Analyzing the Effectiveness of Constraints on Reducing the Number of Necessary Hints in Sudoku Variants using Z3

Noah Antisseril

September 2021

Abstract

The minimum number of clues possible in a Sudoku is the number of clues required to ensure a unique solution to the puzzle. It has been proven that the lowest number of hints possible in Sudoku is 17. However, variants of the game impose further constraints on the player. As a result, the number of hints becomes lower when finding the lowest number of hints on a variant of Sudoku. In this article, different variants are used to analyze the effect of constraints on reducing the minimum number of hints. In the process, the absolute minimum number of hints on a board is also found.

1 Introduction

Sudoku is a widely popular logic puzzle consisting of a 9x9 board with 9 boxes of 3x3. Within the puzzle, each 3x3 box, row, and column must contain the numbers 1-9. Each of these groups must contain each number once. Most Sudoku puzzles are created and solved using backtracking algorithms, which is a type of brute force algorithm designed to test out possibilities. However, satisfiability modulo theories (SMT) solvers can be used to a similar effect. In particular, the SMT solver Z3 can be utilized to create a satisfiable model to create and solve most Sudoku puzzles. The more interesting question to be asked regarding Sudoku puzzles is the minimum number of hints possible on a given board. This is a thoroughly researched question, with mathematicians and puzzle enthusiasts asserting the absolute minimum number of hints is 17. The evidence for this claim came from the fact that 16 hint boards could not be found, while 17 hint boards could. However, variants of Sudoku impose extra sets of rules that could either increase or decrease the number of necessary hints required. The goal of this paper is to investigate the effect of further constraints on the minimum number of hints possible. Constraints come in the form of extra rules, usually created as a variant of the game to increase the difficulty on the player. Please note that this paper will not find the absolute minimum number

of hints for a given variant, rather it aims to investigate the effects of the variant on the least number of hints. In other words, the question this paper asks is how effective are certain variants in reducing the amount of necessary hints in a Sudoku puzzle?

1.1 The Minimum Number of Hints

A "proper" Sudoku puzzle is a board that, when solved, can only produce one solution. Therefore, the necessary hints on a Sudoku board are the hints that make the board unique. In other words, if a necessary hint is taken out, then additional solutions can be introduced when solving. Throughout this paper, when the term "minimum number of hints" is referenced, it refers to the number of necessary hints on the board.

1.2 Descriptions of Sudoku Variants

Variants of Sudoku take the base game and add extra constraints to the puzzle, which can both help and hinder the player. The variants discussed within this paper will be "Hypersudoku", "Anti-Knight Sudoku", and "Anti-King Sudoku". This paper will investigate the least number of hints possible on each board, and how further constraints affect the number of hints.

1.2.1 Hypersudoku

Hypersudoku maintains the constraints found in the base puzzle of Sudoku. Four more 3 x 3 boxes are placed on the board, with the constraints that the numbers 1-9 must appear in each of these new boxes only once. Please see the following sample board from the Cross+A website.

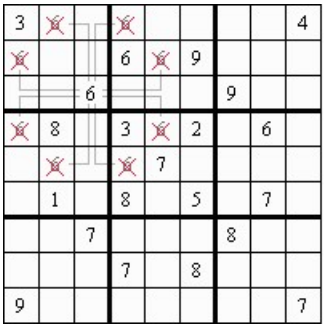
9	8						7	1
7			8		1			5
				4				
	7		9		3		4	
		9				3		
	3		7		8		9	
				8				
6			2		4			9
4	2						5	6

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

1.2.2 Anti-Knight Sudoku

Anti-Knight Sudoku maintains the constraints found in the base puzzle of Sudoku. The added constraint is that all cells in a chess knight's move must contain different numbers. A chess knight can move 2 cells in one direction,

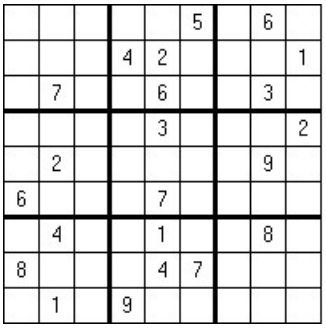
and 1 cell in another direction. Please see the following sample board from the Cross+A website.



