DIFFICULTY ANALYSIS OF SUDOKU PUZZLES

SHARIAR AZAD

MUHAMMAD ZUBAIR HASAN

NAZIFA HAMID

A THESIS SUBMITTED FOR

THE DEGREE OF BACHELOR OF SCIENCE IN
COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY

2020

SUPERVISOR'S APPROVAL

This thesis paper titled **"DIFFICULTY ANALYSIS OF SUDOKU PUZZLES"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in 2020.

Abu Wasif
Assistant Professor,
Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology (BUET),
Dhaka, Bangladesh

DECLARATION

This is to certify that the work presented in this thesis paper, titled, "DIFFICULTY ANALYSIS OF SUDOKU PUZZLES", is the outcome of the investigation and research carried out by the following students under the supervision of Abu Wasif, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh.

It is also declared that neither this thesis paper nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Shariar Azad
Roll: 201714037
14 March 2021

Muhammad Zubair Hasan
Roll: 201714041
14 March 2021

Nazifa Hamid
Roll: 201414044
14 March 2021

ABSTRACT

In this thesis, we have analyzed the difficulty of sudoku puzzles. For this, we have designed and implemented a sudoku solver which is based on human-like solving strategies and also how our solver compares to a brute-force solver. The two solvers were tested to analyze difficulty and the test was based in terms of solving speed, symmetry, number of empty regions. The experiments on our datasets suggest that the human solver is comparable to computer based Sudoku solvers and it is generally more efficient in terms of solving speed. Although the human solver does not guarantee solutions to all Sudokus; but the puzzles it was able to solve suggests that difficulty does not really depend on our assumed parameters.

ACKNOWLEDGEMENT

We are thankful to Almighty Allah for his blessings for the successful completion of our thesis. Our heartiest gratitude, profound indebtedness and deep respect go to our supervisor, Abu Wasif, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis. We are especially grateful to the Department of Computer Science and Engineering (CSE) of Military Institute of Science and Technology (MIST) for providing their all out support during the thesis work.

Finally, we would like to thank our families and our course mates for their appreciable assistance, patience and suggestions during the course of our thesis.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Artificial intelligence is intelligence demonstrated by machines, unlike the natural intelligence displayed by humans and animals [1]. AI problems can be solved using different kinds of methods and CSP is one of them. In AI Constraint satisfaction problems or CSPs are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations [2]. Sudoku is a puzzle invented in Japan and it can be solved using CSP. There are also more efficient ways to solve Sudoku using search based, logic based and computational based algorithms. A lot people daily solve Sudoku of different difficulty on various websites. The science behind the difficulty of the Sudoku puzzle is truly fascinating.

1.1 Motivation

Puzzle games are becoming more and more popular and attracting players of all ages. People take their time playing puzzles in traffic or in leisure time like spending some time playing puzzles in offices during intermission [3]. Also now-a-days some modern games like adventure games and RPGs now have logic puzzles integrated into the game story [4]. In the field of puzzle solving difficulty rating is a very important thing. Most classic puzzles offer problems in a wide range of difficulty so that it is easy to start playing but challenging to master. A good game teaches players everything it has to offer before players stop playing .There is a common notion that a problem too easy offers nothing to be learned rather makes it boring while a problem too hard can make players stop playing before even they have a grasp of the game properly [4]. So researchers have taken quite a bit of interest in finding the intricacies and inner workings of Sudoku difficulty. Our motivation is to design scales in increasing difficulty so that players do not get frustrated too easily trying to solve hard puzzles in the first run; rather they would be able to learn through practice and experience by choosing the proper difficulty and keep the game interesting.

1.2 Sudoku

Sudoku is a puzzle that consists of a 9 x 9 grid partially filled with numbers from 1 to 9. Each box is called a cell. In a puzzle certain no of clues are given and each empty cell has certain no possible candidates. The goal is to place numbers 1 to 9 to each cell in such a way that in each

row, column, and 3 x 3 sub-grid, each number occurs exactly once. Sudoku is also considered as a constraint satisfaction problem where Variable : each cell, Domain: 1 to 9, Constraints: AllDiff (each row, each column, each box). There are many other constraint satisfaction problems which are puzzles and also numerous real life problems such as timetabling, scheduling etc. Our objective is to conduct difficulty analysis and metrics as general as possible, so that the results may be applicable to other constraint satisfaction problems [5]. Sudoku has been the focal point of many research studies, mainly with respect to its mathematical and algorithmic features, e.g., enumerating possible Sudoku grid [5, 6], NP-completeness of generalized version of Sudoku [5, 7], formulation as constraint satisfaction problem [5, 8, 9] and even psychological aspects of the puzzle have been studied [5, 10]. The difficulty rating of Sudoku puzzles is obviously not a unique problem and the disparity of the Sudoku difficulty rating is widely discussed among Sudoku players and researchers. Current rating algorithms are based mainly on personal preferences and random tuning [5]. There are several research papers which discuss methods for difficulty rating [9, 11, 12]. In this thesis, the difficulty of Sudoku puzzles are analyzed with respect to solving speed, symmetry, size of vacant regions and the type of solving algorithm used. Each of the algorithms has a rating which classifies their difficulty. The scales used in this thesis to rate puzzles are easy, medium and hard.

1.3 Objective

The main objective of this thesis is:

- to study and understand logic and search based solving techniques of Sudoku
- to implement these techniques efficiently
- to study features of Sudoku puzzles
- to find correlation between features and hardness.

1.4 Overview of Our Works

This thesis presents an idea to analyze difficulty of Sudoku puzzles. For our work we used Gordon Royle's 17 clue dataset and 3 millions Sudoku puzzles generated by Blagovest Dachev's Sudoku generator from Kaggle [13, 14]. Using these datasets we studied the features of Sudoku puzzles by using human like and computational algorithms and their relation to hardness. This thesis involves the area of artificial intelligence and computational mathematics.

1.5 Application

Sudoku is a very popular puzzle. Many people find it difficult to start playing the puzzle because often Sudoku hardness rating system is defined arbitrarily. Although in our thesis we have only experimented on the features of hardness, in the future works a rating system for hardness maybe proposed so that hardness becomes less ambiguous. This thesis findings would then be finally integrated into online puzzle solving platforms so that players find it easier to learn this game.

CHAPTER 2
LITERATURE REVIEW

In this section some studies that are relevant to our thesis topic is presented. This chapter discusses the previous work done by researchers on difficulty analysis of Sudoku puzzles, difficulty rating systems, how the rating system can be implemented to other puzzles and so on. Active research is being conducted at present times on puzzle difficulty.

2.1 Related Works

A to Z of Sudoku by Narendra Jussien discusses a general overview of the puzzle, history of Sudoku and mathematical representations.The first part consists of rules and techniques to solve Sudoku grids by hand; the second part discusses necessary software development for solving Sudokus (solving, evaluating and generating grids).Finally, the third part lists a set of grids to improve Sudoku solving skills [15]. The rules and techniques described in the book can be divided into 3 categories; basic, advanced and expert techniques. Basic techniques include single position, single candidate, the candidate lines and multiple lines techniques. Advanced techniques include naked pair, naked tuples, hidden pair, hidden tuples, subset rules based duality and region based reasoning.Expert techniques include the X-Wing , swordfish, disjunctive construction, reductio ad absurdum. For our thesis proposed Sudoku solver was based on these techniques and many other human techniques. Finally the difficulty of puzzles were analyzed using human algorithms and brute force.
Taking Sudoku seriously is a book written by Rosenhouse et al and this book describes Sudoku as a math problem and also as a puzzle. This book analyzes the mathematical reasoning behind the unique solutions of puzzles and how this can also be related to other fields of mathematics.In the first chapter various techniques for solving Sudoku puzzles were examined and it also discusses the general question of what constitutes a math problem. Chapter 2 discusses the concepts of a Latin square, an object of huge interest to mathematicians of which Sudoku squares are a special case. Chapter 3 discusses Greco-Latin squares, which are an extension of the idea of a Latin square. Chapters 4 and 5 elaborate on two counting problems related to Sudoku. Specifically, it determines the total number of Sudoku squares and the total number of "fundamentally different" squares.Chapter 6 introduces the problem of how Sudoku puzzles are found interesting and how this problem is within the context of search problems generally. Chapters 7 and 8 investigate connections between Sudoku, graph theory, and polynomials.Chapter 9 is a collection of Hard

Sudoku analysis. Basically it looks for puzzles with the maximal number of empty regions, with the minimal number of given clues, and numerous others. The main goal of this book is to find interesting insights about probability theory and its applications through puzzle solving [16].
Difficulty Rating of Sudoku Puzzles by a Computational Model by Radek Pelánek mainly discusses the parameters that affect the difficulty of a Sudoku puzzle. This paper also analyzed various difficulty rating systems and tested them on 1700 Sudoku puzzles. The best results were found using a computational model of human solving activity. Using this model the conclusion was that there are two sources of the problem difficulty: complexity of individual steps (logic operations) and structure of dependency among steps. Metrics based on analysis of solutions under relaxed constraints were also described which is a novel method inspired by phase transition events in the graph coloring problem. In the article the focus was just not on the performance of individual metrics on the Sudoku puzzle, but also on their generalizability and applicability to other problems [5].
There is no 16-clue Sudoku: solving the Sudoku minimum number of clues problem via hitting set enumeration by Gary McGuire et al answers the question "what is the minimum number of clues that a Sudoku puzzle can have?" For many years it has been assumed that the answer is 17. Rigorous search was conducted to find 16 clue puzzles but none were found. This article mainly discusses the actual search methods and algorithms developed to reach that conclusion. A novel way for enumerating hitting sets was proposed to help with the proof. The hitting set problem is computationally hard; it is one of Karp's 21 classic NP-complete problems. A standard backtracking algorithm for discovering hitting sets would not be adequately fast to look for a 16-clue Sudoku puzzle thoroughly, even at the present supercomputer speeds. To make a comprehensive search possible, an algorithm was designed which efficiently enumerated hitting sets of suitable size [17].
The Chaos Within Sudoku by Ercsey-Ravasz et al proposes a novel way of rating Sudoku puzzle difficulty and uses a novel k-SAT continuous solver to solve puzzles. This paper goes on to analyze the mathematical construction of Sudoku puzzles which is similar to hard constraint satisfaction problems. With a precise planning of Sudoku into a deterministic, continuous time dynamical framework, this article shows that the difficulty of Sudoku puzzles convert into transient chaotic behavior displayed by the developed system. The article additionally shows that the escape rate , an invariant of transient chaos, gives a scalar proportion of the riddle's hardness that associates well with human difficulty ratings. As needs be, $\eta = log10\kappa$ can be utilized to characterize a "Richter"- type scale for puzzle hardness, where simple puzzles has range within $0 < \eta \leq 1$, medium ones $1 < \eta \leq 2$, hard within $2 < \eta \leq 3$ and very hard within $\eta > 3$. In this novel research no puzzle with rating $\eta > 4$ was found [18].
Generate and Solve Sudoku Puzzle by Li Gao et al proposes four levels of difficulty for Sudoku puzzles: Easy, Mild, Hard, Very hard. They also propose a system to rate solving algorithms based on their hardness. After that topics such as generating a full puzzle, removing some of the cells, solving the puzzle and Single- Solution are discussed. At last, a mathematical model to ensure a unique solution was proposed [19, 20].

Automated puzzle difficulty estimation by Van Kreveld et al introduced an interesting method for automatically rating the difficulty of Sudoku puzzles. The proposed method takes multiple aspects of the levels of the puzzles, such as level size, and combines these into a difficulty function. The article discusses how they can be adapted to most puzzle games, and the test was conducted on three different games: Flow, Lazors and Move. A user study was conducted among the players to find out how difficult they found the puzzles to be and later that data was used to train the difficulty function to match the user-provided ratings. These experiments showed that the difficulty function were capable of rating levels with an average error of approximately one point in Lazors and Move, and less than half a point in Flow, on a difficulty scale of 1–10 [4].
Rating Logic Puzzle Difficulty Automatically in a Human Perspective by Hao Wang et al discusses the way to rate the difficulty level of Sudoku problems with human oriented, general difficulty criteria so that the methods can be used to evaluate problems of most Sudoku puzzles. They found that very few previous Sudoku difficulty research are based on real playing data and the rating methods are limited to Sudoku or at most, constraint satisfaction problems (CSP). The proposed method, despite its simplicity and generality, can sort Sudoku problems in an order similar to average player solving time and how a player perceived difficulty [12].

So, in the above discussion many novel Sudoku rating systems and their applications are presented. In our thesis we have simply classified our human-like solving algorithms into scales such as easy, medium and hard. Our work mainly experiments on certain assumed difficulty parameters with a motivation to simplify current Sudoku rating systems so that players do not get confused so much.

CHAPTER 3
ALGORITHMS

To solve the Sudoku puzzles we have implemented two kinds of algorithms in our solver. One is logical rules-based algorithm and another is search-based algorithm. Logical rules-based algorithm is combination of techniques used to solve Sudoku in real life i.e. with pen and paper whereas search-based algorithm is based on backtracking and commonly used to solve Sudoku puzzles in computer programs [21].

3.1 Logical Rules based Algorithm

There are many techniques or strategies to solve Sudoku puzzles starting from easy to advanced. Among them we have implemented 14 in our logical rules-based algorithm. We divided these 14 techniques into three categories: easy, medium and hard. Logical rules-based algorithm solves a puzzle by applying these 14 techniques to the puzzle sequentially starting with easy techniques to hard. Following is a description of the human solving techniques we incorporated in our solver to solve Sudoku puzzles.

3.1.1 Easy Techniques

Singles Candidate: Each row, column or box in a Sudoku board is called a house. Which means a Sudoku board has 27 houses. Firstly, all possible candidates for each square have to be determined. Candidates can also be called "pencil marks" [22]. After determining all the candidates for a house if there is any cell with only one possible candidate left, then that candidate belongs to that cell and we call it a Single Candidate. Also, that candidate can then be removed from the pencil marks of other squares in that particular row, column and box since according to Sudoku rules, same number cannot appear twice in a house [22].
Hidden Singles: Among the candidates or pencil marks of a square if there is one that is a candidate for only that square and no other in that particular house it is called a hidden single.
In the bellow figure [22] among the pencil marks, we can see a 4 being a candidate for the last cell only.

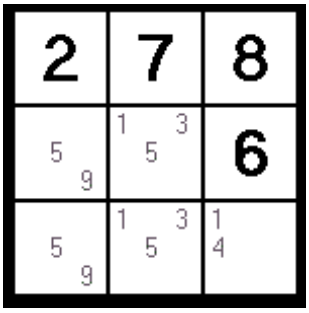

Figure 3.1: Hidden Singles

3.1.2 Medium Techniques

Naked Pair: If two cells in the same house have same two candidates then they are called a naked pair [22]. It means these two candidates belong to these two cells. Although we can not immediately tell which candidate belongs to which cell, this helps remove these two candidates from other cells in that box/row/column. This is a technique to narrow down possible candidates for other cells.

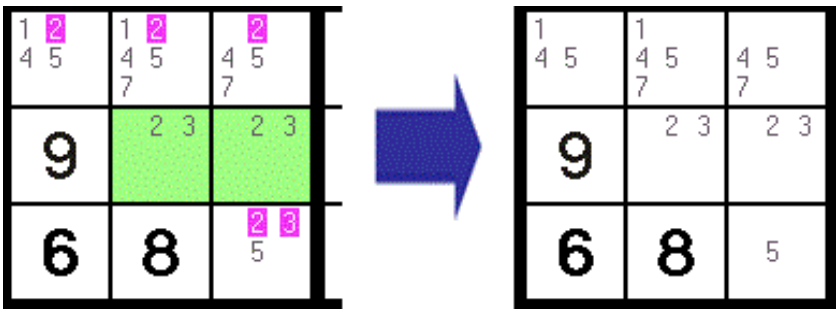

Figure 3.2: Elimination with Naked Pair

Above box has two cells (green marked) having same two candidates 2 and 3. This means 2 and 3 can belong to any of these two cells although we can not yet assign them to cells. This helps us narrow down candidates of other cells in this box by removing 2 and 3 from the pencil marks of other cells [22].

Naked Triples: Naked Triple is rare and has the same principle as Naked Pair except that it involves three candidates instead of two. If there are same three candidates in three cells belonging to the same house or if union of the pencil marks of three cells result in three candidates, it is called a Naked Triple. When a Naked Triple is spotted, we can say that these three candidates belong to these three cells and therefore, they can be removed them from other cells of the house [22].

Figure 3.3: Elimination with Naked Triple

As 1, 5 and 9 are candidates for three cells in the same row we can conclude that they belong

to these three cells and no other in the same row. Therefore, they are removed from the pencil marks of other cells (pink marked) [22].

Naked Quads: Naked Quads have the same principle but involve four candidates and four cells in the same house. They are not easy to find and even rarer [22].

Hidden Pair: Hidden pair is same as a naked pair but they are hidden among other candidates and therefore are hard to find.

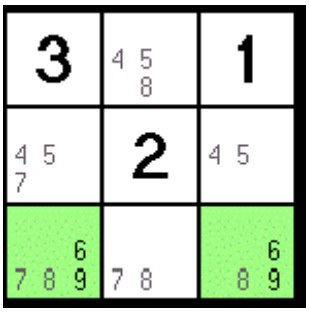

Figure 3.4: Hidden Pair

In the green marked cells 6 and 9 make the hidden pair. This pair belongs to these two cells and therefore other candidates (pink) can be removed from these two cells [22].

Hidden Triple and hidden Quad: Similar to Hidden Pair in a Hidden Triple three possible candidates appear in only three cells in a house hidden among other candidates. Upon finding a hidden triple these candidates are removed from other cells of that house.

A Hidden Quad has the same principle with four candidates. It is even harder to find and not common [22].

Pointing Pairs: If in the same box a number has two cells it is a candidate of and both cells are also in the same row or column, then we can say that this candidate belongs to one of these cells and therefore can be removed from other cells in the same row or column [23].

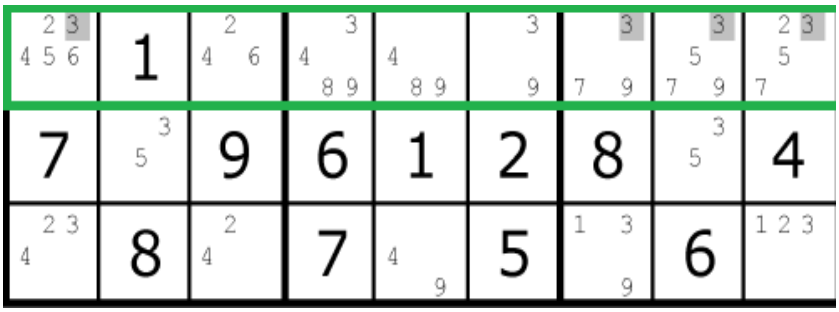

Figure 3.5: Pointing Pair

In the below figure [23], the middle box contains exactly two potential cells for candidate 3. Also these two cells reside in the same row (top). This means that 3 must occur in one of these two cells and can be eliminated from the other cells of the same row (marked). Thus, this is a pointing pair.

Single Chains: Single chain occurs when a pair is chained with another by sharing one same

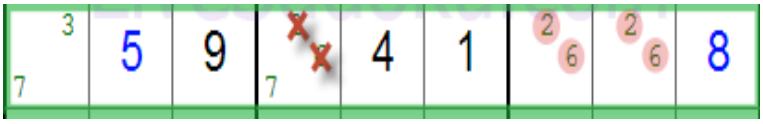

Figure 3.6: Conjugate Pair

candidate [24, 25].

We can see a naked pair in the above figure [24]. Two cells sharing two same candidates. We can also say that 2 forms a conjugate pair and can occur either in the 7th cell or in the 8th. Same goes for 6.

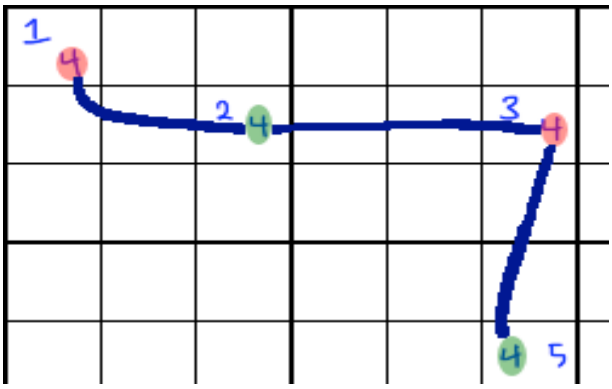

Figure 3.7: Single Chain

For single chains, in the above figure [24], we see that candidate 4 forms a conjugate pair in the upper left box since it is shared by two cells containing 1, 4 and 2, 4. Similarly, 4 forms another conjugate pair in the 2nd row and thus chains these 3 cells. Again, 4 in the two cells in the 6th column forms a conjugate pair and elongates the chain. We use this single chain and apply simple coloring. we take two colors and color each 4 alternatively. In the above figure first 4 is colored red and the next in the chain is colored green, then red and green again. Now, if any of the red colored 4's is correct we can eliminate the green colored 4's. For example, if the red colored 4 in the first cell ends up being true for this cell, it cannot belong to the 2nd cell since they are in the same box and therefore, the it (green) is removed from 2nd cell. This is followed by the next cell in the chain being a 4 (red) since each row must have a 4. Similarly, the next 4 (green) cannot belong to that cell since it shares the same column with the previous one. So, when a single chain is found and one cell is assigned a value, candidates can be eliminated and assigned to cells alternatively in the chain [24, 25].

3.1.3 Hard Techniques

X-wing: X-wing occurs when a particular pencil mark/candidate occurs in two cells of a row and the board contains another row that mirrors it having the same candidate in the same two cells [26].

Blue marked candidates in the above figure [26] form a X-Wing. Since each row in a Sudoku

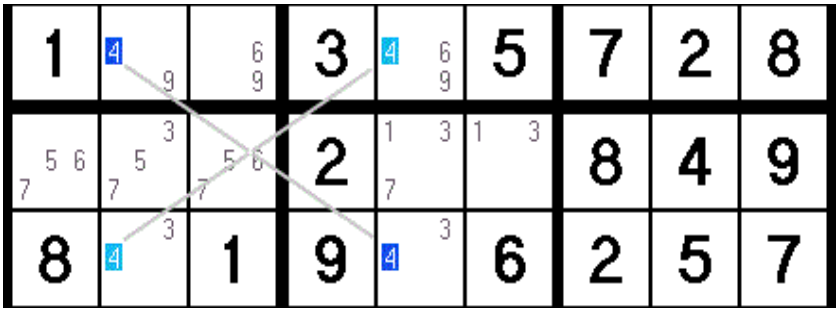

Figure 3.8: X-Wing technique

board must contain values 1-9, each row must have a 4. So, in both rows one of the two cells will have a four. But 4 cannot be in the same column twice which means between cell R1C2 and cell R3C2 only one will be 4. Same goes for the other two cells containing 4 in the same column. This gives us an X-wing [27]. We can now safely say that cells that are in diagonal position will both be 4. which in this case means either the light blue marked cells or the dark blue marked cells will have 4 in them. This way we can eliminate all the other 4's from the pencil marks in these two columns [26, 27].

