SOLVING SUDOKU PUZZLES WITH SEARCH BASED BACKTRACKING CONSTRAINT SATISFACTION ALGORITHMS

Shubha Swamy, Curtis Lin
CSCI: 3202 - Introduction to Artificial Intelligence
December 13, 2018

I. Methods and Implementation

Sudoku stands as one of the most popular past-times for many across the world. Although one might often find it hard to solve a sudoku puzzle with pen and paper, the solution is quite trivial to attain through a search-based backtracking algorithm. In this paper, we explore several studies that have delved into studying the algorithm that efficiently solves sudoku puzzles. We then proceed to implement our version of the algorithm and discuss the results.

Sudoku is played on a 9x9 board, with a total of 81 cells in the entire grid. The grid is further broken down into nine sets of 3x3 cells that do not overlap. Few values are placed on the board initially. These values form the constraints which need to be satisfied by solving it. The Sudoku puzzle is then solved by placing the number that satisfies three restrictions. The first restriction makes sure that there are no duplicate values in a row with unique values from 1-9 inclusive. The second restriction makes sure that there are no duplicate values in a column with unique values from 1-9 inclusive. The third rule enforces that each 3x3 grid within the Sudoku grid has unique values from 1-9 inclusive. A solution is found when each of the 81 cells has a number ranging from 1-9, while also satisfying all of the rules. While solving a puzzle like this with a pen might take some time, using a search based backtracking algorithms along with constraint satisfaction can provide us with a solution in a matter of seconds.

There have been various implementations of different kinds of algorithms which aim to solve the Sudoku puzzle. For example, harmony search which is a musician's behavior inspired evolutionary algorithm has been used as an optimization algorithm to solve for the sudoku puzzle (Geem, 2007). The algorithm frames the sudoku solving problem as an optimization problem with number-uniqueness penalties. Although the algorithm was able to find solutions optimally for sudoku boards with about 40 given values, it failed to solve for hard level sudoku puzzles which had 26 given values.

Another method implements MITS: Mixed-Initiative Intelligent Tutoring System for Sudoku. MITS is a technique that uses intelligent tutoring systems for games and mixed-initiative systems (Cain and Cohen, 2007). MITS is applied to Sudoku puzzles as a way of guiding the moves towards the correct solution through intelligent learning. Another algorithm uses the GAuGE method which stands for Genetic Algorithm using Grammatical Evolution. This is a genetic algorithm is applied to Sudokus by a sequence of instructions is evolved and applied to the board, and a fitness reward is given back (Ryan and Nicolau, 2006). This essentially mimics the way a human solves a Sudoku puzzle. The GAuGE method is generally solvable if a sufficient set of constraints are provided. However, this method does not necessarily provide a guaranteed solution to all cases.

The method implemented in this project uses a search-based backtracking algorithm along with constraint satisfaction in order to solve for Sudoku boards. Using a naive backtracking algorithm would be insufficient to solve for the problem. It is important to use backtracking along with an algorithm that accounts the constraint propagation in order to arrive at an optimal solution (Weber, 2006). Therefore in our implementation of the algorithm, we implement a search-based backtracking algorithm which accounts for constraint satisfaction. We ended up using the backtrack-based algorithm because it provides the most accurate results in reasonably good time (Simha, Suraj & Ahobala, 2012).

The first function we implemented defines a Python class that has the various attributes of our Sudoku board. We include several methods in our program to process various parts of the algorithm. The *checkIfNumberOk* function that checks if adding a number to the board is

possible. The function takes in the location of the value from the board as a coordinate tuple and the number value at that position, and it returns a boolean value accordingly. The constraints and the rules that make up the Sudoku game are checked in this function to make sure that they are satisfied. The primary backtracking function, *recursive_backtracking*, is defined in a recursive structure. This function works by iterating through every possible value for the existing position the program is at. It first checks that the value does not violate any of the constraints placed upon it through the *checkIfNumberOk* function. If it does not violate any constraints, and assigns the current position to the number and modifies the sudoku board. It then continues through the recursion for the next values and backtracks if a value does not satisfy the conditions. If the board returns a board with any square still filled with 0, then the board is either unsolvable or the algorithm failed to solve the board.

