

A General Grading Method for Sudoku Puzzles Based on Minimal Unsatisfiable Sets (MUSes)

Benjamin Simon Jonsson

220023823

Supervisor: Ruth Hoffmann

University of St Andrews

School of Computer Science

14 August 2023

Abstract

The varied difficulty of Sudoku puzzles offers an exciting domain for measuring the difficulty of logic problems for humans. Current approaches focus on directly modelling how humans solve Sudoku puzzles by summing the perceived difficulty of the solving techniques used when solving them. This paper proposes a more general method for measuring the difficulty of Sudoku puzzles using a solving algorithm based on Minimal Unsatisfiable Sets (MUSes). Based on insight from existing grading methods, the formula thoughtfully accounts for the different aspects of difficulty in solving a Sudoku puzzle by combining the number and size of MUSes. The paper compares the formula grades to puzzle grades from notable Sudoku publishers. The results indicate that MUSes can assess the difficulty level of Sudoku puzzles. Additionally, this paper presents a Backtracking Search algorithm that can generate new Sudoku (and Miracle Sudoku) puzzles to create a library of puzzles for further statistical analysis.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 14694 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

B.S. Jonsson

A handwritten signature in black ink, appearing to read 'B.S. Jonsson', with a stylized, cursive script.

15/08/2023

Acknowledgement

I would like to thank Ruth Hoffmann for her help and guidance with this project. I am grateful for the enthusiasm you showed and the interesting insight you provided into Sudoku puzzles. I would also like to thank Ian Gent for his time and advice during the final few weeks.

Table of Contents

1. Introduction.....	6
1.1 Overview	6
1.2 Aims and Objectives	7
1.3 Structure	8
2. Theory	9
2.1 The Sudoku Puzzle.....	9
2.2 The Miracle Sudoku Puzzle	9
3. Background	11
3.1 Puzzle Generation	11
3.1.1 The Backtracking Algorithm	11
3.1.2 Constraint Propagation	12
3.1.3 Other Approaches	12
3.2 Puzzle Difficulty	13
3.2.1 Grading a Sudoku Puzzle.....	14
3.2.2 Logical Strategies	15
3.2.3 Miracle Sudoku Difficulty	16
3.3 Minimal Unsatisfiable Sets (MUSes).....	17
3.3.1 Explanations and Difficulty as MUSes.....	17
3.3.2 DEMYSTIFY	18
4. Design and Implementation	20
4.1 The Backtracking Search Algorithm.....	20
4.1.1 Design choices	20
4.1.2 Generate a Filled Sudoku Puzzle	21
4.1.3 Generate a Starting Sudoku Board.....	23
4.1.4 Modifications for the Miracle Sudoku Puzzle	26
4.2 The Grading Method.....	26
4.2.1 Grade Formula and Design Choices	27
4.2.2 The Grading Algorithm	29
5. Testing.....	32

5.1 Functionality Testing	32
5.1.1 Runtime of the Algorithms	32
5.1.2 Testing the Generation and Solving Process	32
5.2 Invalid Board.....	35
6. Evaluation	36
6.1 Analysis of the Grading Metric.....	36
6.2.1 Variable Weights	36
6.2.2 Grade Level.....	37
6.2.3 Performance.....	39
6.2.3 Limitations of Variable Weights.....	41
6.3 Original Objectives	42
7. Conclusions.....	43
7.1 Summary	43
7.2 Limitations and Future Work.....	43
Appendix.....	49
Appendix A: User Manual	49
A1: Running the Whole Process.....	49
A2: Testing Pre-Generated Puzzles	52
A3: Running the Process for Miracle Sudoku	53

1. Introduction

1.1 Overview

Creating and solving Sudoku puzzles is interesting for two reasons. First, Sudoku is an NP-complete problem and offers an attractive domain to test and challenge different types of solving algorithms (Kendall, Parkes, & Spoerer, 2008; Garey & Johnson, 1979; Yato & Seta, 2003). Most Artificial Intelligence (AI) algorithms are focused on the efficiency of solving the problem and can quickly solve a Sudoku, while others focus on guiding human players (Russel & Norvig, 2015; Lloyd & Amos, 2019; Simonis, 2005; Stuart, 2008). The latter provide interesting cases of designing AI systems that output solving explanations that humans can reasonably understand. For instance, Espasa et al. (2023) use Minimal Unsatisfiable Sets (MUSes) to solve Sudoku puzzles as a series of logical deductions with interpretable explanations.

Second, Sudoku puzzles vary significantly in difficulty and thus offer an exciting domain for measuring the difficulty of logic problems for humans. The most naïve approach is to base the difficulty on the number of clues (filled cells), i.e., the more clues a puzzle has, the easier it is. While this is sometimes true and often is for easier puzzles, it is the complexity and number of logical decisions needed to solve a given puzzle that determines its difficulty (Stuart, 2007; Green, 2006; Pelánek, 2014; Tomlinson, 2005; Fowler, 2009).

The primary method for grading puzzles involves directly modelling the way humans solve puzzles by creating a Sudoku solver that uses the same logical solving techniques as humans and assigning numeric values to the perceived difficulty of the solving techniques, which can be summed to extract an overall metric of difficulty (Green, 2006; Stuart, 2008; Tomlinson, 2005). The work in Pelánek (2011) combines the difficulty of individual logical solving steps and the level of dependency between these individual steps, i.e., whether steps can be solved independently or not. A significant limitation of these systems is that they require an extensive library of sudoku puzzles and a hard-wired and ranked list of human-solving techniques. Additionally, many approaches, such as in Stuart (2007), have access to a lot of human-solving data, which allows the grading metrics to be calibrated based on the correlation of human-solve time and perceived difficulty.

If Sudoku solvers that use MUSes can solve puzzles in the same way as human-produced guides (like in Espasa et al., 2023) and current grading methods are based on human solving techniques (like in Stuart, 2008), then it's reasonable to consider whether a grading approach that uses MUSes can be developed to measure puzzle difficulty. This could reveal whether the difficulty of a logic problem, such as Sudoku, can be determined by some combination of the number and size of the MUSes used to solve it. If so, this would present an alternative and more general method to grading a pen-and-paper puzzle without the need for different variants of a puzzle solver based on puzzle specific solving techniques.

1.2 Aims and Objectives

This paper aims to provide a more general method for measuring the difficulty of Sudoku puzzles by combining the insight from existing grading methods with a solving algorithm based on MUSes. The goal is to create an algorithm that can generate a Sudoku puzzle and grade it based on the number and size of MUSes used to solve the puzzle (with Espasa et al. (2023)’s DEMYSTIFY algorithm) and the grading method presented in Stuart (2007).

The primary objectives set out when the research began were to (1.) generate Sudoku and (2.) generate Miracle Sudoku puzzles, and (3.) use DEMYSTIFY to provide step-by-step solving explanations to the puzzles that are meaningful to players, along with a grade rating system. A secondary objective was determining the number of unique Miracle Sudoku puzzles. However, two things quickly became apparent during the project – miracle sudoku is not that complicated or interesting after the first puzzle, and MUSes have not been used to grade sudoku puzzles before, with the grading of sudoku puzzles being a fascinating aspect of quantifying what determines the difficulty of a problem for humans. As the project progressed it became clear that the area to guide the research toward is using MUSes in some capacity to provide a difficulty measurement of Sudoku puzzles.

Therefore, the main objectives of this paper developed into (1.) generate Sudoku puzzles using a constraint solver and (2.) use a solver based on MUSes to provide a difficulty metric for the generated puzzles. In doing so, this paper provides a system that allows players to generate Sudoku puzzles with a general difficulty metric derived from combining the number and size of MUSes used to solve the puzzle, in a way that matches the grading formula in Stuart (2007). Achieving primary objective (1.) involves representing Sudoku generation as a Constraint Satisfaction Problem (CSP) and solving it in two parts based on the guidance in Stuart (2007) – generate a filled Sudoku puzzle and solving backward to get the starting Sudoku puzzle. Achieving the primary objective (2.) involves developing a thoughtful method of combining the MUSes to solve each puzzle and calculate a reasonable difficulty metric.

The secondary objective of this paper is to generate Miracle Sudoku puzzles, as they offer an exciting challenge when solving a Constraint Satisfaction Problem (CSP).

The main contribution of this work is to expand on the work of Bogaerts, Gamba, Claes, and Guns (2020) and Espasa et al. (2023), who use MUSes as a basis for solving puzzles in a way that is interpretable by humans, by providing insight into how the size and number of MUSes used when solving the puzzle can be a method to measuring the difficulty of a Sudoku puzzle. The grading metric presented also aims to provide a reasonable alternative to grading Sudoku puzzles – which is the “greatest concern of a puzzle maker” (Stuart, 2007, p. 3).

1.3 Structure

This paper proceeds as follows. Section 2 provides some theory, explaining how to play Sudoku. Section 3 provides background information on existing sudoku-generating and solving algorithms, the current notable methods used to grade sudoku puzzles, and the concept of explanations as MUSes. Section 4 explains the design and implementation of the sudoku-generating algorithm and the sudoku-grading algorithm, along with the method of grading puzzles based on MUS size. Section 5 discusses testing the algorithm's functionality, and its efficiency (runtime). Section 6 evaluates the ability of the grading algorithm to grade sudoku puzzles accurately, along with a critical appraisal of meeting the original objectives. Section 7 concludes and discusses future work.

2. Theory

2.1 The Sudoku Puzzle

Sudoku is a logic-based puzzle that involves filling in a 9x9 grid of cells with numbers from 1-9. The Sudoku board is divided into nine 3x3 sub grids (or boxes). The goal of the puzzle is to fill in the board so that every row, column, and 3x3 box contains the numbers 1-9 without any repetition of a number.

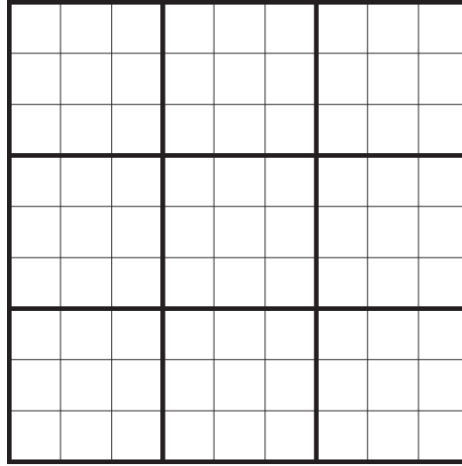


Figure 1: Empty Sudoku Grid

A Sudoku puzzle will contain some initial numbers (called givens or clues) and a user will try to fill in the remaining empty cells according to the following rules: (1.) Cells in the same row cannot contain the same number, (2.) Cells in the same column cannot contain the same number, (3.) Cells in the same 3x3 box cannot contain the same number.

Additionally, for a Sudoku puzzle to be valid there must only exist one possible solution to the current configuration of the puzzle (i.e., the starting puzzle must have a unique solution). From a mathematical perspective, Felgenhauer & Jarvis (2005) prove that the number of possible fully filled Sudoku puzzles is 6670903752021072936960. Furthermore, McGuire et al. (2012) uses an exhaustive computer search to show that a Sudoku puzzle must have at least 17 filled cells for it to have a unique solution. This notion is furthered by Puskas (2016), who looks at puzzles which contain less than 17 clues and proves why they cannot possess a unique solution.

2.2 The Miracle Sudoku Puzzle

Miracle Sudoku is an intriguing variant of Sudoku created by Mitchell Lee. All the rules from generic Sudoku apply. But, now there are three new rules added: (1.) Cells that are separated by a king's move in chess cannot contain the same number, (2.) Cells that are separated by a knight's move in

chess cannot contain the same number, (3.) No two orthogonally adjacent cells may contain consecutive numbers.

Below is an example of the Miracle Sudoku puzzle displayed on Cracking the Cryptic, whereby the constraints for number 1 (i.e., where a 1 cannot be placed) are highlighted in colour on the board (blue: generic constraints, green: king move, pink: knight move, yellow: consecutive numbers) (Anthony, & Goodliffe, 2020)..

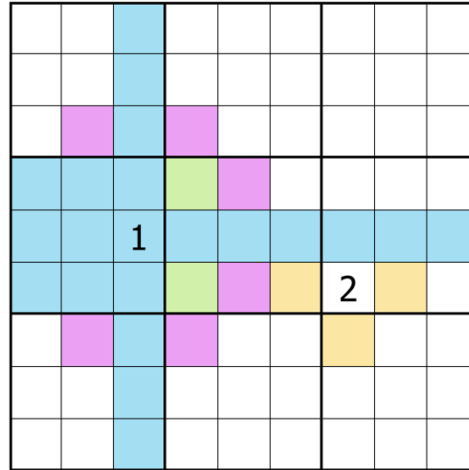


Figure 2: Miracle Sudoku Constraints

From a mathematical perspective the possible number of fully filled Miracle Sudoku puzzles is 72 (McCarthy, 2020). Additionally, a Miracle Sudoku must have at least 2 clues, and these 2 clues must be in distinct rows and columns, for it to have a unique solution. From this, there are a total of 174,816 ways to set a Miracle Sudoku board with 2 clues such that the puzzle is uniquely solvable. However, as there are only a possible 72 fully filled Miracle Sudoku puzzles, many of the starting positions will be identical symmetrically (i.e., have the same two numbers in different positions and are the same when rotated or mirrored). Moreover, even the starting puzzles which are not identical symmetrically, many of them will quickly converge on each other as the puzzle is solved (McCarthy, 2020).

3. Background

3.1 Puzzle Generation

According to Stuart (2007), a good Sudoku puzzle must meet three standards: (1.) A puzzle must have a unique solution, (2.) There must be a logical solution to the puzzle (i.e., a solution without guessing), and (3.) A puzzle must be aesthetic (i.e., have symmetry).

The first stage of a generating algorithm is to fill the Sudoku puzzle with a solution, which means filling in each cell with numbers from 1 to 9 such that the rules of Sudoku are satisfied (Stuart, 2007). In this way, generating a Sudoku puzzle can be viewed as a Constraint Satisfaction Problem (CSP) (Simonis, 2005; Russel & Norvig, 2015). In a CSP, a value from a limited domain (numbers 1-9) is assigned to each variable (empty cell) such that all constraints relating the variables (Sudoku rules) are satisfied (Brailsford et al., 1998). The usual methods for solving a CSP are Backtracking, Constraint Propagation, or a combination of Backtracking and Constraint Propagation (Russel & Norvig, 2015; Norvig, n.d.; Liu, 1998).

The second stage of a generating algorithm is to remove numbers from cells to create the starting puzzle but to ensure that after each subtraction, only one solution exists (satisfying the first standard of a good puzzle). In addition, the algorithm should also remove numbers from cells in such a way as to achieve symmetry (satisfying the second standard) (Stuart, 2007). Finally, the algorithm should be able to grade the Sudoku, which is the most challenging aspect (Meng & Lu, 2011; Stuart, 2007). This final aspect of the generating algorithm is the priority of this paper.

It is important to note that Sudoku-generating algorithms will vary based on the different aspects the puzzle generator prioritizes, such as the efficiency of the generator or desired characteristics of the puzzle (like symmetry or difficulty). This section discusses notable works on Sudoku-solving algorithms or relevant CSP solving literature. Note that the logic utilized in Sudoku-solving algorithms can be applied to puzzle generation as the Sudoku-solving algorithm would start from an empty board instead of a partially filled board.

