**ZIGUZAGU**

**-Squiggly Sudoku**

***A project report submitted to***

***MALLA REDDY UNIVERSITY***

***in partial fulfillment of the requirements for the award of degree of***

**BACHELOR OF TECHNOLGY**

**in**

**COMPUTER SCIENCE & ENGINEERING (AI & ML)**

**Submitted by**

**M.SANDEEP: 2111CS020475**

**M.SANJANA : 2111CS020476**

**A.SANJANA: 2111CS020477**

**B.SANJANA : 2111CS020478**

**P.SANJANA : 2111CS020479**

**U.SANJANA : 2111CS020480**

***Under the Guidance of***

**Dr. G. Hariharan**

**Professor**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING (AI & ML)**



2023

**CONTENTS**

CHAPTER NO. TITLE PAGE NO.

Abstract

1. INTRODUCTION
2. Literature Review
3. Problem Statement
4. Methodology
5. Experimental Results
6. Conclusion
7. Future Work

.

REFERENCES:

1. Author name, Title of paper/books with page numbers, publisher’s name, year of publication
2. Full URL address.

ANNEXUREI : Code of the project(Line spacing 1)

ANNEXUREII : Any other material (Optional)

**ABSTRACT**

In today's world of hectic life people tend to use more phones and lose the ability of basic logical reasoning and do not do much brain related exercise due to which our project Sudoku Game is very important as it helps in logical reasoning and brain development. Nowadays Sudoku puzzles are becoming increasingly popular among people all around the world. The Sudoku game is now very famous again and therefore many developers have tried to generate even more complicated and more interesting puzzles.

Hence the purpose of this project is to create a Sudoku game that would help the user to solve their puzzle in certain time and also would provide hints if the user could not crack it. In this report, we present the detailed development and implementation of a simple Sudoku game. The Sudoku game consists of a graphical user interface, and a puzzle solver; implemented using python. The solver finds the solution to the puzzles entered by the user. This project gives an insight into the different aspects of python programming.

**INTRODUCTION**

To complete this puzzle requires the puzzler to fill every empty cell with an integer between 1 and 9 in such a way that every number from 1 up to 9 appears once in every row, every column and every one of the small 3 by 3 boxes highlighted with thick borders.

Sudoku puzzles vary widely in difficulty. Determining the hardness of Sudoku puzzles is a challenging research problem for computational scientists. Harder puzzles typically have fewer prescribed symbols. However, the number of prescribed cells is not alone responsible for the difficulty of a puzzle and it is not wellunderstood what makes a particular Sudoku puzzle hard, either for a human or for an algorithm to solve.

By Sudoku puzzle of box size n, in this paper, is meant a partial assignment of values from {1,...,n 2} to the cells of an n 2 × n 2 grid in such a way that at most one of each symbols occurs in any row, column or box. A solution of a Sudoku puzzle is a complete assignment to the cells, satisfying the same conditions on row, columns and boxes, which extends the original partial assignment.

With sudoku.py, the process of building models of Sudoku puzzles, which can then be solved using algorithms for computing solutions of the models, is a simple matter. In order to understand how to build the models, first it is necessary to explain the two different representations of Sudoku puzzles in sudoku.py. The dictionary representation of a puzzle is a mapping between cell labels and cell values. Cell values are integers in the range {1,...,n 2} and cell labels are integers in the range {1,...,n 4}. The labeling of a Sudoku puzzle of boxsize n starts with 1 in the top-left corner and moves along rows, continuing to the next row when a row is finished. So, the cell in row i and column j is labeled (i−1)n 2 + j. For example, the puzzle from the introduction can be represented by the dictionary

>>> d = {1: 2, 2: 5, 5: 3, 7: 9, 9: 1, 50

11: 1, 15: 4, 19: 4, 21: 7, 25: 2, ...

27: 8, 30: 5, 31: 2, 41: 9, 42: 8, ...

43: 1, 47: 4, 51: 3, 58: 3, 59: 6, ...

62: 7, 63: 2, 65: 7, 72: 3, 73: 9, ...

75: 3, 79: 6, 81: 4}

A Sudoku puzzle object can be built from such a dictionary. Note that the box size is a parameter of the Puzzle object constructor.

