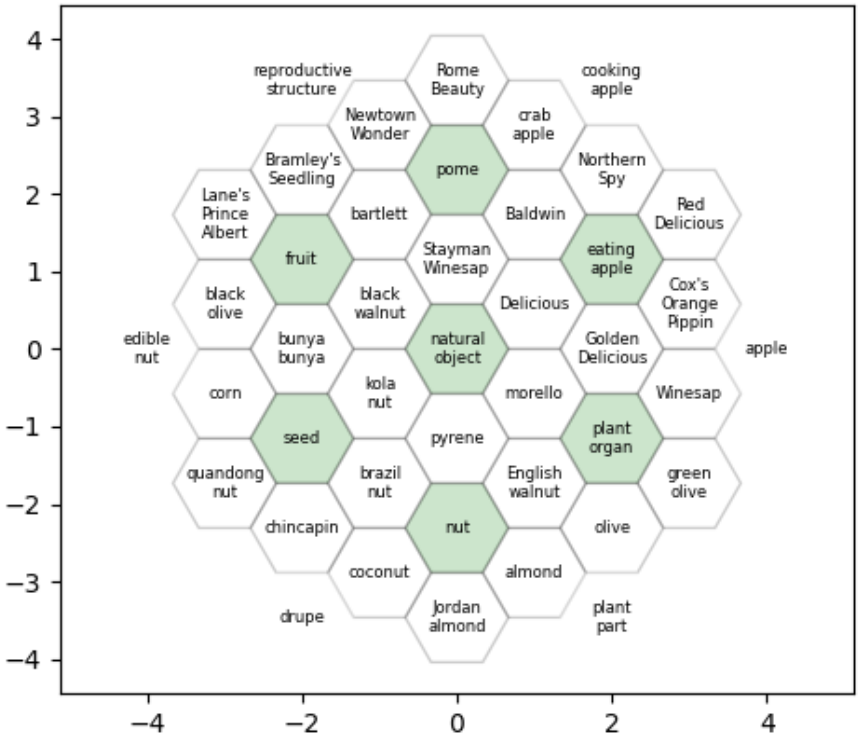


Figure 8.3: An improper computer generated Raatsel with the initial word *Apple*.

Results

The time-expanded rating method was unable to terminate within a reasonable time on the 16 Sudoku instances. The amount of possible states indeed grew exponentially quick. A clear difference can be seen between the amount of states for a beginner versus a medium Sudoku in Figure 8.5. In the beginner instance the amount of states grows much more quickly. The more numbers that can be placed at a given time, the more states that can be reached the next time-step.

Another interesting observation is that the medium Sudoku at time-step 4 only has 1 state. No matter in what order you try to solve this puzzle, this state will always be reached. This is valuable information for puzzle generators, and can be used by expert puzzle setters to improve their puzzle quality.

While this method is unable to calculate a difficulty rating for the Sudoku instances, it does provide new information about the solving process, and can be used alongside existing rating methods.

8.4. Rating Raatsel

The rating method will also be applied to the example Raatsel in Figure 4.2. Because there is a limited number of Raatsels available, there has been no research done on the actual difficulty based on human solving times. The Raatsel is included in this experiment mainly to see whether this method truly works on generic puzzles. A Raatsel always consists of 37 steps, making the largest amount of states at a time-step $\binom{37}{18} \approx 1.77 \times 10^{10}$. This is more feasible, and it is expected the method will terminate for the example Raatsel.