Y- wing: A Y-wing is formed when a cell contains exactly two potential candidates and each of them forms a pair in two other cells and thus links these three cells. Also, the two other cells must share a row/column or a box with the first cell [26]. In the above figure [28] three cells are

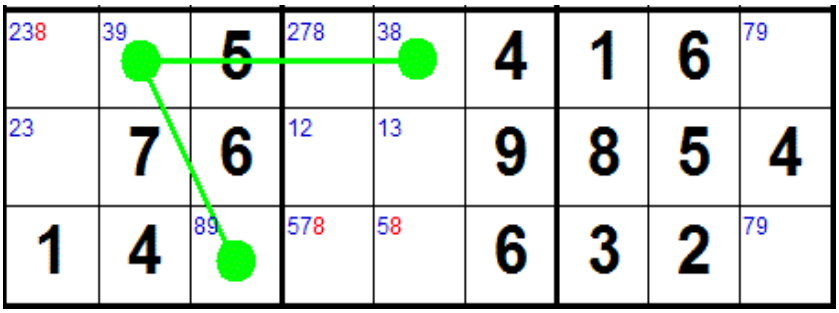

Figure 3.9: Y-Wing technique

connected with a green line. The cell in the middle contains 3 and 9. This cell shares the same box with one of the connected cells and shares row with the other. With the cell in the same row, it forms a link by conjugating a pair with one of its candidates 3 and with the cell in the same box it does the same by conjugating a pair with its other candidate 9. Also, both these other cells called "wings" of the middle cell, share the same other candidate 9.

If the middle cell ends up being 3, it means the cell in the same row with 3, 8 will be 8. But if the middle cell ends up 9, then the cell in the same box will be 8. Which means in either case the one of the wing cells is going to be 8. From this we can come to the conclusion that any cell that shares a house with both these wing cells cannot be an 8. Therefore, we can eliminate 8 from those cells [26, 28]. 8's that can be eliminated are marked red in the figure.

XYZ-Wing: XYZ-wing is similar to Y-wing except that the middle cell has three candidates and shares two of them with other two cells.

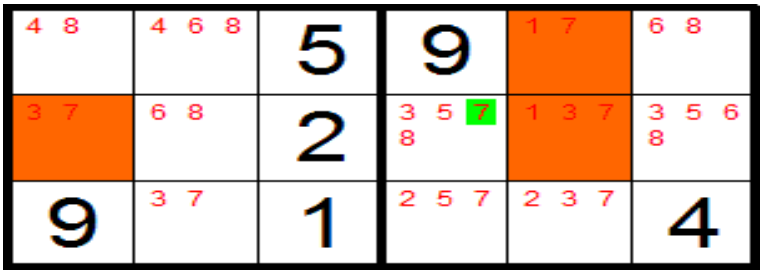

Figure 3.10: XYZ-Wing technique

Here, cells marked orange form a XYZ-wing. The middle cell is the one with three candidates 1, 3 and 7. It shares 1 and 7 with the cell above and 3 and 7 with the cell on its left. If the middle cell is a 1, it means the cell above will be a 7. Therefore, we can eliminate 7 from other cells it shares a house with. Similarly, if the middle cell is 3, the cell on the left will be a 7, and 7 from other cells can be eliminated. If middle cell is 7, the same rule applies. This means that when a XYZ-wing occurs, the candidate shared by all three cells (7 in this case) can removed from cells that share same house with them (marked green) [29].

Swordfish: Swordfish is a variation of X-wing and involves three rows, three columns, two or three cells and one candidate [26].

3.2 Search-based Algorithm

Search based algorithm can also be referred to as brute force. It is based on backtracking which means depth first search is performed. Each cell of a Sudoku puzzle is visited and assigned numbers 1 to 9 sequentially. When it finds a value that is invalid according to the constraints of Sudoku it backtracks whereas when a valid value is found, it is assigned to that cell [21]. Search based algorithm tries all configurations until finds a valid one. Naturally this makes it require longer run time.

We decided to implement the 14 human solving techniques and search based algorithm discussed here in this chapter. The implementation of the algorithms will be discussed in the next chapter.

CHAPTER 4
METHODOLOGY

In this chapter we discuss the approach we took to analyse the difficulty of Sudoku puzzles. We will discuss our take on the evaluation of difficulty, different parameters of evaluation, the datasets we used, how our solver works, solver implementation and testing environment.

4.1 Our approach

After studying other solvers and their approaches we decided to implement logical rules-based algorithm in order to try and experiment with the techniques. As we implemented 14 among the many other human solving techniques, in most cases applying only these techniques resulted in partially solved puzzles. So, we decided to implement the logical rules-based algorithm and search-based algorithm combined. We built a solver that applies the logical rules-based techniques in a specific order until it cannot solve the puzzle anymore with these techniques alone. Then the search-based algorithm or brute force is applied and solving is completed.
With this solver working, in this thesis we analyzed the correlation between difficulty of Sudoku puzzles and various parameters such as region holes, symmetry, clue count, time etc. We collected information from experiments we ran and generated graphs to analyze. We conducted our experiments on two datasets which we will discuss in details in the later section. We analyzed correlations between these parameters or features for each dataset. Also, we analyzed the correlation of each human solving technique with number of clues and size of empty regions.

4.2 Features of Evaluation

To analyze Sudoku hardness we decided to analyze on four parameters. They are discussed below.

4.2.1 Speed

It is the run time required to solve a Sudoku puzzle by our solver. We wanted to verify that if time varied with change of hardness. The time was measured in seconds.

4.2.2 Number of Clues

Secondly no of clues; every puzzle has a certain no of clues. We tried to verify that if hardness varied with change of amount of given clues. Our intuition was that from a given number of clues if we decrease or increase clue count it might change hardness. But for 17 clue dataset we only increase the number of clues as a puzzle can not have clues less than 17. This is the very reason we chose two different datasets so that we can conduct more experimentations.

4.2.3 Rectangular Empty Region

Each Sudoku puzzle has a certain number of clues and most of the time clues are in such an arrangement that there are large empty regions in the puzzle. They can be compared to empty rectangular sub-matrices. The goal was to analyze if their sizes have any effect on difficulty. Figure which shows rectangular empty regions of size 9 and 4. The green marked region in figure 4.1 shows examples of empty regions.

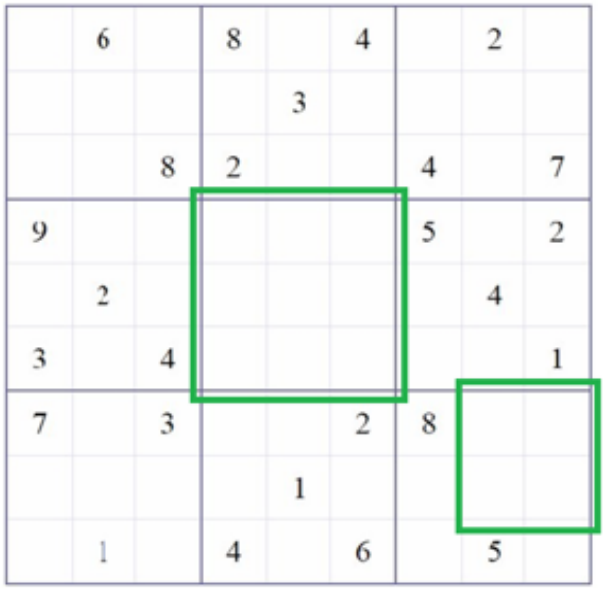

Figure 4.1: Empty Region of different sizes

4.2.4 Symmetry

Symmetry is an active research topic in the field of Sudoku solving.The number of distinct Sudoku grids has been determined to be N= 6.671×1021. But sometimes a partially filled Sudoku puzzle can be rearranged kind of in a hands on way. For example, rotating a valid Sudoku puzzle by 180 degree can create a different puzzle, but is still valid. Similarly, if all the 5's with the 6's in the grid is swapped another valid grid would be found. Let's say for another case if we take a Sudoku grid and switch the places of the fifth and seventh rows, and another valid grid

may be found. When any of these operations are performed on a valid grid, validity property is preserved. These kinds of operations are called symmetries of a grid. A symmetry of an object is a characteristic that shows a certain property of the object. It is interesting to note that if two symmetry operations are performed consecutively, the resultant transformation would also be a symmetry. Additionally, performing one symmetry then a second then a third should be possible by gathering either the first and second together or the second and third together, and the resultant change is the same either way [30].

Figure 4.2: 4-Fold Symmetry

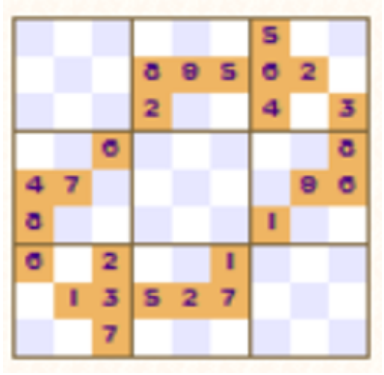

Figure 4.3: 2-Fold Symmetry

For our thesis only 4 types of symmetry of Sudoku puzzles were analyzed. Firstly the 4-fold symmetry shows a symmetry pattern where each quarter (top left, top right, bottom left and bottom right) are the same except that they are rotated through a right angle (90°) [31]. Figure 4.2 shows an example of 4-fold symmetry where we can clearly see 4 regions in top-left, top-right, bottom-left, bottom-right which would remain unchanged if rotated 90°.
The 2-fold symmetry is the pattern if turned upside down then the pattern remains the same. They all have 180° rotational symmetry [31]. This symmetry is shown in figure 4.3.

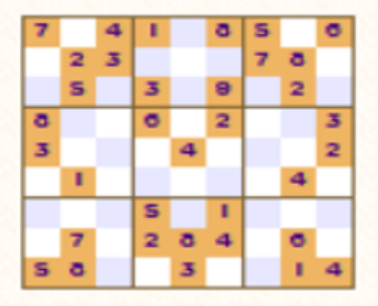

Figure 4.4: Vertical Symmetry

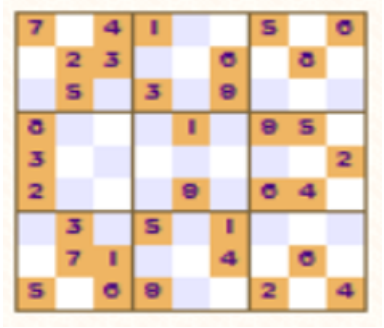

Figure 4.5: Horizontal Symmetry

Finally there is the mirror symmetry where if the puzzle is divided vertically or horizontally it shows a pattern. So when an image is divided vertically and one part is the mirror image of the other that is called the vertical symmetry and when an image is divided horizontally and one part is the mirror of the other that is called the horizontal symmetry [31].Figure 4.4 is an example of vertical symmetry and figure 4.5 is an example of horizontal symmetry.

When a puzzle shows no pattern that is called asymmetry. Figure 4.6 is an example of an asymmetric puzzle.

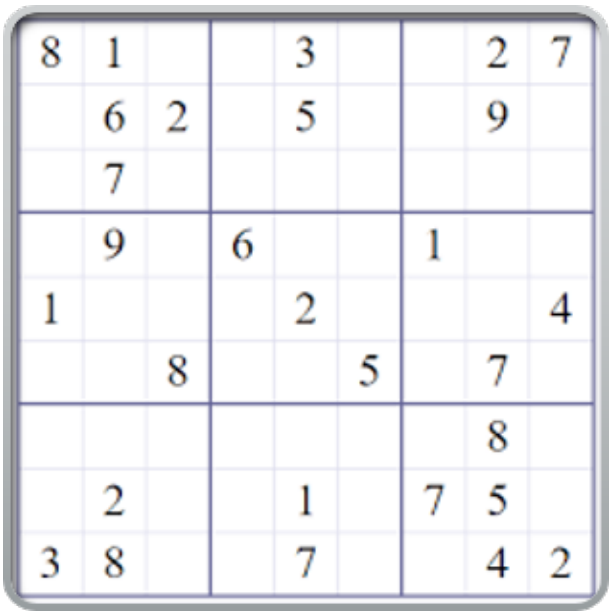

Figure 4.6: Asymmetry

4.3 Dataset

For our analysis we chose 2 datasets. The first one is Gordon Royle's 17 clue dataset [13]. This is a special dataset because 17 clues is the minimum no of clues required to solve a puzzle. We had an intuition that the lesser no of clues would make the puzzles harder to solve. Also the lesser number of clues means there is more room for empty regions and we wanted to verify if there is any pattern to these empty regions which contribute to the hardness of puzzles. So to verify these theories we chose this dataset. Figure 4.7 shows hardness distribution in this dataset.

Table 4.1 shows exact count of the puzzle distribution based on hardness.

17 Clue Dataset	
Difficulty	Count
Easy	15850
Medium	7254
Hard	4311

Table 4.1: 17 clue Puzzle count of different difficulties

We also found that all the puzzles in this dataset, clues were in asymmetric shape which is showed in figure 4.8.

For our 2nd dataset we chose the 3 million Sudoku dataset which was generated by Blagovest

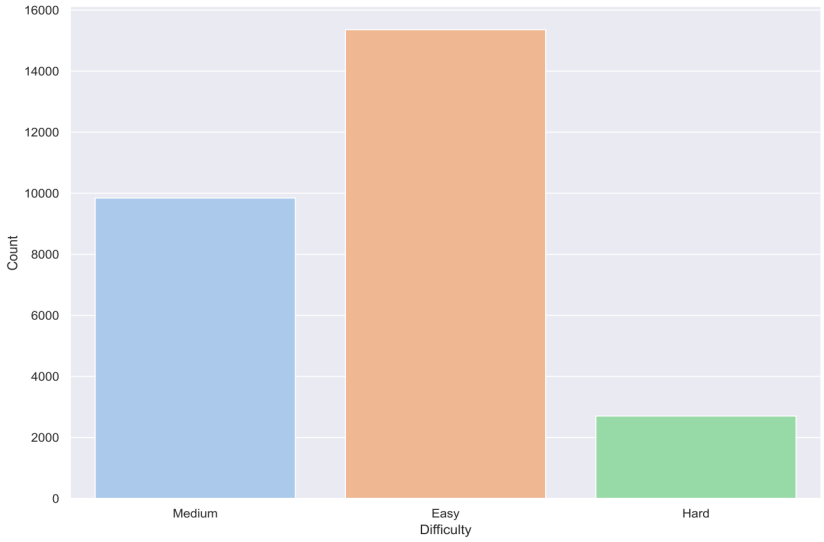

Figure 4.7: 17 clue puzzle count vs difficulty

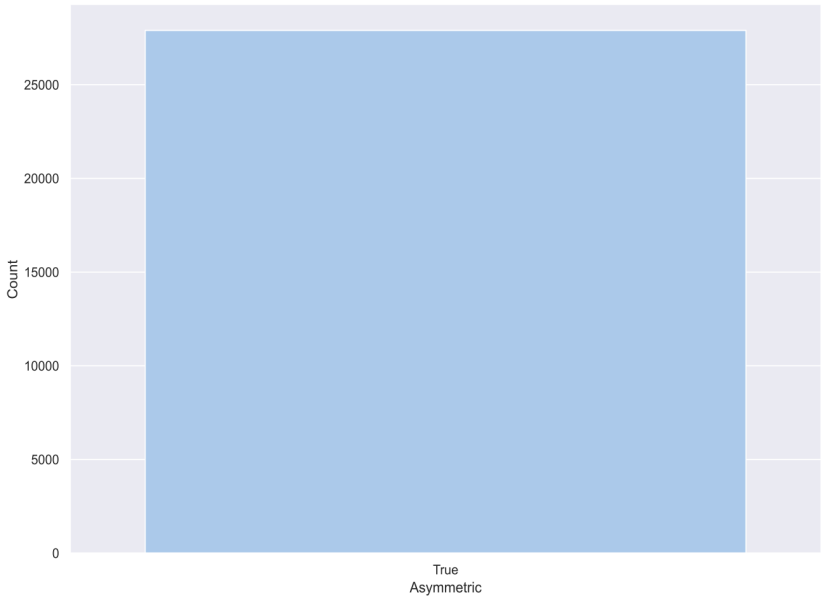

Figure 4.8: Symmetry distribution

Dachev's Sudoku generator and collected from kaggle [14]. This dataset has variable clues Which makes it an even more interesting dataset to check hardness with respect to clue counts, empty regions and symmetry.The minimum number of clues found in the dataset is 19, and the maximum is 31.below. Figure 4.9 shows the clue count distribution with respect to different difficulties.

Table 4.2 shows a summary of variable clue count and different difficulties.

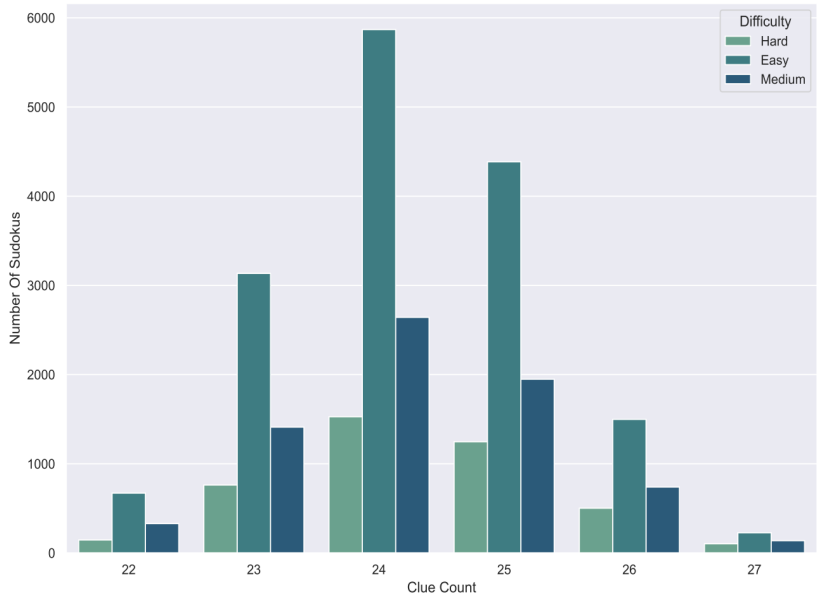

Figure 4.9: Variable clue difficulty vs count

Variable Clue Dataset	
Difficulty	Count
Easy	15358
Medium	9843
Hard	2702

Table 4.2: Variable clue puzzle count of different difficulties

Table 4.3 shows the summary of variable clue empty region size and their count.

Variable Clue Dataset	
Empty Region Size	Count
12	4747
14	16561
15	632
16	5474
20	489

Table 4.3: 17 clue Puzzle count of different difficulties

Figure 4.10 shows empty regions and difficulty wise distribution of the dataset.

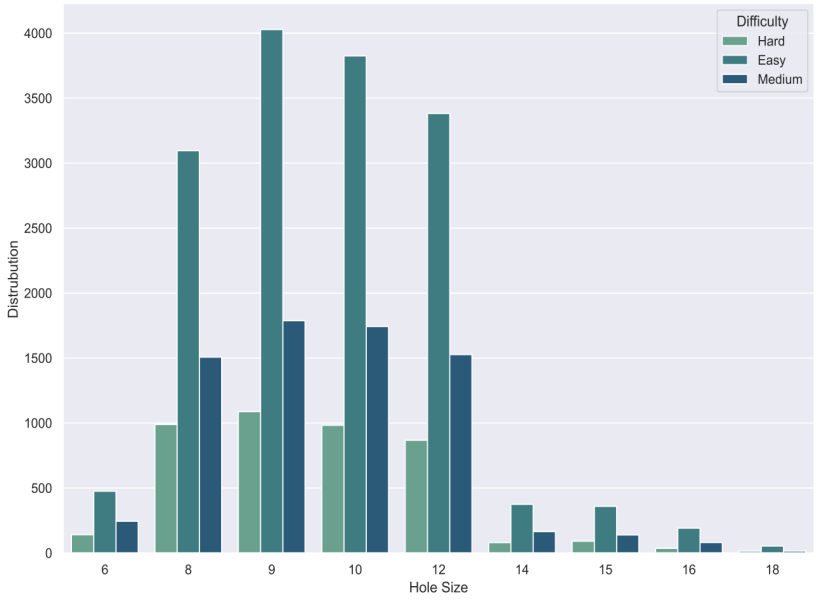

Figure 4.10: Variable clue empty region distribution

4.4 Implementation

Our solver is written in Python language and packages required were NumPy and pandas. A solver function calls all the implemented human solving techniques in a sequential order from easy to hard and then calls brute force or the search-based algorithm (found in appendix B, shared). It assigns a puzzle's difficulty based on the category (easy, medium, hard) of the techniques used and keeps a count on the number of times they are used. Time to solve the puzzle is also saved. This solver function is called in SudokuSolver where the boards are taken as input, initialized, checked for validity and symmetry first (found in appendix B, shared).
Information on difficulty level, count of the techniques used and their count, run time is saved in a text file. Detailed log information about the solution such as which technique is applied in which cell, what value is assigned, when brute force is called is saved in another text file.
To find the hole sizes or vacant regions another script is written separately in python which takes the boards as input, parses in a similar fashion, then finds the size of rectangular vacant regions. It keeps count of boards with that hole size and returns those boards.

4.5 Testing Environment

This section describes the testing configuration of softwares and hardwares used in our thesis experimentation.

4.5.1 Software Configuration

For our testing we have used 3 devices. All the devices were configured with windows 10 operating system. We used Microsoft Visual Studio Code as our code editor and all the necessary python packages were installed. All the devices used python version 3.8.4 and packages required were :

- NumPy == 1.18.4
- matplotlib == 3.2.1
- Pandas == 1.14.0
- seaborn == 0.11.0

4.5.2 Hardware Configuration

Three devices were used for testing. Their configurations are given below :
Device 1:

- CPU intel core i7 8700 @ 4.2 GHz
- RAM DDR4 16GB
- GPU GTX TITAN XP

Device 2:

- CPU Ryzen 9 3900X @ 4.6 GHz
- RAM DDR4 32GB
- GPU GTX 1080 Ti

Device 3:

- CPU intel core i7 6700 @ 3.20 GHz
- RAM DDR4 16GB
- GPU GTX 965m

CHAPTER 5
EXPERIMENTATION AND RESULT

We have used two dataset for our experimentation. The first dataset is Gordon Royle's 17 clue dataset, which helps us eliminate number of clues in a sudoku as a parameter for difficulty. The second dataset is kaggle's 3 million generated sudoku dataset which has varying number of clues. The experiments were conducted based on logic based and search based algorithms.

5.1 Experimentation on Gordon Royle's 17 Clue Dataset

In this dataset every sudoku has 17 clues. We perform numerous analyses and the first one is the time it takes to solve sudokus of our classified difficulties namely easy, medium and hard. We can clearly see from the graphs and the table that easy sudokus take the least amount of time to solve and they are pretty much consistent for all sudokus. But hard sudokus take the most time and it varies a lot for each sudoku.