3.1.1 The Backtracking Algorithm

A *generate-and-test* algorithm solves a CSP in the most straightforward, and naive, means possible. This method will systematically generate every possible assignment of values to variables and then test to see if the assignment of values satisfies the constraints (Barták, 2005; Liu, 1998). A more efficient approach would be to only examine those possible assignments of values that lead to a solution. A *backtracking* search algorithm is a depth-first search approach that tries to assign values to variables one at a time until no legal values are left to assign to the current variable in the systematic search, at which point the algorithm backtracks to try another value for the previous variable (Russel & Norvig, 2015). In this way, the *backtracking* algorithm does not examine all possibilities but only

those that will lead to a solution – it searches the solution tree (all possible solutions) and explores as far as possible along each branch for an answer before backtracking (Lloyd & Amos, 2019; Herimanto, Sitorus, & Zamzami, 2019; Cormen, 2009; Barták, 2005).

In Sudoku generation, a *backtracking* search algorithm would start by finding the first empty cell of the Sudoku grid. Next, the algorithm utilizes a recursive function to try and fill the selected empty cell with a valid number from 1 to 9. It tries each number and checks if it violates any of the Sudoku rules. If the inputted value violates any of these rules, the algorithm will backtrack and try the next number (Russel & Norvig, 2015). If the number is valid, the algorithm will move onto the next empty cell and repeat the process. This systematic search will continue until the Sudoku grid is fully populated with a valid puzzle layout.

3.1.2 Constraint Propagation

Instead of searching for the correct solution, an algorithm can also use *constraint propagation*. The focus of *constraint propagation* is to reduce the domain size of the variables in the CSP; when assigning a value to a variable, this value can also reduce the number of legal values from the domain for another variable, and the effect can continue to propagate through the constraints to impact other variables. (Russell & Norvig, 2015). For example, if a 2 is assigned to a cell (a.), this immediately means that 2 cannot go in any of the other cells in the same row, column, and 3x3 box as cell (a.). Therefore, it is possible to remove 2 from all associated cell domains, with the effect of the removals propagating to affect other cells. However, while *constraint propagation* may effectively reduce the domain size of variables and thus simplify the CSP, it is usually ineffective at solving the whole problem (Liu, 1998). In the case of Sudoku, it will often get stuck at some point during the solving process.

Backtracking search can be combined with *constraint propagation* by either using both in unison with one another or, as in Norvig (n.d.), using *constraint propagation* until the algorithm gets stuck and initiate a *backtracking search* process. Sometimes, the *constraint propagation* algorithm can solve/populate the whole Sudoku puzzle without needing a *backtracking search* (Russell & Norvig, 2015).

3.1.3 Other Approaches

The objective of the Exact Cover Problem, which is a type of CSP, is to determine whether it is possible to choose a specific subset from a given set in a means where every element from that given set is included in precisely one of the selected subsets (Chabert & Solnon, 2020). The most sophisticated approach to solving this problem is described in Knuth (2000), where the author introduces a "dancing links" data structure to implement Algorithm X (called the DLX), which applies

an efficient 'brute-force' backtracking algorithm for the Exact Cover Problem (Lloyd & Amos, 2019; Knuth, 2000). Because any Sudoku puzzle can be converted into an Exact Cover Problem, the DLX provides an efficient approach to solving Sudoku puzzles (Fletcher, Johnson, & Morrison, 2007; Hunt et al., 2007).

Other important work relating to Sudoku algorithms includes entropy minimization (Gunther & Moon, 2012), using formal logic (Weber, 2005), evolutionary algorithms (Mantere & Koljonen, 2007; Wang, Yasuda, & Ohkura, 2015; Deng & Li, 2013; Segura, Penã, Rionda, & Aguirre, 2016), an artificial bee colony algorithm (Pacurib, Seno, & Yusiong, 2009), and partial swarm optimization (Moraglio & Togelius, 2007; Hereford & Gerlach, 2008).

It is important to note that automated puzzle-generation methods prioritize speed and efficiency rather than replicating human-solving techniques (Lloyd & Amos, 2019). These methods utilize propagators and search trees to solve puzzles quickly (Simonis, 2005). Currently, the most efficient algorithms for solving Sudoku puzzles are Ant Colony Optimization for the largest and most complicated Sudoku puzzles, while direct search algorithms such as the 'Dancing Links' Algorithm perform best on easier and smaller Sudoku puzzles (Lloyd & Amos, 2019).

On the other hand, rule-based algorithms designed to mimic human-solving techniques suggest alternative strategies, such as tracking possible values, pattern recognition, and logical deduction, with small search trees as a last resort (Espasa et al., 2023; Stuart, 2008). In general, rule-based algorithms tend to be slower. However, they are more intuitive to understand as they build/solve puzzles based on how humans would logically do so. Rule-based algorithms also make it easier to grade the puzzles as the algorithm can simultaneously measure the puzzle's difficulty by assigning grade points to the strategies described by puzzle-solving instructions (Pelánek, 2014).

3.2 Puzzle Difficulty

This section first discusses what determines the difficulty of a Sudoku puzzle, based on the analysis by Stuart (2007) and Pelánek (2014), where the authors describe a difficulty rating metric and then evaluate the correlation of the difficulty rating and human performance (measured in time). Second, it summarizes the specific Sudoku strategies (logical deductions made to determine what value can legally go into an empty cell) grouped by difficulty level, as Stuart (2008) describes in Sudokuwiki.org. It is important to note that grading a Sudoku puzzle is subjective. While Stuart (2007) may develop a difficulty ranking of Sudoku strategies which is not too controversial, some people will inevitably find specific strategies harder to identify and solve than others as people have different innate abilities and different degrees of skill in identifying strategies. Thus, some puzzles will always be more difficult for some people, indicating the complexity of creating a Sudoku grading system.

3.2.1 Grading a Sudoku Puzzle

Two main aspects of solving a Sudoku puzzle influence its difficulty. Firstly, the dependency between puzzle-solving opportunities. For example, it will be easier to find a solution when a group of cells can be solved independently of each other (i.e., solve the logical steps in parallel) rather than when a group of cells must be solved dependently on each other (i.e., solve the logical steps in strict sequential order) (Pelánek, 2014). As Stuart (2007) explains, it is possible to extract some difficulty measures by counting the number of opportunities to solve at all stages of finding a solution to the puzzle. In this way, so-called 'bottlenecks' will occur when there is only one or a few chances to make a valid deduction.

The second aspect is the difficulty of each logical step during the solving process (Pelánek, 2014). Specifically, the complexity of the strategy required to identify and solve a solving opportunity—this is the typical approach to grading Sudoku puzzles (Green, 2006; Pelánek, 2011; Tomlinson, 2005; Fowler, 2009). For example, a puzzle that requires identifying empty cells where only one number is possible (so-called 'eyeballing') will be easier than a puzzle that requires detailed notetaking to identify possible empty cells a number might go in (Stuart, 2007). In this way, the 'strategies' in Sudoku are the logical deductions or thought processes a player makes when attempting to fill an empty cell.

To quantify the difficulty of a strategy, Stuart (2007) first ranks each strategy by three factors; (1.) How difficult the strategy is to identify in reality, (2.) how often the strategy is needed, and (3.) how many numbers does the strategy eliminate. Second, by carefully assigning weights and scores to each strategy, a total score is derived from the whole solve route (Stuart, 2007). It is important to note that this difficulty metric is not a strict mathematical formula; it is founded on a subjective view of what makes a puzzle difficult.

However, over a large amount of statistical analysis based on data from people submitting their solving times to the Daily Competition Sudoku (2000 to 3000 submissions a day over 330 days), Stuart (2007) calibrates the difficulty of the puzzles based on how long it takes people to solve the puzzle. For example, if solving a puzzle requires more complicated strategies (according to the metric for difficulty), it should take people longer to solve than a puzzle with easier strategies. Overall, the difficulty of a puzzle is calibrated and ranked on the difficulty of strategies used and the solve times (Stuart, 2007). 'Sudoku of the Day' takes a similar approach when grading puzzles whereby the creators assign a 'cost' to each Sudoku technique (Tomlinson, 2005). The harder the strategy, the higher the cost. Then, based on the total cost of all strategies used in the puzzle, it is classified as Gentle, Moderate, Tough, or Diabolical.

3.2.2 Logical Strategies

This section covers all the different types of strategies included in the basic, tough, and diabolical strategies DEMYSTIFY uses to solve puzzles. The purpose is to explain how the complexity increases with the difficulty of strategy, as this will provide an important insight into puzzle grading methods. For a detailed explanation of each strategy, please see sudokuwiki.org.

Basic Strategies

The starting point is 'Eyeballing,' which covers two fundamental Sudoku strategies.

- The ***Last Remaining Cell*** strategy is when you identify an empty cell which is the last possible place a certain number can go in the row, column, or box.
- The ***Last Possible Number*** strategy is when you identify a value for an empty cell by noticing that every other number from 1 to 9 apart from that number is present in either the row, column, or box.

Then there are the *Naked Candidates* (*Naked Single*, *Naked Pair*, *Naked Triple*, and *Naked Quad*) and the *Hidden Candidates* (*Hidden Single*, *Hidden Pair*, *Hidden Triple*, and *Hidden Quad*) strategies. Note that a candidate is a possible value for an empty cell. For example, in the *Last Possible Number* strategy, the value you identified for the empty cell would be the only candidate in that cell (Stuart, 2008).

- The ***Naked Candidates*** strategy reasons that if one, two, three, or four cells in the same row, column, or box contain the same one, two, three, or four candidates (possible values), then those candidates can be removed from all cells in that corresponding row, column, or box.
- The ***Hidden Candidates*** strategy reasons that if two, three, or four cells within a row, column, or box share the same two, three, or four candidates, then eliminate all other candidates from those cells.

While all these strategies fall under 'Basic,' it must be noted that Naked and Hidden Triples/Quads are not necessarily easy to spot and use. For example, in sudokuoftheday.com, 100 difficulty points are assigned to the Last Remaining Cell strategy (referred to as 'Single Candidate' in their table), while 2400 points are assigned to the Hidden Triple and 5000 points to the Naked Quad. The increase in points is substantial.

Tough Strategies

The explanations of tough strategies are summarised, please see sudokuwiki.com for detailed explanations with annotated diagrams.

- The ***X-Wing*** strategy reasons that if there are two candidates for an empty cell in each of two different units of the same type (i.e., two rows, columns, or boxes), and these candidates are

also in two other units of the same type, then all other candidates for that empty cell can be removed from the latter two units (Stuart, 2008).

- The ***Simple Colouring*** Strategy involves assigning colours (usually red or blue) to candidates for a bi-value empty cell (cells that can only take on two possible values). Then, look for cells with the same colour in the same row, column, or box, and if there is an odd number of cells with the same colour, remove the uncoloured candidate in those cells.
- The ***Y-Wing*** strategy reasons that if three cells have two candidates each, and two cells share a common candidate, then the third cell can be used to eliminate a candidate in another cell if the two cells with a common candidate are 'seen' (in the same row, column, or box) by both the third cell and the other cell you wish to eliminate the candidate from (Stuart, 2008).
- The ***Swordfish*** strategy uses the same logic as the X-Wing strategy but is extended to three columns or rows instead of two.
- The ***XYZ Wing*** Strategy is a more complex extension of the Y-Wing strategy. It reasons that if three cells (referred to as the "Extended Trio") have three different candidates between them, where one cell 'sees' the other two, and the other two have one candidate in common; eliminate the common candidate from the cells that see all three cells in the "Extended Trio" (Stuart, 2008).

Diabolical Strategies

For diabolical strategies, the explanations go beyond the scope of this paper, and are very complicated to explain. Therefore, strategies under the umbrella diabolical strategies are only be listed: X-Cycles, XY-Chain, 3D Medusa, Jellyfish, Unique Rectangles, SK Loops, Extended Unique Rectangles, Hidden Unique Rectangles, WXYZ Wing, Aligned Pair Exclusion.

3.2.3 Miracle Sudoku Difficulty

In the case of Miracle Sudoku, this paper assumes that all the puzzles will be similar in difficulty as most of them will involve the same solving process because they essentially are the same puzzles with rotations and mirroring (McCarthy, 2020). Moreover, because there are so few filled Miracle Sudoku boards, a more interesting question is how complex a Miracle Sudoku puzzle is relative to generic Sudoku puzzles rather than determining the difference in difficulty between Miracle Sudoku puzzles.

3.3 Minimal Unsatisfiable Sets (MUSes)

This section aims to explain Minimal Unsatisfiable Sets (MUS), how Minimal Unsatisfiable Sets (MUSes) can provide human-understandable explanations of logical output, and how difficulty links to the size of the MUSes. This section also provides an overview of DEMYSTIFY.

MUSes are a concept in the field of Boolean logic and Boolean Satisfiability (SAT) (Dasgupta & Chandru, 2004). Boolean formula are logical expressions constructed with variables (which are either true (1) or false (0)), Boolean operators (negation (\neg), or (\vee), and (\wedge), implication (\rightarrow), equivalence (\leftrightarrow)), and parenthesis, to logically reason about simple or complicated events (Fried, 2017). A Boolean formula is satisfiable if there exists an assignment of truth values to its variables that makes the entire formula true. Conversely, if there is no such assignment, the formula is unsatisfiable. SAT involves searching for a solution to a Boolean formula by satisfying assignments to the formula expressed in a conjunctive normal formula (CNF) (Biere et al., 2021). CSPs can be solved by converting them into SAT problems. The advantage of first expressing problems as CSP lies in its ability to represent constraints more concisely, expressing what might otherwise necessitate numerous SAT clauses (Espasa et al., 2023).

Espasa et al. (2023) state that “an unsatisfiable set of an unsatisfiable constraint problem is any unsatisfiable subset of the set of constraints of the problem”. Therefore, only problems that are already unsolvable can have unsatisfiable sets. The definition of unsatisfiable sets, usually defined on the clauses of a CNF, can be expanded to general CSP problems, which may contain many unsatisfiable sets of different sizes. Solving a puzzle with unsatisfiable sets can be viewed as trying to fill in a value to a cell and identifying all the reasons why it cannot go there, i.e., input a value into a cell that is not correct according to the puzzle's solution, and then determine the number of unsatisfiable sets to this unsolvable problem (Espasa et al., 2023). In this way, Espasa et al. (2023) solve puzzles with unsatisfiable sets to test their hypothesis, which suggests how humans' reason to solve puzzles aligns closely with unsatisfiable sets. In addition to solving puzzles, there is extensive research on other use cases for unsatisfiable sets, such as repairing knowledge bases (Mazure, Sais, & Grégoire, 1998), model checking (McMillan, 2003), or interactive applications (Junker, 2001).

3.3.1 Explanations and Difficulty as MUSes

A previous study by Bogaerts et al. (2020) has already utilized MUSes as a basis for providing interpretable explanations. In their model, they deploy a flexible enough framework that allows for a problem-specific cost function to quantify the interpretability of one explanation at a time. This model is evaluated with a size-based metric on the extracted MUSes.