>>>for i in range(amount):

>>>y = random.randint(0,len(grid)-1)

>>>x = random.randint(0,len(grid)-1)

>>>num = random.randint(1,len(grid))

>>>allow = 0

>>>for e in range(len(grid)):

>>>if num not in grid[x] and num != grid[e][y]:

>>>allow +=1

>>>grid[x][y] = num

>>>tempo = grid

>>>tempo = rearrange(tempo)

>>>for e in range(len(grid)):

>>>if(duplicate\_checker(tempo[e])):

>>>allow = 0

>>>if allow !=len(grid):

>>>grid[x][y] = 0

2 5 . . 3 . 9 . 1

. 1 . . . 4 . . . 4

. 7 . . . 2 . 8 . .

5 2 . . . . . . . . .

9 8 1 . . . 4 . . .

3 . . . . . . 3 6 . .

7 2 . 7 . . . . .. 3

9 . 3 . . . 6 . 4

Random puzzles can be created in sudoku.py by the random\_puzzle function.

>>> import random

>>> q =scramble()

>>> q

. . . . 5 . . . 1

. 5 . . . . . . 7

. . 1 9 . 7 . . .

. . . . . . . . .

. . 5 . . . 7 . .

. . 6 . . . . 9 .

. . . . . 5 . . .

5 . . . . . 4 . .

1 . . . . . . . .

The first argument to srumble() is the number of prescribed cells in the puzzle. Solving of puzzles in sudoku.py is handled by the solve function. This function can use a variety of different algorithms, specified by an optional model keyword argument, to solve the puzzle. Possible values are CP for constraint propagation, lp for linear programming, graph to use a node coloring algorithm on a graph puzzle model and groebner to solve a polynomial system model via a Groebner basis algorithm. The default behavior is to use constraint propagation.

>>> from sudoku import solve

>>> s = solve(q)

>>> s

7 3 2 8 5 6 9 4 1

8 5 9 4 2 1 6 3 7

6 4 1 9 3 7 8 5 2

9 7 8 5 4 3 1 2 6

3 2 5 6 1 9 7 8 4

4 1 6 7 8 2 5 9 3

2 9 4 1 6 5 3 7 8

5 6 3 2 7 8 4 1 9

1 8 7 3 9 4 2 6 5

**Literature Review:**

D.H. Lehmer was the first to introduce the backtracking algorithm in 1950. The backtracking algorithm is one of the problem-solving methods included in a strategy based on searching the solution space, but it does not have to examine all possibilities, only those that lead to only solutions will be processed. Algorithms backtracking is also an algorithm that works recursively, where the search process is based on the Depth-First Search (DFS) algorithm, which is to search for systematic solutions to all possible solutions and search for answers is done by tracing a tree-shaped structure rooted [2]. Therefore this algorithm is quite powerful and very good to be applied in problem-solving and to provide artificial intelligence in the game. Several types of digital games that are generally commonly known by the public, such as Chess, Math Maze, Tic Tac Toe, to Sudoku can be found a solution by implementing the backtracking algorithm.

The backtracking algorithm is an improvement of the brute force algorithm, which is to find solutions to problems among all possible solutions systematically. Backtracking is a typical form of recursive algorithm and is based on DFS (Depth-First Search) in finding the right answer. In another sense, the backtracking algorithm works like experimenting with several possibilities that lead to the solution until it finds the most appropriate one. So there is no need to check all possible solutions, but it is enough that only leads to the solution, namely by sorting pruning the nodes that do not lead to the solution. Thus the search time can be saved. The difference with the brute force algorithm is the basic concept, namely, in backtracking, all solutions are made in the form of a solution tree (tree), and then the tree will be traced in DFS (Depth-First Search) to find the best-desired solution.

Related Works

The previous studies related to research conducted by the author include: Llyod (2019) Conduct research published by IEEE in international journals entitled “Solving