II. Results

The runtime of this algorithm scales exponentially to the size of the board. Since the Board is 9x9 squares, we are guaranteed that the algorithm has a runtime of at least $O(n^2)$. This is because in order to make sure all the constraints are satisfied, we have to iterate through the 2-D array values of the board. The actual runtime of the algorithm we implemented is $O(n^m)$, where n is the number of possibilities for each square (nine for the case of Sudoku) and m is the number of remaining spaces that are blank and still need to be filled. The runtime for this algorithm primarily stems from the recursive backtracking that is implemented. The algorithm essentially performs a search depth wise while searching through possible solutions. Through this perspective, n can be considered as the branching factor and m equates to the depth, resulting in a worst case of $O(n^m)$ for the algorithm we have implemented.

In our implementation, we also generate random boards by placing random numbers when we initiate a board. We did this to primarily test the outputs and performance of our algorithm. The random boards are generated by first iterating by row in an "empty" board filled with zeroes, and choosing between 1 and 3 (inclusive) random indices. The random boards are required to have a minimum of 17 clues. Using this method did not always yield an answer, as boards generated in this manner can often result in boards that have no solution. The main consideration that needs to be applied to creating a Sudoku board is that there should be enough starting values. For example, although having only one value would make a Sudoku board solvable, it is considered an inaccurate solution because having only one constraint produces multiple solutions which make the Sudoku invalid. It is a requirement that well-formed Sudoku boards have only one unique solution. Through research studies, it has been proven that the minimum number of starting clues required to solve a Sudoku puzzle is 17 (MIT Review, 2012). Nobody has been able to prove so far that a unique solution can be produced with 16 or fewer clues.

Figure 1 below depicts the world's hardest Sudoku puzzle (Collins, 2012). The algorithm we implement was able to solve this case in 9.8 seconds with 49,498 number of backtracks. This proves that search based backtracking with constraint satisfaction is an optimal algorithm that yields a solution even for the hardest possible case for Sudoku puzzles. The algorithm iterates through various possibilities and checks constraints ultimately producing a valid solution.

Figure 1: Solving the World's Hardest Sudoku Puzzle

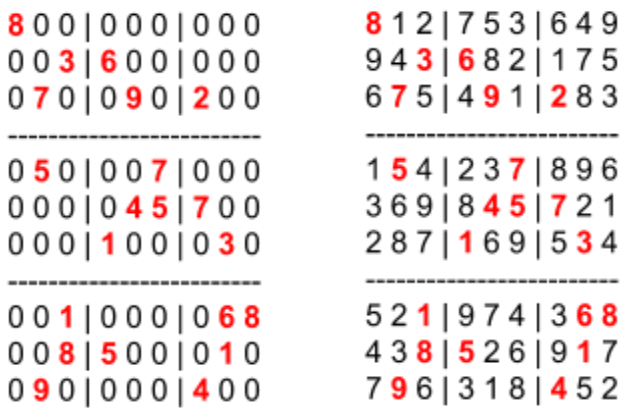

Figure 2 Algorithm Performance : Average Runtime and Backtracks Based on Initial Number of Clues

Clues	Runtime (seconds)	Backtracks
17	0.504968623	1559.416667
18	1.499152946	4049.933333
19	2.30812851	4579.785714
20	1.699907414	1927.533333
21	1.220800877	23496.58333
22	1.121959283	11684
23	1.345962572	9123.4
24	1.171564969	16624.18182
25	0.266493368	4910.9
26	3.302043785	21909
27	0.621702035	3245.75