Following this, Espasa et al. (2023) develop a model that uses MUSes as a basis for human-understandable explanations. Their paper reasons that people solve puzzles in steps and that, with

some limitations, MUSes can reasonably be viewed as explanations for each solving step in a puzzle, such as Sudoku.

The idea is that, by viewing puzzles as a CSP, people will find new information at each step in the solving process. This new information can be represented as removing possible values (remaining candidates) from the domain (possible valid numbers from 1-9) of a variable (empty cell). Furthermore, Espasa et al. (2023) suggest that if a player determines the value for a variable, this is the same as proving that no other values in that variable's domain can be placed in that variable. Therefore, deducing why an empty cell must take a certain value can still be interpreted as a MUS.

As explained in Espasa et al. (2023) and noted by Bogaerts et al. (2020), easy-to-comprehend explanations should only involve a small number of constraints to the problem. Looking at Sudoku, when we explain why a value cannot go into an empty cell, we usually only apply a few constraints (i.e., only a small subset of constraints is needed). This idea aligns well with the definition of a MUS, which is “something that minimizes the number of constraints that makes a problem unsatisfiable” (Espasa et al., 2023, p. 8).

According to Stuart (2007) and Pelánek (2014), the strategies required to solve the puzzles become more complex when Sudoku puzzles increase in grade rating. For example, when solving an easy puzzle, a player should only have to use Basic strategies, which have a small subset of constraints at most and can generally be easily explained. However, when solving a very hard puzzle, a player may have to use Tough or Diabolical strategies, which have a significantly larger subset of constraints (sometimes more than 20 in Diabolical strategies) and can become increasingly difficult to explain and comprehend. Based on the descriptions of what makes a strategy difficult, this would mean more constraints to the problem. If MUSes are used to solve puzzles, then as the number of constraints involved in the problem increases, the size of the constraints involved in MUS should also increase based on the requirement that that a MUS will be easier to interpret if it involves fewer constraints and harder to interpret if it involves more constraints (Espasa et al., 2023). Therefore, it may be possible to quantify the difficulty of a puzzle with a Sudoku solver that is based on MUSes by thoughtfully utilizing the size of the MUS used to solve each step of the puzzle, such as the solver in Espasa et al. (2023) called DEMYSTIFY.

3.3.2 DEMYSTIFY

There are many different algorithms to compute MUSes (Siqueira N. & Puget, 1988; Chinneck & Dravnieks, 1991; Hemery, Lecoutre, Sais, & Boussemart, 2006; Marques-Silva, Janota, & Belov, 2013). This paper uses the approach presented in Espasa et al. (2023), which uses a randomized algorithm (DEMYSTIFY) to compute MUSes for pen-and-paper puzzles. The details on how the algorithm works are beyond this paper's scope. For a detailed documentation on how it works, please see (Espasa et al. (2023), sections 3 and 4, pages 6 to 16). However, essentially DEMYSTIFY gets

explanations for why value v cannot be assigned to variable x by stating that $v = x$ and then asking their solver for a MUS (Espasa et al., 2023))

Given that inner functionality is not analysed in this paper, it is challenging to compare the efficiency and relevance of DEMYSTIFY to other puzzle-solving algorithms based on MUSes (such as in Bogaerts et al., 2020). However, this paper uses the DEMYSTIFY tool for two reasons. Firstly, my supervisor is one of the creators of DEMYSTIFY, and in coming up with the idea for the project, it was always discussed based on the output of the DEMYSTIFY algorithm. In addition, it made sense to use the algorithm I could get direct help with and easily ask for any explanations or insight. Secondly, the results of Espasa et al. (2023) indicate that their randomized algorithm can produce MUSes that balance size and practicality. In addition, the results indicate that, even on the most complicated puzzles tested, their randomised algorithm produces relatively small MUSes. Overall, DEMYSTIFY performs well on a range of pen-and-paper puzzles (Espasa et al., 2023).

4. Design and Implementation

4.1 The Backtracking Search Algorithm

This section discusses the Sudoku-generating algorithm developed in this paper. First, there is a discussion of the design decisions taken, explicitly focusing on why this paper uses a backtracking search algorithm based on the design described in Stuart (2007). Second, there is an overview and step-by-step discussion of the Sudoku generating algorithm (`Simple_Backtracking.js` in JavaScript). The algorithm is divided into two sections; (1.) generate a filled Sudoku board (i.e., the end solution to the puzzle) and (2.) remove numbers to generate a starting Sudoku board with a unique solution (i.e., the starting Sudoku puzzle). Finally, there is an explanation of the modifications made to the algorithm so that it can generate a Miracle Sudoku puzzle (`Miracle.js`).

4.1.1 Design choices

As discussed in Section 3.1, many different algorithms and methods exist for generating and solving Sudoku puzzles. There are more efficient approaches out there than the one developed in this paper, such as the Dancing Links Algorithm X (DLX) (Lloyd & Amos, 2019; Knuth, 2000), Ant Colony Optimization (Lloyd & Amos, 2019), or a combination of Backtracking Search, with Heuristics, and Constraint Propagation (Norvig, n.d.), and many others. This paper did try combining Backtracking Search with Heuristics and Constraint Propagation. However, there are two reasons why this paper uses a simple Backtracking search algorithm.

First, it is not as complex to implement. As an IT&M student, the jump from Backtracking to solving an Exact Cover Problem with Knuth's Algorithm X using a 'dancing links' data structure is substantial. Additionally, the Backtracking approach is still efficient and manages to solve/generate puzzles with almost no weight time (see Section 5.1 for run times). Thus, it did not make sense to try implementing the most sophisticated Sudoku-generating algorithm by sacrificing understanding and quality. Second, since every puzzle goes through DEMYSTIFY when being graded, which takes a long time to run, there is little reward to increase speed slightly on generating the puzzles if the bottleneck comes with the grading.

Another important design decision is deciding how to maintain some form of puzzle symmetry. Symmetry is important purely for aesthetic reasons, and most published Sudoku puzzles contain symmetry. Symmetry in Sudoku means that the starting pattern of clues is symmetric. While there are different degrees of symmetry, the easiest way to ensure some form of symmetry is to follow the approach in Stuart (2007), where an algorithm removes two or four numbers diagonally opposite each other simultaneously. This approach is simple enough to implement and ensures that a starting puzzle does not end up with significantly fewer empty cells on one side or section.

A final important design decision is determining when removing numbers from two cells will likely not lead to a single solution. The cut-off used in this algorithm is thirty (i.e., if thirty or fewer clues remain, only one number is removed). Based on the findings by Stuart (2007), who has generated thousands of puzzles, once thirty clues remain, the numbers should be tested individually to see if they can be safely removed and still maintain a single solution to the puzzle.

4.1.2 Generate a Filled Sudoku Puzzle

The goal of the first part of the backtracking algorithm is to create an empty Sudoku board (a 9x9 grid of zeros) and fill it in so that each number from 1 to 9 occupies each row, column, and 3x3 box just once. To do this, the algorithm creates a 9x9 2D array of 0's (empty board), then it iteratively goes over the board, and each time it encounters an empty cell, it will try to fill it with a random number from 1 to 9. When placing a number into a cell, the algorithm checks if the number placement is valid according to the rules of Sudoku – only one occurrence of each number per row, column, and 3x3 box. If there is a contradiction, the algorithm will backtrack and try another number from 1-9 (excluding the number/s already tried). If no valid number can be placed in the current cell, the algorithm backtracks to the previous cell, tries a new number, and so on. If the number is valid, the algorithm will move on to the next empty cell. This recursive search will continue until the algorithm fully populates the board with a valid Sudoku Puzzle.

Step-by-Step Description

Step 1: Initialize Process

The (1.) *generateSudoku* function initializes the process by calling the *createEmptyBoard* function to create a Sudoku board (board of 0's). Next, the *generateSudoku* function calls the *backtrackSearch* function to fill in the Sudoku board with valid numbers. The *generateSudoku* function returns a filled Sudoku board with a valid solution.

Step 2: Create Empty Board

The (2.) *createEmptyBoard* function generates an empty 9x9 board which is represented as a 9x9 2D array. Each cell's value is set to 0 initially to indicate an empty cell. A 2D array is a matrix of rows and columns. For example, A 2x3 2D array would be matrix $[2][3] = \{[1, 3, 5], [2, 4, 6]\}$, where row 0 and column 2 has a value of 5.

Step 3: Recursive Backtracking Search

The (3.) *backtrackSearch* function is centred on using a recursive approach to fill in the empty cells of the Sudoku board with numbers until a valid solution is found. This function is the core of the

algorithm and is responsible for calling the other functions (*shuffleArray*, *findNextEmptyCell*, *isValidPlacement*) to help the generation process.

Step 3.1: Locate Next Empty Cell

The *backtrackSearch* function starts by finding the first empty cell's coordinates by calling the (4.) *findNextEmptyCell* function, which iterates through the board systematically to locate the next cell with a value of '0' and return its coordinates. If no empty cell can be found, the *backtrackSearch* function returns 'true' to indicate that the board is filled with a valid solution. If an empty cell is located, row and column variables are set equal to the position of the empty cell.

Step 3.2: Try to Place Number in Cell

Next, an array of numbers from 1 to 9 is shuffled using the (5.) *shuffleArray*, which takes an array as an input and randomly shuffles it using the Fisher-Yates algorithm (Eberl, 2016). A shuffled array is used to ensure that the values to be tested are chosen randomly, which will ensure randomness in the generated boards. Following this, the *backtrackSearch* function iterates through the shuffled array and tries to place each number in the empty cell.

Step 3.3: Check if Number Placement is Valid

Before placing a number in the cell, the *backtrackSearch* function checks if the number placement is valid according to the rules of Sudoku by calling the (6.) *isValidPlacement* function, which iterates over every cell in the same row, column, and 3x3 box as the cell being checked, and tests whether the chosen number is already present in any of these cells. If the number is already present (*isValidPlacement* returns 'false') the placement is invalid. If the number is absent (returns 'true') the number is valid.

If the number placement is valid, it is assigned to the empty cell. If the placement is invalid, the *backtrackSearch* function moves to the following number in the array and repeats the process. Suppose the function iterates through all numbers in the array and none can be legally assigned to the empty cell. In that case, the function returns 'false', indicating that the current number placement has led to an invalid solution.

Step 3.4: Recursive Call with Updated Board

After a valid number is assigned to a cell, the *backtrackSearch* function calls itself recursively with the updated board and repeats through the previously mentioned steps. This process will continue until either (1.) the function returns 'true', indicating a valid solution, or (2.) the function returns 'false', indicating that the current number choice has led to an invalid solution.

Step 3.5: Check if Recursive Call Returns True or False

If `true` is returned, then during the recursive call of itself, the *backtrackSearch* function will satisfy the conditional, and the success will be propagated up the call stack, and `true` will be returned – stopping the generation process. If `false` is returned, then during the recursive call of itself, the *backtrackSearch* algorithm will fail the conditional, and the function will backtrack and reset the current cells value to '0' and try the following number in the array. The backtracking search will continue until the chosen number configuration leads to a valid solution.

4.1.3 Generate a Starting Sudoku Board

The second part of the backtracking algorithm aims to take the filled Sudoku puzzle and start subtracting numbers until there is a starting puzzle with only one valid solution. First, the algorithm iterates over all the cells on the board and randomly selects a cell each time. If more than thirty clues remain when a cell is selected, the algorithm will remove two diagonally opposite cells to ensure symmetry (Stuart, 2007). When less than thirty cells remain, the algorithm will remove one because the chances of removing two cells and having a single solution to the puzzle become less likely.

Each time there is a removal, the algorithm will try to solve the puzzle and test to see if it has a single solution. The puzzle-solving process follows the same process as the previous Sudoku generating function. However, instead of returning true when finding a solution and ending the process, a count of one is returned, and the algorithm backtracks to try and find a different solution. If more than one solution is found at any point, the algorithm will reset the removed cell's value and choose a different cell (or pair of cells) to remove and test. This process continues until it cannot remove another cell without the puzzle having more than one solution.

Step-by-Step Description

Step 1: Initialize Process

The (1.) *generateStartingSudoku* function initializes the solving process by calling the *getAllCellPositions* function to get a list of all cell coordinates. Next, it calls the *selectCellsToBeRemoved* function, with all the cell coordinates, to take the filled Sudoku board and output a starting Sudoku board with only one valid solution.

Step 2: Get all Cell Positions

The (2.) *getAllCellPositions* function iterates over all the cell coordinates of a Sudoku board and stores them in an array of [row, column] pairs representing cell positions. Before the array of cell positions is returned it is shuffled using *shuffleArray* function to ensure cells are selected at random – this is done for uniqueness.

Step 3: Determine Cells to Remove Numbers From

The (3.) *selectCellsToBeRemoved* function is responsible for iterating over the array of cell positions, and selecting a [row, column] pair. At each iteration, the *selectCellsToBeRemoved* function determines the number of filled cells remaining in the current configuration of the board by calling the *countFilledCells* function. The (4.) *countFilledCells* function iterates over the board and when a filled cell is found it increments a count by one. After the whole board has been checked, the count is returned.

From this, two checks take place; first a check is made to see if the chosen cell is empty, if it is then the conditional fails and the next cell is tested (i.e., want to select a filled cell). If the first check is passed, then a check is made on how many filled cells remain to determine whether to remove one or two cells. If more than thirty clues remain, the *removeTwoNumbersAndValidateSolution* function will remove numbers from the selected cell and its diagonally opposite cell and test that boards configuration for a unique solution. If thirty or less clues remain then the *removeOneNumberAndValidateSolution* function will only remove a number from the selected cell and test for a unique solution.

The functions will either output a Sudoku board with some subtracted cell values or, if the unique solution check failed, the same Sudoku board. In either case, the *selectCellsToBeRemoved* function will iterate to remove another cells value and test for a unique solution.

Step 4: Remove One/Two Numbers and Validate Unique Solution Check

The (5.) *removeTwoNumbersAndValidateSolution* function gets the coordinates of the diagonally opposite cell by subtracting eight from each of the selected cell's coordinates (i.e., diagonal row = [8 - row]). Then, the cells are set to zero and the current configuration of the puzzle is tested to see if only a single solution exists by calling the *uniqueSolutionCheck* function. If the current puzzle has a unique solution (output is '1'), the numbers can safely be removed, and the function returns the updated Sudoku board. But, if the current puzzle has more than one solution (output is greater than '1') those numbers cannot be removed, and the function resets the cells back their original numbers and returns the original Sudoku board.

The (5.) *removeOneNumberAndValidateSolution* function follows the exact same process, but only removes the number from the chosen cell instead of the chosen cell and its diagonally opposite cell.

Step 5: Unique Solution Check

First, the (6.) *uniqueSolutionCheck* function creates a deep clone of the original board to avoid modifying it during the checking process. Next, it calls the *countSolutions* function, with the cloned

board as a parameter, to determine the number of solutions to the current configuration of the puzzle. The solution count is returned to the `removeTwoNumbersAndValidateSoluton` and `removeOneNumberAndValidateSolution` functions to indicate whether a unique solution was found.

Step 6: Counting Number of Solutions

The (7.) *countSolutions* function uses the same recursive backtracking approach as the `backtrackSearch` function (in Section 4.1.2, Step 3) to fill in the empty cells of the Sudoku puzzle until a valid solution is found. The only differences occur when an output is returned because the *countSolutions* function is focused on finding all the possible solutions to the current Sudoku puzzle configuration and not just on filling the board with a valid solution.