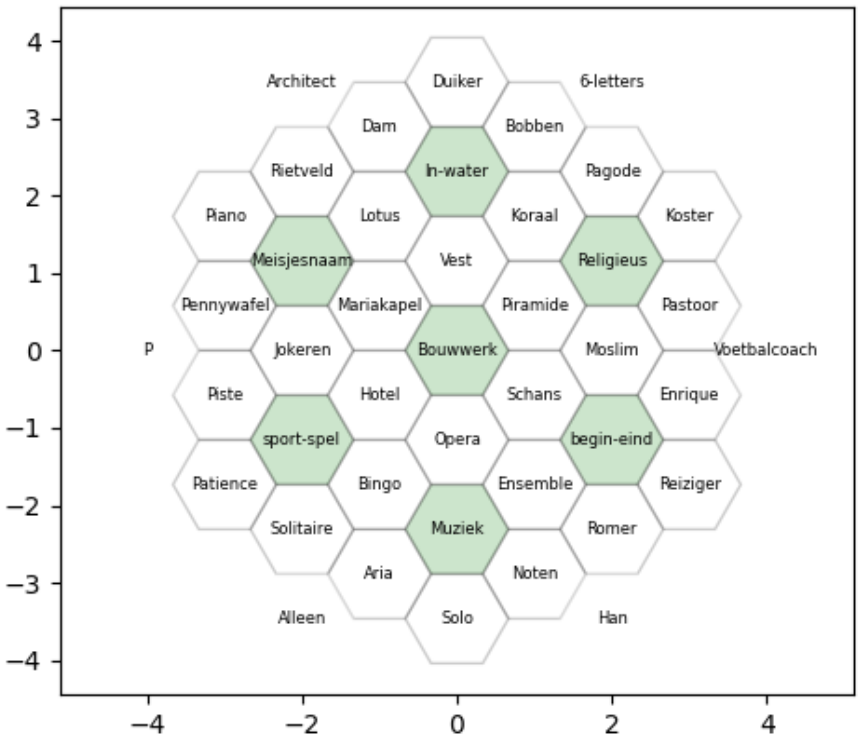


Figure 8.4: A new (Dutch) Raatsel generated from two hand-crafted Raatsels which had significant overlap.

Results

Because Raatsel was implemented in the solving framework explained in Section 5.1, the rating method worked seamlessly. Fortunately, the time-expanded rating does terminate for the example Raatsel.

Figure 8.6 shows the amount of states the example raatsel can be in at each time-step. It can be seen that the amount of states is increasing up to $t = 25$. After that point, there is no more information to be deduced in any of the states, and the amount of states shrinks until the single solved state is finally reached. Even thought we can not assign a numerical score to this instance, it can be concluded this instance is not trivial to solve, as the maximum amount of states does not come close to the theoretical amount.

The Raatsel always having 37 steps makes it computationally feasible, while the Sudoku search space grows too quickly. The time-expanded rating is therefore better suited for Raatsel than for Sudoku.

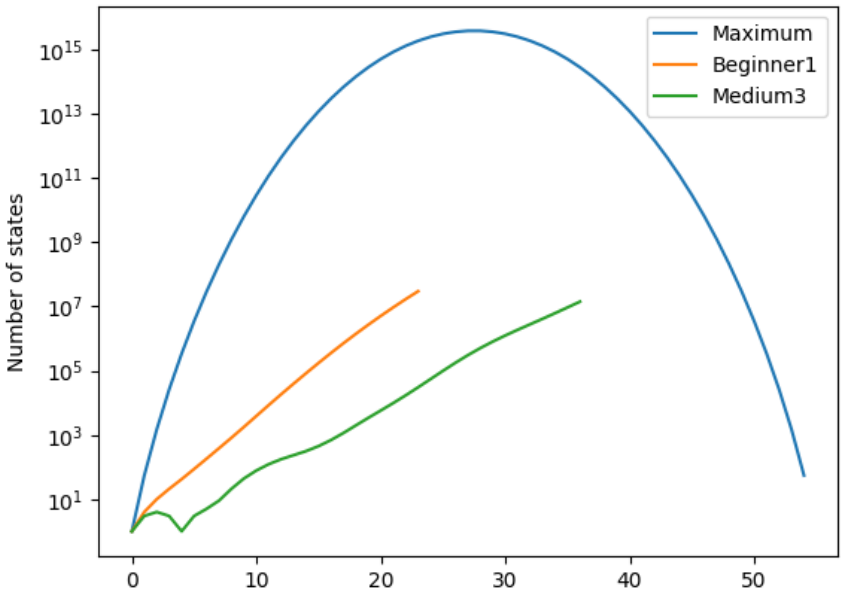


Figure 8.5: The amount of states per time-step for two different Sudoku instances.