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

1.2.3 Anti-King Sudoku

Anti-King Sudoku maintains the constraints found in the base puzzle of Sudoku. The added constraint is that all cells in a chess king's move must contain different numbers. A chess king can move 1 cell in any direction around itself, including vertically, horizontally, and diagonally. Please see the following sample board from the Cross+A website.



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

1.3 SMT/SAT Solvers and Z3

SMT solvers are designed to find satisfiability in first order logic problems, while satisfiability (SAT) solvers are designed to find satisfiability in boolean logic problems. SAT solvers do this by creating a model that assigns truth values to each boolean. In a similar manner, SMT solvers can check the satisfiability and find a model that can solve a given formula. The software Z3 released by Microsoft is both an SMT and SAT solver, meaning it can handle boolean and first order logic, and return models satisfying both. Z3 is highly efficient, with the ability to solve complex problems in a matter of seconds. Throughout this paper, Z3 is used extensively as a means of finding solutions to a Sudoku with specific constraints.

1.4 Related Works

No other papers investigate the effect of constraints on the minimum number of hints in the way this paper does. However, similar problems have been solved before. Gary McGuire of University College Dublin and his associates tackled the issue of definitively proving that 17 hints was the minimum for the base version of Sudoku. In their remarkable paper, they discuss the methodology they used. Their approach was essentially a search of all possible Sudoku boards for a board with a 16 hint minimum. They developed clever solutions to reduce the number of boards they checked, like checking for board transformations or reflections, as well as developing an efficient algorithm. Once these were removed, they were able to reduce the total number of boards that they checked from 6.67 x 10^{21} to 5.47 x 10^{9}. From this, they never found a 16 hint board, proving exhaustively that no 16 hint board exists.

This paper will not take the same route as Gary McGuire and his associates. For the purpose of determining the effect of constraints, the absolute minimum number of hints for each variant is not necessary. While that information could be useful, it would ultimately be time-consuming and pointless when a similar conclusion could be drawn from the mean minimum number of hints of a variant.

2 Methodology

To answer the questions proposed by this paper, the following steps were taken:

1. Created a solver to find the solutions to a given Sudoku
2. Added the constraints from the variants of Sudoku
3. Created an algorithm to generate random boards according to given constraints
4. Created an algorithm to calculate the minimum number of hints

2.1 Z3 in Python

Z3 software can be used to create a rudimentary Sudoku solver. This is done by creating assertions in the program to constrain the model to the rules of Sudoku. For instance, one could write an assertion that would ask Z3 to produce a model with the numbers 1-9 in a row. However, there are two major flaws with this approach. First, it is time consuming to type out the code in Z3. Simply initializing each board piece would take 81 lines, and the rest of the solver would span hundreds of lines of code. Second, the time to generate a single board is long. It would take upwards of 5 minutes to generate a single board from only assertions.

The use of Z3 in Python drastically cut down the amount of code that was needed. The entire board could be created using a formatted list to initialize each cell. This board is what Z3 used to solve a Sudoku. To streamline the

process, a function was created to apply constraints to certain groups. The function called for each term in a certain group to be unique, and utilized tuples to ensure that the numbers 1-9 appeared in each group. The use of tuples cut the time to solve puzzles by a significant amount.

Each group was created with respect to the rules of Sudoku. For example, a 2D list was created for rows, with each list within the 2D list corresponding to a row in the board. The process was repeated for each column and box, so that there were three 2D lists in total. Each of these lists/groups were then passed through the function to apply the Sudoku constraints to it. At this point, Z3 had the assertions to solve a normal Sudoku puzzle.

2.2 Variant Constraints

To create the variants of Sudoku in Z3, their constraints had to be translated into code, then added as assertions. These constraints were declared in a very similar way to the way the constraints for rows, columns, and boxes were declared. The exceptions were the Anti-Knight and Anti-King Sudoku constraints, as these did not require the Sudoku constraints function to be applied to it, since the only constraint was that cells in a Chess Knight or King's move, respectively, were not the same.

2.2.1 Hypersudoku Constraints

Creating the Hypersudoku constraints was very similar to creating the original box constraints. To create the original box constraints, the top left corner of each of the 9 boxes were designated in a 2D list. From there, a loop was implemented to fill out a 3 x 3 grid. This meant that the 2D list corresponding to the boxes on the board consisted of 9 1D lists, each consisting of 9 cells. The Sudoku constraints function was then called on the 2D list, to ensure that each box had the numbers 1-9 without an digits repeating. To create the Hypersudoku constraints, the top left corners of each new box were identified. Considering the first row and column to be index 0, the corners of each of the four new boxes occurred at (1,1), (5,1), (1,5), and (5,5). From there, the 3 x 3 grids were again filled out, creating a 2D list corresponding to the four new boxes. From there, the exact same process as the normal boxes was followed.