Step 6.1: Locate Next Empty Cell

The *countSolutions* function finds the coordinates of the first empty cell. If no empty cell can be found (i.e., valid solution) the function returns `1` (instead of `true`) to indicate one valid solution has been found. If an empty cell is located, row and column variables are set equal to the position of the empty cell. In addition, a *count* variable is initialized and set equal to `0`.

Step 6.2: Try to Place Number in Cell

Next, an array of numbers from 1 to 9 is shuffled, and the *countSolutions* function then iterates through the numbers in the shuffled array and tries to place each number in the empty cell.

Step 6.3: Check if Number Placement is Valid

Before placing the number in the cell, the *countSolutions* function checks if the number placement is valid according to the rules of Sudoku.

If the number placement is invalid, then the *countSolutions* function moves onto test the next number in the array. If the function iterates through all numbers in the array and none can be legally assigned to the empty cell, the *countSolutions* function returns the *count* variable (instead of `false`) indicating that the current number placement has not led to a valid Solution. If the number placement is valid, it is assigned to the empty cell.

Step 6.4: Recursive Call with Updated Board

After a valid number is assigned to a cell, the *countSolutions* function calls itself recursively with the updated board and repeats through the previously mentioned steps. This process will continue until either (1.) the function returns `1`, indicating a valid solution, or (2.) the function returns *count*, indicating that the current number choice has led to an invalid solution and therefore the number of solutions remains the same. The output of the recursive call is added to the *count* variable.

Step 6.5: Check if Recursive Call Returns True or False

Finally, the function checks whether the *count* variable is greater than 1. If it is, it means more than one solution is found and this information is propagated up the call stack. If *count* is less than or equal to one, the function undoes the previous choice and tries the next number to see if it can find another solution. The backtracking will continue until all possible configurations are tested.

4.1.4 Modifications for the Miracle Sudoku Puzzle

The Miracle Sudoku generating algorithm ('Miracle.js') follows the same approach as the Simple_Backtracking.js file but accounts for the additional three rules of the Miracle Sudoku puzzle. So, only the 'isValidPlacement' function needs to be modified.

The modifications are as follows. First, the function checks that the chosen number is not already present in any of the cells separated by a King's move in chess to the chosen cell. For example, if the coordinates of the chosen cell are [row, col], then to access the cell straight above, the function must check that the cell [row - 1, col] does not already contain the chosen cell number.

Second, the function checks that the chosen number is not already present in any of the cells separated by a Knights move in chess to the chosen cell. For example, if the coordinates of the chosen cell are [row, col], then to access the cell with a Knights jump North-East, the function checks the cell at coordinates [row-2, col+1].

Thirdly, the function checks that consecutive numbers from the chosen number are not already present in any of the cells separated orthogonally to the chosen cell. For example, if the coordinates of the chosen cell are [row, col], then to access the cell straight below, the function needs to check that the cell [row + 1, col] does not already contain a number one less or one more than the chosen cells number.

The function follows this logic to check all the other cells connected by these three constraints to the chosen cell. At each new constraint check, the function adds a check to make sure that whenever a subtraction is made from the row or column, that new value is still equal to or greater than '1' and that whenever an addition is made, that new value is still equal to or less than '8' – to ensure the checks do not go outside the board's parameters.

4.2 The Grading Method

This section first discusses the design choices in developing the grade formula utilized to quantify the difficulty of the Sudoku puzzles generated. In doing so, the discussion provides the method used to link MUSes to difficulty level, explaining how Sudoku puzzles are graded in this paper using a combination of the largest MUS Size, the Sum of MUS size, and the number of empty cells. This discussion is followed by a brief overview of the algorithm ('Difficulty_API.js') and a step-by-step

description of how the algorithm provides a grade score to each of the generated puzzles based on the MUS Size's extracted during each solving step by making running the DEMYSTIFY solving algorithm.

4.2.1 Grade Formula and Design Choices

The purpose of the formula presented in this paper is to provide some quantifiable, numerical measurement of a Sudoku puzzles difficulty. Fundamentally, this means measuring 'what makes a Sudoku strategy difficult' (Stuart, 2008). This paper measures difficulty by extracting the size of the MUSes used in each solving step as a proxy for difficulty, specifically, the complexity of solving strategy. A larger MUS size indicates more constraints in the solving process, making it harder to explain and, thus, a more complicated strategy (for further details, please refer to Section 3.3.1). From this, the formula uses the size of the largest 'smallest' MUS as an indicator of the hardest strategy required to solve a puzzle and the sum of the 'smallest' sized MUSes as an indicator of the number of complicated strategies required to solve a puzzle. In doing so, this can provide a framework to evaluate the difficulty of a Sudoku puzzle.

The formula is as follows,

$$(1.) DS = X1(largestMUSSize) + Y1(sumOfMUSSizes) + Z1(emptyCellCount)$$

Where DS is difficulty score, $largestMUSSize$ is the Largest 'smallest' MUS Size, $sumOfMUSSizes$ is the sum of all the 'smallest' MUS sizes, and $emptyCellCount$ is the number of empty cells the initial Sudoku puzzle starts with. The $X1$, $Y1$, and $Z1$ variables indicate the weights that will be assigned to each metric during the evaluation (Section 6.2.1).

The first variable, $largestMUSSize$, is extracted from an array of all the 'smallest' MUSes used to solve the puzzle by DEMYSTIFY. For example, if DEMYSTIFY solves a puzzle with MUSes [5, 3], the $largestMUSSize$ in this puzzle would be [5].

Initially, the idea was to grade the puzzles based only on the $largestMUSSize$ value because a puzzle with a larger MUS size would always be more difficult than a puzzle with a smaller MUS size, *ceteris paribus*. However, this comparison is sufficient only if DEMYSTIFY uses one MUS while solving a puzzle, which becomes obsolete for most puzzles above a medium difficulty level.

So, the critical question becomes how the formula would distinguish, for example, between a puzzle with MUSes of size [4, 3] and a puzzle with MUSes of size [4, 2]. A simple solution to this problem is to compare each puzzle's second-largest MUS size, then if there is still a tie, the third largest, and so on. However, the purpose of the algorithm is to provide a quantifiable grade to each

puzzle and not to make individual comparisons. Hence, the formula also uses a *sumOfMUSSizes* variable to account for the total size of all MUSes that DEMYSTIFY uses when solving a puzzle.

The *sumOfMUSSizes* variable takes the sum of all the 'smallest' MUS sizes used to solve the puzzle. The *sumOfMUSSizes* variable aligns well with most Sudoku difficulty rating formulas. Such formula assign points to solving strategies depending on how difficult they are, and then during solving, the points of each strategy used are added up to get an overall grade scale (see, for example, Stuart, 2007; Pelánek, 2014; Tomlinson, 2005). If MUS size indicates the difficulty of a solving strategy, then the sum of all MUS sizes is the same as adding up the points of each strategy. Thus, the *sumOfMUSSizes* value accounts for everything the *largestMUSSize* value accounts for and more.

However, it is still important to include the *largestMUSSize* value due to the subjectivity surrounding what players find more difficult when solving a puzzle. For example, does a player find it more challenging to solve a puzzle with one very tricky step and the remainder being easy (i.e., a puzzle with MUS sizes [9, 3, 2]) or a puzzle with a lot of somewhat tricky steps (i.e., a puzzle with MUS sizes [5, 5, 5, 4, 4, 3])? According to the *sumOfMUSSizes* value, the second puzzle is harder, but according to the *largestMUSSize* value, the first puzzle is harder. The underlying aspects of this discrepancy go beyond the scope of this paper as it comes down to an individual's innate ability and puzzle-solving experience or practice. Solving this question would require an extensive statistical analysis of players' solving puzzles. In this paper, the formula includes both variables. However, it adds more weight to the *largestMUSSize* value (see section 6.2.1) to account for its size being limited compared to the *sumOfMUSSizes* value.

Finally, the formula must distinguish between easy and medium puzzle difficulties, which is not always possible by only looking at the *sumOfMUSSizes* and the *largestMUSSize* value – players can often solve easy and medium puzzles with only simple deductions. Therefore, the formula utilizes an *emptyCellCount* variable as a numerical count of the number of empty cells in the starting board. It is important to note that many published Sudoku puzzles will only use the initial number of filled cells to measure difficulty, as this is the easiest, yet most naïve, and quickest way to grade a puzzle.

Regarding the actual algorithm, a crucial design decision is to grade the Sudoku puzzles after they have been generated rather than trying to grade a puzzle of a specific difficulty. This decision is due to the time it would take to make recursive calls to the DEMYSTIFY solver at each number removal to ensure that each solving step's difficulty grade (MUS size) remains below or above a certain threshold. Rather, it made more sense to generate a puzzle first and then assign it a difficulty rating second. The idea is that overtime, the algorithm will generate a library of puzzles with different difficulties. So, when a newly generated puzzle is not of the difficulty grade the player wants, they can select a different puzzle from the library.

4.2.2 The Grading Algorithm

The algorithm takes the starting Sudoku puzzle generated in `'Simple_Backtracking.js'` and runs it through the DEMYSTIFY solver. Then, the algorithm saves the output and begins interacting with it to extract the smallest size MUS used at each step to solve the puzzle and save them in an array. From this, the algorithm determines the size of the largest MUS used in solving a solution to this specific puzzle. The algorithm also collects data on how many empty cells there were in the starting Sudoku puzzle, and a sum of all the MUS Sizes, which are used to provide the final difficulty rating metric and grade level for each of the puzzles generated. The algorithm outputs the starting Sudoku puzzle, the solution of the puzzle, the difficulty metrics, and a puzzle grade for each generated puzzle.

Step-by-step Description

Step 1: Import and Temporary Save Starting Sudoku Board

The (1.) *saveStartingBoard* function is responsible for saving the imported starting Sudoku board to a file which contains all the parameters required for the algorithm to run, in the format that can be read by DEMYSTIFY. The function saves the board to a temporary storage folder in the working directory.

Step 2: Convert 2D Array to String for Saving Board

Next, the starting board (9x9 2D array) is converted to a string (i.e., a string with the board grid in 1D form, like '03200...') with the *arrayToString* function, which loops over all cell positions in the starting board and adds them to a string. This conversion is so that when the board is solved, assigned a difficulty rating, and saved it can be saved with the starting grid as its name to be uniquely identifiable.

Step 3: Run DEMYSTIFY and Save Output

Next, the (3.) *call_demystify* function initializes the DEMYSTIFY Python algorithm and saves its output as a JavaScript Object Notation (JSON) file (with its board configuration string). The function uses the `'spawn'` Node module to run DEMYSTIFY from JavaScript. The `'spawn'` module enables the JavaScript function to access Operating System (OS) functionalities by running any terminal commands inside a "child process" (Buna, 2017).

First, the function sets a variable `'terminalInput'` equal to the commands required to run DEMYSTIFY directly from the terminal (for instructions on running DEMYSTIFY, see <https://github.com/stacs-cp/demystify/blob/master/README.md>). Next, a new child process is spawned that runs DEMYSTIFY with the command-line arguments from the `'terminalInput'` variable and sets the current working directory equal to where you want the output to be saved.

Following this, the function listens for any error output during the process, and if there is an error it will log that error (the same one seen in the terminal) to the console. Finally, the function listens for the `close` event, which indicates the solving process has finished. If the process exists with a code of `0`, this indicates a success, and the function logs a success message to the console and calls the *readFileAndGradePuzzle* function to grade the provided puzzle based on the *DEMYSTIFY* output. If the process exists with a code not equal to `0`, this indicates an error in the solving process has occurred, and the functions logs an error message to the console.

Step 3: Fetch the JSON file and Grade Sudoku Puzzle

The (4.) *readFileAndGradePuzzle* is responsible for reading the *DEMYSTIFY* output, extracting the size of the MUS used at each solving step and other metrics (such as number of starting empty cells), and using this information to provide a grade to each puzzle. The function uses the `fs` Node module, with the `readFile` function to read in the JSON file.

Step 3.1 Determine Largest MUS from Array of Smallest MUS Sizes

The function creates an array of the smallest MUS sizes at each step of the solving process by calling the (5.) *getSmallestMUSSizes*, which converts the JSON output file to an object. An object is a collection of properties where each property has a name and an associated value (MDN, 2023).

Following this, the function filters the object array based on each solving step and returns the smallest MUS size at each step if it is not undefined (i.e., in cases where simple deductions were made so no MUSes were needed). After creating a new array of all the smallest MUS sizes, the *largestMUSSize* variable is extracted from the MUS array by using `Math.max`, which returns the highest number from the array.

Step 3.2 Get Remaining Difficulty Metrics

Following this, the function gets a sum of all MUS sizes used when solving the puzzle by calling the (6.) *getSumOfMUSSizes* function, which iterates over each *MUSSize* in the array and adds it to a *sumOfMUSSizes* variable. Next, the function gets a count of the number of empty cells by calling the (7.) *countEmptyCells* function, which iterates over the starting Sudoku board and every time a cell position is equal to zero it increments an *emptyCellCount* variable by one.

Next, the function combines these metrics into the grade formula, where the X1, Y1, and Z1 variables are adjustable depending on the weights wanted. In section 6.2.1, the weights are set to the values they are used to assign grades going forward. Finally, based on the calibrating done during section 6.2, the algorithm gives each puzzle a grade level depending on its difficulty score.

To end, the `saveBoardWithDifficultyGrade` function is called with all the difficulty metrics to output all the information to the user when generating a new Sudoku puzzle. Additionally, it calls

the `printBoardToFile` function to display the starting board in a user-friendly way. An example of the final output is displayed in Figure 3 below.

The Puzzle:		
4 0 0	8 0 0	5 0 0
0 0 0	0 0 3	0 0 0
2 0 3	7 4 9	0 0 0

0 0 8	0 0 0	0 0 9
7 0 0	6 0 8	0 0 5
9 0 0	0 0 0	1 0 0

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

List of Smallest Minimal Unsatisfiable Sets (MUSes) at Each Solving Step (Desc. Order):		
6,6,6,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,4,4,4,4,4,4,4,3,3,3,3,3,0		

Difficulty Metrics:		
Largest 'smallest' MUS Size (6), Sum of MUS Sizes (143), Empty Cell Count (55)		

Difficulty Score: $2460 = X1 * \text{largestMUSSize} + Y1 * \text{SumOfMUSSizes} + Z1 * \text{EmptyCellCount}$		
Where, X1: 80, Y1: 10, Z1: 10		
Assigned Grade: Very Hard		

The Solution:		
4 7 9	8 1 6	5 2 3
1 8 6	2 5 3	7 9 4
2 5 3	7 4 9	6 8 1

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

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

Figure 3: Final output from generating and grading algorithm

5. Testing

5.1 Functionality Testing

This section discusses the testing of the algorithms for efficiency and correctly handling errors. The first part discusses the run time of the generation and grading algorithms. The second part discusses the testing to ensure the algorithms were running as they should (i.e., that a backtracking search is solving the puzzles with backtracking). Thirdly, the section discusses the two main areas where a significant problem could occur; generating an invalid puzzle and DEMYSTIFY being unable to solve the puzzle.