Figure 3 : Average Runtimes Based on the Number of Initial Clues

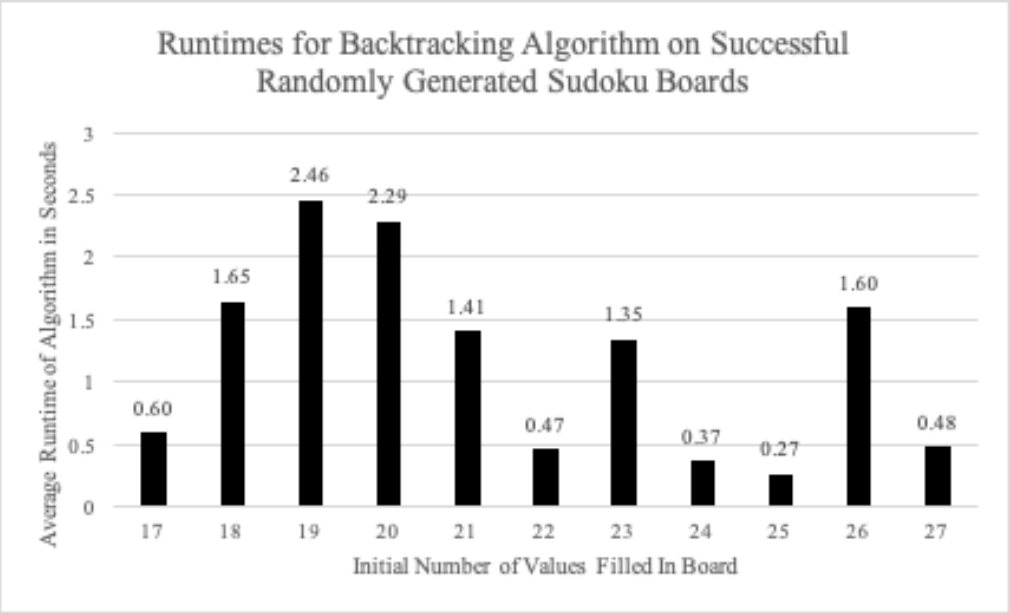

Figure 4: Trends in the Number of Backtracks Based on the Number of Initial Clues

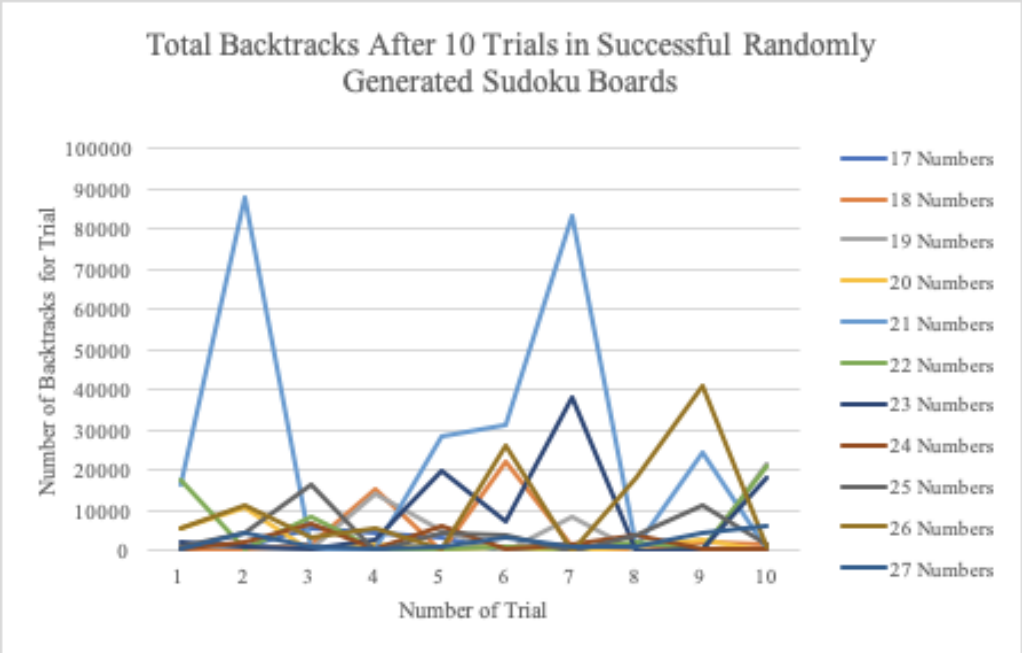

Figure 2 displays the average results we gained from running tests on several randomly generated Sudoku puzzles. As a side note, we did not account for the data when a Sudoku puzzle took more than 10 seconds to run, and only included the tests in which the randomly generated sudoku board was successfully completed. Figure 3 helps us analyze the runtime of the algorithm. We observe from the graph that the average runtime peaks at 19 and 20 clues. The runtime of the algorithm is primarily depended on the depth of the backtrack that is performed. In Figure 4, we see that the number of backtracks peaks during a trial with 21 initial clues.

III. Discussion and Conclusions

Our implementation of a search based backtracking algorithm to solve for Sudoku was tested against several "expert" sudoku problems with known solutions, and produced the correct answer for every test. However, the algorithm we implemented is very bare-bones and leaves a lot of potential to make the algorithm faster.