* Sudoku with Ant Colony Optimization.” Says that sudoku game is a famous puzzle game that is very computationally challenging, so it requires the most powerful and most sophisticated form of algorithm to solve it.[2]
* Ghosh (2017) International journals published by IEEE did the research entitled “A SimpleRecursive Backtracking Algorithm for Knight’s Tours Puzzle on Standard 8×8 Chessboard” says the backtracking algorithm is not entirely the best solution in the case of a knight’s tour game because it executes too long, making it less efficient and practical.[3]
* Schottlender (2014) Conduct research published by IEEE in international journals entitled “The Effect of Guess Choices on the Efficiency of a Backtracking Algorithm in a Sudoku Solver.” Using the initial element of a randomly generated sudoku puzzle is better in the backtracking algorithm compared to using numbers that are already available as starting elements. [4]
* Szabó (2014) on International journals published by IEEE did the research entitled “Creation of the Chips Placement Game with Backtracking Method in Borland Pascal” the application of backtracking algorithm in the creation of the chips placement game is very efficient because the algorithm can calculate all combinations with a concise code
* Fermuller (2014) International journals published by IEEE did the research entitled “Semantic Games with Backtracking for Fuzzy Logics.” implement the backtracking algorithm in Hintikka’s classical game .
* Li et al., (2011) International journals published by IEEE did the research entitled “The Research on Departure Flight Sequencing Based on Improved Backtracking Algorithm.” The application of the backtracking algorithm in scheduling has worked optimally and produced a way that will be and as expected.

**Problem Statement:**

This section provides a clear and concise statement of the problem, this should include a description of the data used in the project and the research questions and hypotheses that guided the project. This problem statement should clearly identify the problem that the research paper is trying to solve and how it will be addressed in the project.

**Methodology:**

**Implementing solving algorithms:**

In order for there to be no difference other than the algorithms themselves, all algorithms were written in the same programming language(python) and all testing was done on the same device.

To get as similar conditions as possible, all algorithms were written as, which also had all the test cases defined. The test cases, after being constructed, were then sent to each of the solving algorithms in turn.

>>>def solver():

>>>global grid,done

>>>if(done ==False):

>>>for y in range(9):

>>>for x in range(9):

>>>if grid[y][x] == 0:

>>>for num in range(1,10):

>>>if stay(num,x,y):

>>>grid[y][x]=num

>>>solver()

>>>grid[y][x]=0

>>>return

>>>done =True

>>> for a in entry\_list:

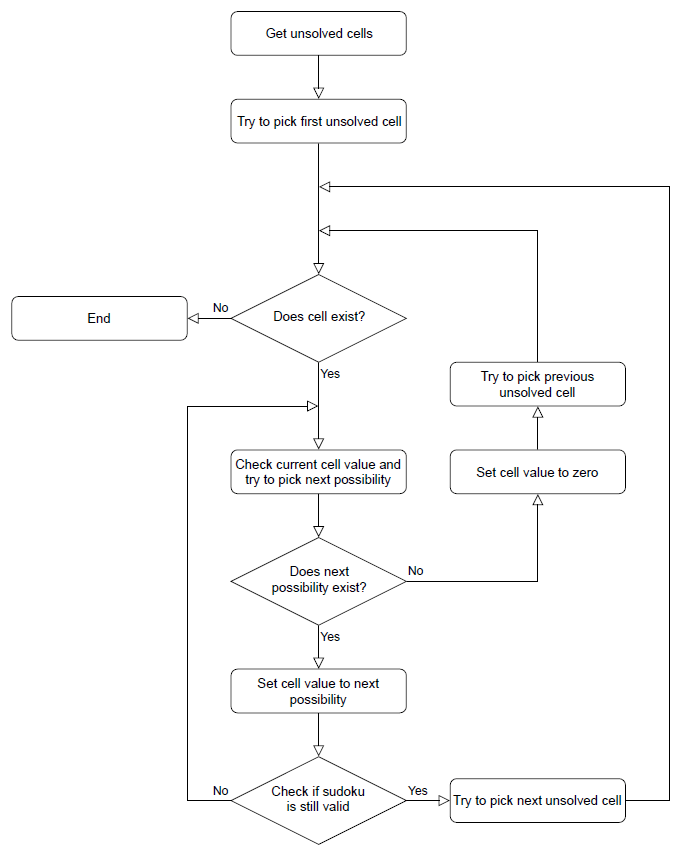
>>>for b in a:

>>>b.delete(first=0,last=100)

>>>display\_val()

>>>return

The solve method uses backtracking algorithm.



To ensure as little bias as possible, test cases were also divided into tiers depending on difficulty, and then these tiers were solved both individually and together.

**Implementing test cases :**

In order to ensure the quality of the test cases be sufficient, the sudoku puzzles used are taken from the book “A to Z of sudoku” by Narendra Jussien. The book divides puzzles into six categories based on difficulty but here we took only three categories. These categories can be found in the result section, named “Easy”, “Medium” and “Hard”.