5.1.1 Runtime of the Algorithms

First, both the efficiency of the (1.) Generate a Filled Sudoku Board and (2.) Generate a Starting Sudoku Board sections of the Backtracking Search algorithm are tested in Figure 4 and 5 respectively. Overall, it rarely takes over 5 milliseconds to generate a fully filled Sudoku puzzle, but it can sometimes take up to 1 second to solve for the starting board. The reason for this discrepancy is because the solving process must search for all possible solutions every time to ensure that when numbers are removed from cells there is always a unique solution.

It is important to note, as discussed in Section 4.1.1, that the method of solving a Sudoku puzzle with backtracking is not the most efficient compared to other methods out there (for example, Lloyd & Amos, 2019; Knuth, 2000; Norvig, n.d.). However, the overall performance of the Backtracking Search algorithm is sufficient for the purpose of this paper as almost all the wait time during the process comes from the call to the *DEMYSTIFY* solving algorithm.

In general, the run time of the *DEMYSTIFY* algorithm will depend on the difficulty of the puzzle generated – easier puzzles will be quicker to solve than harder puzzles. The approximate run times of the algorithm, based on difficulty level are as follows: Easy (7 seconds), Medium (10 seconds), Hard (45 seconds), Very Hard (90 seconds), Evil (150 + seconds). However, it is important to note that performance, in time, is not the concern of this algorithm.

5.1.2 Testing the Generation and Solving Process

To check the generating and solving algorithms run as they supposed to, this paper uses a combination of logging the output at each generating step and checking the process. An example from the generating a filled board testing is displayed in Figure 6 below – notice the backtracking between Step 44 and Step 45 (cannot place a ‘2’ in last position of row), and the final output – and an example from solving a filled board backwards testing is displayed in Figure 7 below – notice how in the beginning two numbers are removed at a time which are diagonally opposite to ensure symmetry.

▶ Array(9) [9, 3, 8, 7, 5, 4, 1, 2, 6] ▶ Array(9) [7, 9, 5, 4, 1, 2, 8, 6, 3] ▶ Array(9) [9, 5, 7, 8, 2, 1, 3, 6, 4]	▶ Array(9) [4, 7, 5, 2, 6, 1, 8, 9, 3] ▶ Array(9) [4, 8, 1, 9, 6, 3, 2, 7, 5] ▶ Array(9) [2, 8, 1, 6, 4, 3, 9, 5, 7]	▶ Array(9) [2, 1, 6, 9, 8, 3, 7, 5, 4] ▶ Array(9) [3, 6, 2, 5, 8, 7, 1, 9, 4] ▶ Array(9) [6, 3, 4, 7, 5, 9, 1, 2, 8]
▶ Array(9) [5, 4, 1, 3, 7, 6, 2, 8, 9] ▶ Array(9) [6, 4, 8, 1, 2, 5, 7, 3, 9] ▶ Array(9) [1, 9, 5, 3, 7, 4, 2, 8, 6]	▶ Array(9) [3, 8, 7, 5, 9, 2, 6, 4, 1] ▶ Array(9) [2, 3, 9, 6, 7, 4, 5, 8, 1] ▶ Array(9) [3, 7, 8, 5, 6, 2, 4, 9, 1]	▶ Array(9) [6, 9, 2, 4, 1, 8, 3, 7, 5] ▶ Array(9) [5, 1, 7, 3, 9, 8, 6, 4, 2] ▶ Array(9) [4, 2, 6, 9, 1, 8, 5, 7, 3]
▶ Array(9) [7, 5, 3, 1, 2, 9, 4, 6, 8] ▶ Array(9) [1, 2, 4, 7, 3, 6, 9, 5, 8] ▶ Array(9) [5, 1, 3, 2, 8, 7, 6, 4, 9]	▶ Array(9) [8, 2, 4, 6, 3, 5, 9, 1, 7] ▶ Array(9) [8, 7, 3, 2, 5, 9, 4, 1, 6] ▶ Array(9) [7, 4, 2, 1, 9, 6, 8, 3, 5]	▶ Array(9) [1, 6, 9, 8, 4, 7, 5, 3, 2] ▶ Array(9) [9, 5, 6, 8, 4, 1, 3, 2, 7] ▶ Array(9) [8, 6, 9, 4, 3, 5, 7, 1, 2]
Generation Time: 3ms – timer ended	Generation Time: 5ms – timer ended	Generation Time: 2ms – timer ended
▶ Array(9) [5, 4, 8, 3, 2, 1, 9, 6, 7] ▶ Array(9) [1, 7, 6, 9, 4, 2, 3, 8, 5] ▶ Array(9) [1, 6, 7, 5, 8, 3, 2, 4, 9]	▶ Array(9) [9, 1, 2, 7, 6, 4, 3, 5, 8] ▶ Array(9) [5, 2, 3, 7, 6, 8, 1, 9, 4] ▶ Array(9) [4, 9, 5, 1, 6, 2, 7, 8, 3]	▶ Array(9) [3, 6, 7, 5, 9, 8, 2, 4, 1] ▶ Array(9) [9, 4, 8, 1, 5, 3, 6, 2, 7] ▶ Array(9) [8, 2, 3, 7, 9, 4, 5, 6, 1]
▶ Array(9) [6, 5, 4, 8, 7, 3, 1, 2, 9] ▶ Array(9) [8, 9, 4, 6, 1, 5, 7, 3, 2] ▶ Array(9) [2, 5, 8, 9, 3, 7, 4, 1, 6]	▶ Array(9) [7, 9, 3, 1, 5, 2, 4, 8, 6] ▶ Array(9) [6, 5, 7, 3, 2, 9, 4, 1, 8] ▶ Array(9) [3, 7, 6, 4, 2, 1, 9, 5, 8]	▶ Array(9) [8, 2, 1, 6, 4, 9, 5, 7, 3] ▶ Array(9) [3, 1, 2, 8, 7, 4, 9, 5, 6] ▶ Array(9) [9, 1, 4, 8, 5, 6, 3, 2, 7]
▶ Array(9) [1, 8, 5, 2, 3, 6, 7, 9, 4] ▶ Array(9) [7, 8, 9, 2, 3, 6, 5, 4, 1] ▶ Array(9) [5, 3, 9, 2, 1, 8, 6, 7, 4]	▶ Array(9) [2, 3, 9, 4, 8, 7, 6, 1, 5] ▶ Array(9) [4, 3, 1, 5, 8, 7, 2, 6, 9] ▶ Array(9) [7, 8, 2, 6, 4, 9, 1, 3, 5]	▶ Array(9) [4, 7, 6, 9, 1, 5, 8, 3, 2] ▶ Array(9) [2, 6, 5, 4, 9, 1, 8, 7, 3] ▶ Array(9) [6, 4, 1, 3, 7, 5, 8, 9, 2]
Generation Time: 2ms – timer ended	Generation Time: 3ms – timer ended	Generation Time: 3ms – timer ended
▶ Array(9) [2, 8, 6, 3, 4, 5, 7, 1, 9] ▶ Array(9) [4, 1, 6, 3, 9, 2, 5, 7, 8] ▶ Array(9) [1, 5, 8, 6, 7, 4, 9, 3, 2]	▶ Array(9) [3, 7, 9, 6, 1, 2, 4, 5, 8] ▶ Array(9) [7, 3, 9, 1, 5, 8, 6, 2, 4] ▶ Array(9) [9, 7, 4, 3, 5, 2, 6, 8, 1]	▶ Array(9) [1, 4, 5, 9, 7, 8, 6, 2, 3] ▶ Array(9) [2, 8, 5, 6, 4, 7, 9, 1, 3] ▶ Array(9) [2, 3, 6, 1, 9, 8, 4, 7, 5]
▶ Array(9) [5, 1, 7, 8, 2, 6, 9, 3, 4] ▶ Array(9) [1, 5, 7, 9, 2, 4, 3, 8, 6] ▶ Array(9) [7, 8, 3, 2, 6, 9, 1, 5, 4]	▶ Array(9) [9, 2, 4, 5, 3, 7, 8, 6, 1] ▶ Array(9) [3, 9, 8, 5, 6, 1, 2, 4, 7] ▶ Array(9) [4, 2, 1, 5, 8, 3, 7, 9, 6]	▶ Array(9) [8, 6, 3, 4, 9, 1, 2, 7, 5] ▶ Array(9) [6, 4, 2, 8, 7, 3, 1, 9, 5] ▶ Array(9) [6, 9, 5, 4, 1, 7, 3, 2, 8]
▶ Array(9) [6, 5, 1, 2, 8, 9, 3, 4, 7] ▶ Array(9) [9, 6, 4, 2, 8, 5, 7, 3, 1] ▶ Array(9) [3, 6, 7, 8, 4, 5, 2, 1, 9]	▶ Array(9) [4, 9, 2, 7, 5, 3, 1, 8, 6] ▶ Array(9) [5, 7, 1, 4, 3, 9, 8, 6, 2] ▶ Array(9) [8, 4, 2, 9, 3, 1, 5, 6, 7]	▶ Array(9) [7, 3, 8, 1, 6, 4, 5, 9, 2] ▶ Array(9) [8, 2, 3, 7, 1, 6, 4, 5, 9] ▶ Array(9) [5, 1, 9, 7, 2, 6, 8, 4, 3]
Generation Time: 3ms – timer ended	Generation Time: 2ms – timer ended	Generation Time: 3ms – timer ended

Figure 4: Testing Runtime of Sudoku Generating Algorithm

▶ Array(9) [7, 0, 0, 0, 0, 2, 8, 0, 3] ▶ Array(9) [7, 0, 0, 8, 0, 0, 0, 9, 0] ▶ Array(9) [7, 0, 9, 0, 0, 6, 0, 0, 1]	▶ Array(9) [4, 0, 0, 0, 0, 0, 6, 0, 7] ▶ Array(9) [9, 8, 3, 0, 0, 0, 0, 0, 5] ▶ Array(9) [5, 2, 0, 3, 0, 0, 7, 0, 0]	▶ Array(9) [0, 0, 0, 0, 0, 0, 0, 9, 0] ▶ Array(9) [0, 0, 0, 0, 0, 0, 0, 4, 2] ▶ Array(9) [0, 0, 4, 2, 0, 0, 0, 0, 0]
▶ Array(9) [0, 5, 2, 0, 4, 0, 0, 7, 0] ▶ Array(9) [0, 0, 0, 4, 9, 0, 7, 0, 0] ▶ Array(9) [0, 0, 0, 0, 0, 0, 8, 0, 2]	▶ Array(9) [0, 0, 0, 8, 0, 8, 0, 0, 0] ▶ Array(9) [0, 0, 0, 6, 0, 8, 0, 2, 1] ▶ Array(9) [0, 8, 0, 6, 4, 3, 0, 0, 0]	▶ Array(9) [0, 8, 0, 9, 0, 0, 2, 4, 0] ▶ Array(9) [0, 0, 0, 0, 1, 2, 0, 0, 0] ▶ Array(9) [4, 0, 0, 0, 0, 0, 0, 0, 0]
▶ Array(9) [0, 4, 0, 0, 0, 0, 0, 0, 0] ▶ Array(9) [0, 4, 0, 0, 0, 0, 0, 0, 0] ▶ Array(9) [0, 0, 0, 0, 0, 0, 1, 0, 0]	▶ Array(9) [3, 0, 6, 0, 0, 0, 0, 0, 9] ▶ Array(9) [1, 0, 0, 0, 0, 0, 0, 8, 9] ▶ Array(9) [0, 0, 6, 0, 0, 9, 0, 5, 3]	▶ Array(9) [8, 0, 7, 3, 0, 5, 0, 0, 2] ▶ Array(9) [0, 7, 0, 0, 0, 5, 0, 0, 3] ▶ Array(9) [2, 0, 0, 1, 0, 0, 9, 0, 8]
Generation Time: 324ms – timer ended	Generation Time: 97ms – timer ended	Generation Time: 139ms – timer ended
▶ Array(9) [6, 3, 0, 0, 0, 0, 0, 9, 1] ▶ Array(9) [0, 5, 7, 0, 2, 0, 0, 0, 0] ▶ Array(9) [0, 0, 0, 0, 0, 0, 5, 2, 1]	▶ Array(9) [0, 9, 7, 0, 0, 0, 0, 0, 0] ▶ Array(9) [8, 0, 0, 0, 1, 0, 6, 4, 0] ▶ Array(9) [6, 0, 7, 5, 0, 0, 0, 0, 3]	▶ Array(9) [0, 8, 1, 7, 0, 0, 0, 0, 0] ▶ Array(9) [0, 0, 0, 7, 0, 0, 0, 0, 0] ▶ Array(9) [0, 0, 0, 0, 3, 2, 0, 0, 0]
▶ Array(9) [0, 0, 0, 1, 0, 0, 0, 3, 4] ▶ Array(9) [0, 3, 0, 6, 9, 0, 7, 0, 0] ▶ Array(9) [8, 0, 1, 2, 0, 0, 6, 0, 0]	▶ Array(9) [0, 0, 2, 0, 0, 0, 0, 0, 0] ▶ Array(9) [0, 4, 0, 0, 0, 0, 0, 0, 0] ▶ Array(9) [0, 0, 2, 0, 6, 0, 9, 0, 0]	▶ Array(9) [4, 7, 0, 2, 0, 5, 0, 0, 0] ▶ Array(9) [0, 0, 6, 0, 0, 5, 0, 1, 0] ▶ Array(9) [0, 0, 6, 0, 0, 7, 3, 0, 2]
▶ Array(9) [0, 0, 0, 0, 0, 2, 5, 7, 0] ▶ Array(9) [0, 0, 0, 0, 0, 1, 0, 0, 0] ▶ Array(9) [0, 0, 0, 4, 7, 0, 0, 0, 0]	▶ Array(9) [0, 0, 0, 0, 0, 0, 8, 0, 9] ▶ Array(9) [0, 6, 2, 0, 0, 0, 0, 9, 3] ▶ Array(9) [3, 0, 0, 0, 0, 9, 2, 0, 4]	▶ Array(9) [7, 5, 0, 0, 0, 0, 0, 4, 2] ▶ Array(9) [0, 0, 0, 4, 0, 0, 5, 2, 0] ▶ Array(9) [4, 7, 8, 0, 0, 0, 0, 0, 0]
Generation Time: 88ms – timer ended	Generation Time: 428ms – timer ended	Generation Time: 103ms – timer ended
▶ Array(9) [0, 0, 0, 0, 0, 0, 0, 0, 0] ▶ Array(9) [7, 3, 0, 0, 0, 2, 0, 0, 0] ▶ Array(9) [0, 0, 0, 0, 0, 0, 4, 0, 2]	▶ Array(9) [0, 2, 0, 4, 0, 0, 6, 9, 0] ▶ Array(9) [0, 0, 2, 3, 0, 6, 0, 0, 8] ▶ Array(9) [6, 0, 0, 0, 0, 3, 0, 0, 0]	▶ Array(9) [6, 0, 0, 0, 7, 0, 0, 3, 0] ▶ Array(9) [0, 0, 0, 0, 5, 0, 1, 0, 0] ▶ Array(9) [0, 5, 0, 8, 0, 4, 3, 0, 0]
▶ Array(9) [1, 0, 0, 0, 0, 3, 0, 0, 8] ▶ Array(9) [0, 2, 1, 0, 0, 0, 0, 0, 4] ▶ Array(9) [0, 0, 9, 5, 0, 0, 0, 0, 8]	▶ Array(9) [0, 9, 0, 0, 5, 0, 0, 0, 0] ▶ Array(9) [0, 0, 4, 0, 0, 0, 6, 0, 0] ▶ Array(9) [8, 0, 0, 0, 0, 0, 0, 0, 3]	▶ Array(9) [5, 6, 0, 1, 0, 0, 4, 0, 0] ▶ Array(9) [6, 0, 0, 0, 0, 0, 9, 5, 0] ▶ Array(9) [0, 0, 2, 0, 0, 6, 0, 0, 0]
▶ Array(9) [8, 7, 0, 0, 6, 0, 0, 0, 5] ▶ Array(9) [0, 0, 8, 9, 6, 0, 0, 0, 0] ▶ Array(9) [0, 0, 0, 1, 0, 9, 0, 7, 0]	▶ Array(9) [0, 0, 4, 0, 0, 1, 0, 7, 0] ▶ Array(9) [4, 0, 0, 8, 0, 7, 0, 0, 0] ▶ Array(9) [0, 0, 5, 3, 0, 0, 0, 0, 1]	▶ Array(9) [0, 0, 0, 0, 0, 0, 0, 0, 0] ▶ Array(9) [0, 0, 0, 0, 2, 0, 0, 6, 1] ▶ Array(9) [2, 0, 0, 0, 0, 0, 0, 0, 0]
Generation Time: 296ms – timer ended	Generation Time: 125ms – timer ended	Generation Time: 532ms – timer ended

