Fun Sudoku Solver

Dean G. Huang

***Abstract*—This project dives into the sudoku solving realm, with the sole purpose of eliminating sudoku headaches. The approach is to write a python script that will solve any sudoku. After one quarter of effort, my program has been battle-hardened to solve almost all solvable sudoku it encounters.**

***Index Terms*—Algorithms, Combinatorics, Graph theory, Group Theory, Programming**

I. Introduction

A. History

To keep it succinct, Sudoku's history started due to human's curiosity toward grid permutations. Unrepeating patterns in rows, columns, and boxes. For example, the first documented puzzle of this strain, the Magic Square, dates to China two thousand years ago [1]. It was conjured by filling a 3x3 grid with numbers 1 through 9; the sum of the rows, columns, and diagonals must add up to the same number. For example, this magic square adds up to 15 in every contract:

4	9	2
3	5	7
8	1	6

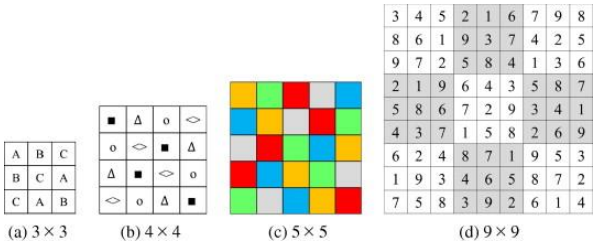
Magic Squares [1]
https://www.sudokudragon.com/sudokuhistory.htm

The venerable Swiss Genius, Leonhard Euler, is later credited for devising the "Greek-Roman Squares" in the 18th century:

A α	B γ	C δ	D β
B β	A δ	D γ	C α
C γ	D α	A β	B δ
D δ	C β	B α	A γ

Graeco-Roman Squares [1]
https://www.sudokudragon.com/sudokuhistory.htm

This puzzle is a 4x4 grid filled by the 16 members of the 2 sets of 4 combination, where the members with similar attributes can not "see" each other. This puzzle later came to be loosely named the "Latin Squares" with a single variable, constructed by filling any NxN grid with random non-repeating symbols, such as:



Latin Squares Varying Complexity [2]
https://www.sciencedirect.com/science/article/abs/pii/S0020025513008359

That concludes the logic aspect of the Sudoku history. The trivia history begins with the official rules being set in stone by Howard Garns in 1979, where each 3x3 sub-grid also had a contract; he published the puzzle in *American Dell Magazine*, named "Number Place". The Japanese media soon adopted the puzzle and called it "Su-Doku", which literally translates to "number alone" [1].

B. Global Constraints

My program can solve any solvable sudoku, but what is a solvable sudoku? This question is mystical; like how prime numbers are unique since no other numbers could divide them whole, Sudoku solutions are unique depending on the number of clues provided.

So how is a puzzle created? Imagine a number line, from 0 to 81, attributed as clues given. If 0 clues were given (blank grid) the puzzle would be easy to solve since you could solve it any way you want (trillions of solutions). If 81 clues were given, the puzzle would be easy since its already solved (one solution). To create a puzzle, its about balancing in between 0 and 81 clues while ensuring only one unique solution.

Nowadays, computer programs simply create puzzles by reducing the solved units in a solution grid. It keeps omitting clues until the difficulty level is satisfied.

To further dive into this question of how many clues, we must first understand the relation between solvable and unique. First, any sudoku must be solvable, or else it would violate the very principles of the Latin Squares. The moment a contradiction happens, it signals an error has been made. This should settle the idea that all published solvable sudokus are solvable.

"This work was supported in part by the UCLA Samueli School of Engineering."

D. G. Huang is with the University of California Los Angeles, CA 90024 USA (e-mail: deanthebean@ucla.edu).

Now comes the unique solution. Mathematicians and programmers used statistics and exhaustive brute force computation to confirm that 17-clue puzzle is indeed the bare minimum. They performed this with the reductionist approach, which involves inspecting a solved grid and providing multiple instances of 16 clue grids. The team then checked the 16-clue grid instance to see if its solution is the same as the original grid. If the same grid is the solution, then an 16-clue grid with an unique solution exists. The team spent seven years on this problem without ever discovering a single 16-clue grid [3].