Creating Sudoku puzzle:

>>>def scramble():

>>>global grid

>>>clear()

>>>for a in entry\_list:

>>>for b in a:

>>>b.delete(first=0,last=100)

>>>amount = 20

>>>for i in range(amount):

>>>y = random.randint(0,len(grid)-1)

>>>x = random.randint(0,len(grid)-1)

>>>num = random.randint(1,len(grid))

>>>allow = 0

>>>for e in range(len(grid)):

>>>if num not in grid[x] and num != grid[e][y]:

>>>allow +=1

>>>grid[x][y] = num

>>>tempo = grid

>>>tempo = rearrange(tempo)

>>>for e in range(len(grid)):

>>>if(duplicate\_checker(tempo[e])):

>>>allow = 0

>>>if allow !=len(grid):

>>>grid[x][y] = 0

>>>display\_val()

For different level we change the amount which is mentioned in the above algorithm.

These levels of difficulty are based on the number of different sudoku solving techniques necessary to solve them. The very easy puzzles only need one or two of the most basic solving techniques, whereas the Hard puzzles require every available solving method.

Ten puzzles from each category were used as test cases, and to make the numbers easier to observe, each test case was run 100.000 times, totalling one million solved puzzles per level of difficulty.

**Comparing the results :**

The results of the tests were compared primarily on how quickly the algorithms solved all the test cases. They were also reviewed to ascertain whether any puzzles were unsolvable for any of the algorithms.

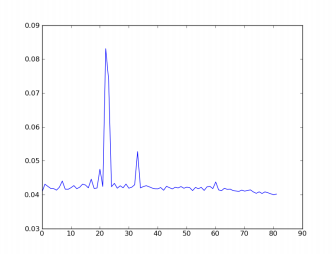
With the backtracking algorithm, the second factor was redundant as it tries every possible solution. If this algorithm returns unsolved test cases, it means that it was either incorrectly implemented, or the test cases themselves were unsolvable.

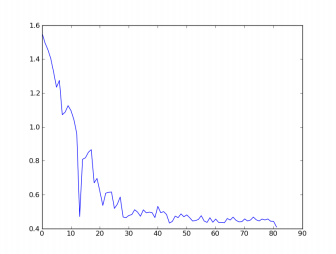
The results were also given in seven different versions. First all of the test cases together, then the results for the testing the different levels of difficulty for the unsolved puzzles.

These results were reviewed individually as well as grouped together, to see if there was any difference between the different results.

**Experimental Results:**

We introduced the random\_puzzle function in the introduction. The method by which this function produces a random puzzle is fairly simple. A completed Sudoku puzzle is first generated by solving the empty puzzle via constraint propagation and then from this completed puzzle the appropriate number of clues is removed.

An interesting problem is to investigate the behavior of different models on random puzzles. A simple script, available in the investigations folder of the source code, has been written to time thesolution of models of random puzzles and plot the timings via matplotlib. 



Two plots produced by this script highlight the different behavior of the constraint model and the integer programming model.

The first plot has time on the vertical axis and the number of clues on the horizontal axis. From this plot it seems that the constraint propogation algorithm finds puzzles with many or few clues easy. The difficult problems for the constraint solver appear to be clustered in the range of 20 to 35 clues.

A different picture emerges with the linear programming model. With the same set of randomly generated puzzles it appears that the more clues the faster the solver finds a solution.

**System Testing:**

System testing is done by running the Sudoku program using the Android platform. Several functions can be run on the system, including generating puzzles, checking the puzzle, and running the backtracking algorithm. Implementation of the operation carried out at Smartphone android with the following specifications:

* Model ASUS T00F API 19.
* CPU Quad-core 1.2 GHz Cortex-A7
* Memory RAM 2048 MB

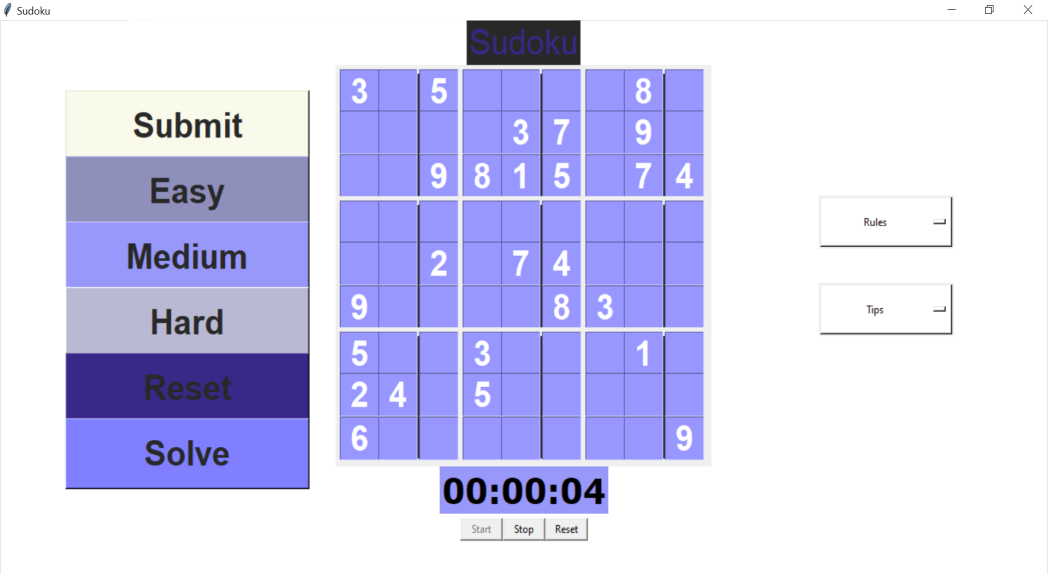
****

Figure 1:the Sudoku generated is of Easy level

