Backtracking Algorithm Implementation for Sudoku Solver

Jonathan Dervin, Roman Higginson, Zaryaab Qasmi

20 April 2024

**Abstract**

This report outlines the development of a Sudoku Board Solver, undertaken to enhance our understanding of backtracking algorithms and elementary artificial intelligence. Our primary emphasis was on implementing a backtracking algorithm to efficiently solve Sudoku puzzles by systematically exploring possible solutions. Additionally, we incorporated strategic methods to augment the algorithm's effectiveness. This endeavor provided invaluable insights into the integration of logical algorithms for solution determination and the utilization of generative algorithms to optimize solution-seeking methodologies

**Introduction**

Sudoku is a puzzle game that involves the player entering certain numbers from 1 to 9 into a 9x9 grid while being mindful of not violating the rules of the game. The rules are that when a player inputs a number, the number must not be the same as another number in the same row, the same column, and the same 3x3 sub-grid. There are many strategies to this game that an experienced player can use to solve the puzzle with minimum errors. Some solving strategies for Sudoku are Hidden Singles, Naked Pairs/Triples and Hidden Pairs/Triples.

The problem that the backtracking algorithm addresses are the rules that this game abides by. The fact that each number in each row, column, and sub-grid must be unique. For this project, the goal was to use several solving strategies to solving 5 boards. But when none of the solving strategies can be used, the program will resort to using the backtracking algorithm to solve them.

**Methodology**

The application comprises a straightforward front-end and back-end architecture facilitating the display of pages on a webpage. Concerning the front-end, the application utilizes Bootstrap, HTML, CSS, and JavaScript to construct a minimalistic interface presenting a Sudoku board to users. Notably, two web pages share a similar layout. The initial page enables users to input a Sudoku board and subsequently request its solution. Upon requesting the solution, users are redirected to a second page where the results are displayed. In instances where a solution is unattainable, an error notification indicating the absence of a solution is presented. Users are then provided with an option to return to the initial page. Both pages leverage JavaScript's fetch requests to interact with the server.

Regarding the back-end infrastructure, Flask is employed to establish the server. Flask facilitates the creation of a rudimentary server and the routing of diverse client requests to distinct functions. Two primary functions govern the back-end's response to requests: "serve\_html" and "customBoard." The "serve\_html" function serves HTML files upon client requests, locating and returning the specified file as necessary. Conversely, the "customBoard" function enables users to submit Sudoku game boards, initiating the solver's search for a solution. Subsequently, if a solution is found, it is transmitted to the client. Additionally, the server incorporates several auxiliary functions to facilitate the transformation of data into JSON format, facilitating seamless communication between the front-end and back-end components.

A Sudoku Board is considered to be a Constraint Satisfaction Problem (CSP), which is a problem where we need to assign a set of variables that satisfy all of the restrictions given. The backtracking algorithm is a searching algorithm for CSPs where we use a search tree to assign a single variable per level. Since Sudoku is a CSP, the backtracking algorithm will also utilize its key components:

Variables: These represent every cell inside the Sudoku board

Domains: These represent a list of numbers that each cell can have.

Constraints: These represent the limitations each value in the cell must follow.

The backtracking algorithm is meant to assign a number from a domain to an empty cell while being consistent to the constraints of the Sudoku board. If the cell is not consistent, unassign the cell and go to the next number in the domain. If the cell is consistent, go to the next empty cell. If no number in the cell’s domain is consistent to the constraint, backtrack to the last cell that was assigned.

For representing the Sudoku board, we decided to represent it as its own class which contains instances of the Cell class, which represents a value on the board. To represent the CSP and its components, we decided to use a dictionary. The keys of the CSP would be a string named after the components: variables, domains, and constraints. The value of the variable key would have a list variable containing the cells of the Sudoku Board. The value of the domain key would contain a dictionary of each variable containing a list of numbers from 1 to 9. The value of the constraint key would be a dictionary of each constraint type: rows, columns and sub-grids. Each constraint holds a lambda function for checking if a given value violates the constraint or not. In order to solve the board, we used a class called solve\_sudoku\_csp that takes the current board and the CSP variable as parameters. The solve\_sudoku\_csp class contains three functions: the first function is the backtracking function, the second function is for checking for available empty cells in the board, and the third function is for checking if the recently assigned board is consistent with the constraints.

**Results**

It solved all five puzzles using the backtracking algorithm. Each puzzle was tested 5 times to calculate the average time each one took to solve. Puzzle 1 took 1516 backtracks for it to be solved. The average time to solve was 49.5 ms. Puzzle 2 took 4209 backtracks for it to be solved. The average time to solve was 113.6 ms. Puzzle 3 took 201 backtracks for it to be solved. The average time to solve was 12.8 ms. Puzzle 4 took 70373 backtracks for it to be solved. The average time to solve was 1.8 s. Puzzle 5 took 1525 backtracks for it to be solved. The average time to solve was 43.7 ms.