In conclusion, no puzzles can be less than 17-clue. After it reaches the 17-clue level, only a few exists. As you increase the number of clues given, the number of unique solved patterns decrease. The 23-clue level is the theoretical level where the puzzle must have an unique solution. After this level, the puzzle will be guaranteed an unique solution. The popular Japanese publisher, for example, never provided more than 32 clues [3] in their puzzles.

II. Motivation

The reason for why I chose to develop a Sudoku Solver is because during my friend's visit, he brought out a Sudoku book and suggested solving one to pass time. I have solved Sudokus as a child, so I was intrigued. While solving it, I kept thinking to myself: "I wish I could just write a program to do all these logical checks for me." I pondered for days about my potential algorithm that would solve the sudokus. Sure enough, after visually scouring through the suggested senior design projects I stumbled upon the Sudoku Solver. I thought the universe was aligned, calling to me to write the code already.

The simple answer to the purpose of this project is to eliminate Sudoku headaches. Why rack your brain on a puzzle when you could let a program check everything for you?

In hindsight, I had no idea the difficulty of the puzzles would prove my initial strategies helpless. It was due to this failure that I had to develop a better, more robust Guess All solver.

III. Approach

A. Team Organization

This one is straightforward since I am the sole contributing member on the team. On a more abstract level, motivating the lazy half of myself to push past a bug problem was difficult here and there, but I would remember professional software programmers probably write this in a day so the lazy side of me has no grounds to complain.

B. Plan

In all honesty, I came into this project with lofty expectations. I imagined Sudoku could be solved by machine learning algorithms. It was after researching the subject that I realized Sudoku solving does not require machine learning. After that, my plan was to create a sudoku solver with the easiest tools available: python, and arrays. Since sudokus are simply 81 numbers on a grid, I thought I could tackle this problem by setting up the architecture of the sudoku as an 81 unit long array. I enjoyed the simplicity of a 1D array since its easy to access, pass through functions, and edit.

By solving the puzzle in the past, I knew the importance of suspected numbers list. My plan was to simply use the suspected number list to fill in all the squares. This turned out successful for easy to medium puzzles, not for difficult ones.

It was after the completion of the contract solvers did I begin the new plan of implementing the binary search tree to solve difficult sudokus. This was a reactionary plan, but a plan that succeeded nonetheless.

C. Standard

No standards available.

D. Theory

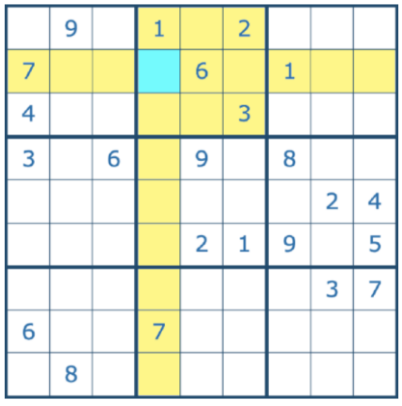
In this project, I utilized indexing algebra, exclusionary logic, and algorithms.

For indexing, it is helpful to not only switch from current unit to row number, but also vice versa. If I require all the numbers in row 4 for example, I would need to be able to index them correctly. The general formulas for converting current unit ("i" for index) to the column, row, or box identification:

```
col_ID = (i%9) + 1
row_ID = int(i/9) + 1
box_ID = int(i/27)*3 + (int(i/3)%3)+1
```

The only one that really required any math is box_ID since converting unit to box number is trickier than row or column. Therefore, I usually work on row functions first and box functions last. Also worth mentioning is that the contract IDs are indexed from 1 to 9 for quality of life.