2.2.2 Anti-Knight Constraints

Creating the Anti-Knight constraints was a much more complex problem than the Hypersudoku constraints. This was due to the fact that each cell on the board had to have the constraint applied to it, rather than four 3 x 3 boxes. Additionally, the cells near the corners of the board could not move out of the board, meaning that the constraints had to be different for certain cells. To solve this problem, a function was created to check if a given row and column was on the Sudoku board. Then, the movement of the chess knight was added as a 2D list. The chess knight can move two cells in one direction, and one cell in

another. This gave a total of eight spots where a number could not repeat. Once the knight movements were added, it was a matter of iterating through each cell on the board, and applying the constraint for the 8 places in the knight's move. To account for cells near the edge of the board, the function designed to check if a row/column was on the board was utilized. If the considered spot was not on the board, it was ignored by Z3.

2.2.3 Anti-King Constraints

The Anti-King Constraints shared a lot of similarities to the Anti-Knight constraints. Similar to the process for the Anti-Knight constraints, the movement of a chess king was added as a 2D list. A chess king can move 1 cell in any direction around itself, including vertically, horizontally, and diagonally. This again gave a total of eight spots where a number could not repeat. From there, each cell on the board was iterated through and the Anti-King constraints were applied to it. Cells near the edge of the board were accounted for in the exact same way they were for the Anti-Knight constraints.

2.3 Board Generation Algorithm

To find the least number of hints on a given Sudoku puzzle, a way was needed to randomly generate Sudoku puzzles. It was important that the generated board are random, as otherwise the board generation algorithm would create the same board every time. This alone would not be useful, as in order to compare the effect of the constraints, a wider set of data was needed.

To create random boards, the first step was to pass the indices of the board through Python's shuffle feature, which reorganized the indices of the board in a random order. The indices of the board were then iterated through, and for each cell that was iterated through, the following steps were taken:

1. Attempted to place a number variable, starting at 1, at the cell's location in the original board's list

2. Passed the given value of the number through Z3 to check if it violated the Sudoku/Variant constraints

   (a) If Z3 returned unsatisfiable, the number violated the Sudoku constraints. The next number from 1-9 was then passed through, following steps 1 and 2

3. Process was repeated until Z3 returned satisfiable, which meant that the number was in a valid spot. The number that returned satisfiable was placed at that cell

By placing the numbers in each cell, a board was quickly generated. This method shared many similarities with backtracking algorithms. The difference between the two is that while backtracking works recursively, the algorithm utilized here does not. This is because of Z3. While a backtracking algorithm may place

numbers, it would eventually run into a spot where it cannot place a number. At this point, the algorithm would move back one or more cells, until it found a solution that satisfies the cell. By using Z3, a model is being generated to check the validity of a number placement. This ensures that putting a number at a point does not mean that previous cells need to be changed to generate the board.

2.4 Hint Minimizing Algorithm

To find the least number of hints, an algorithm was created to determine and remove non-essential hints. The first step to doing this was saving the randomly generated board. The output of Z3's model() command was the finished Sudoku board. To best understand the algorithm, there are 3 things to notice:

1. C1: a copy of the output of Z3's model command. It is a 2D list used to keep track of the original, fully solved Sudoku board
2. C2: a second copy of the output of Z3's model command. It is a 2D list used to keep track of the removed spaces in the puzzle, or the spaces where hints were able to be taken out
3. Grid: the board that will be passed into Z3 to be solved

It is important to take out hints at random from the board. This is because as more hints are taken out, the harder it is to find a hint that is non-essential. If you were to iterate through the board normally, hints could be easily taken out of the top rows, but it would be much harder to take them out of the bottom rows. This would result in a board that had no hints at the top, but many hints at the bottom. Similar to how random board generation was done, the hints were also taken out at random to create a more even board.