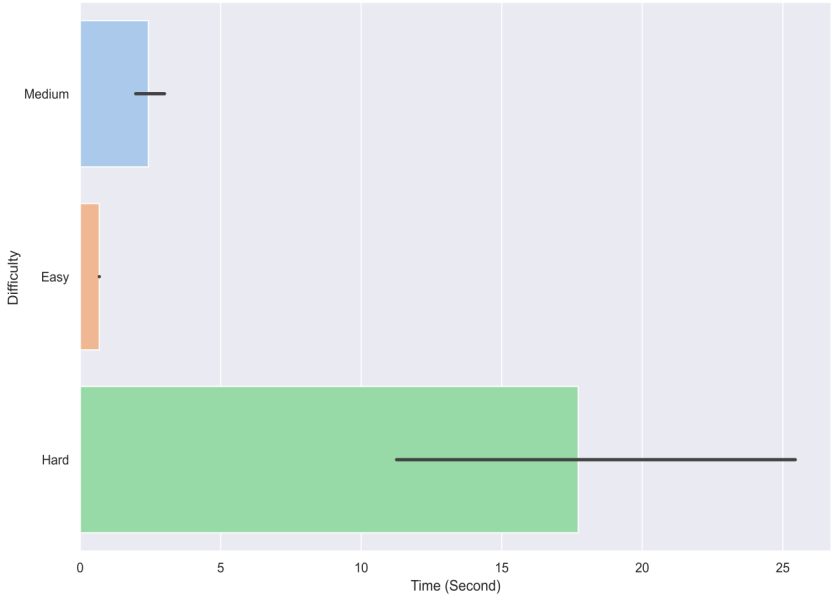

Figure 5.1: Mean Time to solve a Sudoku for different difficulties

In the next analysis we try to find out how much does hole size affect the difficulty of a

Difficulty	Mean	Standard Deviation
Easy	0.682620	0.099542
Medium	2.426535	25.306068
Hard	17.718415	191.142118

Table 5.1: Mean Time to solve a Sudoku for different difficulties

puzzle. In Fig 5.2 we see that the probability of sudoku of all difficulty stays about the same for all the hole sizes. We also see that the distribution of easy medium and hard sudoku is very similar for all the hole sizes in Fig 5.3

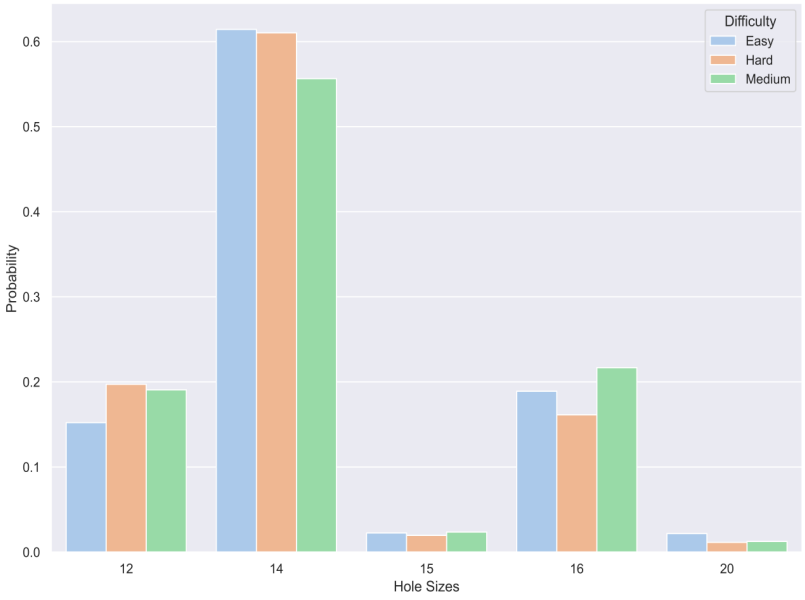

Figure 5.2: Probability of a sudoku having a certain hole size

Hole Size	Easy Probability	Medium Probability	Hard Probability
12	0.152168	0.190694	0.197261
14	0.614338	0.556436	0.610289
15	0.022594	0.023570	0.019615
16	0.189087	0.216804	0.161362
20	0.021813	0.012496	0.011473

Table 5.2: Probability of a sudoku having a certain hole size

In Fig 5.4 we see a line graph with mean time to solve compared to the hole sizes. Previously we established that there was no significant skew to any particular difficulty for any given hole

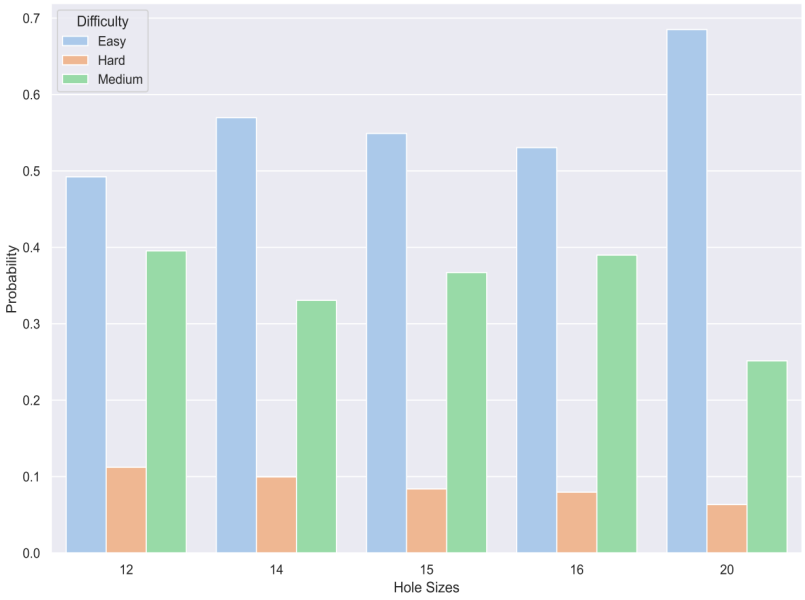

Figure 5.3: Distribution of difficulty among a hole size

Hole Size	Easy Probability	Medium Probability	Hard Probability
12	0.492311	0.395408	0.112281
14	0.569712	0.330717	0.099571
15	0.549051	0.367089	0.083861
16	0.530508	0.389843	0.079649
20	0.685072	0.251534	0.063395

Table 5.3: Distribution of difficulty among a hole size

size. So there is no possibility of hard sudokus of skewing the mean time to solve for a clue count. In our case we can clearly see that when the hole size increases the time to solve a sudoku decreases. Moreover, as the sudokus for all hole sizes are in the same distribution there is no doubt that the downward trend can be attributed to the rising hole sizes.

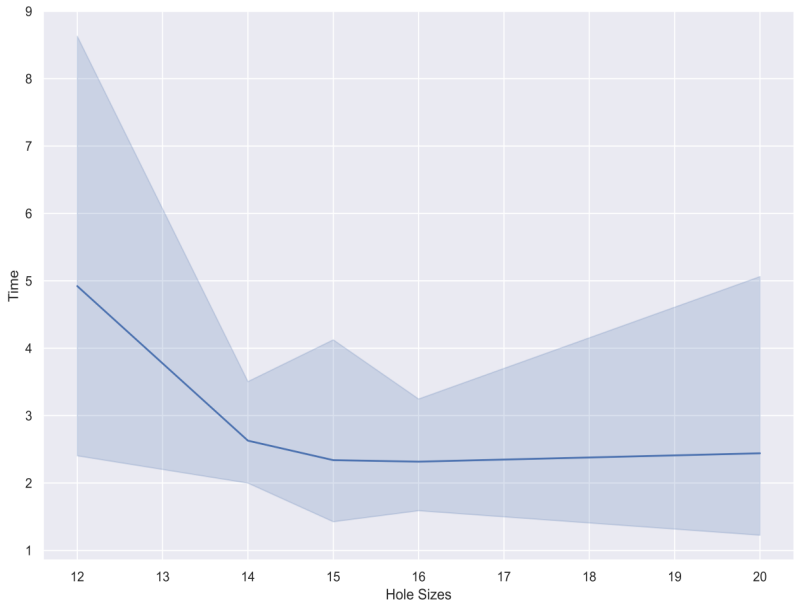

Figure 5.4: Average time to solve for different hole sizes

Hole Size	Time to Solve
12	4.921738
14	2.628236
15	2.339045
16	2.316700
20	2.440943

Table 5.4: Average time to solve for different hole sizes

5.1.1 Logical Rules and Search Based Methods

Finally we try to analyze the count of moves for different difficulties. The most simple and common observation is that easy methods are utilized for all of the puzzles and hard methods are utilized for only hard puzzles. Medium puzzles are utilized for medium and hard puzzles as expected. The other important observation that we can make from these graphs is the fact that easy methods are utilized less on average for hard puzzles. This could be attributed to the fact that hard methods can be used to eliminate more candidates and thus easy methods does not need to be called as much. But, at the same time hard sudoku puzzles utilize more medium difficulty methods than medium difficult sudokus itself. It is interesting because this means that hard puzzles utilize more hard and medium rules, whereas medium difficult sudokus utilize more easy methods and the usage of easy methods are very similar to easy puzzles. Also we can see that there is no obvious trend that can be related to the puzzles hole size and methods. Sometimes downward trends are observed but it can be attributed to a lack of appropriate data.

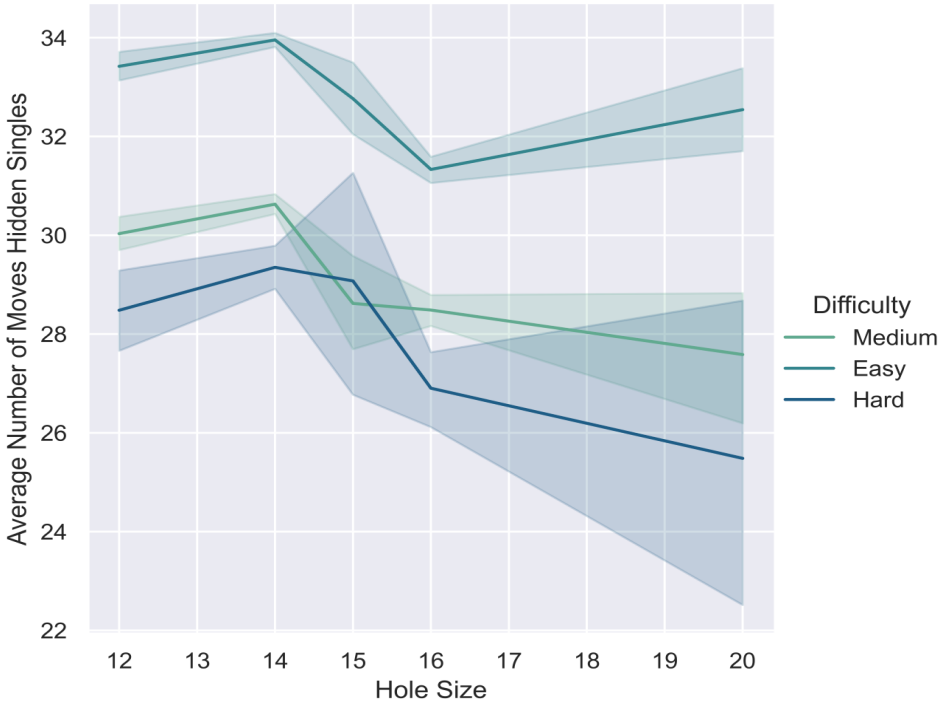

Figure 5.5: Average number of moves for Hidden single for different hole sizes

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Easy Difficulty	Average Number of Moves for Easy Difficulty
12	33.421053	30.033564	28.482176
14	33.955167	30.629907	29.351122
15	32.766571	28.620690	29.075472
16	31.332645	28.486410	26.903670
20	32.543284	27.585366	25.483871

Table 5.5: Average number of moves for Hidden single for different hole sizes

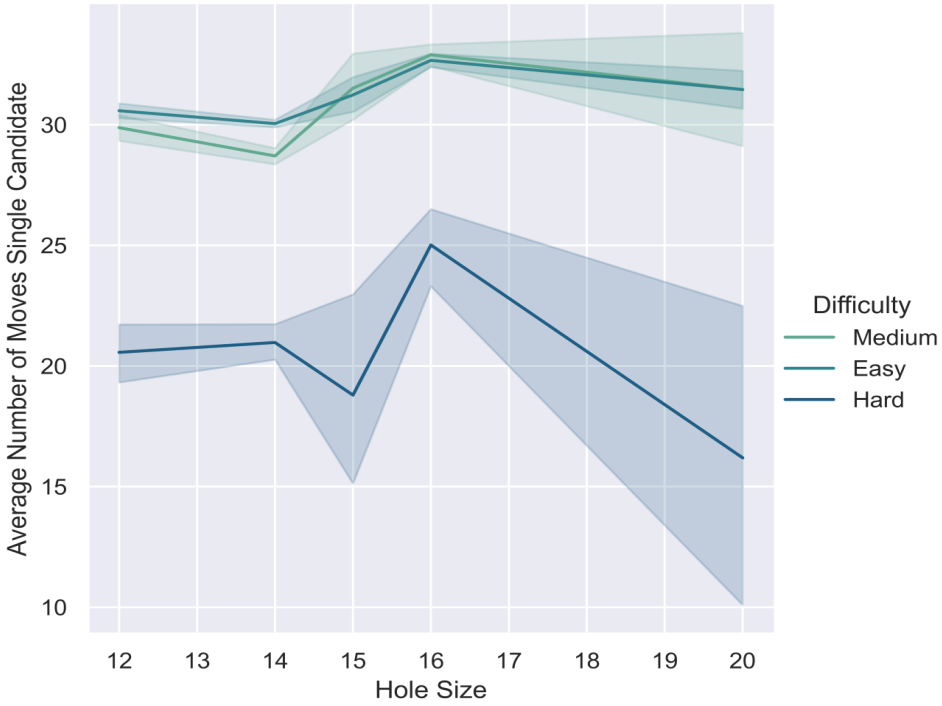

Figure 5.6: Average number of moves for Single candidate for different hole sizes

To end our experiment section we compare logic based method to search based methods or brute force method. Firstly we see that though there are sudokus of different difficulty levels in our dataset the time to solve them does not differ very much when using search based methods. From Fig 5 we can see that the time to solve is more or less the same for all the difficulty levels.

After we see that there is no correlation between the difficulty of a sudoku and the time it takes to solve it with search based methods, we move on to the next step where we see the effect the number of clues have on the time to solve a sudoku using similar methods. Here we see an obvious downward trend where when the number of clues in a sudoku increases the time to solve the sudoku decreases drastically. It is very obvious from the graph, and also we can see that the

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	30.578947	29.876399	20.564728
14	30.044833	28.704583	20.972711
15	31.233429	31.512931	18.792453
16	32.667355	32.895501	25.013761
20	31.456716	31.455285	16.193548

Table 5.6: Average number of moves for Single candidate for different hole sizes

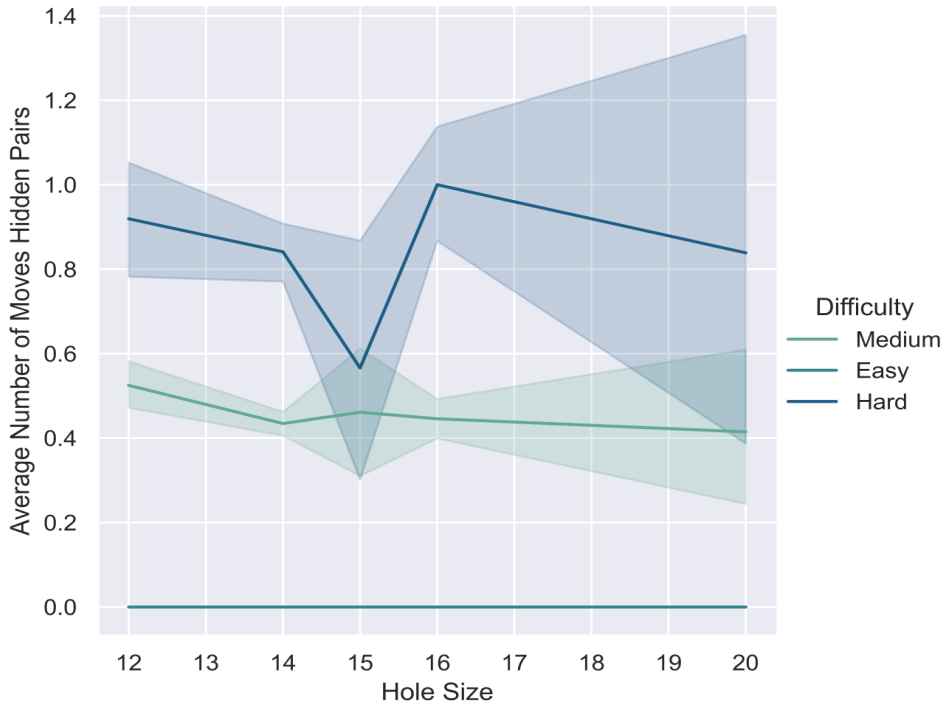

Figure 5.7: Average number of moves for Hidden Pair for different hole sizes

time to solve also stays about the same for all the difficulty levels.

We look at the next figure where time to solve is averaged over different hole sizes. In this figure we can see an opposite picture. Previously we saw that when the clue count increases the time to solve decreases, but in this figure the time to solve increases when hole size increases. But, similar to our previous observation there is no pattern for sudokus of different difficulty with equal hole sizes.

In our final graph we plot a scatter-plot of Search based methods time to solve vs logic based methods time to solve. This gives us some interesting insights. Firstly, search based methods can take a lot of time compared to logic based methods in the worst case. Secondly, logic based methods have much less outliers compared to brute force(search based method). Thirdly, logic

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	0.524774	0.919325
14	0.0	0.434362	0.841116
15	0.0	0.461207	0.566038
16	0.0	0.445642	1.000000
20	0.0	0.414634	0.838710

Table 5.7: Average number of moves for Hidden Pair for different hole sizes

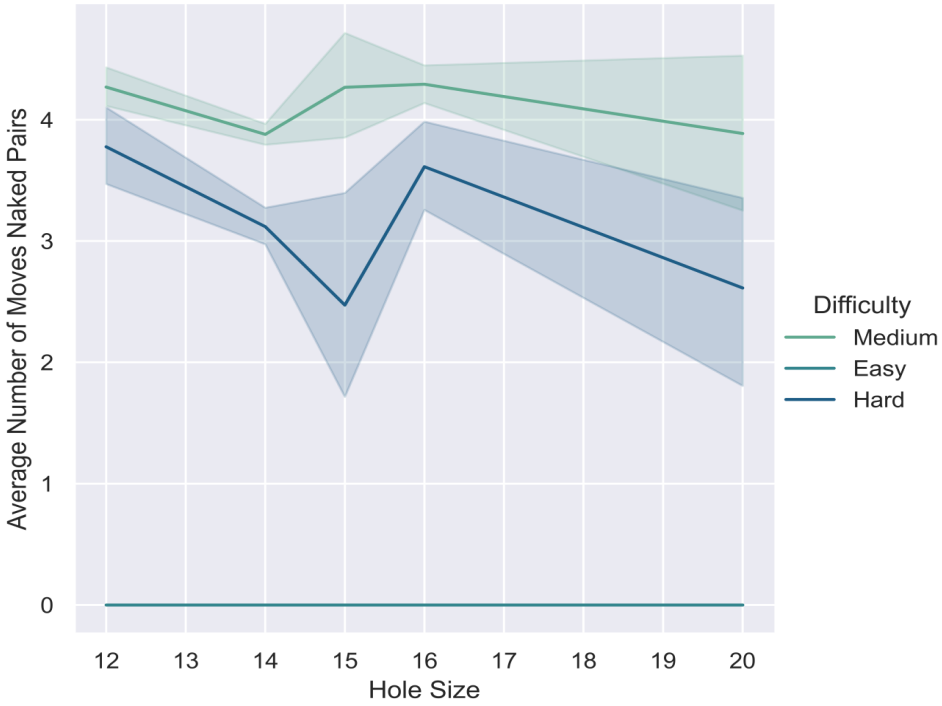

Figure 5.8: Average number of moves for Naked Pairs for different hole sizes

based methods clearly define an area for different difficulties which is not seen in the brute force method.

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	4.269046	3.776735
14	0.0	3.878948	3.118253
15	0.0	4.267241	2.471698
16	0.0	4.292409	3.612385
20	0.0	3.886179	2.612903

Table 5.8: Average number of moves for Naked Pairs for different hole sizes

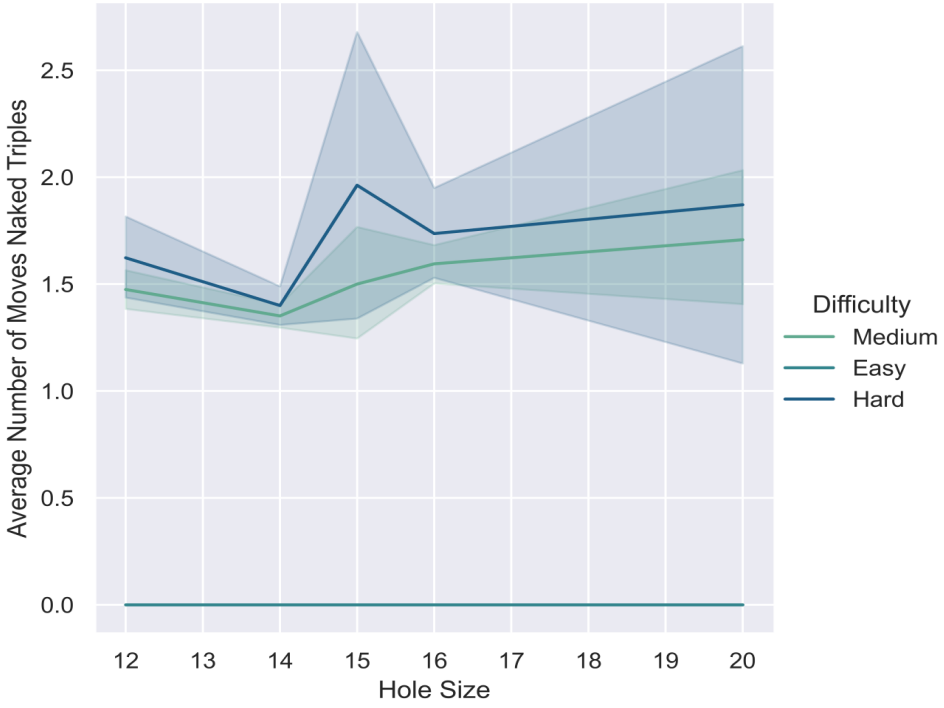

Figure 5.9: Average number of moves for Naked Triples for different hole sizes

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	1.474694	1.622889
14	0.0	1.351105	1.399636
15	0.0	1.500000	1.962264
16	0.0	1.594658	1.736239
20	0.0	1.707317	1.870968

Table 5.9: Average number of moves for Naked Triples for different hole sizes

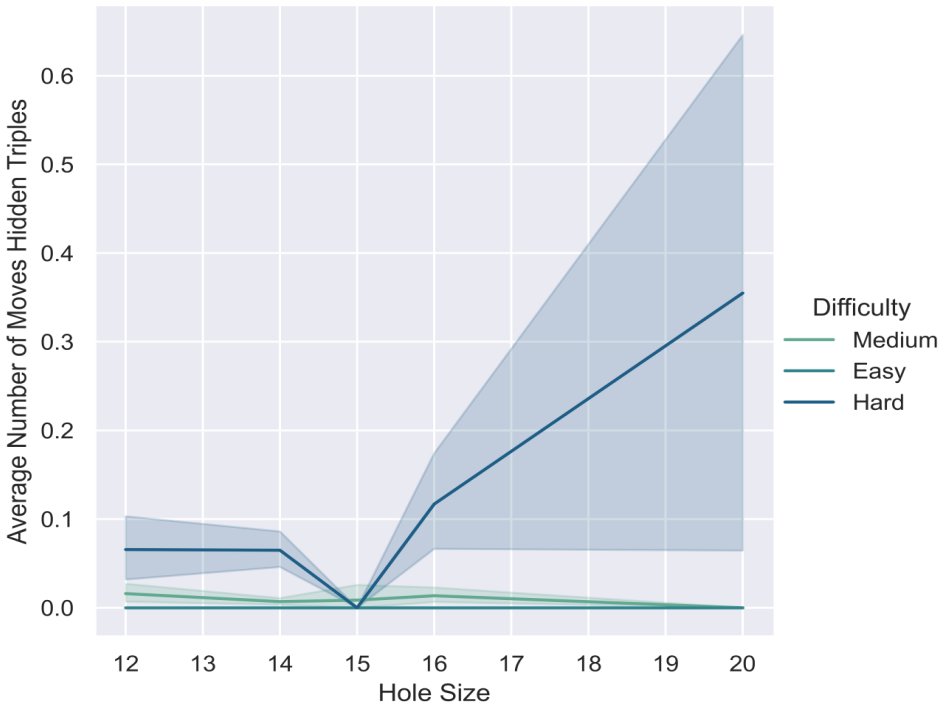

Figure 5.10: Average number of moves for Hidden Triples for different hole sizes

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	0.015983	0.065666
14	0.0	0.006938	0.064888
15	0.0	0.008621	0.0
16	0.0	0.013590	0.116972
20	0.0	0.0	0.354839