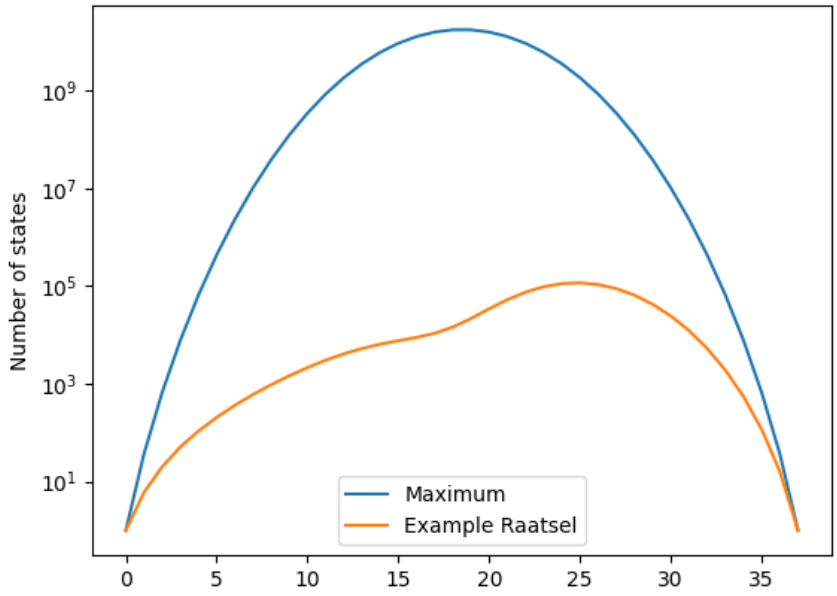


Figure 8.6: The amount of states per time-step for the Example Raatsel.

9 Conclusions

In this final chapter we will reflect on the outcomes of the experiments and answer the research questions.

9.1. Raatsel Introduction and Solving

The Raatsel puzzle was introduced as the *RAATSEL-SOLVE* decision problem. Once the strategies for *RAATSEL-SOLVE* were defined, it was integrated into the solving framework.

With the abstraction of the truth matrix M, the strategy-solver could solve Raatsels in the same way a human would. We conclude that a language-based puzzle can be solved with a strategy-solver, given that the linguistic aspect is provided to the solver.

9.2. Raatsel Generation

The proposed Subgraph Isomorphism method works correctly and generates proper Raatsel instances within a reasonable timeframe that matches state-of-the-art Sudoku generators. The generated Raatsels were not strategically solvable as the quality of the clues was lacking. WordNet is insufficient to create fun and engaging Raatsels. It contains very factual relations (e.g. An *Apple* is a *Fruit*), but to create interesting Raatsels more creativity is needed in the labeling. (e.g. An *Apple* is *Red*, *Sweet* or *A Company*)

This is not a problem with the method, but a problem with the data on which the method operates. As seen, when given hand-crafted clues as input, a new Raatsel could be created. Large puzzle companies have manually made databases of relations, but keep them private for commercial use. Applying these high-quality databases would result in more creative, and most importantly more fun Raatsels.

9.3. Time-Expanded Rating

We have seen that the number of states grows exponentially based on the amount of steps the puzzle has left. The difficulty of the instance also plays a role, with the more difficult instances having fewer states.

It is more suitable for Raatsels, which have fewer time-steps than Sudoku. Since the Raatsel is on the lower end of the amount of symbols that need to be placed, this method is not expected to work on most puzzles.

While the time-expanded rating method was unable to finish on all Sudoku instances, it is capable of revealing a curious property of a puzzle instance. If at any time-step there is only a single state for the solver to be in, every player who attempts to solve the puzzle will be guaranteed to encounter that state. This is valuable information for experienced puzzle setters which they can use to create more engaging puzzles.

The time-expanded rating method is not a substitution for existing rating methods, but can be used alongside to reveal interesting properties about generic puzzle instances.

9.4. Final Remarks

We have shown that strategy-solving is indeed a feasible method to implement a generic puzzle solver, as it can handle language and logic based puzzles. It most closely resembles human solving, showing a clear relation between difficulty for computers and difficulty for humans. Strategy-solving was not able to completely remove the language aspect from the Raatsel, but it could be applied after introducing an abstraction of the language.

The Subgraph Isomorphism reduction has shown to be a viable method to generate strategy-solvable Raatsels. With our solving and rating methods working on both the Sudoku and Raatsel, and our generation method creating strategy-solvable Raatsels, we conclude that the concept of strategy-solvability can be used to solve, generate and rate human-solvable generic pen and paper puzzles.

9.5. Future work

We propose the following improvements to Raatsel generation, possible continuations towards generic puzzle generation, and applications to other fields of research.

Improved Raatsel Generation

An issue with the proposed method for Raatsel generation is that the separation of generating the truth matrix and finding the words makes it unable to find nearly valid Raatsels. Instead, the subgraph isomorphism solver could be expanded to not include patterns leading to unsolvable Raatsels. This would allow it to report instances that are nearly valid, but missing some words. A human could then select the final words from their own memory, instead of the (often incomplete) language graph. This human-machine hybrid method could allow for a much higher success rate for Raatsel generation.