In order to take hints out of the board, they had to be non-essential. To determine if a randomly selected hint is non-essential, the board C2 was used. One by one, the elements of C2 would be set to None. The value of None would mark a space where a hint was taken out of the board. The Grid variable would then be used to assert that every value in C2, except for any occurrences of None, is the same in Grid. This is important, since Z3 would be solving for the spaces where no value is defined. Since the rest of the board is the same as the values in C2, the only spaces Z3 would solve for are the places where hints were taken out.

If a hint was essential, taking it out would mean another puzzle could possibly be made. It is possible to use Z3 to check for this. For every hint taken out of the board, it was asserted that Z3 had to find another value for that cell that was different from what was there before. If the value was different than what it was before in the generated board, then other possible solutions existed. However, if Z3 could not find a solution where the value was different, then taking out the hint would not effect the final puzzle in any way. Therefore, if a different value could be found, the hint had to be kept, and if a different value could not be found then the hint was non-essential and removed.

Since Grid now contained the values of C2 without None values, Z3 could be called to solve for the hints. To ensure that other solutions were not possible, it was asserted that the solved value of Grid is not the same as the value of the cell in C1. If the value of Grid was different than what it was before in C1, then other possible solutions existed. However, if Z3 could not find a solution where the value was different, then taking out the hint would not effect the final puzzle in any way. Therefore, if a different value could be found, the hint had to be kept, and if a different value could not be found then the hint was non-essential and removed. If Z3 returned satisfiable, then the clue was essential and must be put back into the board. If Z3 returned unsatisfiable, then the clue was non-essential and could be taken out of the board. This process repeated for every cell on the board, until only the necessary hints were left.

It is important to note the use of Z3's push() and pop() functions. While checking if a hint was essential or non-essential, multiple assertions were made that should have only been true while checking for the hint. To ensure that these assertions are not permanent, push and pop can be used. The push command created a new stack of assertions, which was used to check the hints. Once the hints had been determined to be essential/non-essential, the pop command was used to clear the stack of assertions created between its corresponding push. The push was placed before the first set of assertions related to hint checking, and the pop was placed after the hint checking was over.

3 Results

To determine the effects of each variant, each program was run 50 times with each constraint. This meant that 50 random boards were created along the specifications of each of the three variants. From these 50 trials, information was gathered regarding the maximum, minimum, and mean number of hints remaining on the board. From each of these trials, the mean was compared to the control group of normal Sudoku, which was also run 50 times. The control group was used as a baseline to measure the effectiveness of the algorithm in reducing the number of hints

3.1 Control Group

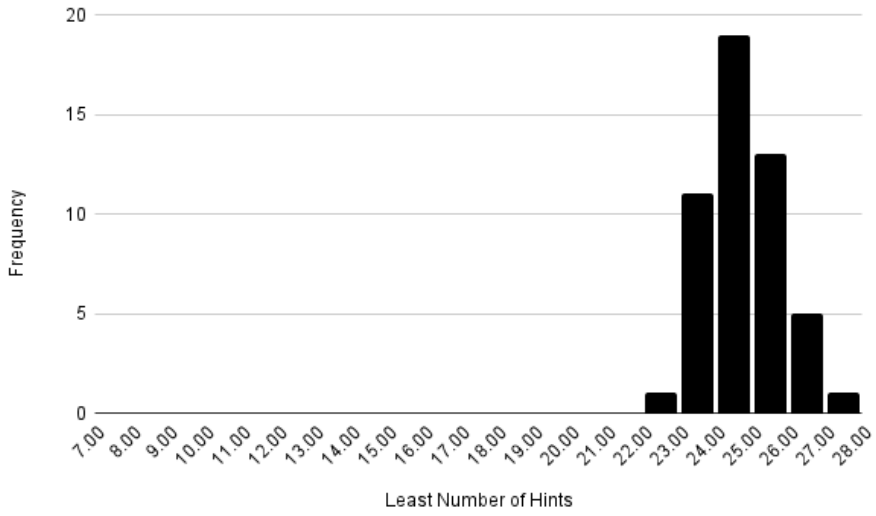
From the 50 trials that were run, the minimum number of hints was determined to be 22. When comparing to the accepted minimum number of hints, 17, we can see that the program was not entirely effective in finding the absolute minimum number of hints. This is because the method used in this paper is not an exhaustive proof of the minimum number of hints on each board. As seen in the Related Works section, the absolute minimum number of hints was not a necessary part of this paper. The method utilized in this paper created randomized Sudoku puzzles, and found the least number of hints on the generated board. The method used means that finding the absolute minimum number of hints would be a matter of luck, since the board generated would

need to be a 17 hint board. However, for the purposes of analyzing the different constraints and their effects on the minimum number of hints, the method used in this paper worked perfectly.