Table 5.10: Average number of moves for Hidden Triples for different hole sizes

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	0.0	0.574109
14	0.0	0.0	0.670710
15	0.0	0.0	0.584906
16	0.0	0.0	0.509174
20	0.0	0.0	0.0

Table 5.11: Average number of moves for Swordfish for different hole sizes

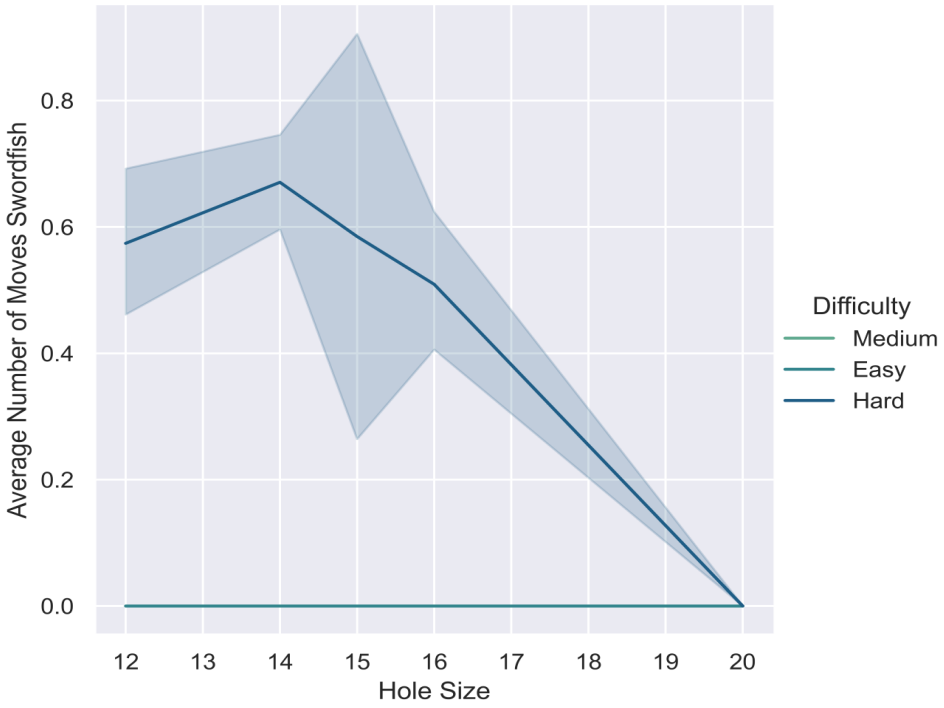

Figure 5.11: Average number of moves for Swordfish for different hole sizes

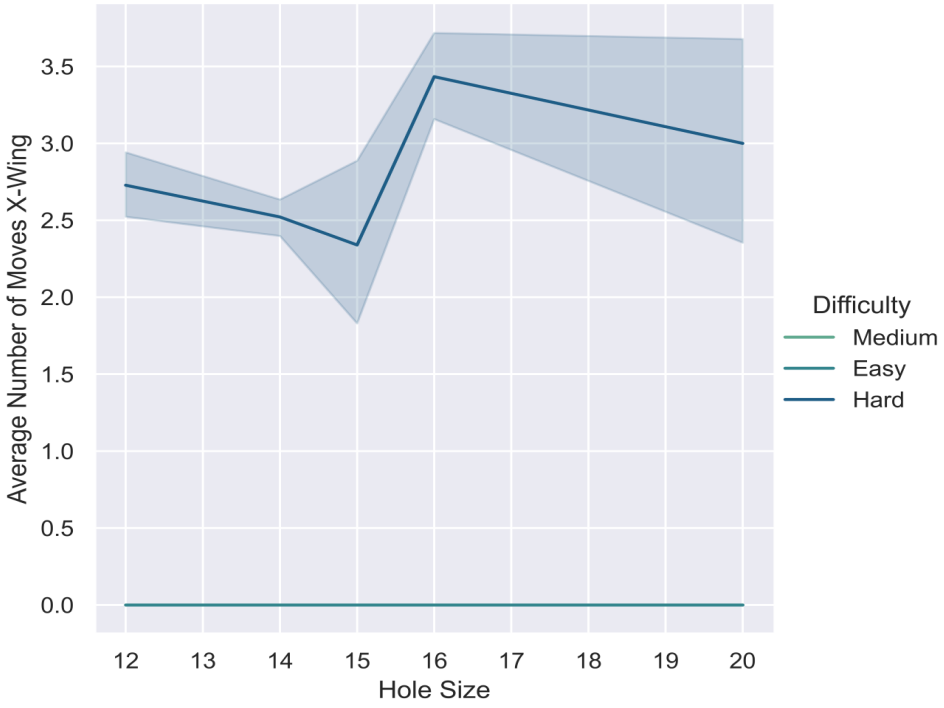

Figure 5.12: Average number of moves for X-Wing for different hole sizes

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	0.0	2.727955
14	0.0	0.0	2.521528
15	0.0	0.0	2.339623
16	0.0	0.0	3.433486
20	0.0	0.0	3.0

Table 5.12: Average number of moves for X-Wing for different hole sizes

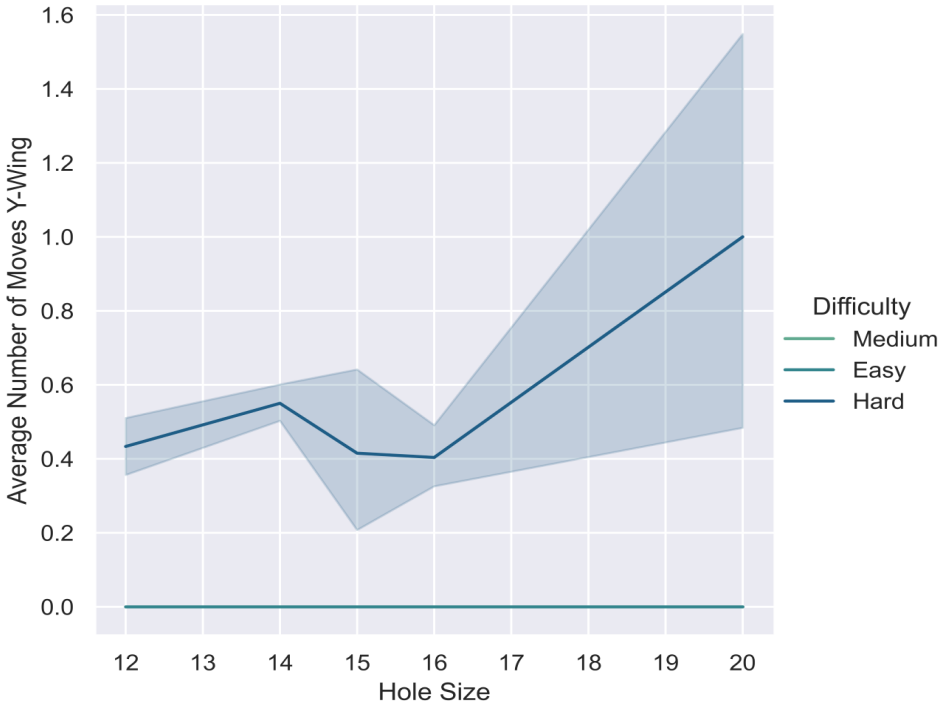

Figure 5.13: Average number of moves for Y-Wing for different hole sizes

Hole Size	Average Number of Moves for Easy Difficulty	Average Number of Moves for Medium Difficulty	Average Number of Moves for Hard Difficulty
12	0.0	0.0	0.433396
14	0.0	0.0	0.550030
15	0.0	0.0	0.415094
16	0.0	0.0	0.403670
20	0.0	0.0	1.0

Table 5.13: Average number of moves for Y-Wing for different hole sizes

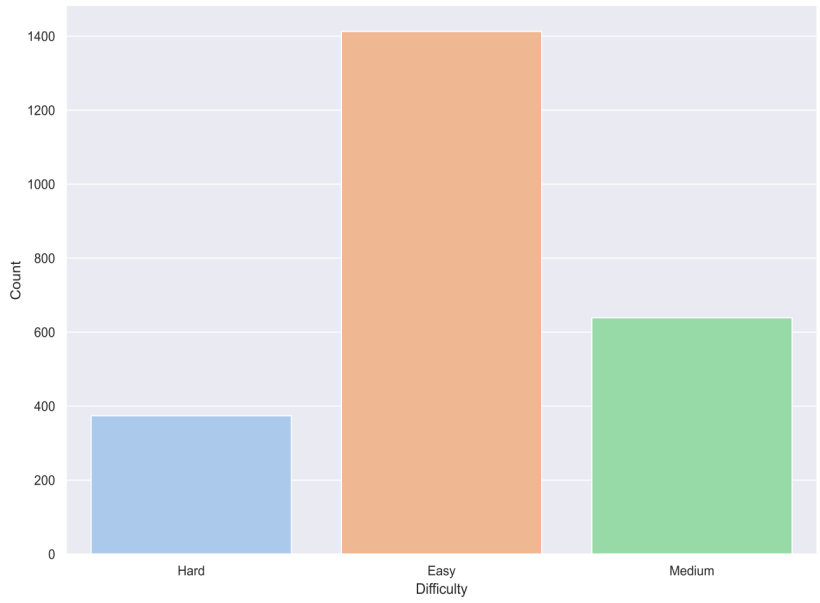

Figure 5.14: Distribution of Sudokus based on difficulty

Difficulty	Count
Easy	1413
Medium	374
Hard	639

Table 5.14: Distribution of Sudokus based on difficulty

Difficulty	Mean Time To Solve	Standard Deviation
Easy	7.984618	23.762563
Medium	8.029962	23.864847
Hard	6.976301	26.139004

Table 5.15: Mean time to solve Sudoku with Brute Force

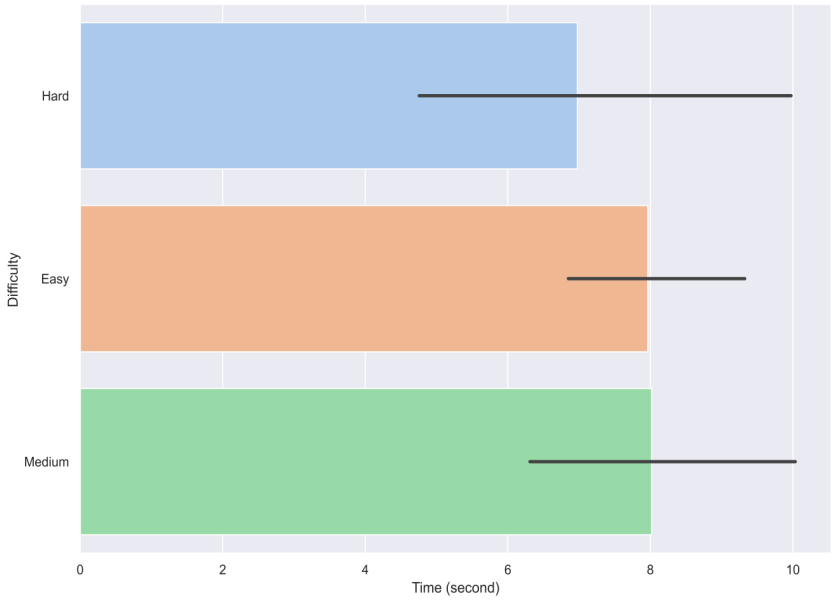

Figure 5.15: Mean time to solve Sudoku with Brute Force

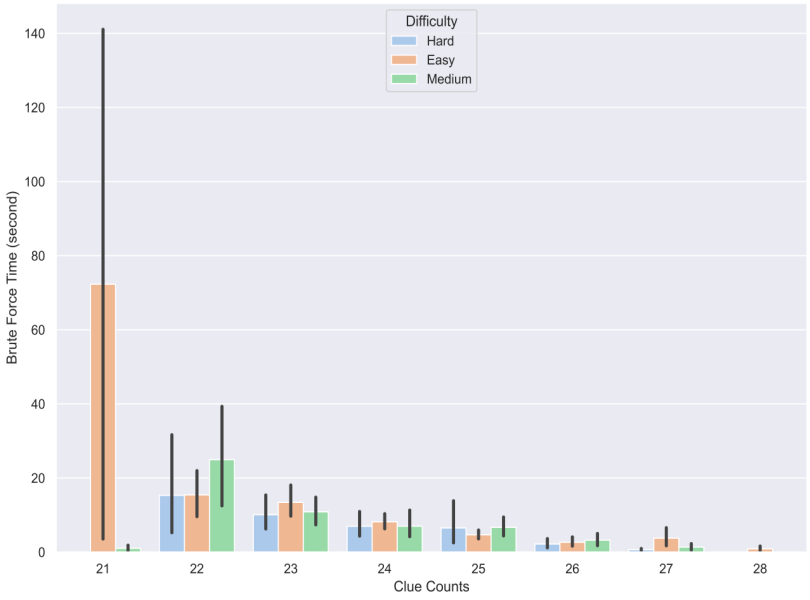

Figure 5.16: Mean time to solve Sudoku with Brute Force for different clue counts

Clue Count	Mean time to solve Easy problems	Mean time to solve Medium problems	Mean time to solve Hard problems
21	72.3223865	1.0362305	Not Available
22	15.4591126071	24.9657004828	15.332734625
23	13.4428463409	10.8659083053	10.0855169444
24	8.1920313897	7.01514656	6.9787636165
25	4.6718373316	6.6917695196	6.5374547107
26	2.6521873435	3.2218156316	2.1808564318
27	3.7878482105	1.3597250625	0.6716358333
28	0.8758395	Not Available	Not Available

Table 5.16: Mean time to solve Sudoku with Brute Force for different clue counts

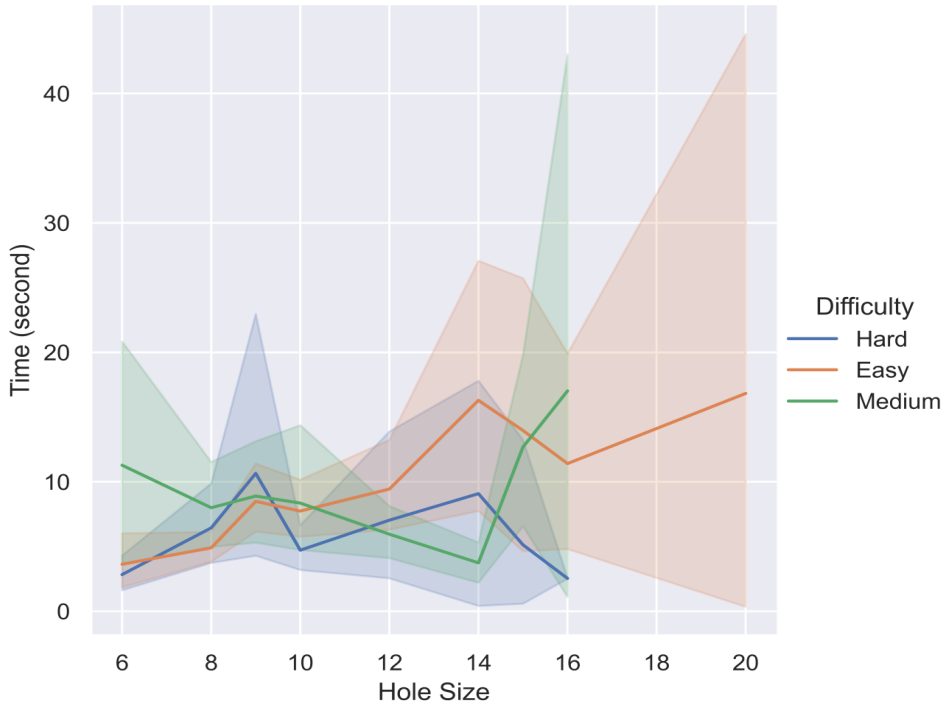

Figure 5.17: Mean time to solve Sudoku with Brute Force for different clue counts

Hole Size	Mean time to solve Easy problems	Mean time to solve Medium problems	Mean time to solve Hard problems
6	3.6324141875	11.281155125	2.83472
8	4.898075806	7.9999658031	6.4401986737
9	8.4886799727	8.9011969139	10.6509270227
10	7.7385810426	8.3567605771	4.7188743913
12	9.4394029505	5.9528811471	7.03582975
14	16.2939607714	3.741112875	9.085337
15	13.9660764839	12.6853857273	5.131629375
16	11.408764913	17.0212451429	2.5371835
20	16.8274506667	Not Available	Not Available

Table 5.17: Mean time to solve Sudoku with Brute Force for different clue counts

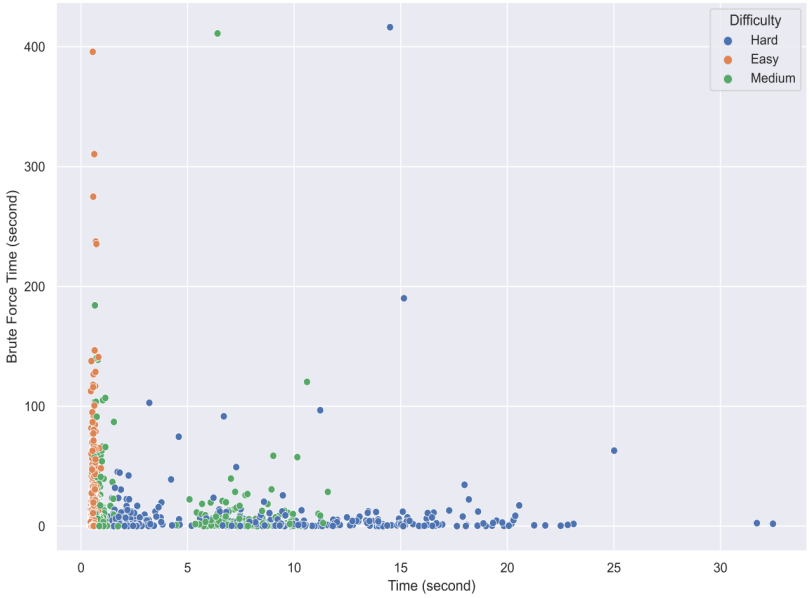

Figure 5.18: Comparison of time to solve for search based methods and logic based methods

5.2 Kaggle's 3 Million Dataset

We ran the same set of experiments on Kaggle's 3 million Sudoku dataset in which puzzles have variable number of clues. From the numerous analyeses we performed we found similar trend in the graphs as we had found for the 17 clues dataset. Graphs generated from these analyses are given below:

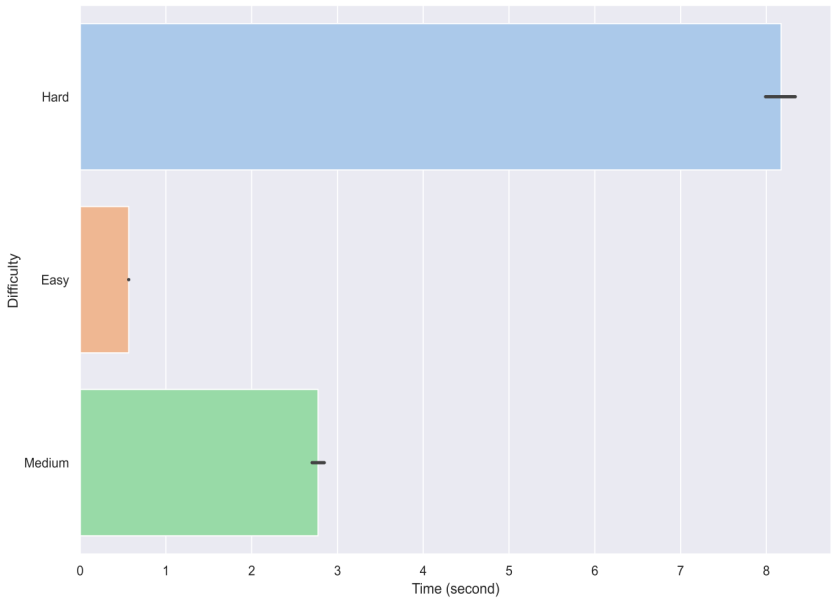

Figure 5.19: Mean time to solve a Sudoku for different difficulties

Figure 5.19 shows Time to solve Sudoku puzzles with respect to difficulties and figure 5.20 shows the correlation of solving time as clue count increases.
After that, we try to see the effects of hole sizes on difficulty of a puzzle. figure 5.21 shows distribution of easy, medium, hard Sudoku for different hole sizes whereas Figure 5.22 shows time to solve with respect to different hole sizes. We also show the downwards trend of these hole sizes with respect to clue counts in figure 5.23.

