Sudoku Solver: Using a Set of Possibilities

Steffen L.

Artificial Intellegence CS 4242
Kennesaw Sate University
Marietta, 1100 South Marietta Parkway 30060

30 Jan 2017

Abstract

The basic puzzle of Sudoku is one of the games that is considered an intellectual challenge similar to chess or go. The human brain uses multiple different strategies and rules to try and solve a Sudoku problem. While a computer may instead use simple methods such as brute force to test a combination of numbers to see if it works. This paper aims to try and simulate how the human would solve a Sudoku problem in a computer program. Rather than using computer based techniques, it will try to use normal human methods to solve the Sudoku. Such a method would be reducing a set of possible values that could fit in empty slots in the puzzle.

1 Introduction

Solving a Sudoku puzzle is a trivial task to a typical computer in this day and age. The computational task of solving a Sudoku problem using a brute force method is trivial if the algorithm is done correctly. The challenge is to attempt to replicate the thought process of a human being when solving a Sudoku puzzle. This is done to represent the difficulties in transferring the human thought process into code. The simple algorithm that most people use is checking all the possible numbers that could go in the empty spots on a row, column, and nonet. A nonet is the 3x3 sub square that divides the Sudoku board into nine sections. Where a human would take their time in going through each number and the relative location of the numbers and empty spaces, the computer can do it quickly so it would be easier when testing multiple puzzles.

2 Related Works

A paper by Imperial College London[3] utilized a constraint problem technique to solve for Sudoku puzzles. They use a similar method to constrain a possible set of numbers. They modeled the problem where the interactions of row, column and nonet are all accounted for. They then take all the constraints and run it through their propagations scheme to produce a valid solution.

Another paper created a metaphor for using ants behavior as their underlying algorithm[1]. They used a similar three dimensional array to store data. Then a simple rule based algorithm would crawl through the board similar to an ant and update the stored data. The ant algorithm is set up into a very small two element check, where they would validate the value of possibilities.

Using membrane algorithm and Particle Swarm Optimization, an algorithm was created to solve a sudoku puzzle in a very optimized and fast method. It displaced all other membrane based algorithms on speed and accuracy for solving the more complex puzzles including higher order sudoku problems. The membrane algorithm simply uses some form of best fit propositions for each cell on the grid. Each membrane proposes a possible value using its own internal calculations and communicates its best fit value to the rest of the membranes. More details can be found in the referenced paper[2].

3 Methodology

The architecture of the algorithm is structured with incremental rules and iterative decrements of possible values. When it starts, the algorithm has collections of possible values ranging from 1 to 9 for every square in

the Sudoku grid. This set of possible values (PV) uses a data structure of a 3D numpy array of booleans (x, y, z) where the x and y represents the grid coordinates of the Sudoku puzzle and the z represents the possible values that can go in (x, y). The initial starting state of the puzzle is imported as a 2D numpy array called the working (W) set. Both the PV set and the W set works on a 9x9 Sudoku grid.

Once the initial starting values of the puzzle have been imported, the set of PV can then begin to decrease in size. The first rule, rule0 will simply be the elimination of the initial values in the PV set. In the first iteration, this rule updates the PV set with the initial values in the W set. In sequential iterations, it updates the PV set with the newly found values from the previous iteration of the W set.

The next 3 rules, rule1, rule2, rule3, will deal with the possible existing values in the W set in the rows, columns, and nonets respectively. Rule1 would loop through all the rows and check for every existing element in the W set. Once it finds an existing element "e", it loops through each element in the current row again to find all the empty spots. It would then remove "e" in the PV set for the row. For example, the first loop would find a number in the working set for the row. Say that it found an "8", then the second loop would find an empty square and remove the possibility of the empty square having an "8". Rule2 and rule3 works similarly to rule1 for the columns and nonets.

The next rules, rule4, rule5, rule6, will iterate through the PV set for any number occurring only once for the row, column, and nonets respectively. In rule4, the algorithm would iterate through the row of PV set and count the occurrence of each number. If a number only occurs once, then it eliminates all other possible numbers for that spot. For example, if there was a row with three empty spots and the first and second empty spots have a possibility of "1 or 7" but the third empty spot has a possibility of "1, 6, or 7". The third empty spot has to be a "6" since it is the only possible spot for the "6".

These rules are used in every iteration. For each iteration, the PV set updates the working set if and only if the PV[x, y] set contains a value of only one number in the Sudoku grid. This means that at the end of the iteration, if there is only one number in a PV[x, y] and that number does not exist in the W[x, y] then W[x, y] is set to z where z is the only number in PV[x, y]. If there are no updates after an iteration, the algorithm considers the algorithm finished and checks itself with the solution. This means that some puzzles may not be filled in entirely.

The rules are set up in such a way that they can easily be inserted or removed from the algorithm. This allows for measuring the weight of each rule against how many puzzles are solved.

The programming language chosen for this algorithm was python. Libraries such as numpy and tkinter was used for data manipulation and GUI moderators [4][5]. The test sample of sudoku puzzles are acquired from genina.com.

4 Experimental Results

In the experiment, a set of three easy, three medium, and three hard puzzles are used to test the algorithms. Based on all 7 rules, the algorithm was able to finish all the puzzles. By picking and choosing only a few rules, we can judge how much the rules weigh to get a solution. The rules are graded on if the puzzle was solved and how many iterations it took to solve it.