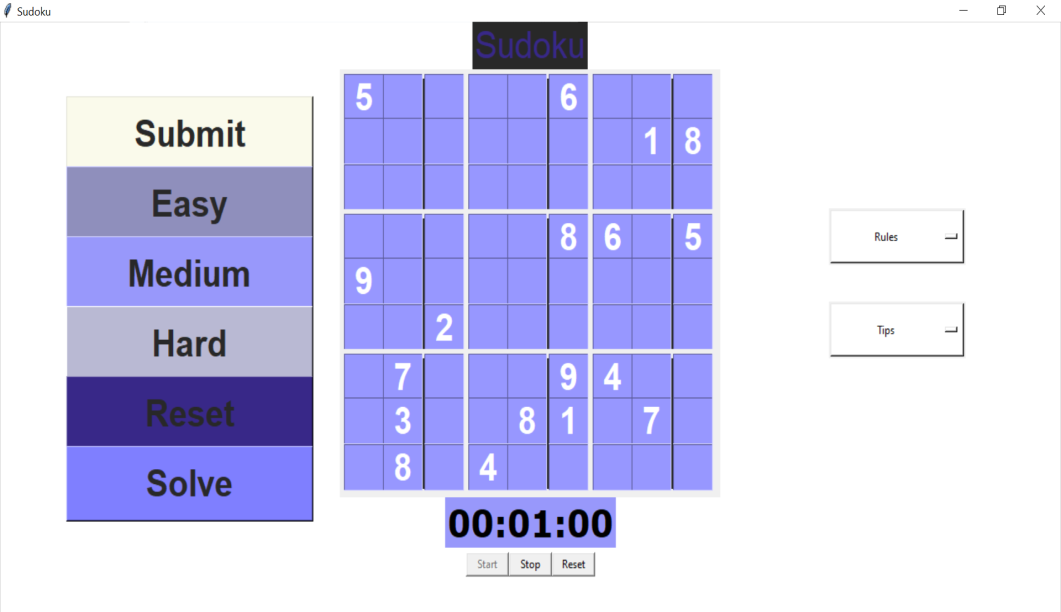


Figure 2:the Sudoku generated is of Medium level

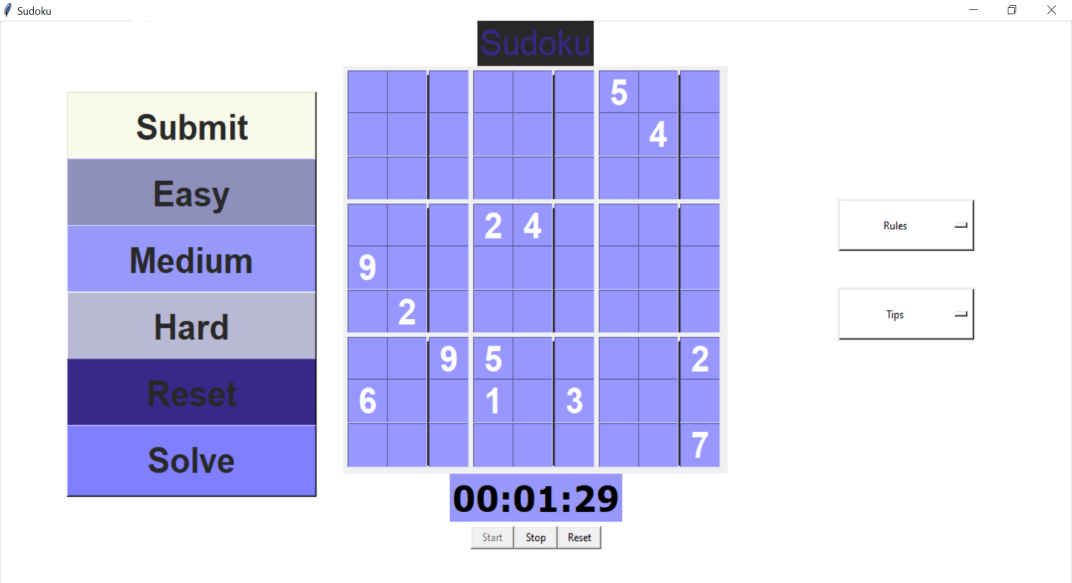
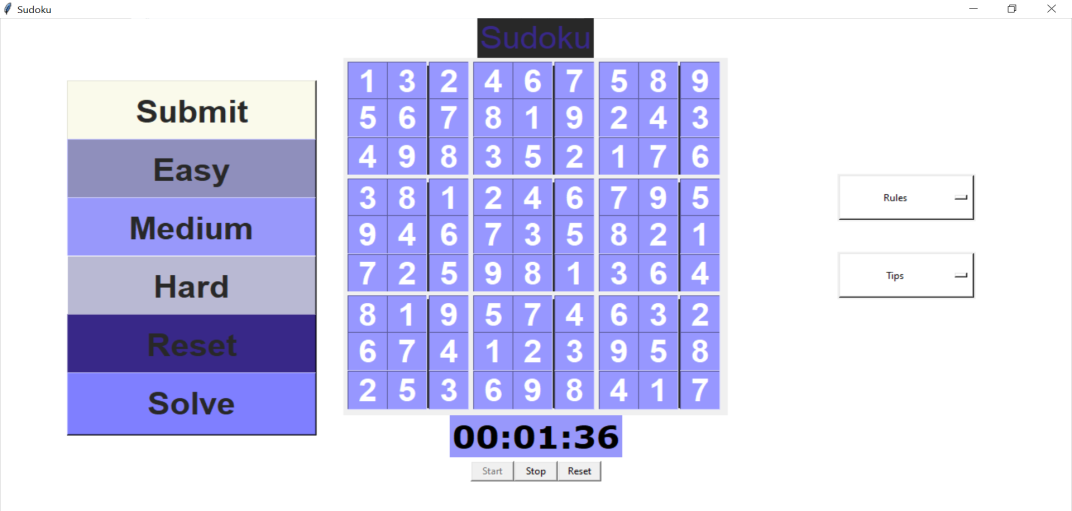