For the other trials, the control group acted as a precedent. The minimum number found from the 50 trial set of each variant should not be taken as a definitive absolute minimum. Although the minimum found could be the absolute minimum, this paper will not prove that, rather it will use it as a statistic to determine the effectiveness of a constraint.

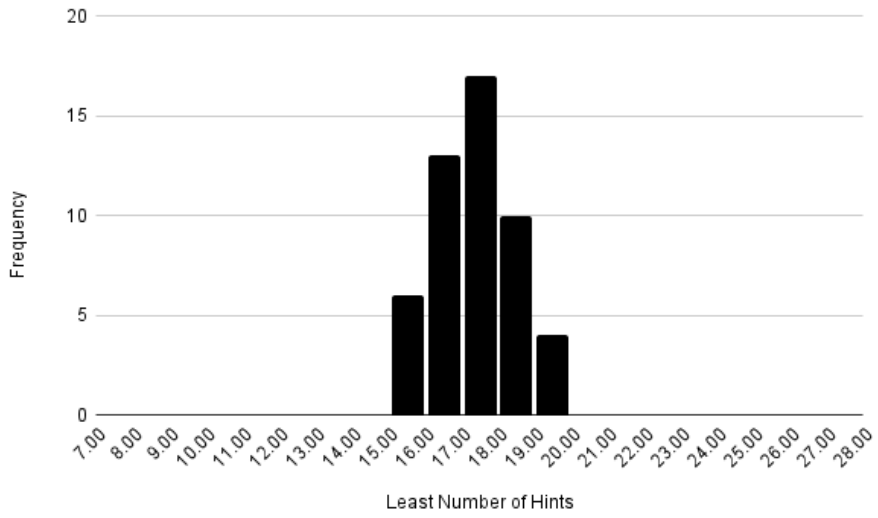
Below, the table displays the collected data from each of the 50 trials. The figure shows the distribution of the minimum number of hints for the 50 trials run with the Control group constraints

	Control	Hyper	Anti-King	Anti-Knight
mean	24.26	16.86	19.26	13.34
median	24.26	16.86	19.26	13.34
minimum	22	15	17	12
maximum	27	19	21	16



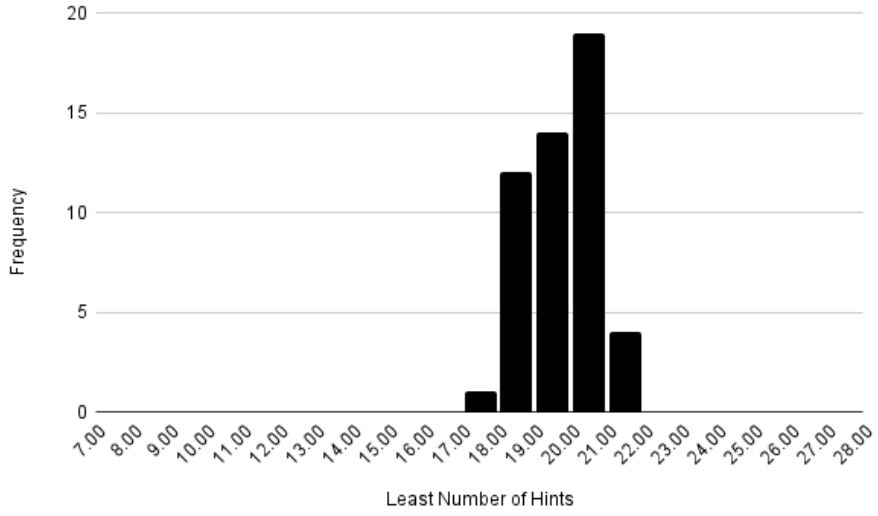
3.2 Hypersudoku

From the 50 trials that were run with the Hypersudoku constraints, the below histogram was created to see the distribution of the results. When comparing these to the control data, it is clear that the Hypersudoku constraints were effective in reducing the number of hints needed. The differences between the means of the two data sets is 7.4, meaning that on average the Hypersudoku constraint reduced the number of hints from the base puzzle by 7.4 hints.