Figure 5: Testing Runtime of Sudoku Solving Algorithm

[illegible]

Figure 6: Snippets from the Sudoku Generating Algorithm Process

[illegible]

Figure 7: Snippets from the Sudoku Solving Algorithm Process

5.2 Invalid Board

A final test takes place for every board generated in `Simple_Backtracking.js`. Before the algorithm removes numbers to generate a starting Sudoku board, the filled Sudoku board is put through the *finalBoardTest* function, which iterates over every cell in the board to check that the sum of each row, column, and 3x3 box adds to 45 (i.e., each row, column, and 3x3 box contains every number from 1 to 9 just once). If false, it will restart the generation process. However, since the generation process involves checking that every number placed is valid according to the rules of Sudoku, it should not be possible for an invalid board to be generated. Nevertheless, generating an invalid board would be a major failure of the algorithm, so it is important to double check.

To handle the errors from the DEMYSIFY algorithm, the code in the grading algorithm logs the DEMYSIFY terminal error output in node.js. An error should not be encountered with the boards generated in `Simple_Backtracking.js` as they are written and saved in the input format DEMYSTIFY requires and checked to ensure that they are valid (and can be solved). However, if someone wishes to test a pre-generated puzzle (such as the boards from `Grade_Testing.js`, DEMYSIFY may produce an error if the board is invalid according to the rules of Sudoku (Figure 8) or the parameters required by DEMYSIFY (Figure 9).

```
const invalid_board1 = [
  [9,9,7,0,8,0,4,0,0],
  [0,4,0,0,0,5,7,8,0],
  [0,6,0,0,0,7,0,1,0],
  [9,0,6,0,7,0,0,4,3],
  [4,0,3,2,0,6,0,0,8],
  [8,0,1,3,0,9,0,0,0],
  [6,8,5,1,3,0,0,0,0],
  [0,3,4,5,9,0,2,6,0],
  [0,1,0,0,6,0,0,0,0]
]
```

```
demystify.parse.ParseError: ERROR: Constraint box_alldiff('0', '0', '0', '0', '1', '9') cannot be satisfied..
Demystify process exited with code 1. An error occurred.
```

Figure 8: Invalid Board according to the rules of Sudoku

```
const invalid_board2 = [
  ['one',9,7,0,8,0,4,0,0],
  [0,4,0,0,0,5,7,8,0],
  [0,6,0,0,0,7,0,1,0],
  [9,0,6,0,7,0,0,4,3],
  [4,0,3,2,0,6,0,0,8],
  [8,0,1,3,0,9,0,0,0],
  [6,8,5,1,3,0,0,0,0],
  [0,3,4,5,9,0,2,6,0],
  [0,1,0,0,6,0,0,0,0]
]
```

```
ERROR: Identifier not defined: one
ERROR: Failed type checking in parameter file:letting fixed be [[one, 9, 7, 0, 8, 0, 4, 0, 0],
[0, 4, 0, 0, 0, 5, 7, 8, 0],
[0, 6, 0, 0, 0, 7, 0, 1, 0],
[9, 0, 6, 0, 7, 0, 0, 4, 3],
[4, 0, 3, 2, 0, 6, 0, 0, 8],
[8, 0, 1, 3, 0, 9, 0, 0, 0],
[6, 8, 5, 1, 3, 0, 0, 0, 0],
[0, 3, 4, 5, 9, 0, 2, 6, 0],
[0, 1, 0, 0, 6, 0, 0, 0, 0]]
Demystify process exited with code 1. An error occurred.
```

Figure 9: Invalid Board based on Parameters.

6. Evaluation

6.1 Analysis of the Grading Metric

This section first discusses the determination of different variable weights used in the formula, and the grade level ‘cut off’ ranges. This discussion focuses on the reason behind why it is crucial to strike a balance between deriving most information from the *sumOfMUSSizes* variable due to its more extensive scale and propping up the *largestMUSSize* variable with more weight to account for the occurrence of ‘bottlenecks’ in a puzzle where a player is likely to get stuck. Secondly, this section provides an overall evaluation of the accuracy of the grading metric in its capacity to provide a reasonable difficulty grade to a Sudoku puzzle. Note that accuracy is generally subjective in this instance. However, the performance of the grading metric is tested against many other online generate puzzles, which have already been graded, to see how this paper’s grading formulae measure up to other notable methods.

6.2.1 Variable Weights

The initial formula weights assume that the difficulty of a Sudoku puzzle comes from both the difficulty of the most challenging Sudoku strategy (i.e., a Tough or Diabolical strategy) and the quantity of Sudoku strategies above ‘eyeballing’ (i.e., Naked and Hidden Candidates to Tough or Diabolical strategies) required to solve a puzzle. This assumption aligns well with most Sudoku difficulty rating formulas which assign points to solve strategies depending on how difficult they are and then add up the points of each strategy used to get the final solution to get an overall grade scale. In general, Basic strategies will contribute points based on the frequency of their occurrence (not high individual points), and the Tough and Diabolical strategies will contribute points based on their high point score (see, for example, Stuart, 2007; Pelánek, 2014; sudokuoftheday.com).

It is reasonable to assume that the *sumOfMUSSizes* value reveals the most about the puzzle – it displays the total size of MUSes used to solve the entire puzzle – and the *largestMUSSize* value reveals the most challenging step a player needs to overcome to solve the puzzle. A grade formula with ‘costs’ could exponentially increase the cost of strategies that are extremely tough to solve and that few people have encountered. Thus, based on a lot of statistical analysis and experience in solving puzzles, such formulas account for this assumption by increasing the points of a more complicated strategy nonlinearly (unlike the linear progression in MUS size) (Stuart, 2007).

By thoughtfully assigning weights, and because the values are not in a standard scale (the *largestMUSSize* is just one of the MUS sizes added in the *sumOfMUSSizes*), the formula can assign extra points to the hardest solving step in the sum of all solving steps. In doing so, the formula can place more value on the hardest solving step relative to all other solving steps but also derive most of the difficulty from the number of hard cells to solve based on the *sumOfMUSSizes* value. By taking

this approach, the formula can balance accounting for the hardest cell to solve (the 'bottleneck') and the number of cells that require more than 'eyeballing' to solve.

After the initial formula weights were set, based on the understanding of what makes a puzzle difficult discussed above (and discussed in more detail in Sections 3.3.1 and 4.2.1), I used a method of personal testing of trying to solve the puzzles myself and reviewing the algorithm's output afterward to observe what caused certain puzzles to seem more challenging than others. It is important to note that the variable weighting requires further research and statistical analysis. With that in mind, the formula, with weights set, is as follows,

$$(1.) DSI = 80(largestMUSSize) + 10(sumOfMUSSizes) + 10(emptyCellCount)$$

Note that if the variables were normalized to a common value scale (e.g., all variable values between 0 and 1), the variable weights should be weighted more closely and would directly depend on what is subjectively determined to provide more insight into a puzzle's difficulty.

The *emptyCellCount* variable carries the same weight as the *sumOfMUSSizes*. However, because the *emptyCellCount* variable is capped at 72 (89 cells – 17 minimum clues, see Section 2.1) and usually capped at 60, it only adds a significant portion of the difficulty score during the easy and medium (and sometimes the easier hard) puzzles. This limitation is in line with the discussion in Section 4.2.1; the purpose of the *emptyCellCount* variable is to differentiate between easy and medium puzzles.

An alternative approach is to manipulate the values directly before weighing them. An idea could be to convert the *sumOfMUSSizes* value to an average MUS size. Doing so would bring the *sumOfMUSSizes* value closer to the *largestMUSSize* value. However, these changes would cause the new value (i.e., average MUS size) to misrepresent what the *sumOfMUSSizes* value currently represents in the formula - the size and number of complicated strategies required to solve a puzzle. For example, average MUS size would rate a puzzle of MUS sizes [8,8] more complicated than a puzzle of [8,8,5,5,3] (the first has a higher average), which would not make sense. It is better to use the variables that most accurately account for a puzzle's difficulty and try to manage values that are out of scale through thoughtful weighting.

6.2.2 Grade Level

Finally, the algorithm provides a grade level (e.g., easy) to the puzzles based on the output of the difficulty formula. The grade levels are presented in Table 1 below.

Grade Scale	Difficulty Points
Easy	0 - 500
Medium	501 - 1360
Hard	1361 - 2400
Very Hard	2401 - 3500
Evil	3500 +

Table 1: Difficulty Levels based on Formula (1.)

The 'cut off' values for the easy and medium puzzles are based on specific criteria, not personal testing. For example, an easy puzzle can only derive its difficulty score from the number of empty cells. Therefore, an easy puzzle generated by this algorithm should only require 'eyeballing' to solve. Therefore, there is a maximum difficulty level score of 500 Difficulty Points (i.e., if there is ever a MUS of size [1] or more, it will move up past the easy difficulty level – unless there are an unreasonable number of empty cells, e.g., 20). Regarding medium puzzles, they should only involve *naked* and *hidden pairs* at most, as *naked* and *hidden triples* can be complicated for new and intermediate players. To determine what MUS size is associated with *naked* and *hidden pairs*, the example of the *naked* and *hidden pair* Sudoku puzzles from [sudokuwiki.com](https://www.sudokuwiki.com) were put through DEMYSTIFY to observe the output (see, https://www.sudokuwiki.org/Naked_Candidates). The largest MUS size from naked pairs is [3], and the largest from hidden pairs is [8]. Therefore, any puzzle with a largestMUSSize equal to or greater than [9] is automatically pushed into the hard category. At the simplest level, a puzzle of MUS sizes [9] with 55 empty cells would receive a grade of 1360. Therefore, the upper 'cut-off' for medium-difficulty puzzles is 1360.

The specific 'cut off' values become significantly more complicated to determine from hard upwards. However, based on a combination of testing puzzles from [sudokuwiki.com](https://www.sudokuwiki.com) with DEMYSTIFY and personal testing, a thoughtful estimation of puzzle difficulty ranges from hard up is derived. From personal experience, most hard Sudokus can be solved with only Basic strategies (albeit the hardest Basic strategies), with the challenge coming from the number of harder Basic strategies, such as hidden and naked pairs, triples, and quads. The Hidden Triples and Hidden Quads examples from [sudokuwiki.com](https://www.sudokuwiki.com) were put through DEMYSTIFY. The largest MUS sizes were [7] and [6], respectively – less than the largest MUS size from the hidden pairs example, highlighting the difficulty of grading puzzles by assigning a MUS size directly to a solving strategy.

An upper 'cut off' of 2700 is set for hard puzzles, assuming the largest MUS size should be between [7 - 10], with many MUSes [7] and less in other solving steps. Already, the 'cut off' ranges have mainly become subjective, as there may be puzzles with MUSes of sizes greater than [7] with fewer MUSes overall and will be graded as medium puzzles. Moreover, it is difficult to account for

puzzles with one substantial MUS size (hard step) and lots of small ones, although this research has rarely encountered such puzzles.

From very hard to evil, it becomes even more complicated as puzzle difficulties found in published puzzles in this category begin to vary substantially in difficulty. Some very hard puzzles are personally too challenging to solve, while others can take ten to fifteen minutes to solve. Moreover, some evil puzzles are impossible, while others are hard but still solvable.

The difficulty of evil puzzles is the most subjective. This subjectivity is because of the pre-existing knowledge required to identify and use an expert-level sudoku-solving technique. For example, if an evil puzzle is very tricky but can be solved without tough or diabolical strategies, then players who are good at Sudoku but not experts (i.e., not learned strategies) could solve it. However, suppose solving an evil puzzle requires some tough or diabolical strategies. In that case, knowledge of these strategies is a prerequisite, and thus, only players with expert knowledge (i.e., learned strategies) can solve them.

6.2.3 Performance

To give some overview of how the MUS grading formula performs relative to other graded puzzles, an example of 25 puzzles from sudoku.com, the New York Times, and Sudoku Wiki were put through the algorithm to see how their grade ratings performed to the ones presented in this paper. Because each of these sites uses different grade categories (i.e., the New York Times only has easy, medium, and hard), table 2 below shows how this paper has grouped their grade categories relative to the ones presented in this paper.

Publisher	Grade Category	Grade Category in Paper
Sudoku.com	Easy	Easy
	Medium	Medium
	Hard	Hard
	Very Hard	Very Hard
	Evil	Evil
SudokuWiki	Easiest	Easy
	Gentle	Easy
	Moderate	Medium
	Tough	Hard, Very Hard
	Diabolical	Evil
NY Times	Easy	Easy, Medium
	Medium	Hard
	Hard	Very Hard, Evil

Table 2: Grade Category Matching Between Publishers

The comparisons are made to observe how the grades provided by different publishers (which would presumably use different formulas) differ from a grade formula that uses MUSes. There is no right or wrong ‘grade.’ The results are presented in Table 3 below.

