Tanner Hart
411 Final Report

Sudoku Solver Final Report: An Overview

Introduction:

Sudoku is a highly popular logic-based puzzle game that has captivated millions of enthusiasts worldwide. Beyond its entertainment value, Sudoku serves as an excellent problem for algorithmic exploration, particularly in the realm of constraint satisfaction problems (CSPs). The Sudoku-solving program discussed in this report uses a recursive backtracking method to efficiently solve Sudoku puzzles as well as being used to validate their correctness. Additionally, the program automates testing using input and output files, with the primary limitation being that I was not able to get an efficient valid board generating algorithm working. This report delves into the motivation, algorithmic design, implementation details, and runtime analysis of the program.

The Sudoku solving program was developed to address two primary objectives: solving valid Sudoku puzzles and ensuring input puzzles adhere to the rules of Sudoku (aka are valid boards). The importance of solving such puzzles extends beyond entertainment, as Sudoku embodies CSPs that are widely applicable in artificial intelligence, optimization, and research. Solving Sudoku puzzles algorithmically allows developers to explore and refine techniques that are generalizable to broader CSPs, such as scheduling problems, path optimization, and resource allocation, among many others.

This program implements a simple backtracking approach, which explores all possible configurations of the Sudoku grid to find a valid solution one cell at a time. By combining simplicity and effectiveness, backtracking ensures that the program is both efficient and easy to understand. To ensure thorough testing, the program accepts .in files containing Sudoku puzzles, solves them, and compares the output against expected .out files. This approach not only validates the program's correctness but also highlights its performance under various conditions. Although no testing framework was implemented, nor a simple tester like the ones used throughout the semester, I instead opted to print the results to the console and allow the user to view the board side by side and compare.

Algorithm Intuition:

At the core of the program is the backtracking algorithm, a powerful recursive strategy for solving problems. The algorithm begins by iterating through the Sudoku grid to locate empty cells. For each empty cell, it attempts to place a number between 1 and 9 that satisfies the constraints of Sudoku:

- **Row Constraint**: No two numbers in the same row can be identical.
- **Column Constraint**: No two numbers in the same column can be identical.
- **Subgrid Constraint**: Each 3x3 subgrid must contain unique numbers.

The program explores each potential placement incrementally. If a valid placement is found, the algorithm recurses to solve the remaining grid. If no valid placement exists, the algorithm backtracks by resetting the current cell and attempting alternative configurations. This process ensures that every possible solution is explored systematically until a valid configuration is found or all possibilities are exhausted.

Implementation Details

The program's implementation is split into two primary functions: *solveBoard* and *isValidPlacement*. The *solveBoard* function employs recursive backtracking to fill the Sudoku grid, while *isValidPlacement* ensures that each placement adheres to Sudoku's rules.

The pseudocode for *solveBoard* is as follows:

1. Iterate through each row and column of the grid.
2. If an empty cell is found:
    - For each number from 1 to 9, check its validity using *isValidPlacement*.
    - If valid, place the number in the cell and recursively attempt to solve the grid.
    - If the recursion fails, backtrack by resetting the cell.
3. If no empty cells remain, return true (indicating the grid is solved).

```
function solveBoard(board):
    for each row in board:
        for each column in row:
            if board[row][col] is empty:
                for number in 1 to 9:
                    if isValidPlacement(board, number, row, col):
                        board[row][col] = number
                        if solveBoard(board):
                            return true
                        board[row][col] = 0  // Backtrack
                return false  // No valid number found
    return true  // Solved

function isValidPlacement(board, number, row, col):
    return number is not in board[row]
           and number is not in board[col]
           and number is not in 3x3 subgrid of (row, col)
```

The *isValidPlacement* function checks the row, column, and subgrid constraints efficiently by iterating over the relevant sections of the grid. This modular design ensures that the program is easy to understand and modify.

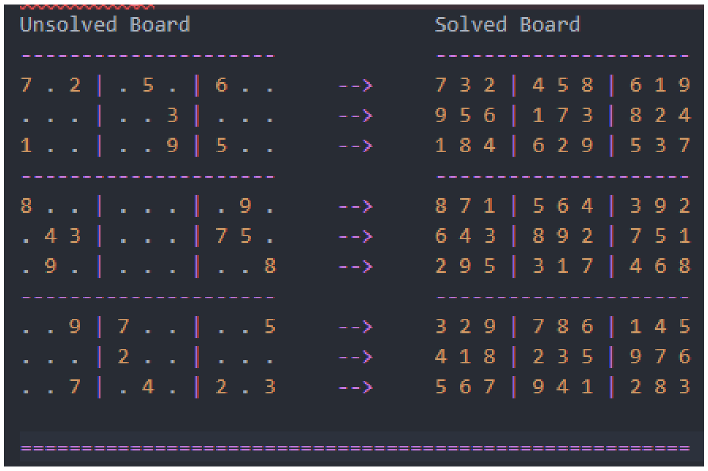
Runtime Analysis

The backtracking algorithm used in the program is inherently exponential in its worst-case time complexity, O(9^n), where n is the number of empty cells in the grid. This complexity arises because each empty cell can have up to 9 possible values, resulting in many possibilities. However, several factors mitigate this exponential growth in practice:

1. **Constraint Pruning**: The *isValidPlacement* function eliminates invalid configurations early, reducing the number of recursive calls.
2. **Structured Inputs**: Sudoku puzzles are highly structured, often requiring far fewer recursive steps than the worst-case scenario.
3. **Early Termination**: For well-designed puzzles, the algorithm often finds a solution before exploring all possibilities.

Results

For each test, the program correctly solved the puzzle and produced outputs matching the expected results. Additionally, the program handles unsolvable puzzles, indicating that the input was invalid without entering infinite loops or producing incorrect results. The side by side print out of the unsolved and solved boards further enhances the program's usability. Users can visually compare the input and output, ensuring the program's correctness.



```
Unsolved Board                      Solved Board
---------------------               ---------------------
7 . 2 | . 5 . | 6 . .       -->     7 3 2 | 4 5 8 | 6 1 9
. . . | . . 3 | . . .       -->     9 5 6 | 1 7 3 | 8 2 4
1 . . | . . 9 | 5 . .       -->     1 8 4 | 6 2 9 | 5 3 7
---------------------               ---------------------
8 . . | . . . | . 9 .       -->     8 7 1 | 5 6 4 | 3 9 2
. 4 3 | . . . | 7 5 .       -->     6 4 3 | 8 9 2 | 7 5 1
. 9 . | . . . | . . 8       -->     2 9 5 | 3 1 7 | 4 6 8
---------------------               ---------------------
. . 9 | 7 . . | . . 5       -->     3 2 9 | 7 8 6 | 1 4 5
. . . | 2 . . | . . .       -->     4 1 8 | 2 3 5 | 9 7 6
. . 7 | . 4 . | 2 . 3       -->     5 6 7 | 9 4 1 | 2 8 3

=======================================================
```

Conclusion

This Sudoku solving program demonstrates the simplicity and power of recursive backtracking algorithms in solving constraint satisfaction problems. By combining simplicity, modularity, and robustness, the program effectively solves Sudoku puzzles of varying difficulty.

Despite its exponential worst-case complexity, the program's runtime remains efficient for real-world inputs due to constraint pruning and the structured nature of Sudoku puzzles. Future additions would include some of the features that I did not quite get implemented correctly into this version. I would still like to include a valid board generator that is efficient. Having this would allow me to then take input from the user on how many boards to run and would allow me to quickly test the program with varying input sizes, highlighting the efficiency. These would be cool features to include in the next version of the program.