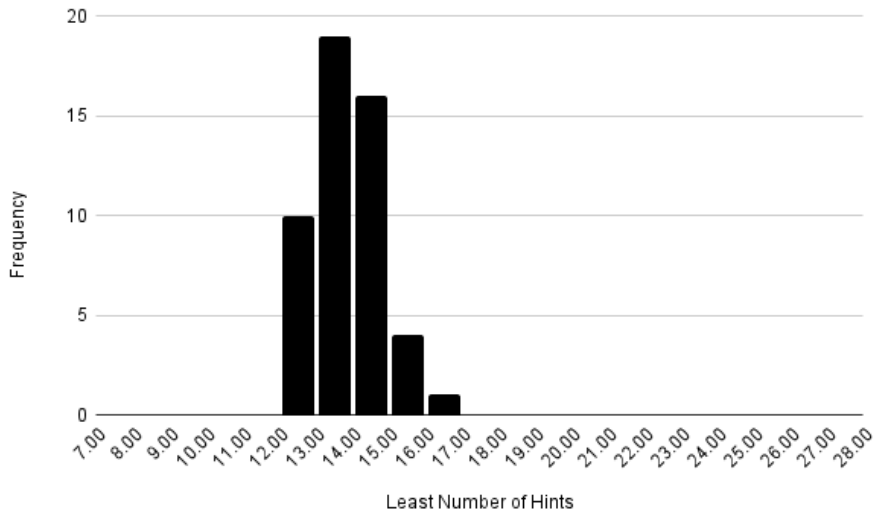
3.3 Anti-King Sudoku

From the 50 trials that were run with the Anti-King constraints, the below histogram was created to see the distribution of the results. When comparing these to the control data, we can see that the Anti-King constraints were effective in reducing the number of hints needed, though not as much as the Hypersudoku constraint. The differences between the means of the control and the Anti-King data set is 5, meaning that on average the Anti-King constraint reduced the number of hints from the base puzzle by 5 hints. However, the Hypersudoku constraint was able to reduce the number of hints on average by 2.4 more than Anti-King constraint, indicating that the Hypersudoku constraint was more effective.



3.4 Anti-Knight Sudoku

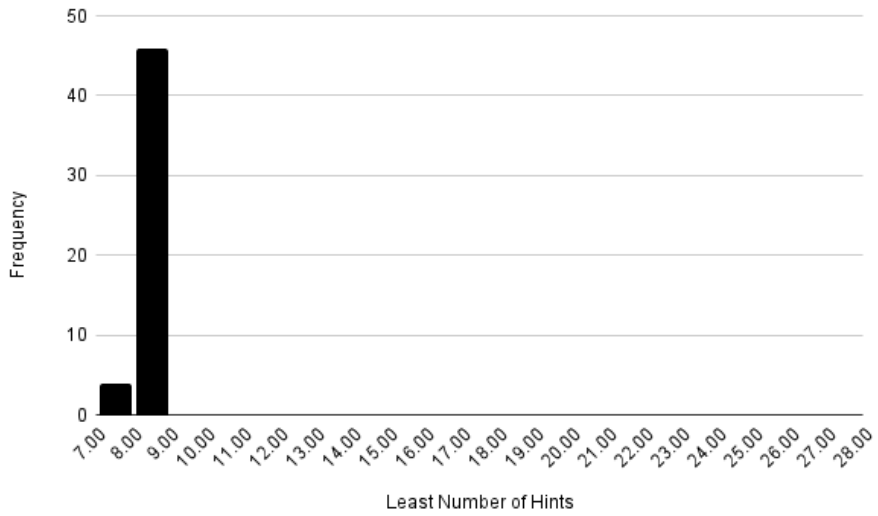
From the 50 trials that were run with the Anti-Knight constraints, the below histogram was created to see the distribution of the results. When comparing these to the control data, it is clear that the Anti-Knight constraints were extremely effective in reducing the number of hints needed, more so than any other constraint. The differences between the means of the control and the Anti-Knight data set is 10.92, meaning that on average the Anti-King constraint reduced the number of hints from the base puzzle by 10.92 hints. This reduction from the Anti-Knight constraint is, on average, 5.92 hints more than the Anti-King constraint and 3.52 hints more than the Hypersudoku constraint. This means that the Anti-Knight constraint was by far the most effective constraint in reducing the number of hints.