For exclusionary logic, I had to work very closely with the contracts. Contract is the term I use to describe the possible numbers an unit could be, such as:

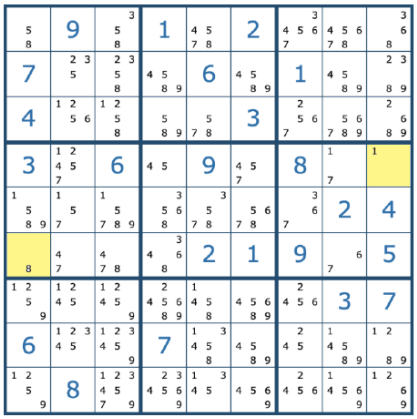


Blue Unit's Contract Space [5]

https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/

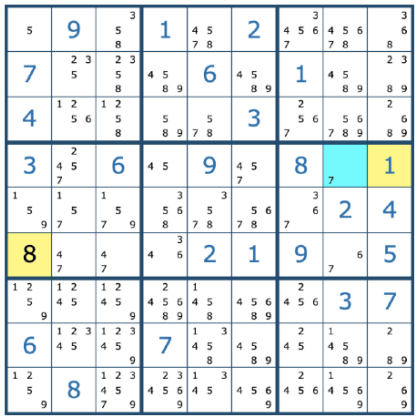
With contracts, I created two different types of solvers called single suspect filler and unique filler. The single suspect filler

will fill up the unit if there is only one suspect. Observe the yellow units below:



Suspect Fill Before [5]
https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/

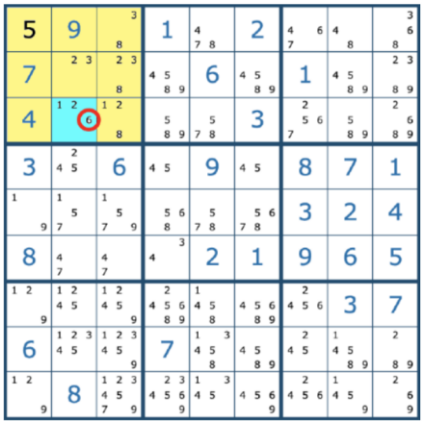
After discovering the unit only has one suspect, the function will update the matrix with the single suspect values:



Suspect Fill After [5]
https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/

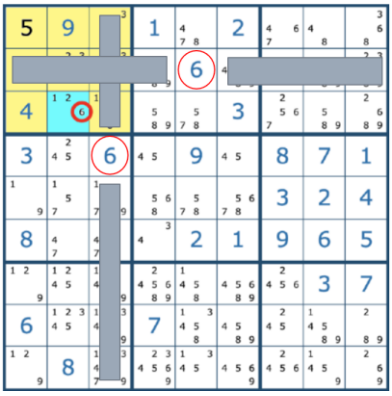
Thanks to the domino effect of updated units and decreasing suspect lists, the grid becomes easier to solve as shown by the new blue unit with a value of 7. In practice, the hardest part of sudoku is to find the first few hints. The grid becomes easier to solve as more clues are discovered.

After finishing the suspect fill, I realized there is a lot more potential to the contract solving methods. For example, simply observing the way the other 6 alienates the circled 6 into the blue unit in Box 1(colored yellow):



Unique Box Fill [5]
https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/

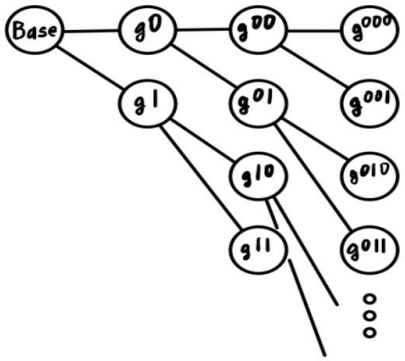
It is apparent that 6 is unique within the box 1 parameter since no other unit could be 6:



Unique Box Fill with Occlusion [5]
https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/

This visualization demonstrates why 6 belongs in the blue unit. Another helpful benefit unique contract provides is that even if only one parameter is true, it must be true for all three. For example: If box unique corners a 6 in that spot and row unique does not agree, box unique is still correct no matter what column or row observe. This means the unique fillers will cover each other's blind spots and fill the puzzle faster than before.

Finally, comes the binary search tree algorithm. This fundamental algorithm is heavily utilized in computer science and honestly parallels with my outlook towards how to deal with people in general. It works by splitting every node into two children:



Simple Depiction of BST

Although binary search tree algorithm is supposed to help reduce the search time in programs, my implementation of the search tree does not parse data by value. In other words, my binary search tree is still a linear one, since I am still required to try and fail every single simulation before the solved puzzle is discovered.