Since rule0 is considered necessary to run the algorithm correctly, it will not be running without rule0. Simply running rule0 by itself will result in only a static modification of the PV set where it only removes values from the initial puzzle. The rules that are applied to rows and columns have been grouped together. This will reduce redundency since the results would be specific to how the rows and columns are set up in the puzzles. This leaves us with four groups: rule1 & rule2, rule 3, rule4 & rule5, and rule6. This is shown in Table 1.

Out of these combinations, rule1 and rule2 produced the most results on their own. Along with rule3, it was able to solve over half the test data. Rules 4 through 6 however are more fine-tuning rules where once rules 1 through 3 are ineffictive, 4 through 6 can help it progress. An interesting event happened when testing rule set 1,2,6 and 1,2,4,5,6. These tests showed that some of the values it was producing was violating the rules of Sudoku. All the other tests only failed when they did not fill in the squares and finish the puzzle, while all the other attempts were correct. My theory is that rule6 requires rule3 to operate correctly. Since rule6 will attempt to select a single number out of a nonet, and rule3 makes sure there are no duplicates within the nonet, rule6 may attempt to select numbers from a duplicate PV. For example, if the initial problem contains a "7" in a nonet, rule3 would eliminate all possible "7s" in that nonet. However since there is no rule3 there may exists more "7s" in

Table 1: Results of Combinations of Rules 1-6

Rules	easy1	easy2	easy3	medium1	medium2	medium3	hard1	hard2	hard3
1, 2	10	N	N	N	N	N	N	N	N
3	N	N	N	N	N	N	N	N	N
4, 5	N	N	N	N	N	N	N	N	N
6	N	N	N	N	N	N	N	N	N
1, 2, 3	5	6	7	11	10	N	N	N	N
1, 2, 4, 5	4	N	N	N	N	N	N	N	N
1, 2, 6	N	N	N	N	N	N	N	N	N
3, 4, 5	N	N	N	N	N	N	N	N	N
3, 6	N	N	N	N	N	N	N	N	N
4, 5, 6	N	N	N	N	N	N	N	N	N
1, 2, 3, 4, 5	2	3	3	4	5	4	10	7	7
1, 2, 3, 6	3	4	5	6	6	8	N	10	13
1, 2, 4, 5, 6	N	N	N	N	N	N	N	N	N
3, 4, 5, 6	N	N	N	N	N	N	N	N	N
1, 2, 3, 4, 5, 6	2	3	2	4	4	4	7	5	6

N = Incorrect Solution

the nonet. If the PV set of "7" only has one instance of "7" then rule6 will select that "7" even though the initial "7" from the starting problem exists within the same nonet.

By extension, rules 4 and 5 should also have the dependent property exhibited by rule6, but since rules 1 and 2 allow the puzzle to progress, the algorithm would not have gone far enough to show such a property. I then broke up the initial groups and tried to test without rules 1 or 2 separately. In a test using a set of rule 2, 3, 4, 5, 6, the result produced the dependent property since rule1 was not in effect. The result had duplicate values throughout the rows, columns, and nonets. With a rule set of 1, 3, 4, 5, 6, the result had similar duplicates. The implementation had the rules run in numerical order, if the rules were ran in a different order such as 6, 5, 4, 3, 2, 1, the result would also produce incorrect values and a few duplicates.

5 Conclusion

Sudoku can be solved up to a level difficulty of hard simply by using a constrained possibility set and 6 simple rules that iterate through the puzzle. These rules are based on a human thought process rather than a brute force method. They require some form of ordering and dependency to run correctly. The first three rules are based on the principle that solutions cannot duplicate in a row, column, and nonet. The second set of three rules are based on single number possibility. This is where the possibility of a number in a row, column, or nonet occurs only once, then the solution must contain that number assuming the first three rules apply beforehand. The main reason for this algorithm is to understand the difficulties of transferring human thought processes into working code.

Acknowledgements

This paper used a Latex template to arrange the formatting and style of the IEEE system. This being the first paper I have used using Latex, I would also like to reference LaTeX .

References

[1] Schiff, K. (2015). AN ANT ALGORITHM FOR THE SUDOKU PROBLEM. *Journal Of Automation, Mobile Robotics & Intelligent Systems*, 9(2), 24. doi:10.14313/JAMRIS_2-2015/14

[2] Singh, G., & Deep, K. (2016). A new membrane algorithm using the rules of Particle Swarm Optimization incorporated within the framework of cell-like P-systems to solve Sudoku. *Applied Soft Computing*, 4527-39. doi:10.1016/j.asoc.2016.03.020

[3] Simonis, Helmut (2005). "Sudoku as a Constraint Problem" (PDF). *H Simonis homepage*. Cork Constraint Computation Centre at University College Cork: Helmut Simonis. Retrieved 8 December 2016. "paper presented at the Eleventh Inter-

national Conference on Principles and Practice of Constraint Programming"

[4] Stfan van der Walt, S. Chris Colbert and Gal Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[5] "Tkinter Python interface to Tcl/Tk Python v2.6.1 documentation". Retrieved 2009-03-12