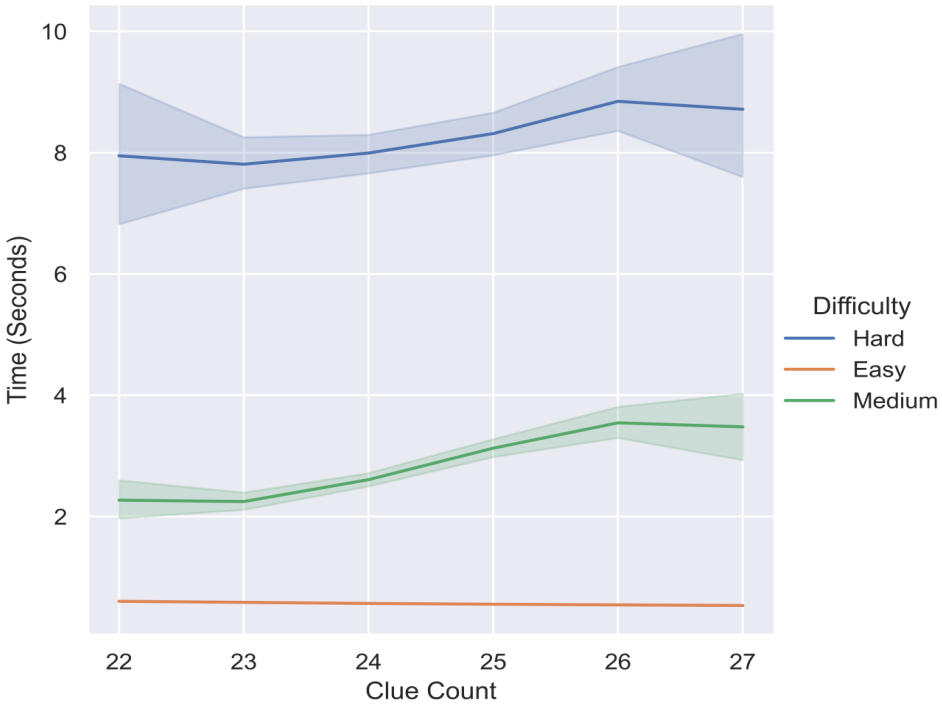

Figure 5.20: Mean time to solve a Sudoku for different clue counts

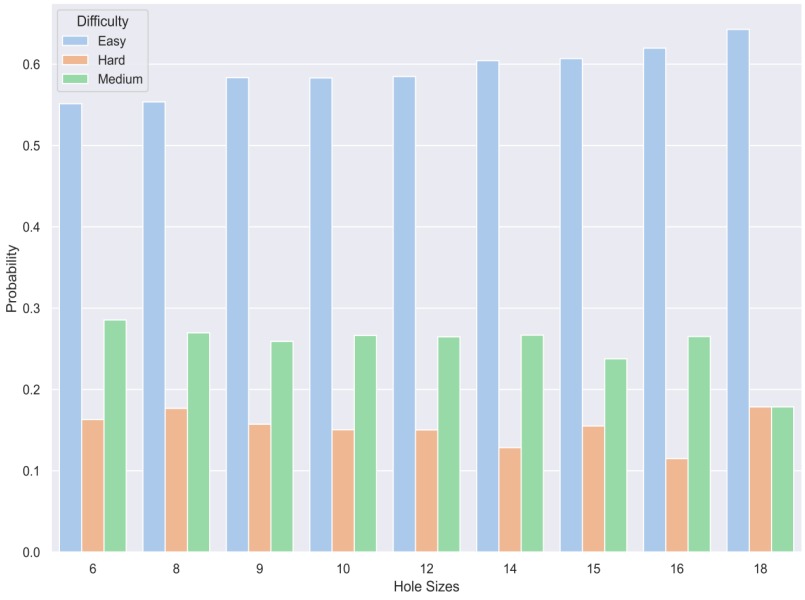

Figure 5.21: Distribution of difficulties among a hole sizes

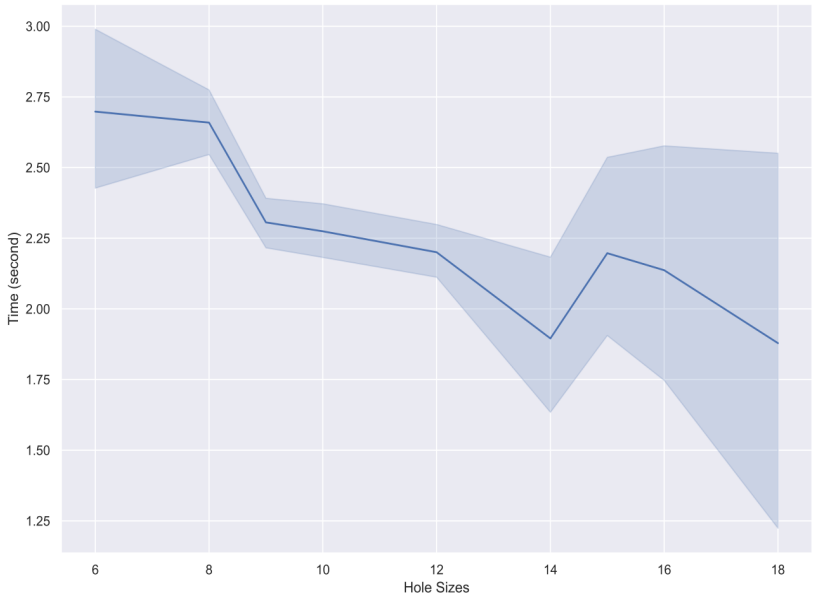

Figure 5.22: Average time to solve for different hole sizes

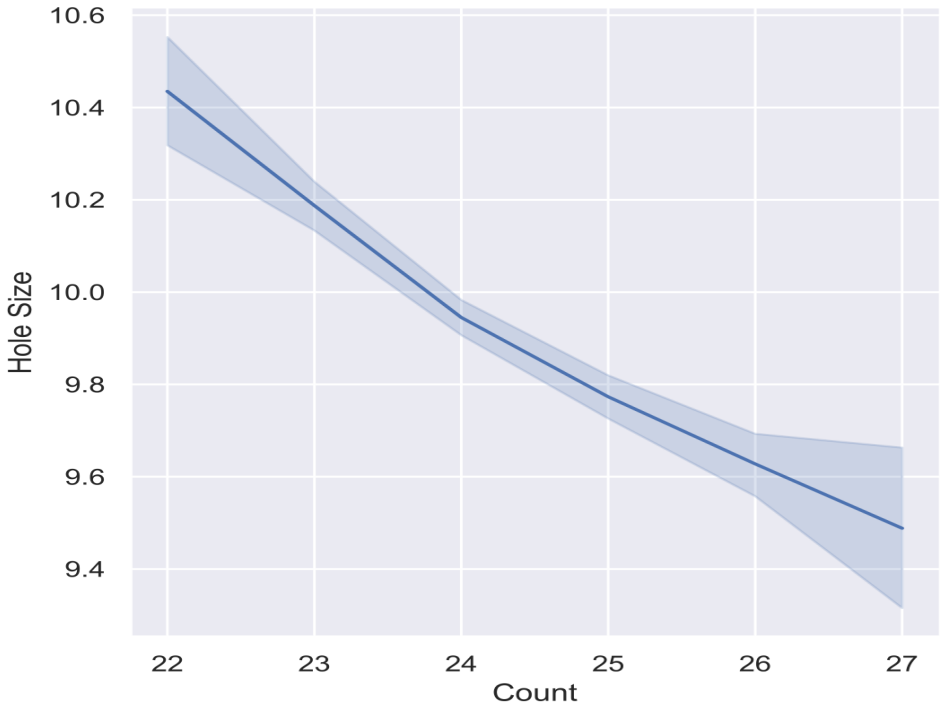

Figure 5.23: Correlation of different hole sizes wit respect to clue counts

Next is the analysis of count of moves of each method for different difficulties with respect to hole sizes. This way we can also analyze the utilization of different human solving techniques. This analysis gives us the same findings we obtained from the 17 clue dataset. The next set of figures from 5.24 are the graphs that represent this analysis.

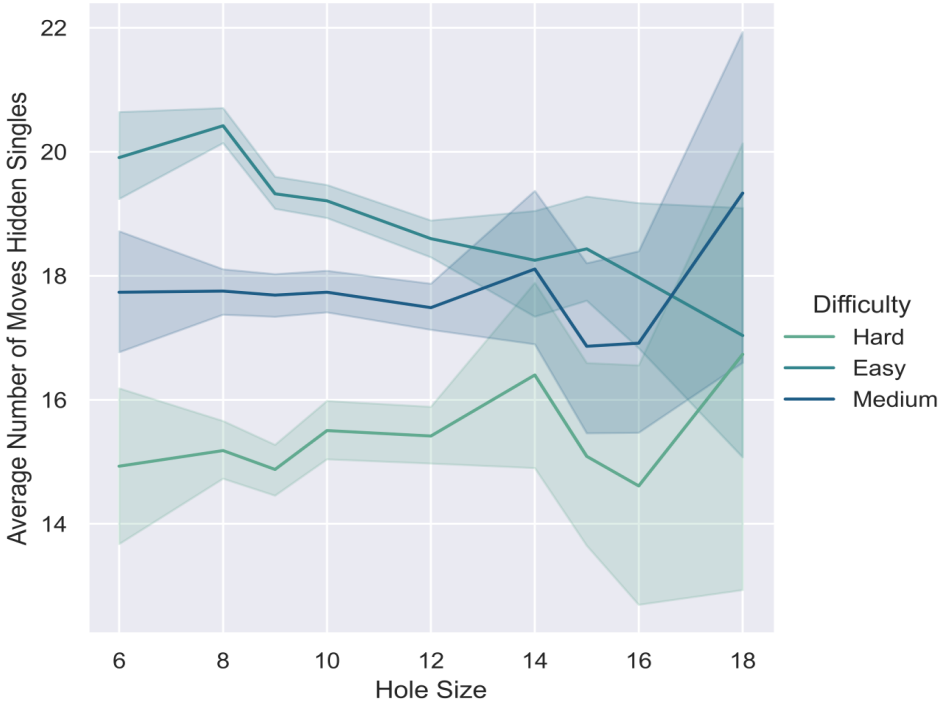

Figure 5.24: Average number of moves for Hidden Singles for different hole sizes

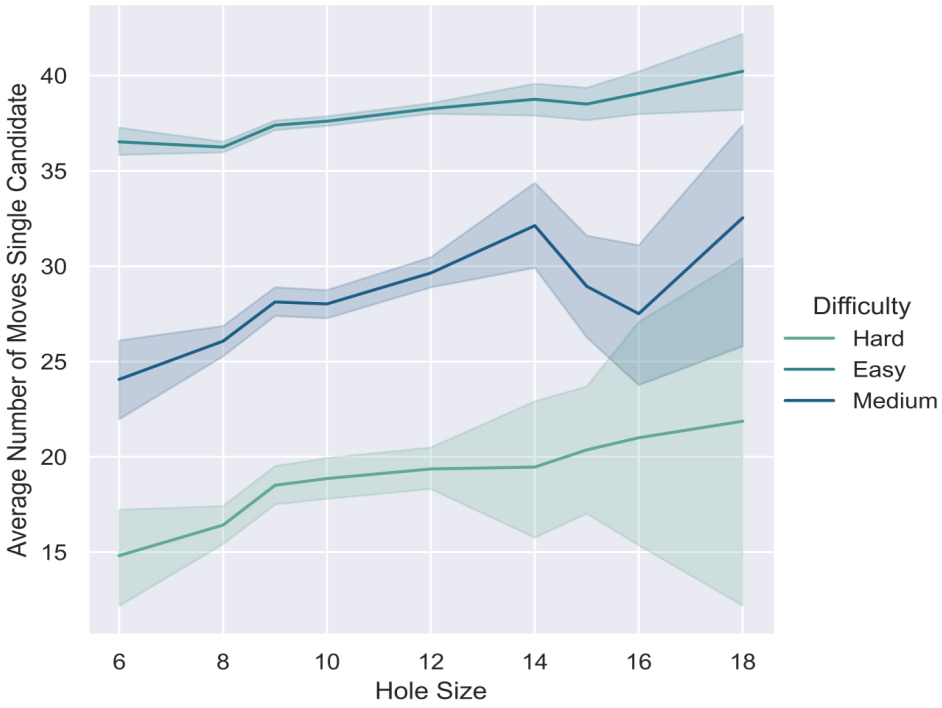

Figure 5.25: Average number of moves for Single Candidate for different hole sizes

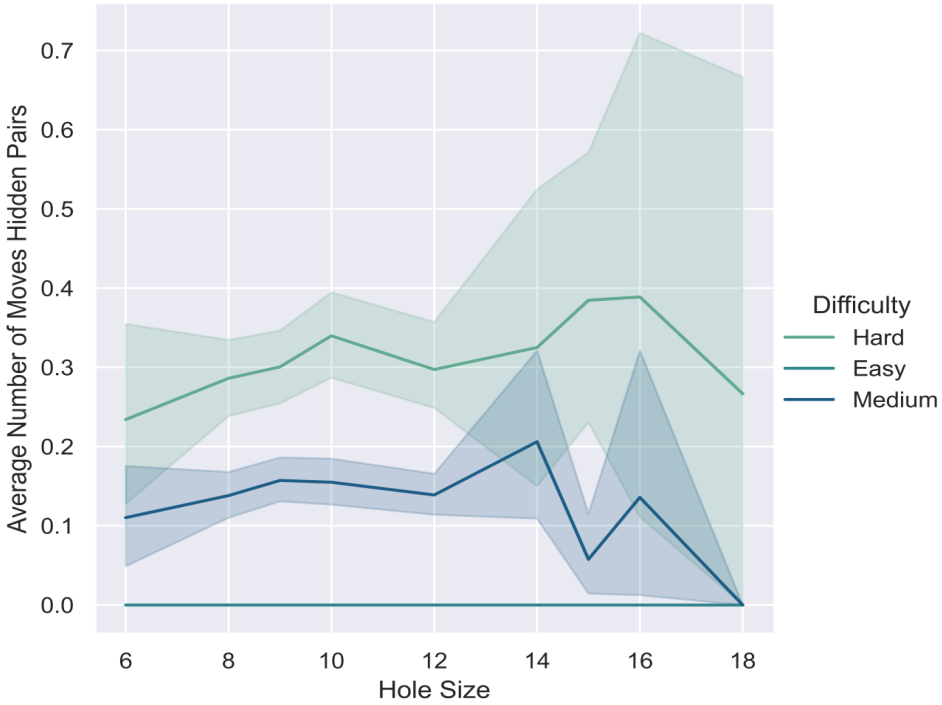

Figure 5.26: Average number of moves for Hidden Pair for different hole sizes

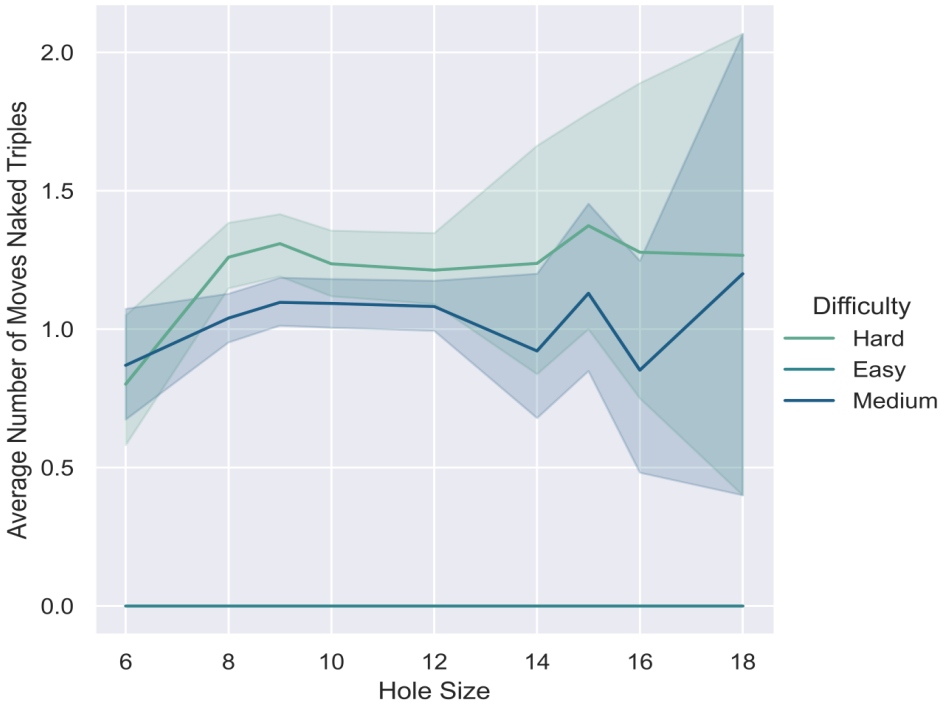

Figure 5.27: Average number of moves for Naked Triples for different hole sizes

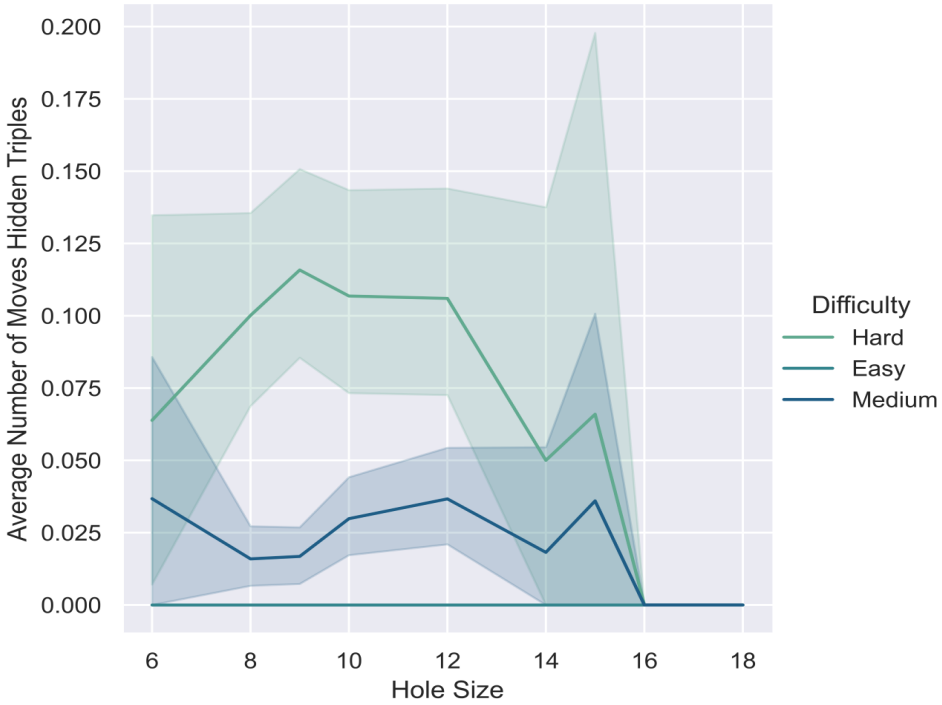

Figure 5.28: Average number of moves for Hidden Triples for different hole sizes

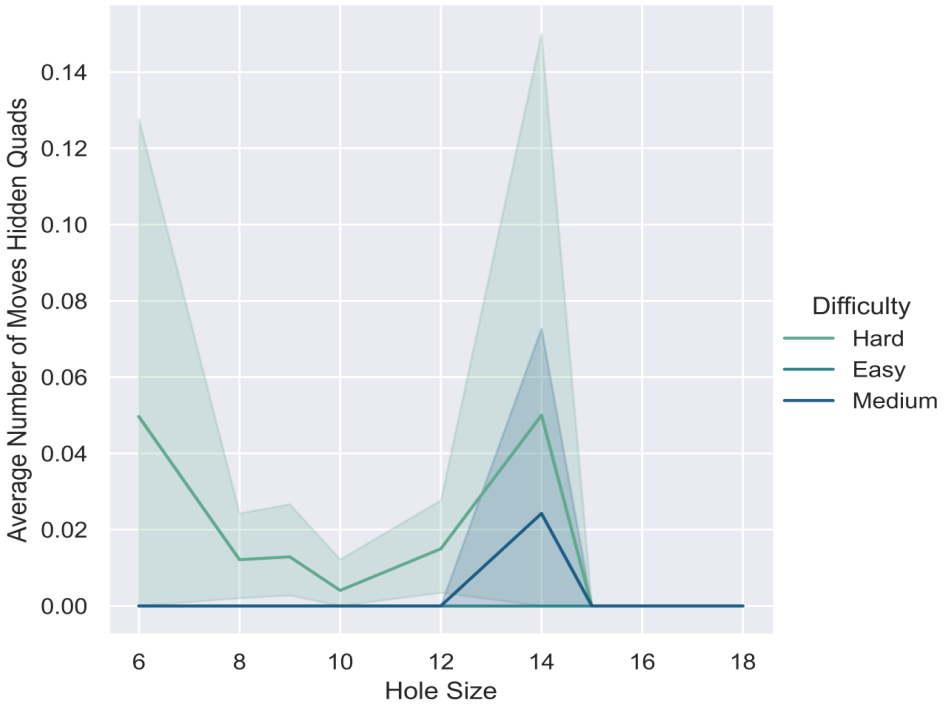

Figure 5.29: Average number of moves for Hidden Quads for different hole sizes

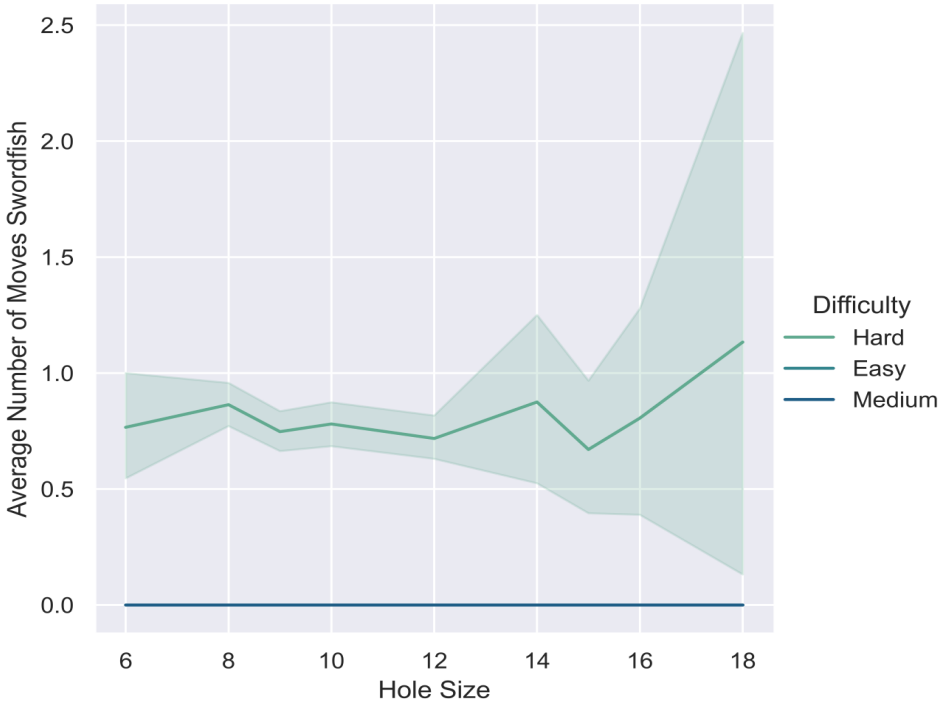

Figure 5.30: Average number of moves for Swordfish for different hole sizes

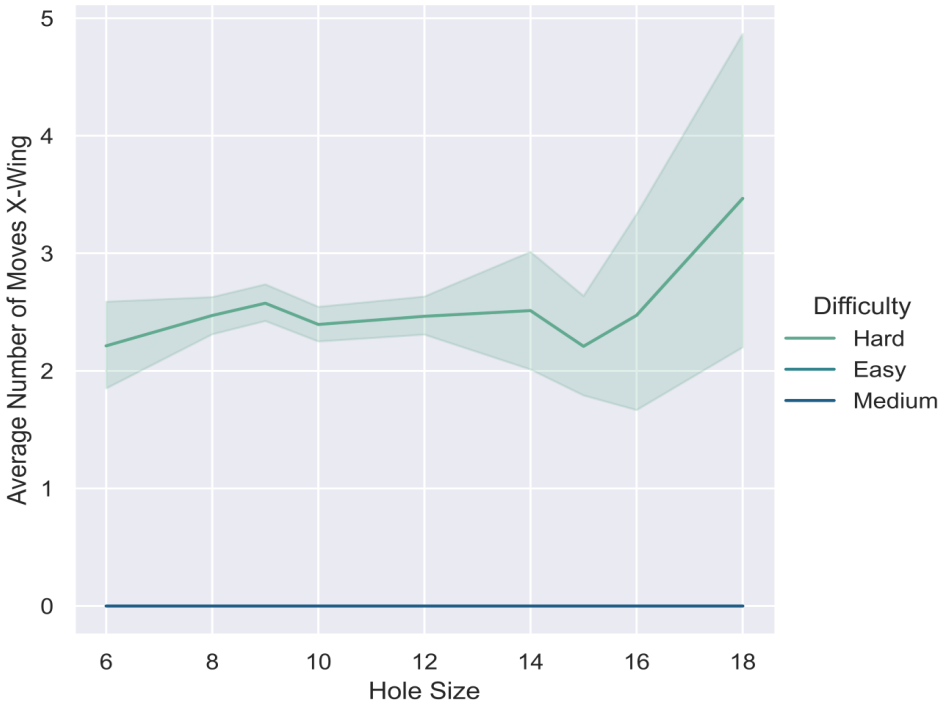

Figure 5.31: Average number of moves for X-Wing for different hole sizes

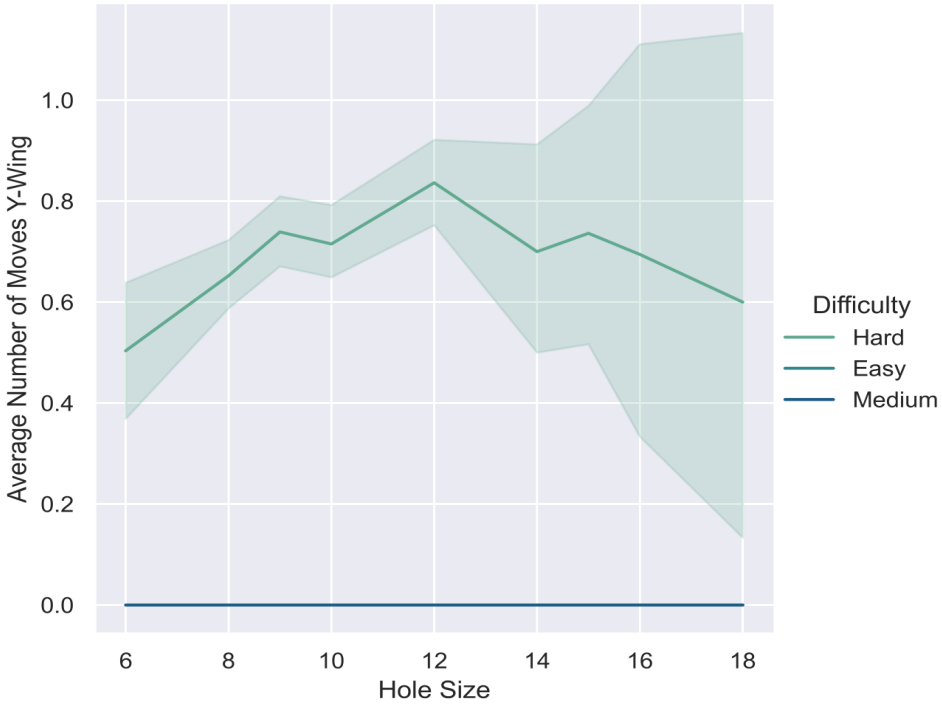

Figure 5.32: Average number of moves for Y-Wing for different hole sizes

Again, we do the same analysis of count of moves for different difficulties but with respect to clue counts. The next set of figures from 5.33 represent this analysis.