To create an algorithm, one must devise a start case and end case. The start is simple, it is when the first unit with two suspects is discovered. The end is when the current puzzle is stuck or successful.

The puzzle is stuck the moment a contradiction happens, or when the puzzle is invalid. A contradiction is when one parameter deems the unit as a certain number, while another parameter disagrees. For example, if there is only one empty unit in a row, then we know the number for sure; however, a contradiction happens when that number is already in the box that houses the unit. In short, two parameters disagree with the validity of a unit, so there is a contradiction between the parameters. The puzzle is invalid when any of the 27 parameters holds two of the same numbers, such as when a box has two 7s.

The puzzle is successful when the total sum of all the numbers is 405 and the grid is valid. If this is the case, the function will return the solved grid immediately to break out of the recursive depths.

This concludes all the theories I implemented while building the sudoku solver.

E. Software / Hardware

My project is purely on the theoretical side; the benefit that comes with this type of project is that no hardware, besides computer, is involved. For software, everything written is in python. The only library imported is the math library. I had to create my own matrix/grid interface to debug and modify the puzzles.

In terms of pure computer science knowledge, as mentioned above in theory, I employed indexing, array manipulation, abstract data structures, and algorithms to overcome the sudoku puzzles. Since my entire project is 914 lines of code (including large blocks of comments), I will refrain from sharing too much code in this section.

My Code Snippet will simply be the helper functions and Data Structures for easier reference.

Code Snippet:

```
### MATRIX HELPER FUNCTIONS
def mat_correct(raw):#Changes . to 0
def mat2grid(in_mat):#Changes 1D to grid
def mat_stuck(in_mat):#If stuck, return 1
def mat_print(in_mat):#Print Matrix

##### INDEX PRINTING STATION
#Print Index
def row_print(in_mat, num):
def col_print(in_mat, num):
def box_print(in_mat, num):

### MATRIX MEMBER GRABBER
#Grab the exact row we want from the matrix
def row_grab(in_mat, num):
def col_grab(in_mat, num):
def box_grab(in_mat, num):
def missing(in_mat): #returns missing numbers


### GRID CONDITION CHECKERS
   def grid_percent(in_mat):#return pop% of matrix
   def grid_filled(in_mat):#return 1 if filled

### DATA STRUCTURE UNIT
class Unit: #DATATYPE 1: Single Unit
    def __init__(self, value, list):
      self.value = value #unit has 1 value
      self.suslist = list #list of suspects

class Grid: #DATATYPE 2: Entire 9x9 Grid
    def __init__(self, data):
      self.data = data

### Grid Helpers
    def package(self):

### Grid Checkers
    def grid_valid(self):#Is Grid a valid Grid?
    def grid_solved(self): #Is Grid Solved?
    def g_sum(self): #Returns the Sum of the Grid
    def in_list(self, array, num):#Is # in list?

### Solvers Nest Here ###
### CONTRACT SOLVERS
# Suspect Solver
    def suspectlist (self):#fill, then patch
    def safesuspectlist(self):#safely fill
    def suspectlistfill(self):#fill single sus
# Unique Solvers
    def token2parent(self, total_array, tokens,
unique): #Unique helper function
    def row_unique(self):#FINDS UNIQUE SUS IN ROW
    def box_unique(self):#FINDS UNIQUE SUS IN BOX
    def col_unique(self):#FINDS UNIQUE SUS IN COLUMN
    def contract_solver(self): #Contract Solve
#Recursively Guess through the Binary Search Tree
    def guess_all(self, in_mat): #Guess Solve
```

F. System Build / Operation

1) High-Level Overview

The entire system works by first entering a raw sudoku puzzle. The program reads the 81 long array and attempts to contract solve it. If the contract solver is successful, the program is done. If contract solver gets stuck for more than 3 iterations, the function will return false and hand the puzzle down to the

guess all solvers Once guess all recursively solves the puzzle, the puzzle will be printed.