Expand to More Puzzles

The integration of Raatsel and Sudoku into the framework went seamlessly, but there is a plethora of other puzzles that currently do not have any strategies defined. While there is no reason to suspect these puzzles being unable to be integrated in the framework, more research has to be done. Especially on puzzles that follow a different structure such as the *Hashiwokakero* (*Bridges*).

Converting Strategies Into Constraints

The main issue with constraint solvers is that their workings are virtually unexplainable in terms of human solving. Currently, the rules are implemented as constraints, which are by definition enough information to solve a puzzle instance. Human solving however could be simulated by instead implementing the *strategies* as constraints. Instead of enforcing that a row of Sudoku cells must contain all different symbols, we can say that when eight different numbers are already placed, the ninth must be the remaining one (*Naked Single*). While a subtle difference, this approach could continue the Logical Modeling generation proposed by Nishikawa and Toda[14] and allow generic instance generation based on pre-defined strategies.

Other Applications

Not only puzzles can be formulated as strategy-solvable problems, other decision problems can as well. A possible application of strategy-solvability is in the field of Software Validation. It could be possible to model a well-known problem such as *Shortest Path* and invent strategies for solving it. Instances could then be generated that are solvable with respect to a certain set of strategies. Varying this set of strategies would allow us to generate instances that *require* a specific strategy to solve. This is very useful when validating a piece of software that is supposed to solve the Shortest Path problem. For example creating an instance that could only be solved with the *Check if start is destination* strategy. Fields which require some form of instance generation could benefit greatly from this strategy-solving approach.

References

[1] T Boothby, L Svec, and T Zhang. “Generating Sudoku puzzles as an inverse problem”. In: *Math. Contest Model.* 24 (2008).

[2] Cameron Browne. “Uniqueness in Logic Puzzles”. In: *Game & Puzzle Design* 1.1 (2015), pp. 35–37.

[3] Tom Davis. *The mathematics of Sudoku*. 2006.

[4] David Eppstein. “Subgraph isomorphism in planar graphs and related problems”. In: *Graph Algorithms and Applications I*. World Scientific, 2002, pp. 283–309.

[5] Lianne V Hufkens and Cameron Browne. “A functional taxonomy of logic puzzles”. In: *2019 IEEE Conference on Games (CoG)*. IEEE. 2019, pp. 1–4.

[6] Graham Kendall, Andrew Parkes, and Kristian Spoerer. “A survey of NP-complete puzzles”. In: *ICGA Journal* 31.1 (2008), pp. 13–34.

[7] Raph Koster. *Theory of fun for game design*. ” O’Reilly Media, Inc.”, 2013, p. 44.

[8] Tung-Ying Liu, I-Chen Wu, and Der-Johng Sun. “Solving the slitherlink problem”. In: *2012 Conference on Technologies and Applications of Artificial Intelligence*. IEEE. 2012, pp. 284–289.

[9] Timo Mantere and Janne Koljonen. “Solving, rating and generating Sudoku puzzles with GA”. In: *2007 IEEE congress on evolutionary computation*. IEEE. 2007, pp. 1382–1389.

[10] Ciaran McCreesh, Patrick Prosser, and James Trimble. “The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants”. In: *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*. Ed. by Fabio Gadducci and Timo Kehrer. Vol. 12150. Lecture Notes in Computer Science. Springer, 2020, pp. 316–324. DOI: 10.1007/978-3-030-51372-6_19. URL: https://doi.org/10.1007/978-3-030-51372-6%5C_19.

[11] WJM Meuffels and Dick den Hertog. “Puzzle—Solving the Battleship puzzle as an integer programming problem”. In: *informs Transactions on Education* 10.3 (2010), pp. 156–162.

[12] Nicholas Nethercote et al. “MiniZinc: Towards a standard CP modelling language”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[13] Nikoli Puzzles. *Nikoli Puzzles (japanese)*. [Online; accessed 3-March-2022]. 2022. URL: https://www.nikoli.co.jp/ja/puzzles/.

[14] Kohei Nishikawa and Takahisa Toda. “Exact Method for Generating Strategy-Solvable Sudoku Clues”. In: *Algorithms* 13.7 (2020), p. 171.