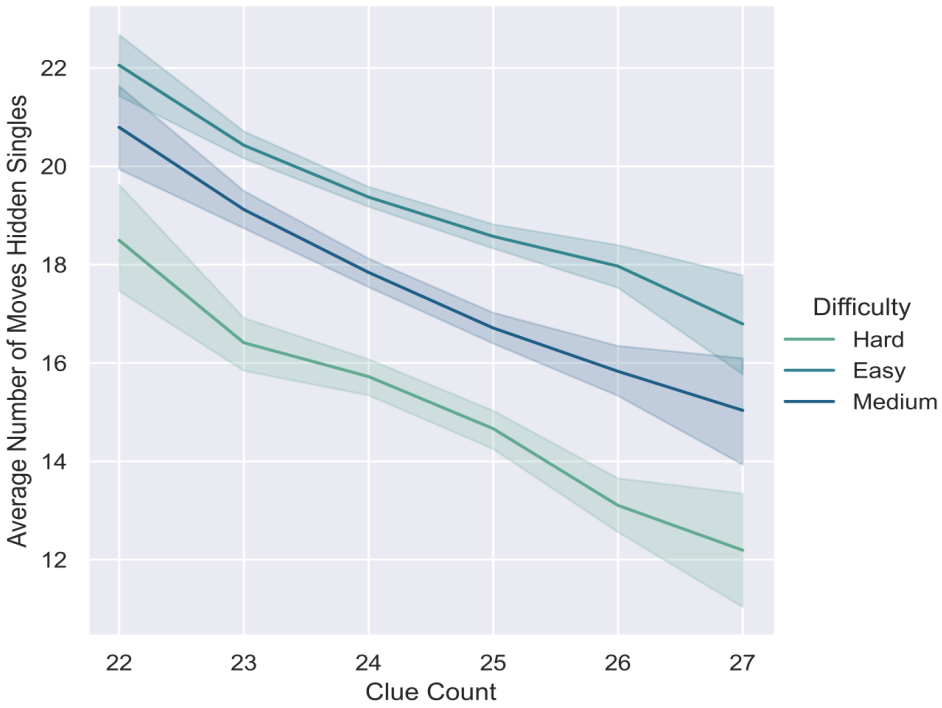

Figure 5.33: Average number of moves for Hidden Singles for different clue counts

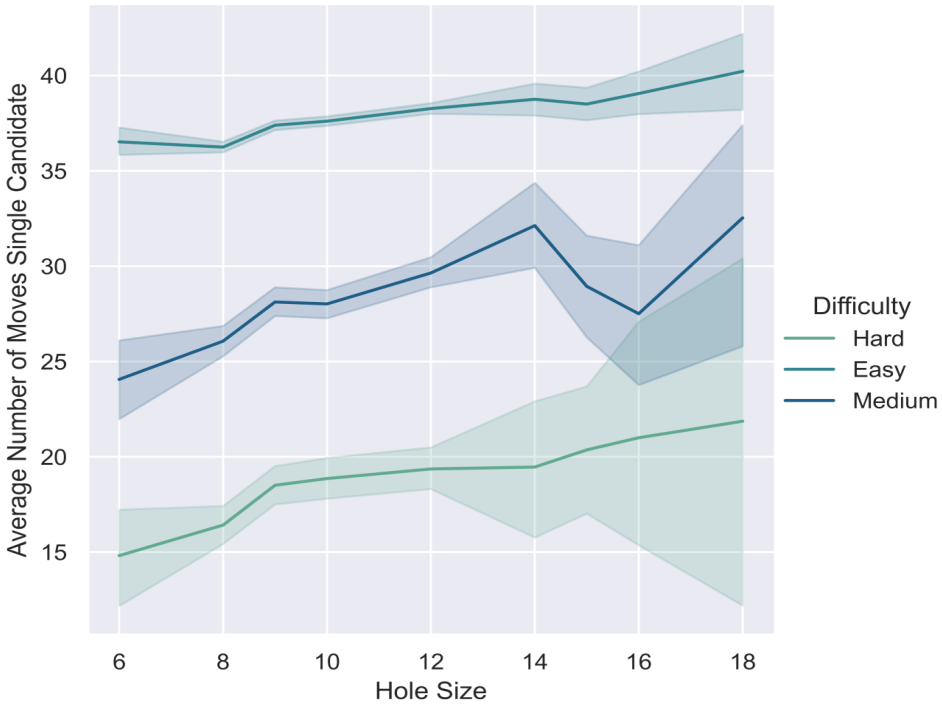

Figure 5.34: Average number of moves for Single Candidate for different clue counts

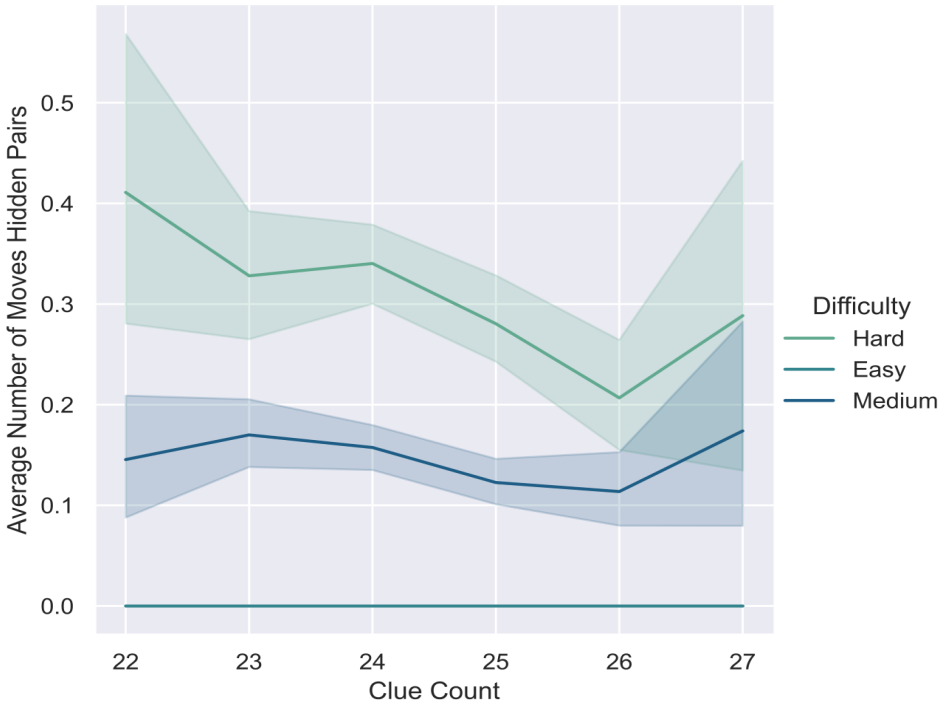

Figure 5.35: Average number of moves for Hidden Pair for different clue counts

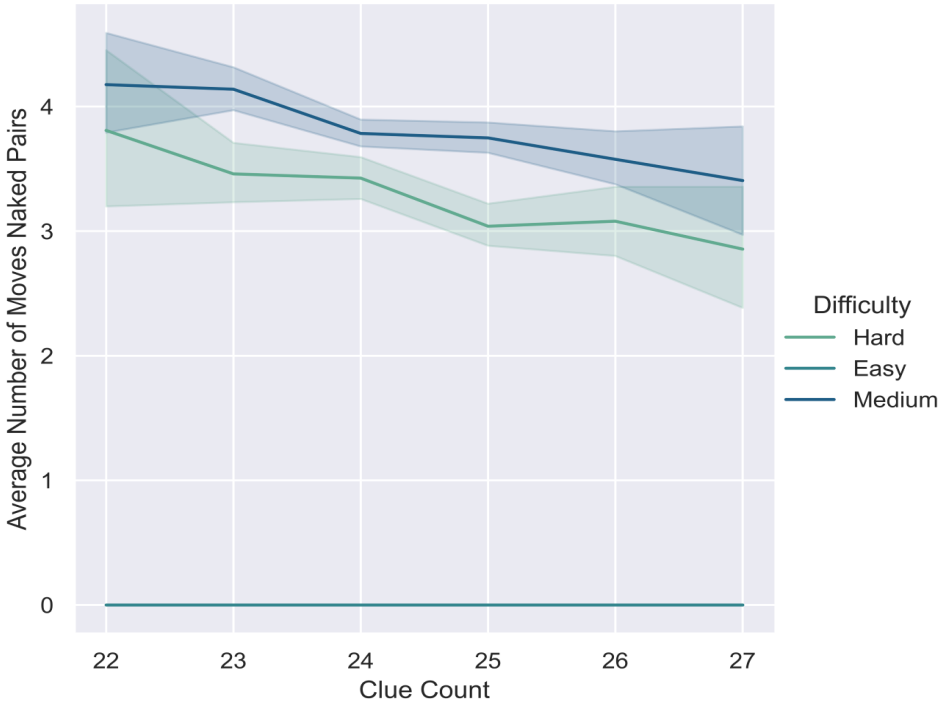

Figure 5.36: Average number of moves for Naked Pair for different clue counts

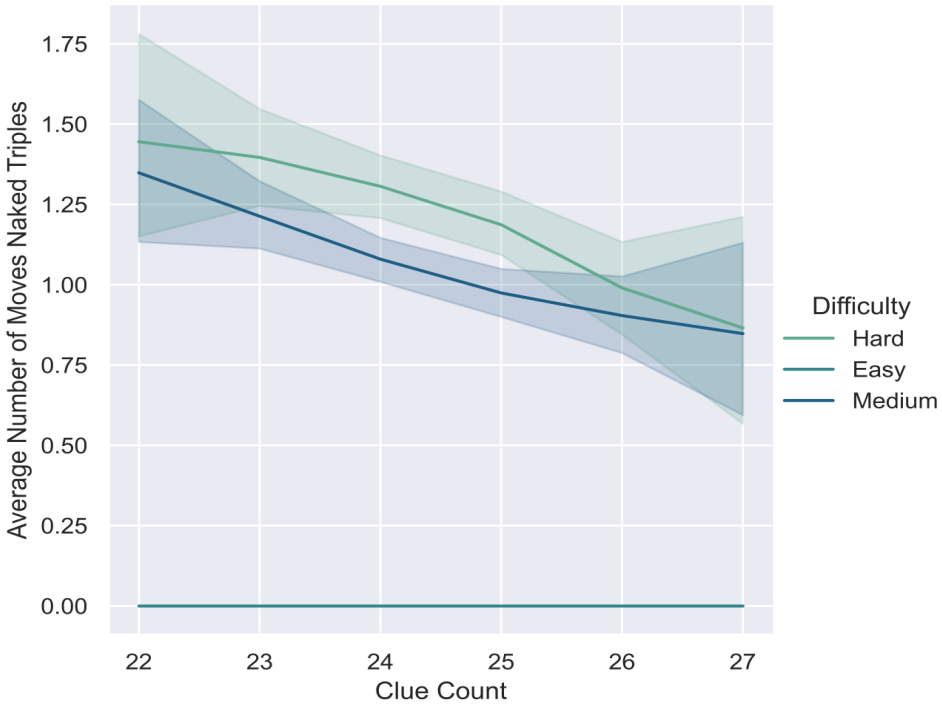

Figure 5.37: Average number of moves for Naked Triple for different clue counts

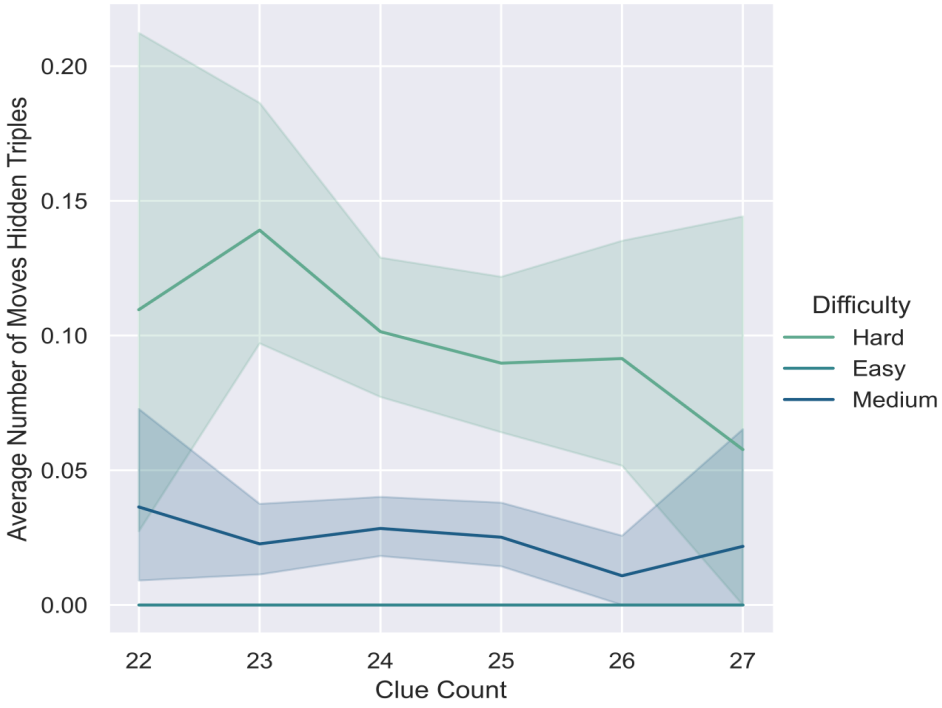

Figure 5.38: Average number of moves for Hidden Triples for different clue counts

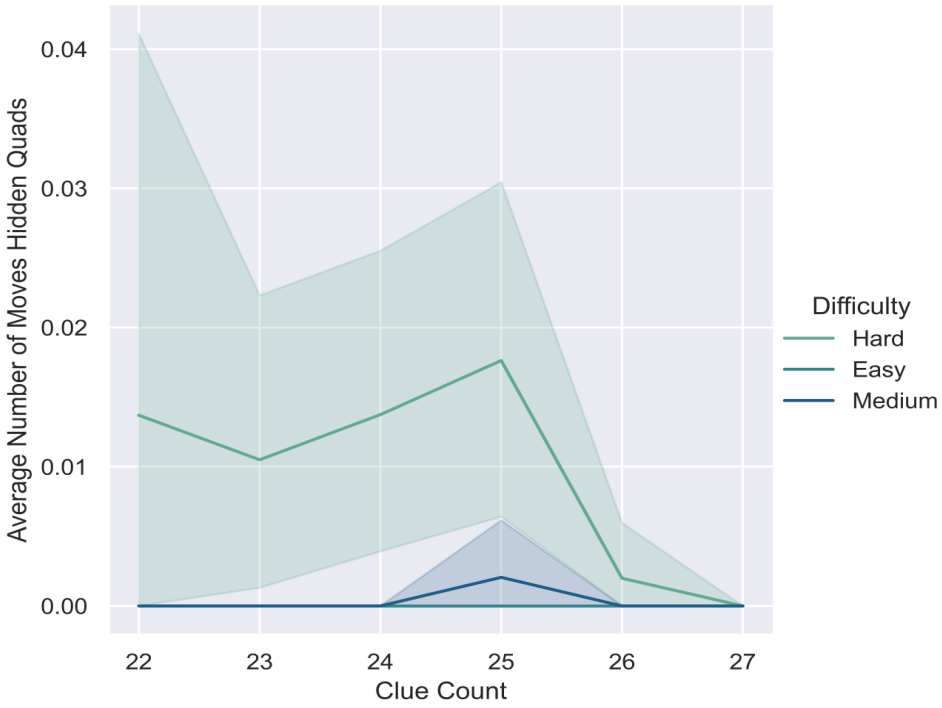

Figure 5.39: Average number of moves for hidden Quads for different clue counts

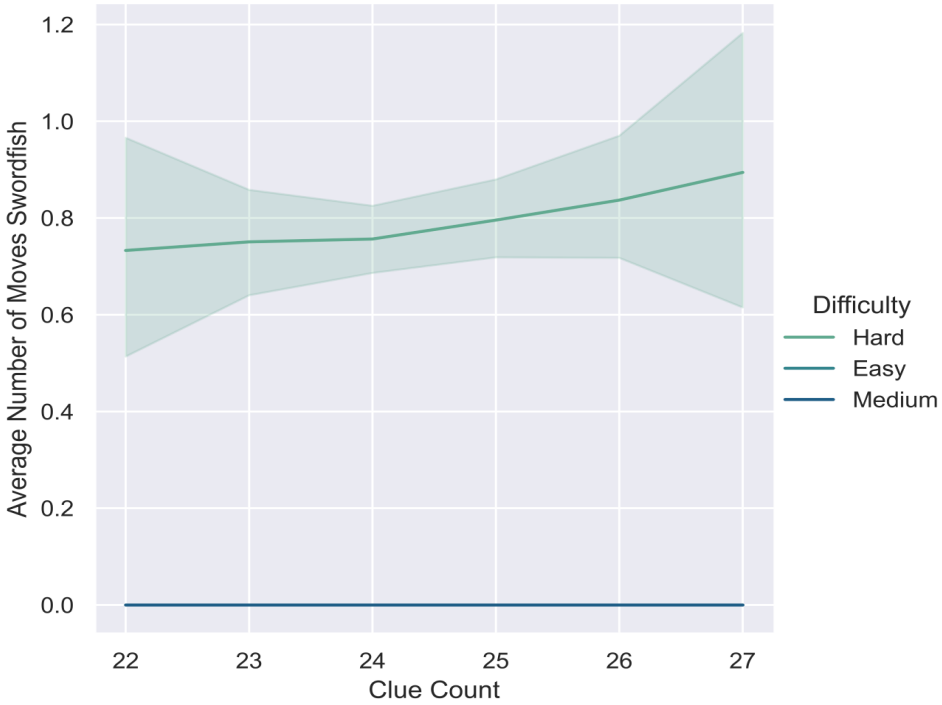

Figure 5.40: Average number of moves for Swordfish for different clue counts

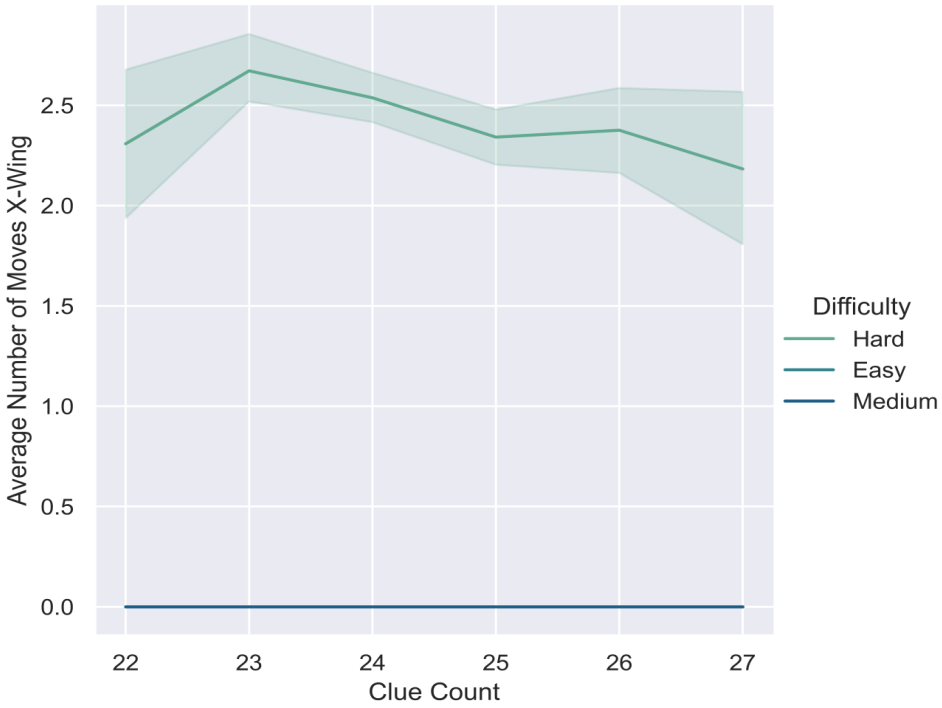

Figure 5.41: Average number of moves for X-wing for different clue counts

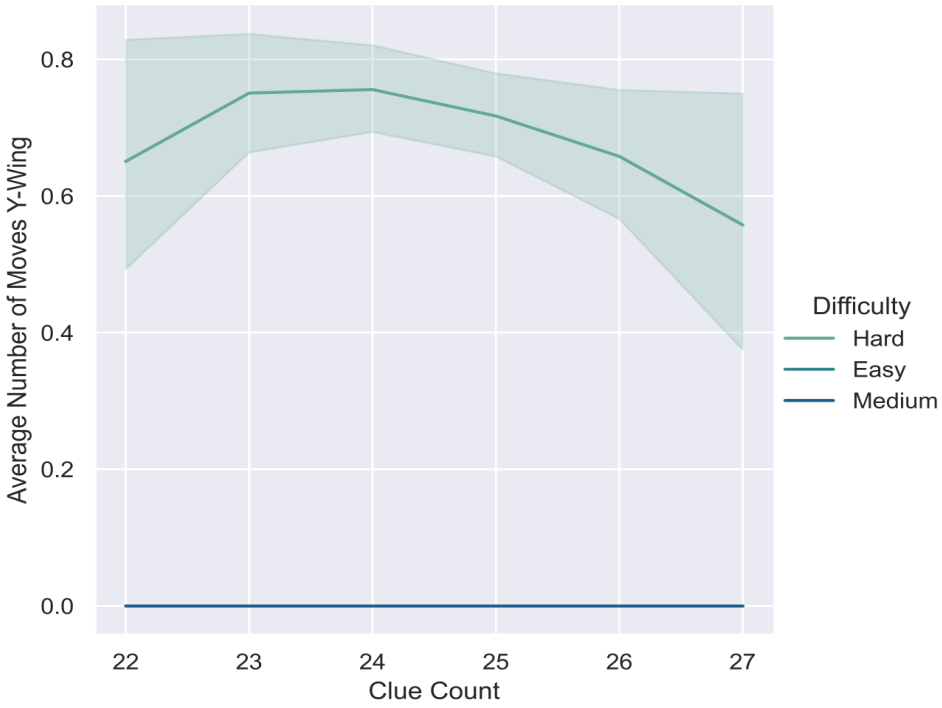

Figure 5.42: Average number of moves for Y-wing for different clue counts

We incorporated results of all our experiments in this chapter. The graphs shown in this chapter has some clear insights which will be summarize in next chapter. Finally, Data for the graphs presented are shared in Appendix C.