Figure 3:the Sudoku generated is of Hard level



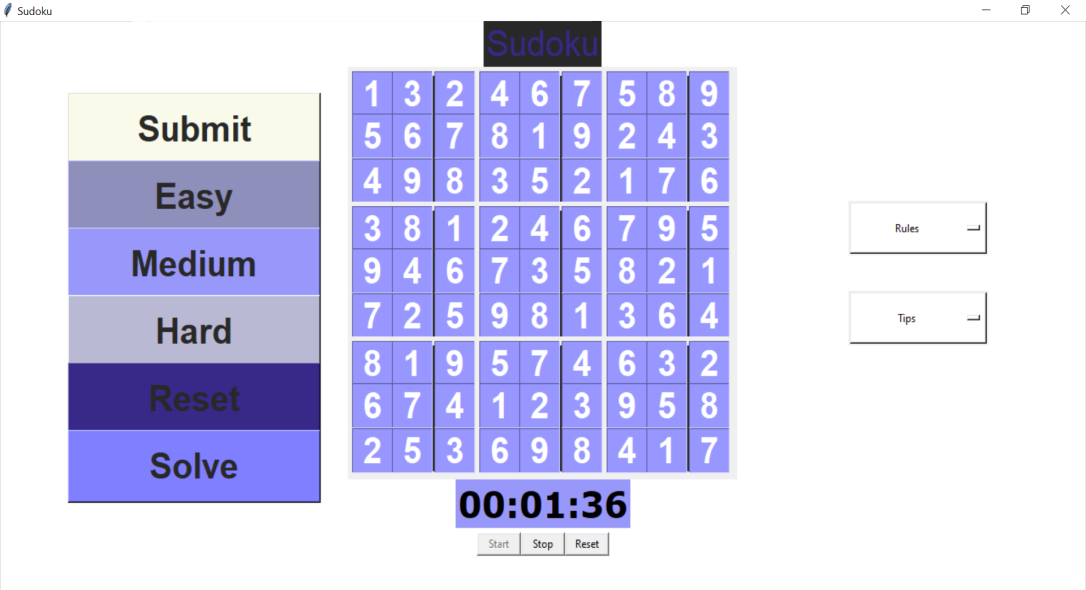


Figure 4,5 and 6: Backtracking Algorithm and Solve Puzzle With Manually

**Conclusion:**

When comparing sudoku solving algorithms written in Java, the backtracking algorithm has been proven to be superior to both the constraint algorithm as well as the rule based algorithm.

The constraint solution was comparably inefficient to such a degree that it should never be considered other than for studying constraint programming in Java.

The rule based algorithm was slower than backtracking even when only counting the initiation and memory allocation, and as such can never be improved to be faster than the backtracking solution by adding more rules. This of course assuming the program is written in Java.

It’s possible there are ways to improve the memory management to such an extent it can outperform the backtracking algorithm, but we consider it unlikely. Since the backtracking algorithm is so efficient as well as easy to implement, we consider finding such a solution unnecessary.

The conclusion we have reached is that the backtracking algorithm is the best method for computationally solving sudoku with the Java programming language.

**Future Work:**

Future plans for sudoku.py are to increase the variety of models. Both by allowing for greater customization of currently implemented models and by implementing new models. For example, we can imagine several different Sudoku models as constraint programs beyond the model presented here. Another approach is to model Sudoku puzzles as exact cover problems and investigate the effectiveness of Knuth’s dancing links algorithm. Also important to us is to compare all our models with models [Lyn06] from satisfiability theory. In [Kul10] a general scheme is presented which is highly effective for modeling Sudoku.

There are great many interesting, unsolved scientific problems involing Sudoku puzzles. Our hope is that sudoku.py can become a useful tool for scientists who work on these problems

**References:**

[Bar08]] A. Bartlett, T. Chartier, A. Langville, T. Rankin. An Integer Programming Model for the Sudoku Problem, J. Online Math. & Its Appl., 8(May 2008), May 2008

[Bre79] Brelaz, D., New methods to color the vertices of a graph, Communications of the Assoc. of Comput. Machinery 22 (1979), 251-256.

[Fel06] B. Felgenhauer, F. Jarvis. Enumerating possible Sudoku grids Online resource 2006 <http://www.afjarvis.staff.shef.ac.uk/sudoku/>

[Kul10] O. Kullmann, Green-Tao numbers and SAT in LNCS (Springer), "Theory and Applications of Satisfiability Testing - SAT 2010", editors O. Strichman and S. Szeider

[Lew05] R. Lewis. Metaheuristics can solve Sudoku puzzles, Journal of Heuristics (2007) 13: 387-401

[Lyn06] Lynce, I. and Ouaknine. Sudoku as a SAT problem, Proceedings of the 9th Symposium on Artificial Intelligence and Mathematics, 2006.

[Sim05] H. Simonis. Sudoku as a Constraint Problem, Proceedings of the 4th International Workshop on Modelling and Reformuulating Constraint Satisfaction Problems. pp.13-27 (2005)

[Var05] J. Gago-Vargas, I. Hartillo-Hermosa, J. Martin-Morales, J. M. UchaEnriquez, Sudokus and Groebner Bases: not only a Divertimento, In: Lecture Notes in Computer Science, vol. 4194. pp. 155-165. 200