[15] Radek Pelánek. *Human problem solving: Sudoku case study*. Tech. rep. Technical Report FIMURS-2011-01, Masaryk University Brno, 2011.

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

[17] Sudoku of the Day. *Sudoku of the Day*. [Online; accessed 29-March-2022]. 2022. URL: https://www.sudokuoftheday.com/difficulty.

[18] Princeton University. *About WordNet.* URL: https://wordnet.princeton.edu/citing-wordnet.

[19] S Zama and I Sasano. *Proposal and implementation of Sudoku problem generation method suitable for initial placement (japanese)*. Mar. 2016.

[20] Giulio Zambon. *Sudoku Programming with C*. Springer, 2015.

A

Solution to Example Raatsel

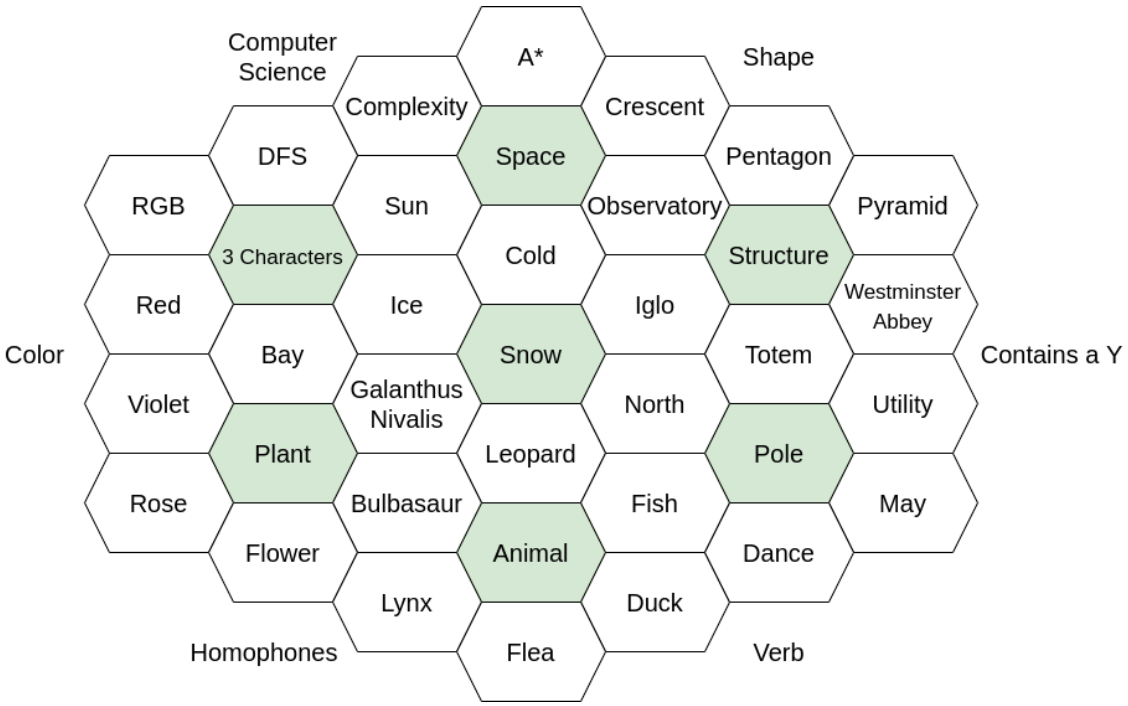


Figure A.1: The solution to the Raatsel presented in Figure 4.2.

B

Minizinc Sudoku Solver

```
include "alldifferent.mzn";

int: S = 3;
int: N = 9;

set of int: numbers = 1..9;

array[1..N, 1..N] of 0..N: clues;
array[1..N, 1..N] of var numbers: solution;

% Add clues to solution
constraint forall(i,j in numbers)(
  if clues[i,j] > 0 then solution[i,j] = clues[i,j] else true endif );

% Alldifferent for rows
constraint forall(i,j in numbers)(
  alldifferent([solution[i,j] | j in numbers]));

% Alldifferent for columns
constraint forall(i,j in numbers)(
  alldifferent([solution[i,j] | i in numbers]));

% Alldifferent for subgrids
constraint forall(x, y in 1..S)(
  alldifferent([solution[(x-1) *S + x1, (y-1)*S + y1] | x1, y1 in 1..S]));

solve satisfy;

output [
  show(solution)
];
```