CHAPTER 6
CONCLUSION

6.1 Findings

In this thesis, we ran many experiments based on speed, symmetry, number of clues and size of empty regions. After interpreting these information we can summarize them into following conclusions:

- For a puzzle to be solvable it has to have a minimum of 17 clues. Puzzles with lesser clues than that have been proven to be unsolvable.
- The hardness of a puzzle does not depend on clue count. Since, results from both datasets do not show any downward or upward trend.
- From the time graphs of puzzles of different difficulties it is evident that harder puzzles take more time to solve.
- Our results show that all puzzles from both datasets are asymmetric. This makes Sudoku hardness with respect to symmetry inconclusive.
- There is no upward, downward or definite trend or skew in the graphs of puzzle difficulty with respect to vacant regions. This gives us the conclusion that hardness of puzzles do not depend on size of vacant regions.
- The graphs generated for search-based algorithm show us that solving time increases when clue count increases. Time graph with respect to hole sizes also gives us the same result that, solving time increases when hole size increases. Likewise, solving time decreases when either of these count or size of these features decreases.
- It is also evident from our time graphs that logical rules based algorithm requires much less time than search based algorithm as search based algorithm does not solve puzzles intelligently.

6.2 Limitations

There are a few a limitations to our work due to shortage of time. They are stated below:

- Our Logical rules based solver can not solve puzzles alone in most cases. Brute forces is required to take over. That is because there are many more human solving techniques other than the ones we have implemented. So, our logical rules based solver is not complete without these techniques.
- Although We divided puzzles into three categories of difficulty: easy, medium and hard, we could not generate rating for puzzles in numerical values.

6.3 Future Work

Following are some features we can add in the future:

- We can add more rules to our logical rules based solver. This will make our logical rules based solver complete. Brute force will not need to take over.
- Generate difficulty rating in integer values for puzzles. Difficulty rating in integer values will give a clearer idea of the hardness of a puzzle.
- We can integrate our solver and difficulty rating scale in an online platform and make our work useful for Sudoku enthusiasts.

REFERENCES

[1] Wikipedia contributors, "Artificial intelligence — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=1011001027, 2021. [Online; accessed 11-March-2021].

[2] Wikipedia contributors, "Constraint satisfaction problem — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Constraint_satisfaction_problem&oldid=1010685221, 2021. [Online; accessed 11-March-2021].

[3] enjoysudoku.com, "The new sudoku players' forum." http://forum.enjoysudoku.com/is-www-setbb-com-alive-t30183.html.

[4] M. van Kreveld, M. Löffler, and P. Mutser, "Automated puzzle difficulty estimation," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 415–422, 2015.

[5] R. Pelánek *et al.*, "Difficulty rating of sudoku puzzles by a computational model.," in *FLAIRS Conference*, Citeseer, 2011.

[6] F. J. B. Felgenhauer, "Enumerating possible sudoku grids." http://www.afjarvis.staff.shef.ac.uk/sudoku/, 2005.

[7] T. Yato and T. Seta, "Complexity and completeness of finding another solution and its application to puzzles," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 86, pp. 1052–1060, 2003.

[8] I. Lynce and J. Ouaknine, "Sudoku as a sat problem," in *PROCEEDINGS OF THE 9 TH INTERNATIONAL SYMPOSIUM ON ARTIFICIAL INTELLIGENCE AND MATHEMATICS, AIMATH 2006, FORT LAUDERDALE*, Springer, 2006.

[9] H. Simonis, "Sudoku as a constraint problem," Citeseer.

[10] N. Y. L. Lee, G. P. Goodwin, and P. N. Johnson-Laird, "The psychological puzzle of sudoku," *Thinking & Reasoning*, vol. 14, no. 4, pp. 342–364, 2008.

[11] M. Henz and H.-M. Truong, *SudokuSat—A Tool for Analyzing Difficult Sudoku Puzzles*, pp. 25–35. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[12] W. Hao, W. Yu-Wen, and S. Chuen-Tsai, "Rating logic puzzle difficulty automatically in a human perspective," in *DiGRA Nordic &#3912: Proceedings of 2012 International DiGRA Nordic Conference*, 2012.

[13] G. Royle, "A collection of 49,151 distinct sudoku configurations with 17 entries." http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php.

[14] Kaggle.com, "3 million sudoku puzzles with ratings." https://www.kaggle.com/radcliffe/3-million-sudoku-puzzles-with-ratings, 2020. [Online; Accessed: 12-March-2021].

[15] N. Jussien, *A to Z of Sudoku*. ISTE, 2007.

[16] J. Rosenhouse and L. Taalman, *Taking sudoku seriously: The math behind the world's most popular pencil puzzle*. OUP USA, 2011.

[17] G. McGuire, B. Tugemann, and G. Civario, "There is no 16-clue sudoku: solving the sudoku minimum number of clues problem via hitting set enumeration," *Experimental Mathematics*, vol. 23, no. 2, pp. 190–217, 2014.

[18] M. Ercsey-Ravasz and Z. Toroczkai, "The chaos within sudoku," *Scientific reports*, vol. 2, no. 1, pp. 1–8, 2012.

[19] G. Li, X. Qi, N. Wang, R. Huang, and J. Zhao, "Genarate and solve sudoku puzzle," in *2019 10th International Conference on Information Technology in Medicine and Education (ITME)*, pp. 671–673, IEEE, 2019.

[20] T. Mantere and J. Koljonen, "Solving, rating and generating sudoku puzzles with ga," pp. 1382 – 1389, 10 2007.

[21] Wikipedia contributors, "Sudoku solving algorithms — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Sudoku_solving_algorithms&oldid=1003749776, 2021. [Online; accessed 11-March-2021].

[22] Learn-Sudoku.com, "Basic techniques." https://www.learn-sudoku.com/basic-techniques.html. [Online; accessed 12-March-2021].

[23] Sudoku.org.uk, "Pointing pairs - intersection removal." http://www.sudoku.org.uk/SolvingTechniques/IntersectionRemoval.asp. [Online; accessed 12-March-2021].

[24] thonky.com, "Simple coloring (singles chains) strategy." https://www.thonky.com/sudoku/simple-coloring. [Online; accessed 12-March-2021].

[25] gamesudoku.blogspot.com, "Advanced-single chains." http://gamesudoku.blogspot.com/2007/02/advanced-single-chains.html. [Online; accessed 12-March-2021].

[26] Learn-Sudoku.com, "Advanced techniques." https://www.learn-sudoku.com/advanced-techniques.html. [Online; accessed 12-March-2021].

[27] sudokuessentials.com, "How to identify sudoku x-wing." https://www.sudokuessentials.com/x-wing.html. [Online; accessed 12-March-2021].

[28] users.telenet.be, "How to solve a sudoku?." http://users.telenet.be/vandenberghe.jef/sudoku/index.html?how2solve. [Online; accessed 12-March-2021].

[29] sudokusnake.com, "Xyz wings." http://www.sudokusnake.com/xyzwings.php. [Online; accessed 12-March-2021].

[30] pi.math.cornell.edu, "The math behind sudoku: Solution symmetries." http://pi.math.cornell.edu/~mec/Summer2009/Mahmood/Symmetry.html. Accessed: 2021-03-11.

[31] sudokudragon.com, "Sudoku puzzle patterns." https://www.sudokudragon.com/symmetry.htm. Accessed: 2021-03-11.

APPENDIX A
SUDOKU PUZZLES

A.1 Easy puzzles

	1	2	3	4	5	6	7	8	9
A	6	1	3	9	8	7	4	5	
B	4				5				
C									3
D		7	8	3		4	9		1
E				2			5		
F	1		2				7		
G						9	6		
H	8	6			1	2		4	
I		9				8	2	7	5

Figure A.1: Easy puzzle 1

	1	2	3	4	5	6	7	8	9
A	6	1	4		7			2	
B			8						
C	9		5			8			
D				8		7		3	
E			3	9	5	2	7	1	
F		9			6				5
G		4					3		
H			6	4	8		2		
I				3	2		1	4	

Figure A.2: Easy puzzle 2

	1	2	3	4	5	6	7	8	9
A			3	4		2	7	6	
B	7	1	6	8		9	2	3	4
C	4	5							
D				9					
E								8	1
F	2			6				7	
G	1	9	8	5					
H		3		2	7		5	1	
I								4	

Figure A.3: Easy puzzle 3

A.2 Medium Puzzles

	1	2	3	4	5	6	7	8	9
A			4	3		7			6
B		6		4			8		
C				1					
D			5					4	
E	4		7		3		2	6	
F						9			
G									5
H		5	8				1		
I			2		7		4		8

Figure A.4: Medium puzzle 1

	1	2	3	4	5	6	7	8	9
A	6					9			
B				3	2			6	
C	3	5	1		8	4			
D	1	6					3		
E			9				8		
F			3					2	7
G				4	9		1	7	3
H		3			1	5			
I				2					4

Figure A.5: Medium puzzle 2

	1	2	3	4	5	6	7	8	9
A	9					6			
B	8	4	5	7					
C	1			9	5			7	
D				2			1		7
E					6				
F	3		7			1			
G		6			2	4			1
H						9	8	3	2
I				3					6

Figure A.6: Medium puzzle 3

A.3 Hard Puzzles

	1	2	3	4	5	6	7	8	9
A	6	9		3		1			
B			4	5					
C					7	9	2		4
D	3		5		8				
E								8	5
F	7		1				3		
G									
H							7	1	
I		4				2			9

Figure A.7: Hard puzzle 1

	1	2	3	4	5	6	7	8	9
A	9				5	1	7	3	
B	1		7	3	9	8	2		5
C	5				7	6		9	1
D	8	1		7	2	4	3	5	
E	2			1	6	5			7
F		7	5	9	8	3		1	2
G		2	1	5	3	7			
H	7	5	8	6	4	9	1	2	3
I	3	9		8	1	2	5	7	

Figure A.8: Hard puzzle 2

	1	2	3	4	5	6	7	8	9
A	1	9	5	3	6	7	2	4	8
B		7	8		5		3	6	9
C	3		6		9	8	1	5	7
D			3	7	8		5	9	
E	7		9			5			6
F	5	8	4	9		6	7	1	
G	8	3	2	5	4	9	6	7	1
H	9		7		1	3		2	5
I		5	1		7	2	9		

Figure A.9: Hard puzzle 3

APPENDIX B
CODES

This section of the book consists of all the codes used for experimentation. Here is how to run our code.

- open `sudoku_solver.py`
- enter input location (an example is given in the code and all the inputs can be found in the input folder)
- run `sudoku_solver.py`
- check count.txt to see how many times each method was called
- check log.txt to check step by step solution

B.1 Human Like Solver

This solver uses the human like solving algorthms to solve Sudoku puzzles. All the methods used this solver is given below.

B.1.1 `sudoku_solver.py`

This is the main function of our solver which takes input and calls the solver function.

```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
import time
import sys
from print_board import print_board
from validation_check import check_valid
from candidate_handler import candidates, candidates_update
from cells_seen import cells_seen
from solver import solver
from init_board import init_board

```

```
#file_name = input("Please enter the file location: ")
grid = ""
#enter input location
main = open("Inputs\\Symmetrical Inputs.txt")
i = 1
c = 0
for single in main:

    if str(i)!=sys.argv[1]:
        i+=1
        continue

    symtest = np.zeros((9,9))

    n = open("Inputs\\input.txt", "w")
    for ind in range(9):
        for j in range(9):
            if(single[ind*9+j] != '.'):
                n.write(single[ind*9+j])
                symtest[ind][j] = 1
                c+=1
                n.write('|')
            else:
                n.write('0')
                n.write('|')
    n.close()

    f = open("Inputs\\input.txt", "r")
    for line in f:
        for num in line.split('|')[:-1]:
            if(int(num) < 10):
                grid+=str(num)

    t1 = time.time()
    board,square_pos = init_board(grid)
    print_board(board)

    f = open('Output/count.txt'.format(i), 'w')
    f.write(f"The Count is {c}\n")
    f.write(f"{t1}\n")

    print(symtest)
    print(np.rot90(symtest))

    if np.array_equal(symtest, np.rot90(symtest)):
        f.write(f"The Input is 90 Degree Symmetric.\n")

    if np.array_equal(symtest, np.rot90(np.rot90(symtest))):
```

```
62          f.write(f"The Input is 180 Degree Symmetric.\n")
63
64      f.close()
65      f = open('Output/log.txt'.format(i), 'w')
66      f.write('Log Start\n')
67      f.close()
68      #check validity before solving it
69      if check_valid(board):
70          cands = candidates(board)
71          print(cands)
72          solver(board,cands,square_pos)
73      else:
74          print("The board is not valid!")
75
76      break
```

B.1.2 solver.py

This is the function which solves the puzzle by calling all the algorithms and classifies difficulty of the puzzle.

```
1  from single_candidate import single_cand
2  from hidden_singles import hidden_singles
3  from hidden_pairs_triples_quads import hidden_pairs_triples, hidden_quads
4  from naked_pairs_triples_quads import naked_pairs_triples, naked_quads
5  from pointing_pairs import pointing_pairs
6  from box_line import box_line
7  from x_wing import x_wing
8  from y_wing import y_wing
9  from singles_chains import singles_chains
10 from xyz_wing import xyz_wing
11 from swordfish import swordfish
12 from print_board import print_board
13 from validation_check import check_valid
14 from brute_force import Brute_Force
15 import sys
16 import numpy as np
17 import time as time
18 #%% SOLVER FUNCTION
19 def solver(board,cands,square_pos):
20     #run strategies in listed order as long as the board has empty cells
       (".")
21     if (board==".").any().any():
22         single_cand(board,cands,square_pos)
23         hidden_singles(board,cands,square_pos)
24         naked_pairs_triples(board,cands,square_pos)
25         hidden_pairs_triples(board,cands,square_pos)
```

```
        pointing_pairs(board,cands,square_pos)
        box_line(board,cands,square_pos)
        naked_quads(board,cands,square_pos)
        hidden_quads(board,cands,square_pos)
        x_wing(board,cands,square_pos)
        y_wing(board,cands,square_pos)
        singles_chains(board,cands,square_pos)
        xyz_wing(board,cands,square_pos)
        swordfish(board,cands,square_pos)
        s = Brute_Force()
        s.load_puzzle_from_file(board)
        get_time = s.solve()
        for i in range(9):
            for j in range(9):
                board.iloc[i,j] = s.puzzle[i*9+j]
    else:
        print("COMPLETE!!!!!\n")
        if check_valid(board):
            board.columns = np.arange(1,10)
            board.index = np.arange(1,10)
            print_board(board)
            print("\n")
            Difficulty = "Easy"
            print(f"{(board == '.').sum().sum()} Missing Elements Left in
The Board!\n")
            f = open('Output/log.txt', "r")
            checks = ['Single Candidate', 'Y-Wing', 'XYZ-Wing', 'X-Wing', '
Swordfish', 'Singles-Chains', 'Pointing Pairs',
                    'Naked Pairs', 'Naked Triples', 'Naked Quads', 'Hidden
Singles', 'Hidden Pairs', 'Hidden Triples', 'Hidden Quads' , 'Naked
Singles']
            counts = {}
            for line in f:
                for i in range(len(checks)):
                    if checks[i] in line:
                        if i in [1,2,3,4]:
                            Difficulty = "Hard"
                        elif i not in [1, 2, 3, 4, 0, 10, 14] and Difficulty
 != "Hard":
                            Difficulty = "Medium"
                        if checks[i] in counts.keys():
                            counts[checks[i]] += 1
                        else:
                            counts[checks[i]] = 1

            f = open('Output/count.txt', 'a')
            for key in counts.keys():
                f.write(f"{key} has occured {counts[key]} times\n")
```

```
            f.write(f"The Difficulty level is {Difficulty}\n")
            t1 = time.time()
            f.write(f"{t1}\n")
            f.close()
        else:
            print_board(board)
            print("Error")
        sys.exit(0)
```

B.1.3 `init_board.py`

This is the function which initializes the board.

```
import sys
import pandas as pd
#%% construct the board from 81 digit grid string
def init_board(grid):

    if len(grid)==81:
        board = []

        for i in range(81):
            board.append(int(grid[i]))

        board = pd.Series(board).replace(0,".")
        board = pd.DataFrame(board.values.reshape((9,9)))

        board.index = ["A","A","A","B","B","B","C","C","C"]
        board.columns = ["A","A","A","B","B","B","C","C","C"]

        #helps finding the boxes when (row,col) pair is known
        square_pos = pd.DataFrame([ [1,1,1,2,2,2,3,3,3],
                                    [1,1,1,2,2,2,3,3,3],
                                    [1,1,1,2,2,2,3,3,3],
                                    [4,4,4,5,5,5,6,6,6],
                                    [4,4,4,5,5,5,6,6,6],
                                    [4,4,4,5,5,5,6,6,6],
                                    [7,7,7,8,8,8,9,9,9],
                                    [7,7,7,8,8,8,9,9,9],
                                    [7,7,7,8,8,8,9,9,9]])
        return board,square_pos
    else:
        print(f"Board length: {len(grid)}")
        sys.exit(1)
```

B.1.4 parse_input.py

This is the function which parses the input and turns it into our desired format.

```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
import time
import sys
from print_board import print_board
from validation_check import check_valid
from candidate_handler import candidates, candidates_update
from cells_seen import cells_seen
from solver import solver
from init_board import init_board

#file_name = input("Please enter the file location: ")
grid = ""
#enter input location
main = open("Inputs\\Symmetrical Inputs.txt")
i = 1
c = 0
for single in main:

    if str(i)!=sys.argv[1]:
        i+=1
        continue

    symtest = np.zeros((9,9))

    n = open("Inputs\\input.txt", "w")
    for ind in range(9):
        for j in range(9):
            if(single[ind*9+j] != '.'):
                n.write(single[ind*9+j])
                symtest[ind][j] = 1
                c+=1
                n.write('|')
            else:
                n.write('0')
                n.write('|')
    n.close()

    f = open("Inputs\\input.txt", "r")
    for line in f:
        for num in line.split('|')[:-1]:
            if(int(num) < 10):
```

```
                grid+=str(num)

    t1 = time.time()
    board,square_pos = init_board(grid)
    print_board(board)

    f = open('Output/count.txt'.format(i), 'w')
    f.write(f"The Count is {c}\n")
    f.write(f"{t1}\n")

    print(symtest)
    print(np.rot90(symtest))

    if np.array_equal(symtest, np.rot90(symtest)):
        f.write(f"The Input is 90 Degree Symmetric.\n")

    if np.array_equal(symtest, np.rot90(np.rot90(symtest))):
        f.write(f"The Input is 180 Degree Symmetric.\n")

    f.close()
    f = open('Output/log.txt'.format(i), 'w')
    f.write('Log Start\n')
    f.close()
    #check validity before solving it
    if check_valid(board):
        cands = candidates(board)
        print(cands)
        solver(board,cands,square_pos)
    else:
        print("The board is not valid!")

    break
```

B.1.5 `cells_seen.py`

This is the function which keeps track of if a cell is visited or not.

```
import pandas as pd

#finds returns all seen cells given the location of the key cell
def cells_seen(inx,square_pos):
    cells = []
    #box
    cells.extend(square_pos[square_pos == square_pos.iloc[inx]].stack().
    index)
    #rows
    cells.extend(square_pos.iloc[[inx[0]],:].stack().index)
```

```
    #cols
    cells.extend(square_pos.iloc[:,[inx[1]]].stack().index)
    cells = pd.Series(cells).unique()
    return cells
```

B.1.6 `select_group.py`

This is the function which eliminates some row or column as candidates.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
```

```
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(temp)
                                cands.set_value(rows,pointcols[0],np.array(temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]} Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(temp)
                                cands.set_value(pointrows[0],cols,np.array(temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols} Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
```

```
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.7 `candidate_handler.py`

This is the function which finds all the candidates for a cell and keeps track of them. If elimination of candidates are required it calls other sub-routines to handle it.

```
import pandas as pd
import numpy as np
from cells_seen import cells_seen

#%% INITIALISE CANDIDATES
def candidates(board):
    cands = pd.DataFrame(np.full((9,9),np.nan))
    for row in range(9):
        for col in range(9):
            temp = []
            if board.iloc[row,col] == ".":
                temp.extend(board.iloc[row,:])
                temp.extend(board.iloc[:,col])
                temp.extend(board.loc[board.index[row]][board.columns[col]].values.flatten())
                temp = pd.Series(temp)
                drops = temp[temp!="."].unique()
                cand = pd.Series(np.arange(1,10),index=np.arange(1,10)).drop(drops).values.astype("O")

                #this part is for handling the "ValueError: setting an array element as a sequence"
                if len(cand) == 1:
                    if cand%1 == 0.0:
                        temp = cand.tolist()
                        temp.append(99)
                        cands[col][row] = np.array(temp).astype("O")
                        temp.remove(99)
                        cands[col][row] = np.array(temp).astype("O")
                else:
                    cands[col][row] = cand

    return cands
#%% UPDATE CANDIDATES
def candidates_update(cands,row,col,val,square_pos):
    #remove the value from candidate matrix first
    cands.iloc[row,col] = np.nan
```

```
35
36      #locate the cells seen by the key cell
37      seen_cells = cells_seen((row,col),square_pos)
38
39      #remove the value from the seen cells
40      for i in seen_cells:
41          try:
42              if val in cands.iloc[i]:
43                  cands.loc[i[0]][i[1]] = np.delete(cands.iloc[i],np.where(
    cands.iloc[i]==val))
44          except:
45              pass
46
47      return cands
```

B.1.8 validation_check.py

This is the function which checks if the constraints of Sudokus are maintained across row, column and box.

```
1  import pandas as pd
2
3  def row_check(board,isprint=False):
4      check = True
5      for row in range(9):
6          a = board.iloc[row,:].value_counts()
7          try:
8              a = a.drop(".")
9          except:
10             pass
11
12         check = check and not any(a>1)
13
14     if isprint:
15         if check:
16             print("Row Check: Passed")
17         else:
18             print("Row Check: Failed")
19
20     return check
21
22 def col_check(board,isprint=False):
23     check = True
24     for col in  range(9):
25         a = board.iloc[:,col].value_counts()
26         try:
27             a = a.drop(".")
```

```
        except:
            pass

        check = check and not any(a>1)

    if isprint:
        if check:
            print("Column Check: Passed")
        else:
            print("Column Check: Failed")

    return check

def square_check(board,isprint=False):
    check = True
    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            a = pd.Series(board.iloc[i,j].values.flatten()).value_counts()
            try:
                a = a.drop(".")
            except:
                pass

        check = check and not any(a>1)

    if isprint:
        if check:
            print("Square Check: Passed")
        else:
            print("Square Check: Failed")

    return check

#validation check function
def check_valid(board):
    rcheck = row_check(board)
    ccheck = col_check(board)
    cucheck = square_check(board)

    return rcheck and ccheck and cucheck
```

B.1.9 `single_candidate.py`

This is the function which implements single candidate algorithm.

```

from candidate_handler import candidates_update
```

```
#interdependent modules, it has to be imported like below,otherwise it wont work
import solver as solver

#%% SINGLE CANDIDATES
def single_cand(board,cands,square_pos):
    ischanged = 0
    for row in range(9):
        for col in range(9):
            if board.iloc[row,col] == ".":
                cand = cands.iloc[row,col]
                # cand = int(cands.iloc[row,col][0])
                try:
                    lenx = len(cand)
                    if lenx == 1:
                        ischanged = 1
                        f = open('Output/log.txt','a')
                        f.write(f"R{row+1}C{col+1}={cand[0]} : Single Candidate\n")
                        print(f"R{row+1}C{col+1}={cand[0]} : Single Candidate")
                        f.close()
                        board.iloc[row,col] = cand[0]
                        cands = candidates_update(cands,row,col,cand[0], square_pos)
                except:
                    ischanged = 1
                    f = open('Output/log.txt','a')
                    f.write(f"R{row+1}C{col+1}={cand} : Single Candidate (except)\n")
                    print(f"R{row+1}C{col+1}={cand} : Single Candidate (except)")
                    f.close()
                    board.iloc[row,col] = cand
                    cands = candidates_update(cands,row,col,cand,square_pos)
    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.10 `hidden_singles.py`

This is the function which implements hidden singles algorithm.