3.5 Anti-Knight and Anti-King Constraints

Following the trials run on the variants, an additional 50 trials were run on the board generated with both the Anti-Knight and Anti-Sudoku boards. These proved to be very effective in reducing the number of hints. Below please see the data table with the results of the 50 trials, and a histogram to better understand the distribution of the data. With both the Anti-Knight and Anti-King constraints, the algorithm was able to find a board with eight hints, the absolute minimum number of hints possible on a Sudoku board. To understand this better, remember that there are 9 possible numbers on a Sudoku board. Assume that the hints placed on the board are 1-9, meaning that there are a total of 9 hints on the board. Each of these hints mark where other hints of the same number will be. For instance, if one of the nine hints was 1, then that hint would set the position of all nine other 1s on the board. Therefore, it is reasonable to take one of the 9 initial hints away. If only one hint is taken away, 8 hints remain, and if each of those 8 hints decide the position of the rest of their numbers, then the only remaining spot is for the ninth hint. This means that 8 hint boards are possible, and can be found as shown by the solver.

	Anti-King + Anti-Knight
mean	7.92
median	8
minimum	7
maximum	8



3.6 Problems with Z3

The Anti-King and Anti-Knight combined constraint trials also exposed a problem in Z3. From the 50 trials, a 7 hint board was found by Z3 four times. Anything below 8 hints is not possible without special constraints. Consider the 8 hint board situation from the earlier section. If another hint is taken away, the remaining number of hints is 7. Those 7 hints would decide the position of all their numbers, leaving 18 positions open, 9 for each of the two remaining number. In these 18 positions, there are 2 configurations that the two numbers can go. The first is the original, and the second is where all nine cells of each number are switched with the position of the other number's cells. The two numbers can always be switched, since there is no constraint preventing them from doing so. Therefore, the board would not be unique, indicating 7 hint boards are not possible.

The fact that the algorithm found a 7 hint board indicated a problem within Z3. The code of the situation was rigorously checked. When the 7 hint board's layout was reproduced and reran through the hint minimizing algorithm, the minimum number of hints was found to be 8, rather than 7. This showed that the issue was not a reproducible problem with Z3, and rather it was problem within Z3's own logic. The error in Z3's logic casts doubt over the results of the paper. Since 7 hint boards should not be possible, it is feasible that Z3 made mistakes in the minimization process of the other variants.

3.7 Relationship between the Minimum Number of Hints and Difficulty

Sudoku puzzles are graded on difficulty based on the techniques it would take to solve the puzzle. The techniques range from reducing the number of possibilities to a single answer as the easiest technique, to trial-and-error being the hardest technique. Trial-and-Error is the definitive hardest technique, as it is not feasible for humans to attempt. Regarding puzzles at their minimum number of hints, the most common technique to solve would be trial-and-error, until other possible logical techniques become available. As such, it stands to reason that the lower the number of hints, the more the trial-and-error technique would have to be used to solve the puzzle. Ranked by their ability to create a difficult puzzle using this metric, the most effective variants in order are Anti-King and Anti-Knight combined, Anti-Knight, Hypersudoku, and Anti-King.

4 Conclusion

In this paper, the SMT solver Z3 was used in conjunction with Python to determine the effectiveness of Sudoku variants. Using Python and Z3, random boards were generated and then in each of these boards, the minimum number of hints was found. Then, the minimum number of hints per constraint was generalized by calculating the mean of 50 trials on each board. From this, the Anti-Knight

constraint was determined to be the most effective in reducing hints, followed by Hypersudoku and Anti-King Sudoku. Additionally, the absolute minimum number of hints in a Sudoku was found by combining the Anti-Knight and Anti-King constraints, and finding the minimum number of hints on the board.

In the future, two different approaches can be taken to refine the contents of this paper. First, the algorithms and techniques applied in this paper can be used in conjunction with a solver other than Z3. While this may reduce speed, it could improve the accuracy of the model considering the issues this paper faced with Z3. Secondly, this paper could find the absolute minimum number of hints through the same exhaustive method as Gary McGuire. While this would be time intensive, it would eliminate the need for statistics in the comparison of variants.

5 Bibliography

McGuire, G. (2013). There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration (thesis).