2) Lower-Level Description

To bring the description down to coding level, I created a small library of matrix/array helper functions to edit, reference, grab, and check the inputted array or puzzle. Some helper functions take in a whole puzzle (81 Units) or a simple array (9 digits).

The abstract data structures I created are called: Unit and Grid. Unit is simple, it only contains a value and the suspect list of that unit. Grid is simply an 81-long array of Units. The architecture of a Grid allows me to create Grid functions to call upon itself. There is not much within the Unit structure, but within the Grid structure contains the grid helpers and grid solvers.

3) Detailed Description

With the high and low level descriptions given, a super-detailed narrative following the grid as it embarks on the heroes journey could be found here:

A raw sudoku array is read by the program and turned into a Grid data structure. The Grid object is then contract solved. When the grid calls contract solver, it first populates the entire grid with correct suspect lists. This is done by creating a temporary neighbor array in each empty Unit to fill in all three parameters: row, column, and box. Once all 27 numbers are appended to the temporary neighbor list, the array is then passed into the "missing" function to grab all the potential suspects of that unit. This list is then given to the Unit attribute, so that later when we call Grid functions the current Unit will house the correct suspects. While traversing down this array, if the single suspect filler finds an Unit with only one suspect, then that Unit will be filled with that value and its suspect list will be deleted.

While working on this, a bug occurred where the suspect list did not reflect the true state of the puzzle. I had to overcome this bug by creating a new "safe" suspect filler that DID NOT fill in the single suspects. The reason behind this, is that when the Units down the line change, the Units in the front will not "realize" they changed, so their suspect list will be outdated, leading to contradictions down the road.

That is the extent of single suspect filler. The next step is to utilize the unique fillers. The logic behind unique fillers is a little trickier. For the sake of repetition, only row unique will be broken down in detail, since the same principle applies to column and box.

Row unique only has 9 rows to check. It starts with Row 1, and appends to a huge array that will hold all the suspects in Row 1. Each Unit, filled or unfilled, will end by appending a "0" to the total suspect neighbor list. This will come in handy for later reference. Now that we have the long list of suspects in this parameter, we must find unique numbers out of 1 through 9. Once the unique number(s) has been found, it will receive a token. For example, if the number 7 is unique in Row 1, the unique token array will look like this:

[0 0 0 0 0 0 1 0 0]

This signifies that we must fill 7 in somewhere. Now that we know 7 is unique in Row 1, we must find which exact Unit will house 7. This is where the "0" comes in handy. By keeping track of the number of times "0" shows up, we know which exact Unit will house the unique suspect.

One important aspect to note is that sometimes the re will be two different unique numbers that live in the same Unit. This is untrustworthy since it implies the Unit does not know which suspect is correct. To avoid further confusion or contradictions, I will ignore cases where two unique numbers come from the same Unit.

Once these gates have all been passed, it becomes clear that the unique number belongs in that Unit. This is only row unique, but column and box unique works in the same exact way just different indexing. This basic algorithm will fill in all the unique suspects in the 27 contracts possible.

Now that we have finally finished the contract solvers, we can finally discuss the real solver that matters. The contract solver is deemed useless the moment it does not progress the puzzle for more than three calls. If contract solver is defeated, the program will then quit contract solver and pass on the Grid to the guess-all solver. The guess-all solver knows the contract solver is stuck, so the first thing it does is guess.

To make this whole process recursive, the function must know when to stop. The function will stop when its stuck. The Grid is stuck when a contradiction happens or when its invalid. The function will also stop when it reaches a valid solution. Armed with these two end cases, the search for the solved Grid can commence.

It is important to remember that the base case (first guess) will always have right answer. The guess solver can trust the contract solver from above.

Here is the exact road map for the guess-all solver to succeed:

1) Try guess 1. If it is stuck, return failed Grid, and go to step 2. If it works, contract solve. If Grid is solved, return solved Grid. If not solved, put Grid into guess-all solve.

2) Try guess 2. If it is stuck, return failed Grid, and go to step 3. If it works, contract solve. If Grid is solved, return solved Grid. If not solved, put Grid into guess-all solve.