As indicated in "MIT Technology Review," having less clues in a sudoku problem doesn't necessarily indicate that a sudoku board will be harder to solve, but rather it is also "because the hardness of the puzzle depends not only on the number of clues but also on their position as well." They interestingly noted that out of all the 17 or 18 clue puzzles they tested, all were easier than the famous "Platinum Blonde," puzzle, which has 21 clues. For this reason, we tested our algorithm against the "Platinum Blonde." If we were to base the difficulty of Sudoku puzzles based on the total number of backtracks, the "Platinum Blonde" would be considered extremely difficult. While running our algorithm with this sudoku board produced a feasible (and presumably unique) solution, the algorithm required a whopping 1,114,711 backtracks to complete with a predictably terrible runtime.

This presents two potential areas of further possible exploration: first, how can we determine the difficulty of a given sudoku board? While we can predict how difficult a board would be the average user based on the number of backtracks needed to complete the board provided by the algorithm, it would be interesting to try to describe the exact characteristics that makes these specific boards to solve, by researching the position of the numbers on the board along with the number of clues provided. The other area of exploration to pursue is to attempt to drastically decrease the number of backtracks needed to solve a given sudoku board by our backtracking algorithm.

Our algorithm currently has a worst-case runtime of $O(n^m)$. For future solutions, we would like to explore solutions that perform better. We can begin to try to improve our algorithm by implementing some type of variable ordering to complete the puzzles. Currently, we are iterating first by row, and then by column within the row to finish the board. However, instead of choosing the next open space arbitrarily, we could choose the variable with the "fewest legal left values in its domain", along with value ordering, by choosing the variable that "rules out the fewest values in the remaining variables." (Heckman). Adding in these two ideas is a good start for a next step in exploring this topic.

Works Cited

arXiv, Emerging Technology from the. "Mathematicians Solve Minimum Sudoku Problem." *MIT Technology Review*, MIT Technology Review, 9 Jan. 2013, www.technologyreview.com/s/426554/mathematicians-solve-minimum-sudoku-problem/.

Xiv, Emerging Technology from the. "Mathematics of Sudoku Leads To 'Richter Scale' of Puzzle Hardness." *MIT Technology Review*, MIT Technology Review, 22 Oct. 2012, www.technologyreview.com/s/428729/mathematics-of-sudoku-leads-to-richter-scale-of-puzzle-hardness/.

Caine , Allan, and Robin Cohen . "Tutoring an Entire Game with Dynamic Strategy Graphs: The Mixed-Initiative Sudoku Tutor." *David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada*, 2007, pdfs.semanticscholar.org/7e77/52098fb05c02e5ca27f2d68684117be83302.pdf.

Collins, Nick. "World's Hardest Sudoku: Can You Crack It?" *The Telegraph*, Telegraph Media Group, 28 June 2012, www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html.

Nicolau, Miguel & Ryan, Conor. (2006). Solving Sudoku with the GAuGE system. 3905. https://www.researchgate.net/publication/221009172_Solving_Sudoku_with_the_GAuGE_system.

Geem , Zong Woo. "Harmony Search Algorithm for Solving Sudoku." *Johns Hopkins University, Environmental Planning and Management Program*, 2007, link.springer.com/content/pdf/10.1007/978-3-540-74819-9_46.pdf.

McGuire, et al. "There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem." *[Astro-Ph/0005112] A Determination of the Hubble Constant from Cepheid Distances and a Model of the Local Peculiar Velocity Field*, American Physical Society, 1 Sept. 2013, arxiv.org/abs/1201.0749.

Simha , Pramod, et al. "Recognition of Numbers and Position Using Image Processing Techniques for Solving Sudoku Puzzles." *An Introduction to Biometric Recognition - IEEE Journals & Magazine*, Wiley-IEEE Press, 2012, ieeexplore.ieee.org/abstract/document/6215962.

Simonis, Helmut. *Sudoku as a Constraint Problem*. citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.2964&rep=rep1&type=pdf.

Weber , Tjark. "A SAT-Based Sudoku Solver*." *Institut Fur Informatik, Technische Universitat Munich*, www.cs.miami.edu/home/geoff/Conferences/LPAR-12/ShortPapers.pdf#page=15.