Sudoku Solving Algorithm

CO4352 - Advanced Algorithms

Name: Pathiraja P.M.M.N

Index No: 17/ENG/080

Registration No: EN86199

1.0 Introduction

Sudoku is a popular puzzle game that is popular because it is entertaining and has easy-to-understand rules. It is a number-based logic puzzle with numbers ranging from 1 to 9. Sudoku is the most popular Sudoku puzzle, which consists of a 9X9 grid with some filled values. In other styles, grids with a scale of 16X16 are used. If we consider the most common puzzle type, it has a 9 by 9 grid and some places on the board have numbers, making it solvable; otherwise, it is difficult to solve because the number of empty cells increases. There are 81 cells on the board, which are divided into nine square girds of three-by-three sub-boards. It is necessary to arrange numbers from 1 to 9 on the board so that no row, column, or sub-board contains the same number twice to win the game. In most cases, a puzzle has only one solution, but some puzzles can have multiple solutions.

2.0 Background and constraints

In The Sudoku puzzle, there are 81 cells on the board, which are divided into nine square girds of three-by-three sub-boards. To win the game, numbers from 1 to 9 must be arranged on the board in such a way that no number appears more than once in any row, column, or box border. In most cases, a puzzle has only one solution, but some puzzles can have multiple solutions.

A backtracking algorithm is the simplest way to solve the algorithm. The algorithm attempts a value in the first empty cell. If that value does not work, it will try a different one. This step will be repeated from 1 to 9. When a number is found, the program moves on to the next empty cell and repeats the procedure. If the value cannot be placed in a particular cell, the algorithm will return to the previous cell and try again. This process continues until all the cells have been filled or all options have been exhausted.

3.0 Implementation of the Algorithm

Initially, the sudoku solving algorithm was implemented as a simple backtracking algorithm. It tries all the possible empty cell combinations. It is necessary to check if it is safe to assign a number before doing so. They did this by writing separate functions to check if the value they are trying to assign already exists in the row, column, or box border. It is safe to assign the number to that cell if the current value is not present in the row, column, or sector. It checks the next empty cell after assigning the number and assigns a number recursively until all the assignments lead to a solution. If the assignments do not yield a solution, it returns to the previous cell and assigns a new value before searching for the next cell. Similarly, it checks all possible combinations until it finds one that works.

The previous approach tried all the different combinations for each empty cell to find a solution. For example, let us say that the first cell of the sudoku board is empty and value “1” is not present in that row, column, and sector. To find the value “1” does not apply to that cell it has to check all the other combinations on the other cells to decide that and it tires all the combinations for the other cells too. Due to that reason, it takes a certain amount of time.

4.0 Optimization techniques

The implementation began with the creation of a simple backtracking algorithm and its optimization by lowering the number of node expansions in the sudoku search tree during the backtracking process. The steps that follow describe how the implemented algorithm works.

Initially, for every empty cell (which represents 0), a domain is defined as a set of numbers between 1 and N (N will be 9 or 16 according to the grid size), except for the numbers which violate the sudoku rules. When the modification occurs on the grid, the domain values of each cell are updated according to the newly assigned values.

Step 1

To find a solution, the previous method tried all the possible combinations for each empty cell. For example, let us say that the first cell of the sudoku board is empty and value "1" is not present in that row, column, and sector. To determine that the value "1" does not apply to that cell, it must examine all the other combinations of the other cells, and it exhausts all the other combinations as well. that reason, it takes a certain amount of time.

As a first step in the optimization process, I attempted to optimize the code by finding all possible values for each cell. The above-implemented algorithm only checks the possible values for each cell when filling the empty cells. This reduces the number of options the algorithm must test in each cell.

Input File Name	Initial execution time (s)
Input1.txt	0.056
Input2.txt	1.9360
Input3.txt	2.5630
Input_hex1.txt	>10
Input_hex2.txt	>10
Input_hex3.txt	>10

Table 01

Optimization technics

We could consider possible positions for a given missing value in one of the rows, columns, or boxes instead of considering possible values for a given empty cell.

We make the following modification to the solver:

1. Pick the cell with the smallest number of candidates. The puzzle is solved if no such cell can be found.
2. Look for missing values in sets (rows, columns, and boxes) and count how many positions they could occupy. Determine the set and value that has the fewest possible positions.
3. Proceed to step 4 if the set-search result yields several positions that are less than the number of candidate values found in step 1. Otherwise, proceed to step 5.
4. Recursively solve by filling the identified value in each candidate position in the set. Signal failure to the caller if all candidate positions have been filled.
5. Place the value in the cell for each candidate value identified in step 1 and try to recursively solve the puzzle. Signal failure if all candidate values have been exhausted.

The algorithm is essentially the same, with the exception that we try the set-oriented approach if it results in a lower branch factor.

This means that hidden singles or tuples produce branch factors that are like their empty counterparts.

Making this change frequently has a significant impact on the results of difficulty estimations.

Input File Name	Initial execution time (s)
Input1.txt	0.016
Input2.txt	0.042
Input3.txt	0.169
Input_hex1.txt	0.170
Input_hex2.txt	0.175
Input_hex3.txt	0.173

Table 02

5.0 Challenges faced

1. Because the backtracking algorithm takes longer, it was necessary to optimize it to speed up the process.
2. Choosing the right data structures.
3. Identify runtime errors by debugging the code.

6.0 Limitations

The following are some of the drawbacks of the current algorithm.

1. The current algorithm is limited to Sudoku puzzles with 9 * 9 and 16 * 16 numbers.
2. The user must use the command line to provide input.
3. Some operations currently use nested loops, which can slightly increase the algorithm's running time.

7.0 Future Improvements

Future enhancements could include the following.

1. The algorithm's efficiency can be improved even more by making it use less memory space, which also reduces computation time.
2. Extend the algorithm to solve Sudoku puzzles of various sizes.
3. Using optimization techniques, remove the nested loops.
4. Design a user interface in a graphical format.