Puzzle	Website	Website Grade	Formula Grade	Formula
1	Sudoku.com	Easy	Easy	430
2	Sudoku.com	Easy	Easy	430
3	Nytimes.com	Easy	Easy	430
4	Nytimes.com	Easy	Easy	430
5	Sudoku.com	Medium	Medium	780
6	Sudoku.com	Medium	Medium	510
7	Sudoku.com	Medium	Medium	1310
8	Nytimes.com	Medium	Hard	1740
9	Nytimes.com	Medium	Hard	2360
10	Sudoku.com	Hard	Hard	2680
11	Sudoku.com	Hard	Very Hard	2990
12	Sudoku.com	Hard	Evil	5730
13	Sudoku.com	Hard	Hard	1480
14	Nytimes.com	Hard	Medium	1270
15	Nytimes.com	Hard	Hard	2670
16	Sudoku.com	Very Hard	Hard	2370
17	Sudoku.com	Very Hard	Very Hard	3060
18	Sudoku.com	Very Hard	Evil	4600
19	Sudoku.com	Very Hard	Very Hard	3050
20	Sudoku.com	Evil	Evil	4050
21	Sudoku.com	Evil	Hard	1480
22	Sudoku.com	Evil	Hard	2250
23	Sudoku.com	Evil	Evil	8410
24	Sudokuwiki.org	Diabolical	Evil	5150
25	Sudokuwiki.org	Diabolical	Very Hard	2800

Table 3: Grades relative to other publishers for the same puzzles

Overall, the grade formula tends to provide grade ratings that are the same or one grade different than those from sudoku.com, nytimes.com, and sudokuwiki.org. This similarity indicates

this paper's method's success in providing reasonably accurate grades relative to notable Sudoku publishers. However, there were three outliers in rows 12, 21, and 22. The Sudoku.com Hard puzzle, in row 12, received a grade of Evil based on this paper's formula, with a score of 5730 – the second highest! In contrast, the Sudoku.com Evil puzzle, in row 21, received a grade of Hard (and could almost be an easy puzzle). Given the large discrepancy in these grade ratings, they were personally tested – it took 16:31 minutes to solve the row 12 puzzle and 11:24 minutes to solve the row 21 puzzle. Thus, the grade formula in this paper evaluated the puzzle difficulties more accurately based on these solve times. However, the difference in solving time is not much. Considering a score of 5370 (with a largestMUSSize of [16]), this puzzle should be much harder to solve and require some tough or diabolical solving techniques, which I did not need to solve the puzzle.

Even more interesting is that I could not solve the Evil puzzle in row 23 and could only manage to fill in a few numbers in over forty-five minutes. However, the puzzle in row 23 (score of 8410 and largestMUSSize of [23]) is closer in score to the puzzle in row 12 (score of 5370) than the puzzle in row 12 is to puzzle in row 21 (score of 1480). It would be interesting to research what specific MUS size causes this sudden jump in difficulty.

Considering that these grades are based on MUS sizes and empty cell count alone is impressive. Even though the specifications of the grade formula need further calibration, i.e., normalization of the variable weights and further testing of specific grade ranges, it is evident that, generally, the size and number of MUSes increase as the puzzles increase in difficulty. These results indicate that MUSes can offer logical solutions for problems expressed as CSPs, making them a reliable source for reasonable explanations.

6.2.4 Limitations of Variable Weights

A shortcoming of the formula is using variable weights to ‘normalize’ the variables. When both are unweighted, the emptyCellCount value will always be larger than the largestMUSSize value (apart from maybe the most complicated puzzles). When both are weighted, it will, generally, be larger than the largestMUSSize value until the largestMUSSize value reaches a size of 6 or 7. Additionally, unweighted, the sumOfMUSSizes value will always be equal to or larger (usually larger) than the largestMUSSize value. The sumOfMUSSizes value also increases significantly as the puzzles increase in difficulty, as more steps require MUSes to solve. Although the formula accounts for this discrepancy, to an extent, through variable weights, a more sophisticated approach would be to normalize the variables directly before weighting them. This approach would only be possible after a substantial number of puzzles are generated and tested to accurately get each variable's minimum, maximum, and mean values from a large dataset.

6.3 Original Objectives

The initial primary objectives of creating a Sudoku and Miracle Sudoku generating algorithm were met. While the focus on using MUSes to provide step-by-step explanations and difficulty measurements quickly converged into only focusing on difficulty measurements, it was useful to have developed a deep understanding of how MUSes can be used as explanations (this is a foundation for understanding difficulty). The focus on the Miracle Sudoku was the biggest change in this paper, as it did not present the interesting challenge it looked to be when it was initially presented on Cracking the Cryptic in 2020 – it is very limited in scope.

The two primary goals of this paper were achieved by designing a system that generates sudoku puzzles and grades them based on the MUSes used when solving the puzzles in *DEMYSTIFY*. Moreover, the work in this paper went further to combine the number and size of the MUSes in such a way as to meaningfully develop a formula to quantify the difficulty of puzzles without needing a pre-set, and ranked, list of solving techniques, i.e., the grades provided are not simply based on the number of MUSes or the sum of their sizes.

The Backtracking Search algorithm in this paper represented the sudoku generating process as CSP and solved the problem by first generating a filled sudoku board and second solving backward to get the starting sudoku board. Moreover, the algorithm creates puzzles so to meet the standards for a good puzzle discussed in Stuart (2007).

While the grading algorithm and formula presented in this paper is by no means perfect, it does meet the objectives set out in the beginning of this project; the approach uses a solver based on MUSes (Espasa et al., 2023) to provide a difficulty metric for the generated puzzles. This difficulty metric has been shown to perform reasonably well against other notable puzzle publishers. Therefore, the grading approach did involve developing a thoughtful method of combining the MUSes to solve each puzzle and calculate a reasonable difficulty metric – satisfying primary objective (2.).

7. Conclusions

7.1 Summary

This paper presents an algorithm that generates a valid filled-in Sudoku puzzle, then solves the puzzle backward to display a unique starting Sudoku puzzle. The Backtracking Search algorithm performs efficiently based on its run times and ability to create valid, unique, and symmetrical Sudoku puzzles, according to the guidelines of good puzzles presented in Stuart (2007). The grading algorithm can assign a thoughtful difficulty rating (and grade level) to a puzzle based on the number and size of the MUSes used when solving the puzzle by DEMYSTIFY. The grading formula used to determine the difficulty rating combines and weights the MUS variables thoughtfully to account for what makes a puzzle difficult at different grade levels.

The algorithm's usefulness is demonstrated by the similarity of its grade output to the grades given by other notable puzzle publishers. Therefore, the paper was able to meet its primary objective and provide a more general method for measuring the difficulty of Sudoku puzzles by combining methods from existing grade formulas (such as from Stuart (2007)) with a solving algorithm based on MUSes (in Espasa et al. (2023)).

Although the grading formula's specifications are subjective, as the variable weights, grade category ranges, and combinations of variables rely mainly on personal testing, the results show that the puzzle's difficulty level increases proportionally with the number and size of MUSes used to solve it. These findings provide valuable insights into the potential of algorithms based on MUSes, like DEMYSTIFY in Espasa et al. (2023), to measure the complexity of logical problems. This insight particularly applies to puzzles like Sudoku, where the previous difficulty measuring methods are usually based on directly modelling human-solving techniques.

This paper also presented an algorithm that generates a valid filled-in Miracle Sudoku puzzle, then solves the puzzle backward to display a unique starting Miracle Sudoku puzzle according to the rules presented by Mitchell Lee.

In summary, the work in this paper suggests that MUSes are a viable means for assessing the difficulty level of Sudoku puzzles. These findings further the previous research that employed MUSes in logic-based pen-and-paper puzzles because they indicate that MUSes have explanatory value when applied to Sudoku puzzles – they have been able to explain something about how difficult a Sudoku puzzle is.

7.2 Limitations and Future Work

Due to the scope of creating a general method for grading something as widely studied and popular as Sudoku, the task of this project became more extensive than initially anticipated. As such, this paper utilized a significant amount of personal testing. Although the reasons for the specific grade

'cut-off' levels are based on thoughtful consideration of work done by Stuart (2007) and Pelánek (2014), they are still subjective. As such, more work is needed to calibrate the grading formula over a large database of puzzles. Existing grading formulas have access to lots of human-solving data, which helps the creators gain insight into what works and what does not. An interesting avenue for further research is to undertake a statistical analysis of human-solve times on puzzles graded with MUSes (e.g., to see at what MUS size players begin to struggle).

This statistical analysis would also allow future work to accurately normalize the MUS and empty cell variables directly before weighting them. This approach would be possible after a substantial number of puzzles are generated and tested to accurately get each variable's minimum, maximum, and mean values from a large dataset. Doing so would enable a more sophisticated structure to the existing formula. The grading formula could be modified to use MUSes to directly account for the different Sudoku-solving strategies. In this way, a formula could develop a better understanding of the exact MUS size of all different strategies, if this is indeed possible.

An interesting aspect would be determining how to grade puzzles of a specific difficulty with MUSes. While it is simple to keep a puzzle easy (if a MUS goes beyond a threshold size when solving in DEMYSTIFY backtrack), it is difficult to determine how to keep a puzzle hard as not all solving steps have to be complicated. Additionally, using MUSes to provide difficulty ratings to Sudoku puzzles can be applied to many other pen-and-paper puzzles. A good place to start would be the several other puzzles analysed in the work of Espasa et al. (2023), as DEMYSTIFY can directly solve such puzzles with MUSes.

Finally, given that the grading algorithm is based on the output of MUSes used to solve the puzzle by DEMYSTIFY, a limitation of the paper is not comparing the approach of finding MUSes in DEMYSTIFY to alternative methods to compute MUSes—particularly the approach by Bogaerts et al. (2020) who also used MUSes to provide interpretable explanations. Further research into the most applicable MUS computing algorithm is needed.

References

- Anthony, S., & Goodliffe, M. (2020). The Miracle Sudoku. Cracking the Cryptic. Retrieved from: <https://www.youtube.com/watch?v=yKf9aUIxdb4>
- Barták, R. (2005). Constraint Propagation and Backtracking-based Search. Charles University, Prag. Retrieved from: <http://jayurbain.com/msoe/cs4881/CPschool05notes.pdf>
- Biere, A., Heule, M., van Maaren, H., & Walsh, T. (2021). *Handbook of Satisfiability - Second Edition, of Frontiers in Artificial Intelligence and Applications*. IOS Press. doi:10.3233/FAIA336.
- Bogaerts, B., Gamba, E., Claes, J., & Guns, T. (2020). Stepwise Explanations of Constraint Satisfaction Problems. In *24th European Conference on Artificial Intelligence (ECAI)*.
- Brailsford, S. C., Potts, C. N., Smith, B. M. (1998). Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, (119), 557-581.
- Chabert, M., & Solnon, C. (2020). A Global Constraint for the Exact Cover Problem: Application to Conceptual Clustering. *Journal of Artificial Intelligence Research*, (67), 509 - 547. 10.1613/jair.1.11870. hal-02507186
- Chinneck, J. W., & Dravnieks, E. W. (1991). Locating minimal infeasible constraint sets in linear programs. *INFORMS J. Computing.*, 3(2), 157–168.
- Cormen, T. H. (2009). *Introduction to algorithms: Third Edition*. Cambridge, Mass: MIT Press.
- Dasgupta, S., & Chandru, V. (2004). Minimal Unsatisfiable Sets: Classification and Bounds. In: *Maher, M.J. (eds) Advances in Computer Science – ASIAN*. Higher-Level Decision Making. https://doi.org/10.1007/978-3-540-30502-6_24
- de Siqueira N., J. L., & Puget, J. (1988). Explanation-based generalisation of failures. In *8th European Conference on Artificial Intelligence*, 339–344.
- Deng, X. Q., & Da Li, Y. (2013). A novel Hybrid Genetic Algorithm for Solving Sudoku Puzzles. *Optimization Letters*, (7), 2, 241–257.
- Ekström, J., & Pitkälä, K. (2014). The backtracking algorithm and different representations for solving Sudoku Puzzles. *KTH Computer Science and Communication*.
- Espasa, J., Gent, I. P., Hoffman, R., Jefferson, C., Lynch, A. M., Salamon, A., McIlree, M. J. (2023). Using Small MUSes to Explain How to Solve Pen and Paper Puzzles. ArXiv, arXiv:2104.15040v2
- Felgenhauer, B., & Jarvis, F. (2005). Enumerating possible Sudoku grids. Retrieved from: <https://www.semanticscholar.org/paper/Enumerating-possible-Sudoku-grids-Felgenhauer-Jarvis/0ea50870ff4968425f984f03f99748a0e2d1553d>
- Fletcher, S., Johnson F., & Morrison D. R. (2007). Taking the mystery out of Sudoku difficulty: an Oracular model. *UMAP Journal*, (29), 3, 327–347.

- Fowler, G. (2009). A 9x9 Sudoku Solver and Generator. AT&T Labs Research.
- Fried, D. (2017). Boolean Formulas. Lecture, COMP 587: Computational Complexity. Rice University, Houston, Texas.
- Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: WH Freeman.
- Green, D. (2006). Conceptis Sudoku Difficulty Levels Explained. <https://www.conceptispuzzles.com/>. Retrieved from <https://www.conceptispuzzles.com/index.aspx?uri=info/article/2#:~:text=There%20are%20very%20hard%20puzzles,puzzle%20is%20going%20to%20be>.
- Gunther, J., & Moon, T. (2012). Entropy Minimization for Solving Sudoku. *IEEE Transactions on Signal Processing*, (60),1, 508–513, 2012.
- Hemery, F., Lecoutre, C., Sais, L., & Boussemart, F. (2006). Extracting mucs from constraint networks. In *ECAI 2006, 17th European Conference on Artificial Intelligence*, 113–117.
- Hereford, J. M., & Gerlach, H. (2008). Integer-valued particle swarm optimization applied to Sudoku puzzles. *IEEE Swarm Intelligence Symposium (SIS)*, 1–7.
- Herimanto, Sitorus, P., Zamzami, E. M. (2019). An Implementation of Backtracking Algorithm for Solving A Sudoku-Puzzle Based on Android. *Journal of Physics: Conference Series*.
- Hunt, M., Pong C., & Tucker, G. (2007). Difficulty-driven Sudoku puzzle generation/ *UMAP Journal*, (290), 3, 343–361.
- Junker, U. (2001). QuickXPlain: Conflict detection for arbitrary constraint propagation algorithms. In *Workshop on Modelling and Solving problems with constraints (IJCAI)*.
- Karp, R., M. (1972). Reducibility Among Combinatorial Problems. *Complexity of Computer Computations, Springer*. 85-103.
- Kendall, G., Parkes, A., & Spoerer, K. (2008). A survey of NP-Complete Puzzles. *ICGA Journal*, (31), 13-34.
- Knuth, D. E., (2000). Dancing Links. arXiv:cs/0011047v1.
- Liu, Z. (1998). Algorithms for Constraint Satisfaction Problems (CSPs). University of Waterloo, Canada.
- Lloyd, H. & Amos, M. 2019. Solving Sudoku With Ant Colony Optimization. *IEEE Transactions on Games*. 10.1109/TG.2019.2942773.
- Mantere, T. & Koljonen, J. (2007). Solving, Rating and Generating Sudoku puzzles with GA. *IEEE Congress on Evolutionary Computation (CEC)*, 1382–1389.
- Marques-Silva, J., Janota, M., & Belov, A. (2013). Minimal sets over monotone predicates in Boolean formulae. In *Computer Aided Verification - 25th International Conference*, 592–607.
- Mazure, B., Sais, L., & Grégoire, E'. (1998). Boosting complete techniques thanks to local search methods. *Ann. Math. Artificial Intelligence*, 22(3-4), 319–331.