In terms of complexity of the puzzles, Puzzle 1 and 5 seem to have similar difficulty levels. Puzzle 5 was found from the Sudoku Solver website. This puzzle was listed as a tough puzzle. In comparison to the other puzzles, all of them except for Puzzle 3 took a larger number of backtracks and took more time to solve. So, both Puzzle 1 and 5 have a decent amount of complexity. Puzzle 2 is the second most complex puzzle. Puzzle 3 had the least amount of complexity to solve. Puzzle 4 was the most complex puzzle to solve. This makes sense as this puzzle was also found from the Sudoku Solver website, and it was listed as Diabolical, meaning it was a very complex puzzle for a human to solve, as it would require a lot of the solving strategies to efficiently solve the puzzle. This is true for the backtracking algorithm as well as it backtracked a large number of times.

**Discussion**

The efficiency of the backtracking algorithm depends on the difficulty level of the board it is trying to solve. The algorithm relies mostly on brute force. The different strategies help reduce the workload of the program and gives it a little more depth. The time complexity is said to be O(kn), k being how many times the function recursively calls itself and n being the number of empty cells on the board. The space complexity is the same as the time complexity, O(kn).

We encountered limitations with the solving strategies, primarily due to the complexity of many available options. The strategies implemented in this project are chosen for their simplicity, aiming to lessen the workload on the backtracking algorithm without significantly slowing down the solving process. However, some strategies were deemed too complex and risked increasing the overall solving time, prompting their exclusion from the implementation. It's important to note that these strategies may not be effective for all puzzles, meaning the backtracking algorithm still carries most of the workload, despite the strategies employed in this project.

**Conclusion**

To summarize, we found that the backtracking algorithm was very effective in solving multiple Sudoku boards. Though, depending on the difficulty of the board, the amount of time it takes to solve it can very, showcasing how the backtracking algorithm at its core is a brute force approach to problem solving. Our program presents opportunities for enhancement on multiple fronts. Regarding display, refining the webpages and strengthening the client-server connection could provide a more comprehensive view of the algorithm's operations. As for the solver itself, additional solving strategies could potentially expedite the project's execution time, while optimizing the backtracking algorithm in various segments could further enhance efficiency. Moreover, integrating solving strategies more seamlessly into the backtracking algorithm offers potential for improvement. Specifically, implementing a mechanism to determine the optimal transition point between different solving methods could enhance efficiency by identifying when certain strategies or the algorithm itself are most effective.

**References**

Backtracking. Interviews.school. (n.d.). <https://interviews.school/backtracking>

GeeksforGeeks. (2023, October 16). Introduction to backtracking - data structure and algorithm tutorials. <https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/>

Gupta, U. (2024, March 26). Recursion & Backtracking Time Complexity. Code 360 by coding ninjas. <https://www.naukri.com/code360/library/recursion-backtracking-time-complexity>

Stuart, A. (2005a, May 28). Sudoku solver by Andrew Stuart. <https://www.sudokuwiki.org/sudoku.htm>

Stuart, A. (2005b, June 9). Naked candidates. SudokuWiki.org - Naked Candidates. <https://www.sudokuwiki.org/Naked_Candidates#NP>

Stuart, A. (2008, April 9). Hidden candidates. SudokuWiki.org - Hidden Candidates. <https://www.sudokuwiki.org/Hidden_Candidates#HP>

Appendices:

For this project, we got some assistance from ChatGPT. We used it for finding the best way to implement the backtracking algorithm. At first, we decided to use what we learned in class to help us implement the algorithm. However, this produced faulty results as it would not solve the boards and would stop at the start of the second row. To help us find the best way to implement the backtracking algorithm using ChatGPT, we first asked it: My team has been tasked with the implementation of Sudoku in Python and we would like guidance on how to proceed using the backtracking algorithm. ChatGPT gave us an explanation of the backtracking algorithm. Then, it told us how we would represent the board in Python, giving us a list of lists as the data type we could use. Then, it gave us a basic explanation and implementation of the program in Python. We also asked it: Since this is a Constraint Satisfaction Problem, can you make a variable with the key components of it? It then gave us an explanation of CSPs and its components, and also told us what each component is represented as in terms of a Sudoku board. It also gave us the implementation of the components in Python. The variable and domains components were both variables, but the constraints were implemented as one function and each constraint was a conditional statement. We wanted the constraints to also be listed as a variable to keep each CSP component consistent with each other. So, we asked it what are the constraints of a sudoku puzzle and can you make a variable from those? ChatGPT then listed the constraints of a Sudoku board and gave us it as Python code. Each constraint was listed as a function. This still was not what we wanted. We then asked ChatGPT to put the key components of a CSP into a dictionary. This was how we first implemented it last time. So, it gave us Python code showing the CSP variable as a dictionary. Each key was the string name for the component. For the variable and domain keys, they contained the same values that ChatGPT produced before. For the constraint key, the value was a dictionary with each constraint type. The value for each key was a lambda function for checking if the constraint is consistent. This was an interesting result because we would have never thought of using a lambda function as a way of checking for consistency. Before, for each constraint, we used a list containing a list of tuples that represented each constraint. The program would then check each tuple in the constraint and compare it. This was a very sloppy approach to turning each constraint into a variable, and ChatGPT was able to give us a solution that was a lot cleaner. We then asked it how do we use this variable to solve the sudoku board. It then gave us Python code for how we can implement it using the CSP variable. The code implemented the backtracking algorithm as a part of a class which also contains a way to check for consistency.

I have neither given nor received unauthorized aid in completing this work, nor have I presented someone else’s work as my own.