Backtracking Algorithm Implementation for Sudoku Solver

Jonathan Dervin, Roman Higginson, Zaryaab Qasmi

20 April 2024

Abstract

* Summary: A concise overview of the report, summarizing the purpose, methodology, main findings, and conclusions of your work on the backtracking Sudoku solver.

Introduction

* Background: Provide context for the Sudoku puzzle and the significance of solving it algorithmically.

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

* Problem Statement: Define the specific problem that the backtracking algorithm addresses in the context of Sudoku.

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.

* Objective: Outline the main objectives of implementing the backtracking algorithm, such as understanding its efficiency, effectiveness, and limitations in solving Sudoku puzzles.

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

* Algorithm Overview: Describe the backtracking algorithm, explaining how it attempts to fill the Sudoku grid one cell at a time and backtracks when it encounters a cell where no legal values can be placed.

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.

* Implementation Details: Discuss the approach taken, data structures, and any particular aspects of the implementation like recursion, the choice of starting point, and how conflicts are detected and resolved. Here is where I’d like a statement and description of the creative/innovative approach taken.

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

* Performance Analysis: Present the results of the algorithm’s performance, including metrics such as execution time, the number of puzzles solved, the complexity of puzzles handled, and the number of backtracks required for various puzzles.

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

* Efficiency Evaluation: Analyze the efficiency of the backtracking algorithm in terms of time complexity and space complexity.

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)

* Limitations: Discuss any limitations encountered with the backtracking algorithm, such as difficulties with certain types of puzzles or inefficiency in particular cases. Make recommendations for improvement.

Conclusion

* Summary of Findings: Summarize the key findings from your experimentation with the backtracking Sudoku solver.
* Recommendations: Offer recommendations based on your findings, such as improvements to the algorithm, strategies for optimization, or areas for further research.

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

Appendices:

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