- McCarthy, E. (2020). On Miracle Sudoku. <https://ethmcc.github.io/>. Retrieved from: <https://ethmcc.github.io/miracle-sudoku/#:~:text=There%20are%20exactly%2072%20fully,digit%20as%201%20through%2009.>
- McGuire, G., Tugemann, B., Civario, G. (2012). There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem. arXiv:1201.0749v2
- McMillan, K. L. (2003). Interpolation and sat-based model checking. In *Computer Aided Verification, 15th International Conference*, 1–13.
- MDN Contributors. (2023). Working with objects. MDN Web Docs, Mozilla Developer. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_objects#
- Meng, J., & Lu, X. (2011). The Design of the Algorithm of Creating Sudoku Puzzle. In: Tan, Y., Shi, Y., Chai, Y., Wang, G. (eds) *Advances in Swarm Intelligence*. ICSI 2011.
- Moraglio, A., & Togelius, J. (2007). Geometric particle swarm optimization for the Sudoku puzzle. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, 118–125.
- Norvig, P. (n.d.). Solving Every Sudoku Puzzle. <https://norvig.com/>. Retrieved from: <https://norvig.com/sudoku.html>
- Pacurib, J. A., Seno, G. M. M., & Yusiong, J. P. T. (2009). Solving Sudoku puzzles using improved artificial bee colony algorithm. *Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, 885–888.
- Pelánek, R. (2014). Difficulty Rating of Sudoku Puzzles: An Overview and Evaluation. Faculty of Informatics, Masaryk University Brno. arXiv:1403.7373v1
- Puskar, L. (2016). An Exploration of the Minimum Clue Sudoku Problem. Sacred Heart University.: Academic Festival. Retrieved from: <https://digitalcommons.sacredheart.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1089&context=acadfest>
Retrieved from: https://doi.org/10.1007/978-3-642-21524-7_52
- Russel, S. J., & Norvig, P. (2015). *Artificial Intelligence: A Modern Approach, Third Edition*. New Jersey: Pearson Education, Inc.
- Segura, C., Penã, S. I. V., Rionda, S. B., & Aguirre, A. H. (2016). The importance of Diversity in the Application of Evolutionary Algorithms to the Sudoku Problem. *IEEE Congress on Evolutionary Computation (CEC)*, 919–926.
- Simonis, H. (2005). Sudoku as a Constraint Problem. In *CP Workshop on modelling and reformulating Constraint Satisfaction Problems*, (12), 126-142.

- Stuart, A. (2008). Sudoku Wiki. <http://www.sudokuwiki.org/>. Retrieved from <http://web.archive.org/web/20210131095006/https://www.sudokuwiki.org/sudoku.htm>
- Stuart, A. C. (2007). Sudoku Creation and Grading. *Mathematica*, 39(6), 126-142. Retrieved from: https://www.sudokuwiki.org/Sudoku_Creation_and_Grading.pdf
- Tomlinson, H. (2005). Puzzle Difficulty Ratings. <https://www.sudokuoftheday.com/>. Retrieved from <https://www.sudokuoftheday.com/difficulty>
- W3schools. (n.d.). JSON.parse(). <https://www.w3schools.com/>. Retrieved from: https://www.w3schools.com/js/js_json_parse.asp
- Wang, Z., Yasuda, T., & Ohkura, K. (2015). An evolutionary approach to Sudoku puzzles with filtered mutations. *IEEE Congress on Evolutionary Computation (CEC)*, 1732–1737.
- Weber, T. (2005). A SAT-based Sudoku Solver. In *The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR): Short Paper Proceedings*, 11–15.
- Yato, T., & Seta, T. (2003). Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 1052 - 1060.

Appendix

Appendix A: User Manual

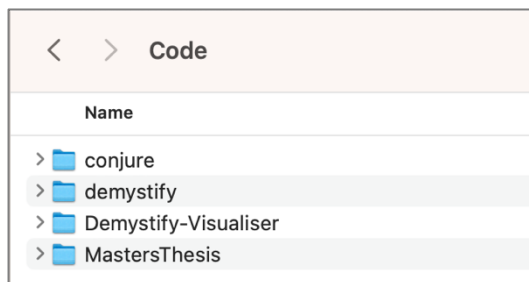
A1: Running the Whole Process

Step 1: Install DEMYSTIFY

Go to <https://github.com/stacs-cp/demystify/blob/master/README.md> and follow DEMYSTIFY and Conjure install instructions.

Step 2: Clone MastersThesis Folder into Repository

Have Conjure, DEMYSTIFY, and the MastersThesis folders in same location to make accessing them easier.



Step 3: Make Modifications to Difficulty_API.js File

Go to line 68. From here you will need to change the locations of where you want to run the python scripts from (which run the DEMYSTIFY solver). To be safe, you can hard code the exact locations of all the specific files. Modify the lines highlighted in blue to the location where you have the Conjure, DEMYSTIFY, and MastersThesis files.

```

71 var call_demystify = function() {
72   const pythonScript = '/Users/benjonsson/Desktop/Masters/CS5099/Code/demystify/demystify';
73   const terminalInput = [
74     pythonScript,
75     '--eprime',
76     '/Users/benjonsson/Desktop/Masters/CS5099/Code/demystify/eprime/sudoku.eprime',
77     '--eprimeparam',
78     '/Users/benjonsson/Desktop/Masters/CS5099/Code/MastersThesis/TemporaryStorage/final_board.param',
79     '--json',
80     `${boardString}.json`
81   ];
82
83   const demystify_python = spawn('python3', terminalInput, { cwd: '/Users/benjonsson/Desktop/Masters/CS5099/Code/MastersThesis/DemystifyOutput' });
84

```

Step 4: Set Working Directory to Sudoku folder.

Open a new terminal at the MastersThesis folder, and then run:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
benjonsson@8afbe871 MastersThesis % cd Sudoku
```

Step 5: Run Algorithm

Then run:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
benjonsson@8afbe871 MastersThesis % cd Sudoku
benjonsson@8afbe871 Sudoku % node Difficulty_API.js
```

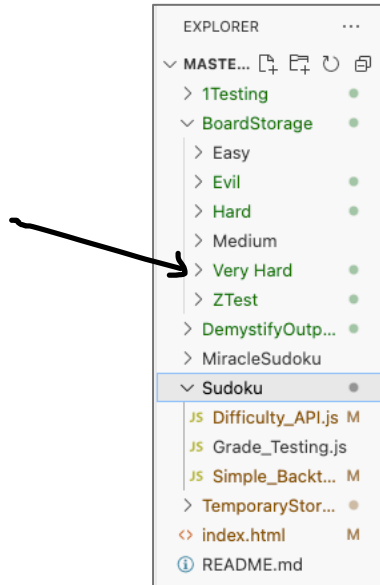
Step 6: Observe Terminal Output

First, you will be displayed this output in the terminal:

```
benjonsson@8afbe871 Sudoku % node Difficulty_API.js
Final Valid Board Test Passed
Starting Sudoku:
[9,5,0,0,0,7,0,0,8]
[0,1,0,0,0,0,9,0,0]
[0,4,0,0,2,0,3,0,0]
[0,0,0,3,6,0,8,0,0]
[0,2,0,0,0,0,0,0,0]
[0,0,8,0,4,0,0,0,0]
[0,0,3,0,8,0,0,6,0]
[0,0,1,0,0,0,0,4,0]
[2,0,0,9,0,0,0,0,0]

Solved Sudoku:
[9,5,2,4,3,7,6,1,8]
[3,1,7,8,5,6,9,2,4]
[8,4,6,1,2,9,3,7,5]
[4,7,9,3,6,1,8,5,2]
[1,2,5,7,9,8,4,3,6]
[6,3,8,5,4,2,1,9,7]
[7,9,3,2,8,4,5,6,1]
[5,8,1,6,7,3,2,4,9]
[2,6,4,9,1,5,7,8,3]
950007008010000900040020300000360800020000000008040000003080060001000040200900000
Easy Time: 74.19ms
Final board saved successfully!
Demystify process completed successfully
[
  3, 3, 3, 3, 3, 3, 3, 3, 3, 5,
  5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
  5, 3, 3, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 3, 3, 3, 0
]
7
59
The sum is: 193. Empty Cells: 59.
Very Hard
Very Hard
New board saved successfully!
benjonsson@8afbe871 Sudoku %
```

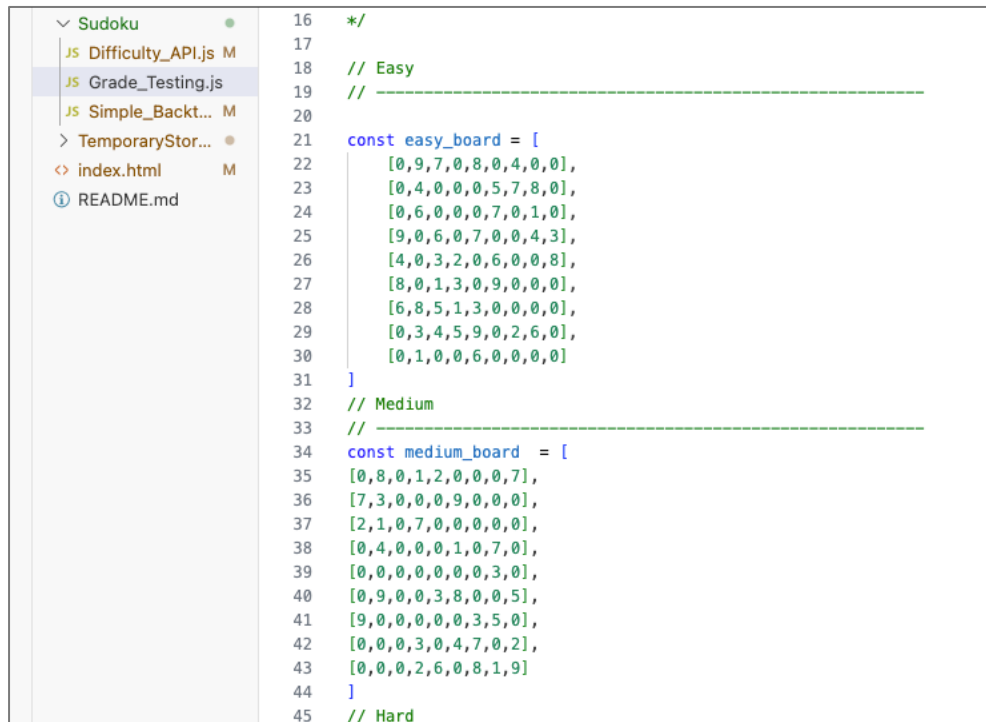
Step 6: Observe Saved Board Output

[illegible]

A2: Testing Pre-Generated Puzzles

Step 1: Input Board

If you want to run existing puzzles, say from other publishers, through the grading formula you can input the boards manually in the Grade_Testing.js file.



```

16  */
17
18  // Easy
19  // -----
20
21  const easy_board = [
22    [0,9,7,0,8,0,4,0,0],
23    [0,4,0,0,0,5,7,8,0],
24    [0,6,0,0,0,7,0,1,0],
25    [9,0,6,0,7,0,0,4,3],
26    [4,0,3,2,0,6,0,0,8],
27    [8,0,1,3,0,9,0,0,0],
28    [6,8,5,1,3,0,0,0,0],
29    [0,3,4,5,9,0,2,6,0],
30    [0,1,0,0,6,0,0,0,0]
31  ]
32  // Medium
33  // -----
34  const medium_board = [
35    [0,8,0,1,2,0,0,0,7],
36    [7,3,0,0,0,9,0,0,0],
37    [2,1,0,7,0,0,0,0,0],
38    [0,4,0,0,0,1,0,7,0],
39    [0,0,0,0,0,0,0,3,0],
40    [0,9,0,0,3,8,0,0,5],
41    [9,0,0,0,0,0,3,5,0],
42    [0,0,0,3,0,4,7,0,2],
43    [0,0,0,2,6,0,8,1,9]
44  ]
45  // Hard

```

Step 2: Modify Code

To have DEMYSTIFY solve a specific puzzle and get a grade output, you need to go the Difficulty_API.js and comment out line 23 and uncomment out line 17 to access the pre generated boards. Change the code in line 25 and set the board variable equal to the board you previously filled in.

```

15
16 // Import for performance analysis.
17 const {easy_board, medium_board, hard_board, very_hard_board, evil_board} = require('./Grade_Testing.js');
18
19 //Import for Miracle Sudoku
20 //const {startingSudokuBoard, populatedSudokuBoard} = require('./MiracleSudoku/Miracle.js');
21
22 // Import
23 //const {startingSudokuBoard, populatedSudokuBoard} = require('./Simple_Backtracking.js');
24
25 var board = hard_board;

```

Also comment out line 237 and remove line 254.

```

230 function saveBoardWithDifficultyGrade(largestMUSSize, EmptyCellCount, smallestMUSSizes, SumOfMUSSizes, DifficultyScore, X1, Y1, Z1, grade){
231 // Save the populated board to a .param file
232
233 // Intial Board
234 let output1 = printBoardToFile(board);
235
236 // Solution
237 let output2 = printBoardToFile(populatedSudokuBoard);
238 const storageContent = ` The Puzzle:
239
240 ${output1}
241
242 -----
243 List of Smallest Minimal Unsatisfiable Sets (MUSes) at Each Solving Step in Descending Order:
244 ${smallestMUSSizes}
245 -----
246 Difficulty Metrics:
247 Largest 'smallest' MUS Size (${largestMUSSize}), Sum of MUS Sizes (${SumOfMUSSizes}), Empty Cell Count (${EmptyCellCount})
248 -----
249 Difficulty Score: ${DifficultyScore} = X1 * largestMUSSize + Y1 * SumOfMUSSizes + Z1 * EmptyCellCount
250 Where, X1: ${X1}, Y1: ${Y1}, Z1: ${Z1}
251 Assigned Grade: ${grade}
252 -----
253 Solution:
254 ${output2}
255

```

A3: Running the Process for Miracle Sudoku

Step 1: Change Some Code to Account for Different Puzzle Format

For DEMYSTIFY to solve a Miracle Sudoku puzzle instead of a Sudoku puzzle you need to change the code in line 73 of the Difficulty_API.js file from sudoku.eprime to miracle.eprime.

```

71 var call_demystify = function() {
72 const pythonScript = '/Users/benjonsson/Desktop/Masters/CS5099/Code/demystify/demystify';
73 const terminalInput = [
74   pythonScript,
75   '--eprime',
76   '/Users/benjonsson/Desktop/Masters/CS5099/Code/demystify/eprime/miracle.eprime',
77   '--eprimeparam',
78   '/Users/benjonsson/Desktop/Masters/CS5099/Code/MastersThesis/TemporaryStorage/final_board.param',
79   '--json',
80   `${boardString}.json`
81 ];
82
83 const demystify_python = spawn('python3', terminalInput, { cwd: '/Users/benjonsson/Desktop/Masters/CS5099/Code/MastersThesis/DemystifyOutput' });
84

```

And uncomment out the line 20.

```
16 // Import for performance analysis.
17 //const {easy_board, medium_board, hard_board, very_hard_board, evil_board} = require('./Grade_Testing.js');
18
19 //Import for Miracle Sudoku
20 const {startingSudokuBoard, populatedSudokuBoard} = require('../MiracleSudoku/Miracle.js');
21
22 // Import
23 //const {startingSudokuBoard, populatedSudokuBoard} = require('./Simple_Backtracking.js');
24
25 var board = startingSudokuBoard;
26
27 function saveStartingBoard(){
28   // Save the populated board to a .param file
29   const content = `language ESSENCE' 1.0
30 }
```