```
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont work
import solver as solver
```

```
5
6  #%% HIDDEN SINGLES
7  def hidden_singles(board,cands,square_pos):
8      is_changed = 0
9      #check rows
10     for row in range(9):
11         temp = []
12         for col in range(9):
13             if board.iloc[row,col] == ".":
14                 temp.extend(cands.iloc[row,col])
15         valco = pd.Series(temp).value_counts()
16         for valun in valco.index[valco == 1].values: #????? there can be 1
       and only 1 single candidate in a row..
17             temp1 = cands.loc[row].dropna().apply(lambda x: valun in x)
18             inx = temp1.index[temp1 == True]
19             if len(inx):
20                 is_changed = 1
21                 for inxt in inx:
22                     f = open('Output/log.txt','a')
23                     f.write(f"R{row+1}C{inxt+1}={valun} : Hidden Singles (
       row)\n")
24                     print(f"R{row+1}C{inxt+1}={valun} : Hidden Singles (row)
       ")
25                     f.close()
26                     board.iloc[row,inxt] = valun
27                     cands = candidates_update(cands,row,inxt,valun,
       square_pos)
28
29     #check columns
30     for col in range(9):
31         temp = []
32         for row in range(9):
33             if board.iloc[row,col] == ".":
34                 temp.extend(cands.iloc[row,col])
35         valco = pd.Series(temp).value_counts()
36         for valun in valco.index[valco == 1].values:
37             temp1 = cands.iloc[:,col].dropna().apply(lambda x: valun in x)
38             inx = temp1.index[temp1 == True]
39             if len(inx):
40                 is_changed = 1
41                 for inxt in inx:
42                     f = open('Output/log.txt','a')
43                     f.write(f"R{inxt+1}C{col+1}={valun} : Hidden Singles (
       col)\n")
44                     print(f"R{inxt+1}C{col+1}={valun} : Hidden Singles (col)
       ")
45                     f.close()
46                     board.iloc[inxt,col] = valun
```

```
                    cands = candidates_update(cands,inxt,col,valun,
    square_pos)

    #check squares
    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            a = cands.iloc[i,j]
            a_flat = pd.Series(a.values.flatten()).dropna()
            temp = []
            for ix in a_flat:
                temp.extend(ix)
            valco = pd.Series(temp).value_counts()

            if any(valco == 1):
                to_change_all = valco.index[valco == 1].values
                for to_change in to_change_all: #loop all values to be
    changed (sometimes multiple changes needed in a single box)
                    for rowx in a.index:
                        for colx in a.columns:
                            try:
                                if board.iloc[rowx,colx] == ".":
                                    if to_change in a.loc[rowx][colx]:
                                        f = open('Output/log.txt','a')
                                        f.write(f"R{rowx+1}C{colx+1}={
    to_change} : Hidden Singles (square)\n")
                                        print(f"R{rowx+1}C{colx+1}={
    to_change} : Hidden Singles (square)")
                                        f.close()
                                        board.iloc[rowx,colx] = to_change
                                        cands = candidates_update(cands,rowx
    ,colx,to_change,square_pos)
                                        is_changed = 1
                            except:
                                print("except")
    if is_changed:
        solver.solver(board,cands,square_pos)
```

B.1.11 hidden_pairs_triples_quads.py

This is the function which implements hidden pairs, triples and quads algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
from select_group import select_group
# from hidden_remove import hidden_remove
```

```
#interdependent modules, it has to be imported like below,otherwise it wont work
import solver as solver
import itertools
#%% HIDDEN PAIRS-TRIPLES
#find hidden pairs-triples-quads in the candidates dataframe
def hidden_pairs_triples(board,cands,square_pos):
    ischanged = 0

    #go through combinations (2 and 3 elements at a time)
    for pair_triple_quad in [2,3]:
        #check for rows, columns and box consecutively
        for rowcolbox in ["row","col","box"]:
            #check each groups of rows, columns or boxes
            for group_no in range(9):
                #get the group
                use = select_group(rowcolbox,cands,group_no,square_pos)

                #find unique values in the group
                vals = []
                for ix in use:
                    vals.extend(ix)
                vals = pd.Series(vals).unique()

                #find pair, triple combinations of the unique values
                for comb in itertools.combinations(vals, pair_triple_quad):
                    inxs = np.full(use.shape,False)
                    for com in comb:
                        inxs = (inxs) | (use.apply(lambda x: com in x))

                    #if it is a pair,triple
                    if sum(inxs) == pair_triple_quad:
                        no_of_hidden = sum(inxs)
                        #determine hidden pair,triple indexes
                        hidden_inxs = inxs.index[inxs]

                        #remove the value from the specific cell of the candidates
                        cands,ischanged = hidden_remove(rowcolbox,group_no,hidden_inxs,comb,cands,square_pos,no_of_hidden)

                        if ischanged:
                            solver.solver(board,cands,square_pos)

#%% HIDDEN QUADS
#find hidden quads in the candidates dataframe
def hidden_quads(board,cands,square_pos):
    ischanged = 0
```

```
51
52      #go through combinations (4 elements at a time)
53      pair_triple_quad = 4
54      #check for rows, columns and box consecutively
55      for rowcolbox in ["row","col","box"]:
56          #check each groups of rows, columns or boxes
57          for group_no in range(9):
58              #get the group
59              use = select_group(rowcolbox,cands,group_no,square_pos)
60
61              #find unique values in the group
62              vals = []
63              for ix in use:
64                  vals.extend(ix)
65              vals = pd.Series(vals).unique()
66
67              #find quad combinations of the unique values
68              for comb in itertools.combinations(vals, pair_triple_quad):
69                  inxs = np.full(use.shape,False)
70                  for com in comb:
71                      inxs = (inxs) | (use.apply(lambda x: com in x))
72
73                  #if it is a quad
74                  if sum(inxs) == pair_triple_quad:
75                      no_of_hidden = sum(inxs)
76                      #determine hidden quad indexes
77                      hidden_inxs = inxs.index[inxs]
78
79                      #remove the value from the specific cell of the
    candidates
80                      cands,ischanged = hidden_remove(rowcolbox,group_no,
    hidden_inxs,comb,cands,square_pos,no_of_hidden)
81
82                      if ischanged:
83                          solver.solver(board,cands,square_pos)
84
85  #removes values except hidden pairs, triples or quads from a group (row col
    or box)
86  #it is different than "naked_remove" function
87  def hidden_remove(rowcolbox,group_no,hidden_inxs,comb,cands,square_pos,
    no_of_hidden):
88      pairtriplequad = {2:"Pair",3:"Triple",4:"Quad"}
89      ischanged = 0
90      for cells in hidden_inxs:
91          if rowcolbox == "row":
92              row = group_no
93              col = cells
94          elif rowcolbox == "col":
```

```
            row = cells
            col = group_no
        elif rowcolbox == "box":
            box_inx = square_pos[square_pos==group_no+1].stack().index
            row = box_inx[cells][0]
            col = box_inx[cells][1]

        #following line is for printing purposes only
        removed_vals = set(cands.iloc[row,col]).difference(set(comb))

        #replace values in the cell with the intersection of combination and the cell
        if len(removed_vals):
            cands.at[row,col] = np.array(list(set(comb).intersection(set(cands.iloc[row,col]))))
            f = open('Output/log.txt','a')
            f.write(f"R{row:<1}C{col:<1}   :  Hidden {pairtriplequad[no_of_hidden]}s ({rowcolbox:<3}), {str(removed_vals):<15} removed, {pairtriplequad[no_of_hidden]:>7}s: {str(comb):>6}\n")
            print(f"R{row:<1}C{col:<1}   :  Hidden {pairtriplequad[no_of_hidden]}s ({rowcolbox:<3}), {str(removed_vals):<15} removed, {pairtriplequad[no_of_hidden]:>7}s: {str(comb):>6}")
            f.close()
            ischanged = 1

    return cands,ischanged
# %%
```

B.1.12 naked_pairs_triples_quads.py

This is the function which implements naked pairs, triples and quads algorithm algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
from select_group import select_group
# from hidden_remove import hidden_remove
#interdependent modules, it has to be imported like below,otherwise it wont work
import solver as solver
import itertools
#%% HIDDEN PAIRS-TRIPLES
#find hidden pairs-triples-quads in the candidates dataframe
def hidden_pairs_triples(board,cands,square_pos):
    ischanged = 0

    #go through combinations (2 and 3 elements at a time)
```

```
    for pair_triple_quad in [2,3]:
        #check for rows, columns and box consecutively
        for rowcolbox in ["row","col","box"]:
            #check each groups of rows, columns or boxes
            for group_no in range(9):
                #get the group
                use = select_group(rowcolbox,cands,group_no,square_pos)

                #find unique values in the group
                vals = []
                for ix in use:
                    vals.extend(ix)
                vals = pd.Series(vals).unique()

                #find pair, triple combinations of the unique values
                for comb in itertools.combinations(vals, pair_triple_quad):
                    inxs = np.full(use.shape,False)
                    for com in comb:
                        inxs = (inxs) | (use.apply(lambda x: com in x))

                    #if it is a pair,triple
                    if sum(inxs) == pair_triple_quad:
                        no_of_hidden = sum(inxs)
                        #determine hidden pair,triple indexes
                        hidden_inxs = inxs.index[inxs]

                        #remove the value from the specific cell of the candidates
                        cands,ischanged = hidden_remove(rowcolbox,group_no,hidden_inxs,comb,cands,square_pos,no_of_hidden)

                        if ischanged:
                            solver.solver(board,cands,square_pos)

#%% HIDDEN QUADS
#find hidden quads in the candidates dataframe
def hidden_quads(board,cands,square_pos):
    ischanged = 0

    #go through combinations (4 elements at a time)
    pair_triple_quad = 4
    #check for rows, columns and box consecutively
    for rowcolbox in ["row","col","box"]:
        #check each groups of rows, columns or boxes
        for group_no in range(9):
            #get the group
            use = select_group(rowcolbox,cands,group_no,square_pos)

```

```
            #find unique values in the group
            vals = []
            for ix in use:
                vals.extend(ix)
            vals = pd.Series(vals).unique()

            #find quad combinations of the unique values
            for comb in itertools.combinations(vals, pair_triple_quad):
                inxs = np.full(use.shape,False)
                for com in comb:
                    inxs = (inxs) | (use.apply(lambda x: com in x))

                #if it is a quad
                if sum(inxs) == pair_triple_quad:
                    no_of_hidden = sum(inxs)
                    #determine hidden quad indexes
                    hidden_inxs = inxs.index[inxs]

                    #remove the value from the specific cell of the candidates
                    cands,ischanged = hidden_remove(rowcolbox,group_no, hidden_inxs,comb,cands,square_pos,no_of_hidden)

                    if ischanged:
                        solver.solver(board,cands,square_pos)

#removes values except hidden pairs, triples or quads from a group (row col or box)
#it is different than "naked_remove" function
def hidden_remove(rowcolbox,group_no,hidden_inxs,comb,cands,square_pos, no_of_hidden):
    pairtriplequad = {2:"Pair",3:"Triple",4:"Quad"}
    ischanged = 0
    for cells in hidden_inxs:
        if rowcolbox == "row":
            row = group_no
            col = cells
        elif rowcolbox == "col":
            row = cells
            col = group_no
        elif rowcolbox == "box":
            box_inx = square_pos[square_pos==group_no+1].stack().index
            row = box_inx[cells][0]
            col = box_inx[cells][1]

        #following line is for printing purposes only
        removed_vals = set(cands.iloc[row,col]).difference(set(comb))

```

```
105         #replace values in the cell with the intersection of combination and
        the cell
106         if len(removed_vals):
107             cands.at[row,col] = np.array(list(set(comb).intersection(set(
        cands.iloc[row,col]))))
108             f = open('Output/log.txt','a')
109             f.write(f"R{row:<1}C{col:<1}   :  Hidden {pairtriplequad[
        no_of_hidden]}s ({rowcolbox:<3}), {str(removed_vals):<15} removed, {
        pairtriplequad[no_of_hidden]:>7}s: {str(comb):>6}\n")
110             print(f"R{row:<1}C{col:<1}   :  Hidden {pairtriplequad[
        no_of_hidden]}s ({rowcolbox:<3}), {str(removed_vals):<15} removed, {
        pairtriplequad[no_of_hidden]:>7}s: {str(comb):>6}")
111             f.close()
112             ischanged = 1
113 
114     return cands,ischanged
115 # %%
```

B.1.13 pointing_pairs.py

This is the function which implements pointing pairs algorithm.

```
1  import numpy as np
2  import pandas as pd
3  from candidate_handler import candidates_update
4  #interdependent modules, it has to be imported like below,otherwise it wont
       work
5  import solver as solver
6  
7  #%% POINTING PAIRS
8  def pointing_pairs(board,cands,square_pos):
9      # print("Pointing Pairs")
10     ischanged = 0
11 
12     for i in [[0,1,2],[3,4,5],[6,7,8]]:
13         for j in [[0,1,2],[3,4,5],[6,7,8]]:
14             use = cands.iloc[i,j]
15             use_flat = pd.Series(use.values.flatten()).dropna()
16 
17             temp = []
18             for ix in use_flat:
19                 temp.extend(ix)
20 
21             valco = pd.Series(temp).value_counts()
22             # try:
23             pair_vals = valco.index[(valco == 2) | (valco == 3)]
24 
```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(temp)
                                cands.set_value(rows,pointcols[0],np.array(temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]} Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.14 box_line.py

This is the function which implements box line algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.15 naked_quads.py

This is the function which implements naked quads algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.16 `singles_chains.py`

This is the function which implements single chains algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.17 `x_wing.py`

This is the function which implements X-Wing algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.18 `y_wing.py`

This is the function which implements Y-Wing algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.19 `xyz_wing.py`

This is the function which implements XYZ-Wing algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.20 `swordfish.py`

This is the function which implements swordfish algorithm.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

```

```
            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[pointrows[0],cols] = np.array(
temp)
```

```
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.21 brute_force.py

This is the function which implements brute force algorithm.Since our solver doesn't include all human like methods when our solver gets stuck it uses brute force to find the solution.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
            pair_vals = valco.index[(valco == 2) | (valco == 3)]
```

```
24
25          for pair_val in pair_vals:
26              pointrows, pointcols = [],[]
27              for ii in use.index:
28                  for jj in use.columns:
29                      try:
30                          if pair_val in use.loc[ii][jj].tolist():
31                              pointrows.extend([ii])
32                              pointcols.extend([jj])
33                      except:
34                          pass
35
36              #pairs point in the column direction
37              try:
38                  if not any(np.diff(pointcols)):
39                      change_col = cands[pointcols[0]].dropna().drop(
    pointrows)
40                      for rows in change_col.index:
41                          temp = change_col[rows].tolist()
42                          try:
43                              temp.remove(pair_val)
44                              # cands.iloc[rows,pointcols[0]] = np.array(
    temp)
45                              cands.set_value(rows,pointcols[0],np.array(
    temp))
46                              f = open('Output/log.txt','a')
47                              f.write(f"R{rows}C{pointcols[0]}
    Pointing Pairs (cols), {pair_val} removed\n")
48                              print(f"R{rows}C{pointcols[0]}     Pointing
    Pairs (cols), {pair_val} removed")
49                              f.close()
50                              ischanged = 1
51                              # solver.solver(board,cands,square_pos)
52                          except:
53                              pass
54              except:
55                  pass
56
57              #pairs point in the row direction
58              try:
59                  if not any(np.diff(pointrows)):
60                      change_col = cands.loc[pointrows[0]].dropna().drop(
    pointcols)
61                      for cols in change_col.index:
62                          temp = change_col[cols].tolist()
63                          try:
64                              temp.remove(pair_val)
```

```
                                # cands.iloc[pointrows[0],cols] = np.array(
    temp)
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.22 print_board.py

This is the function which prints the solution of the puzzle.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
```

```
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
```

```
                                # cands.iloc[pointrows[0],cols] = np.array(
    temp)
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.1.23 `Auto5.bat`

This is the batch file we used to run our solver on our whole dataset.

```
import numpy as np
import pandas as pd
from candidate_handler import candidates_update
#interdependent modules, it has to be imported like below,otherwise it wont
    work
import solver as solver

#%% POINTING PAIRS
def pointing_pairs(board,cands,square_pos):
    # print("Pointing Pairs")
    ischanged = 0

    for i in [[0,1,2],[3,4,5],[6,7,8]]:
        for j in [[0,1,2],[3,4,5],[6,7,8]]:
            use = cands.iloc[i,j]
            use_flat = pd.Series(use.values.flatten()).dropna()

            temp = []
            for ix in use_flat:
                temp.extend(ix)

            valco = pd.Series(temp).value_counts()
            # try:
```

```
            pair_vals = valco.index[(valco == 2) | (valco == 3)]

            for pair_val in pair_vals:
                pointrows, pointcols = [],[]
                for ii in use.index:
                    for jj in use.columns:
                        try:
                            if pair_val in use.loc[ii][jj].tolist():
                                pointrows.extend([ii])
                                pointcols.extend([jj])
                        except:
                            pass

                #pairs point in the column direction
                try:
                    if not any(np.diff(pointcols)):
                        change_col = cands[pointcols[0]].dropna().drop(
pointrows)
                        for rows in change_col.index:
                            temp = change_col[rows].tolist()
                            try:
                                temp.remove(pair_val)
                                # cands.iloc[rows,pointcols[0]] = np.array(
temp)
                                cands.set_value(rows,pointcols[0],np.array(
temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{rows}C{pointcols[0]}
Pointing Pairs (cols), {pair_val} removed\n")
                                print(f"R{rows}C{pointcols[0]}     Pointing
Pairs (cols), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

                #pairs point in the row direction
                try:
                    if not any(np.diff(pointrows)):
                        change_col = cands.loc[pointrows[0]].dropna().drop(
pointcols)
                        for cols in change_col.index:
                            temp = change_col[cols].tolist()
                            try:
                                temp.remove(pair_val)
```

```
                                # cands.iloc[pointrows[0],cols] = np.array(
    temp)
                                cands.set_value(pointrows[0],cols,np.array(
    temp))
                                f = open('Output/log.txt','a')
                                f.write(f"R{pointrows[0]}C{cols}
    Pointing Pairs (rows), {pair_val} removed\n")
                                print(f"R{pointrows[0]}C{cols}     Pointing
    Pairs (rows), {pair_val} removed")
                                f.close()
                                ischanged = 1
                                # solver.solver(board,cands,square_pos)
                            except:
                                pass
                except:
                    pass

    if ischanged:
        solver.solver(board,cands,square_pos)
```

B.2 Brute Force Solver

This is our other solver which completely solves the puzzle using only brute force method.

```
import math
from datetime import datetime

class Brute_Force(object):

    def __init__(self):
        self.n = 0
        self.b = 0
        self.puzzle = []
        self.num_guesses = 0
        self.known_indices = []

    def load_puzzle(self, puzzle_string):
        self.puzzle = []
        self.known_indices = []
        rows = [row for row in puzzle_string.split("\n") if row]
        self.n = 9 #len(rows)
        self.b = 3 #int(math.sqrt(self.n))
        for row_index in range(9):
            for col_index in range(9):
                c = puzzle_string[row_index * self.n + col_index]
                if c == '.':
```

```
                    self.puzzle.append(0)
                else:
                    self.puzzle.append(int(c))
                    self.known_indices.append((row_index * self.n) +
col_index)

    def load_puzzle_from_file(self, path):
        self.load_puzzle(path)

    def to_string(self, pretty=True):
        return "".join(['\n' + str(c) if i % self.n == 0 else str(c) for i,
c in enumerate(self.puzzle)])

    def __repl__(self):
        return self.to_string(pretty=False)

    def __str__(self):
        return self.to_string()

    def solve(self):
        start_time = datetime.now()
        self.num_guesses = 0
        r = self.solve_from(0, 1)
        while r is not None:
            r = self.solve_from(r[0], r[1])
        return datetime.now() - start_time

    def solve_from(self, index, starting_guess):
        if index < 0 or index > len(self.puzzle):
            raise Exception("Invalid puzzle index %s after %s guesses" % (
index, self.num_guesses))

        last_valid_guess_index = None
        found_valid_guess = False
        for i in range(index, len(self.puzzle)):
            if i not in self.known_indices:
                found_valid_guess = False
                for guess in range(starting_guess, self.n + 1):
                    self.num_guesses += 1
                    if self.valid(i, guess):
                        found_valid_guess = True
                        last_valid_guess_index = i
                        self.puzzle[i] = guess
                        break

                starting_guess = 1
                if not found_valid_guess:
                    break
```

```
68
69          if not found_valid_guess:
70              new_index = last_valid_guess_index if last_valid_guess_index is
    not None else index - 1
71              new_starting_guess = self.puzzle[new_index] + 1
72              self.reset_puzzle_at(new_index)
73
74              while new_starting_guess > self.n or new_index in self.
    known_indices:
75                  new_index -= 1
76                  new_starting_guess = self.puzzle[new_index] + 1
77                  self.reset_puzzle_at(new_index)
78
79              return (new_index, new_starting_guess)
80          else:
81              return None
82
83      def reset_puzzle_at(self, index):
84          for i in range(index, len(self.puzzle)):
85              if i not in self.known_indices:
86                  self.puzzle[i] = 0
87
88      def valid_for_row(self, index, guess):
89          row_index = int(math.floor(index / self.n))
90          start = self.n * row_index
91          finish = start + self.n
92          for c_index in range(start, finish):
93              if c_index != index and self.puzzle[c_index] == guess:
94                  return False
95          return True
96
97      def valid_for_column(self, index, guess):
98          col_index = index % self.n
99          for r in range(0, self.n):
100             r_index = col_index + (self.n * r)
101             if r_index != index and self.puzzle[r_index] == guess:
102                 return False
103         return True
104
105     def valid_for_block(self, index, guess):
106         row_index = int(math.floor(index / self.n))
107         col_index = index % self.n
108
109         block_row = int(math.floor(row_index / self.b))
110         block_col = int(math.floor(col_index / self.b))
111
112         row_start = block_row * self.b
113         row_end = row_start + self.b - 1
```

```
        col_start = block_col * self.b
        col_end = col_start + self.b - 1

        for r in range(row_start, row_end + 1):
            for c in range(col_start, col_end + 1):
                i = c + (r * self.n)
                if self.puzzle[i] == guess:
                    return False

        return True

    def valid(self, index, guess):
        return self.valid_for_row(index, guess) and self.valid_for_column(
    index, guess) and self.valid_for_block(index, guess)


if __name__ == "__main__":
    import sys

    s = Brute_Force()
    main_file = open("sudoku-3m.csv", "r")
    i = 1
    c = 0
    print(sys.argv[1])
    for single in main_file:
        if str(i) != sys.argv[1]:
            i += 1
            continue

        sudoku = single.split(',')[1]

        s.load_puzzle(sudoku)

        print("\nPuzzle:\n%s" % (s))
        time = s.solve()
        print("\nSolution:\n%s" % (s))
        print("\nNumber of guesses: %s" % (s.num_guesses))
        with open('output.txt' , 'a') as f:
            f.write("{},{},{}".format(sudoku, time,single.split(',')[-1]))

        break

    main_file.close()
```

APPENDIX C
GRAPHS AND TABLES DATA

Data for the graphs presented in Experimentation and Result chapter can be found in the following links:
Graphs and data for 17 clue dataset
Graphs and data for Kaggle's 3 million dataset