Sudoku Solver Application by State Space Tree Backtracking Algorithm

Arwa Hassan[1], Asmaa Elshabshiri[1], Jumana Moussa[1], Nadine Donia[1]

[1]Department of Computer Science and Engineering, American University in Cairo

Abstract

This report details the design and implementation of a Sudoku application that handles functionality for both generation and solving utilizing a State Space Tree Backtracking algorithm. Sudoku is an NP-complete problem which necessitates an efficient solving strategy. As such, the application utilizes a Depth-First Search (DFS) approach in combination with selection criteria that align with Sudoku's rules to explore the possible solutions and reach a complete, valid solution. The core objective of this project is to develop a functional application that can generate solvable puzzles and solutions with an emphasis on integrating an enjoyable user experience.

Keywords: Sudoku solver, Backtracking, State Space Tree, Depth-First Search

1.1 Introduction

Sudoku is a widely popular logic-based puzzle game that requires filling a 9×9 grid with digits from 1 to 9, ensuring that each row, column, and 3×3 subgrid contains no repeated values. The complexity time of solving a general Sudoku puzzle is a topic of great interest, and many algorithms have attempted to decrease that time complexity. Sudoku remains a type of NP-complete problem. NP-complete simply means that solving it requires a non-polynomial algorithm while verifying its solution can still be done via a polynomial algorithm [1]. As such, the worst-case time complexity for solving an NP-complete problem is generally believed to be exponential [1]. The primary objective of this project was the development of a user-friendly application that solves Sudoku puzzles of varying difficulty. The backtracking algorithm was chosen to be the core solver due to its conceptual simplicity, and effectiveness in traversing the large state space efficiently.

1.2 Problem Definitions

This report focuses on the implementation and analysis of a State Space Tree Backtracking algorithm for solving 9x9 Sudoku puzzles. The primary objectives are the functional implementation of the algorithm to both generate and solve valid Sudoku puzzles in addition to a performance analysis to measure the computational cost (execution time) of the backtracking algorithm to assess the efficiency of the chosen implementation.

1.3 Methodology

Our backtracking algorithm generates a solution by recursively generating children, checking their validity and backtracking if a given node does not contain a valid board. The root node is the initial unsolved puzzle. The root has 9 children, corresponding to the 9 possible numbers that can inhabit the first empty cell. Each node also has 9 children, corresponding to the possible states in the second empty cell, so on and so forth. Each node essentially contains a “state” of the board. For the algorithm to parse through the state space tree, it utilizes a depth first search, proceeding to the deepest possible node before backtracking if the boolean isValid function, which checks the validity of the entire board, returns false. To check the validity of the board, the function iterates through every column, row and box to ensure a given number is only repeated once in each.

1.4 Specification of Algorithms to be Used

Backtracking via State Space Tree

This algorithm aims to find the solution of a Sudoku puzzle by generating the possible states and backtracking when an invalid board is generated. It uses two main classes: node and matrix. The matrix class is created with the objective of having the sudoku board as an object. A class node is also created that represents each node in the state space tree. In other words, each node represents the sudoku board at a specific snapshot. Each node has a pointer to an array of its 9 children. When the application runs, a board is automatically created in the background, to improve user experience.

Solution Generation

In the first iterations of implementation, the function to generate the possible children was initially in the constructor with the reasoning that, every time a new node is created, its children can also be created to ease backtracking. While testing that implementation, however, it was quickly realized that an infinite loop was created and utilized all the available storage of our device. To remedy the error, we decided to move the generate children function to be called during backtracking. Now, when the function Sudoku solver is called, the function populateChildren is called to generate all the 9 possible states of the board at a given position, allowing the program to, via backtracking, find a valid node that contains the complete, solved Sudoku puzzle.

Puzzle Generation

Our application generates a puzzle once the user loads the program. The generate function was implemented by choosing random numbers and filling the board, backtracking when the board is no longer valid. It would then remove cells at random, with the amount of cells removed dependent on the difficulty level chosen by the user. This implementation caused some generated boards to have more than one valid number in a given cell position which contradicts the core functionality of Sudoku. To resolve this, a helper function was created countSolutionsLimit to verify that a generated puzzle only has one valid solution which is then called while emptying cells in the board. If, after emptying a cell, the board can now have more than one valid solution, the removed number is returned.

1.5 Data Specifications

The data expected in our program is simply in the form of integers. Once the application runs, the user is prompted via push buttons with whether they want to solve a Sudoku problem and the level of difficulty they desire. Once chosen, the application generates a new window and displays a Sudoku puzzle and awaits the user to input an integer from 1-9 in each vacant cell. As such, the data the program awaits is in the form of integers which are then saved into a matrix and checked for their current position's validity. While not implemented in the QT application due to its contradiction with the concept of a Sudoku app, the algorithm developed can accept an inputted string of integers, check for the validity of that board and then, if valid, solve it and output its solution.

1.6 Experimental Results

From the experiment, the difference between Backtracking and Dancing Links (DLX) became very clear in both theory and actual performance. Backtracking has an exponential time complexity O(9^E), where E is the number of empty cells, and it uses fairly little memory since it only needs the state space tree and the board. The problem is that this makes it extremely slow once the puzzle becomes harder. In our results, Backtracking took around 8 ms for Easy puzzles, but it quickly jumped to over 1000 ms for Medium puzzles and more than 1600 ms for Hard ones, with some runs taking several seconds. DLX is also exponential in theory, but it uses a much larger data structure (the exact-cover matrix with many linked nodes) to cut down the branching factor early. Because of this extra memory usage, it ended up solving every puzzle, Easy, Medium, and Hard, in about 0.15 ms on average, with almost no variation. Overall, the results line up with what the theory suggests: Backtracking is simple but slows down fast as difficulty increases, while DLX uses more space but is much faster and far more consistent.

1.7 Analysis and Critique

While the results clearly show DLX outperforming Backtracking, the experiment has some limitations. We mainly focused on runtime and did not directly measure memory usage, even though DLX relies on a much larger data structure. Backtracking also has many variations that could improve its performance, but these were not explored. Finally, the tests were limited to 9×9 Sudokus, so the results may not generalize to other puzzle sizes. Despite these constraints, the trends still match the expected theoretical behavior.

1.8 Conclusions

The project successfully developed a functional and efficient Sudoku application based on the State Space Tree Backtracking algorithm. By carefully testing and tweaking the implementation to avoid excessive memory usage incorporating robust constraint checking, the solver performs well across various difficulty levels. While the core backtracking logic is fundamentally sound, the analysis suggests that incorporating stronger search heuristics could provide performance improvements for the more challenging puzzles.

References

[1] F. Vega, "Solving NP-complete Problems Efficiently," *IPI Letters*, vol. 2, no. 2, pp. 76-79, September 2024, doi:10.59973/ipil.122.

Appendix: Listing of all Implementation Codes