3) Both guesses failed. Parent guess has failed, so go to parent's guess 2.

By following these steps religiously, the program will eventually stumble upon the sacred solved Grid.

This full-length description should provide enough detail as to how both contract and guess solver works.

IV. Results

The results of my program will speak for itself. There is not gray area in sudoku solving. Either you solve it, or you do not.

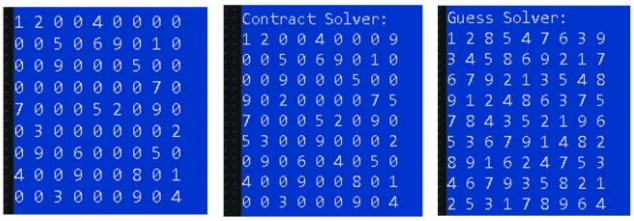
A. Description of Results

The results of my program will speak for itself. There is no gray area in sudoku solving. The puzzle is either solved or is not.

To discuss the results of my project, one must first discuss the type of puzzle was fed into the program. On a scale of

difficulty, my contract solvers could solve any easy to medium puzzle it encountered. For these types of puzzles, the contract solver would bring the puzzle to completion. The difference is solved state is usually around +65%, depending on how many clues were given.

On the difficult spectrum, my guess-all solver could solve all the difficult puzzles I have encountered. A few simple pictures will describe the overall journey a sudoku undergoes from unsolved to solved:



Unsolved (32%) Contract Solved (40%) Guess Solved (100%)

As denoted by the percentage marks, contract solver only solved 8% of the puzzle, but guess-all managed to bring the puzzle to completion, so +60%. Contract solving, on average, only provides +5% increase for difficult puzzles.

From online sudoku banks, I tried around 20-30 different puzzles and all of them were solved by either the contract solver or the guess-all solver.

B. Discussion of results

Despite being able to solve almost any sudoku puzzle it encounters, my program has one major flaw: My program can not solve puzzles that have a minimum suspect list of 3. To put it simply, when guess-all solver approaches this hypothetical grid, it will not be able to solve it since guess-all could only split a Unit with 2 suspects in the list. If the minimum suspect list across the entire Grid is 3, that would mean the guess-all would not begin the recursive solving.

Despite this shortcoming, I have a plan for solving this. I could simply create a more robust guess-all solver that starts at 2 but will elevate to a minimum guess of 3 if necessary. With this in mind, the program could raise the minimum bar to any desired level. In other words, if the puzzle does indeed have a minimum of 3 suspect list length, the robust guess-all would recognize it and increase the search for suspect lengths of 3 and start there. Once it recursively calls itself, it starts at 2.

In summary, I have figured out how to solve any sudoku possible in the world. This entire journey has greatly enriched my problem solving, mathematics, logic, and programming skills.

REFERENCES

https://www.sudokudragon.com/sudokuhistory.htm

[1] "History of Sudoku," *The origin of the Sudoku Puzzle*. [Online]. Available: https://www.sudokudragon.com/sudokuhistory.htm. [Accessed: 21-Mar-2021].

https://www.sciencedirect.com/science/article/abs/pii/S0020025513008359

[2] Y. Wu, Y. Zhou, J. P. Noonan, and S. Agaian, "Design of image cipher using latin squares," *Information Sciences*, 27-Nov-2013. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0020025513008359. [Accessed: 21-Mar-2021].

http://www.math.ie/McGuire_V1.pdf

[3] G. McGuire, *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem*, 01-Jan-2012. [Online]. Available: http://www.math.ie/McGuire_V1.pdf. [Accessed: 20-Mar-2021].

https://norvig.com/sudoku.html

[4] P. Norvig, *Solving Every Sudoku Puzzle*. [Online]. Available: https://norvig.com/sudoku.html. [Accessed: 21-Mar-2021].

https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/

[5] B. Carnes, "How to Play and Win Sudoku - Using Math and Machine Learning to Solve Every Sudoku Puzzle," *freeCodeCamp.org*, 04-Oct-2019. [Online]. Available: https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/. [Accessed: 21-Mar-2021].