Sudoku Solver

CE223 - John Hitti & Audra Benally

23 November 2020

Introduction

For this final project we chose to code a sudoku puzzle solver. The algorithm method chosen was a backtracking algorithm. This algorithm recursively traverses the entire 9x9 puzzle board. It tries potential solutions and backtracks when the sudoku rules are not met. This puzzle solver is displayed using the Tkinter GUI library which is included with python.

GUI: Tkinter

The tkinter package is Python's standard interface to the TK GUI toolkit. This toolkit is a simple yet powerful way to provide a GUI to our algorithm. This greatly increases the algorithm's usability. It also allows us to create a visual representation of the puzzle that can easily and intuitively be interacted with.

The Algorithm

The algorithm backtracks using the recursion method and uses if statements to track the location of a given position. The primary function is called "mainSolutionAlg" and has two arguments that communicate an (i,j) location on the sudoku board. The if statements determine whether: the space is taken or empty, the location needs to move to the next row, or the sudoku puzzle has been solved. The recursion aspect of the algorithm iterates through the location arguments; once an empty cell is filled, the algorithm is called with parameters for the next location. The backtracking is implemented with a line that resets the cell right below the recursion call.

The aim of the algorithm is to find an empty space, iterate from one through nine, assign the first value that works, then recursively move to the next space. It starts from the top left of the board and moves all the way through to the bottom right. When a space comes up that does not work with one through nine, the algorithm kicks out of the current function space, and resets the previous space to 0. Then, within that recently reset space, the iteration continues from where it left

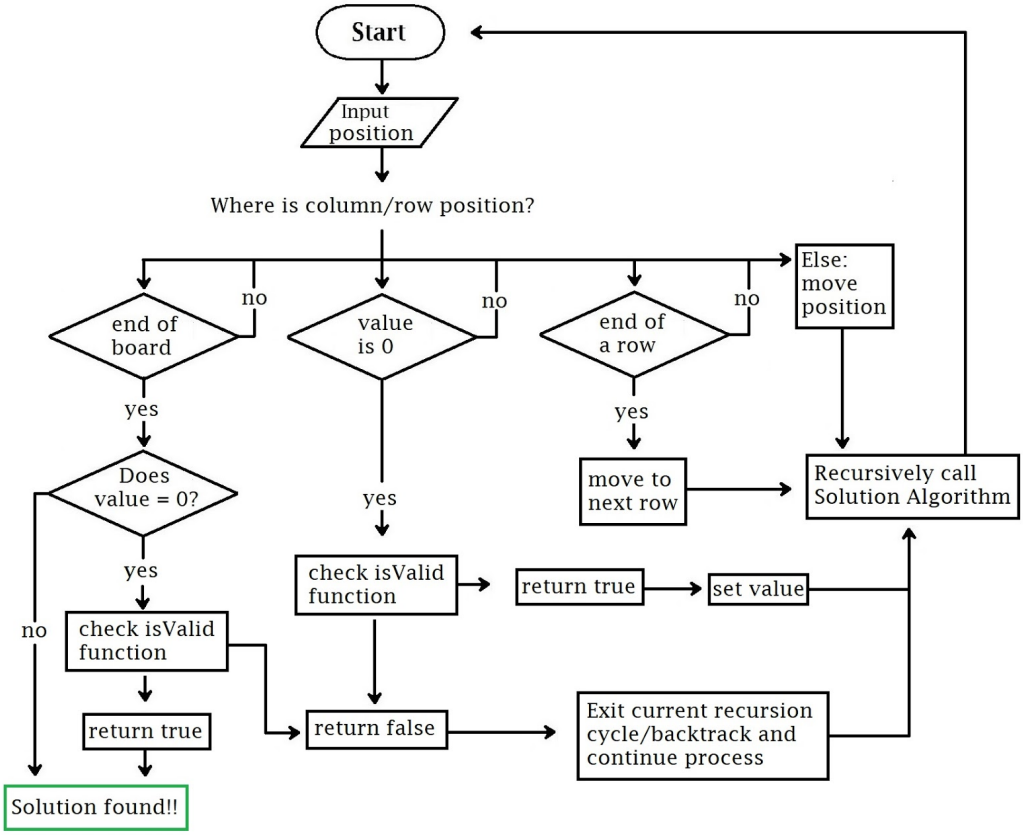
off. If the iteration loop gets to 9 without any numbers working, the function kicks out again. Like before, the previous space is emptied before it continues its iteration to see if any other number will work. The idea is that once the end of the board is reached the puzzle must be solved. Therefore, when the algorithm parameters point to the last space and the space is not equal to zero (i.e. empty), the algorithm makes a deep copy of the array to a display array that cooperates with the python Tkinter GUI. There can be one or multiple solution arrays that can be stored in the display array.

Another important solving function in the code that sits separately from the main algorithm is called the "isValid" function. This function works with three arguments that communicate the (i,j) position on the board, and the value that needs to be validated. The isValid function is used to check the rules of Sudoku. It does this by checking for potential duplicate values in the row, column, and 3x3 block of the given board location. If any of these is equal to the given value, the isValid function returns false. When this function returns true, the number is assigned on the board.

```
Main Solution Algorithm:
Input: i = column and j = row
Check value of position.
    If: position is at the end of the board
        If: the value does not equal 0
            Solution is found!
        Else: check isValid function
            If isValid is true then Solution is found!
    If: position is at the end of a row
        Recursively call solution algorithm and move to the next row
    If: position value is 0
        Check if value can be added to this position
        If isValid returns true
            Set position value, recursively call solution algorithm
            Below recursive call, set value to 0 in case algorithm needs to
            backtrack
    Else: if position already has a value
        Recursively call solution algorithm to move to the next position
```

Figure 1. Backtracking Algorithm Pseudocode.

Figure 2. Backtracking Algorithm Flow Chart.



Data Structures

In this project we used a 3-dimensional array as our primary data type for storing solutions to the sudoku puzzles. Each solution was stored in a nine by nine list in Python and then each solution was appended to a different list where all our solutions were stored. This creates a three dimensional array of solutions. Variables in these lists were stored as integers, however, there was a separate list which stored the currently displayed solution to the gui. This list stored the values as a tkinter native variable class named “StringVar”.

Runtime Analysis

Using a brute force method to solve a sudoku puzzle would yield a time complexity of $O(N^{(N^2)})$ or $2*10^{77}$ (for a 9x9 puzzle). Our backtracking method greatly reduces this to $O(N^X)$ where X is the amount of blank spaces in the puzzle.

The time complexity is affected by the number of blank spaces in a puzzle. Therefore, more difficult puzzles may take longer to complete. This will also cause the amount of data required to store the solutions to rise as more solutions will likely be present.
The space complexity of both algorithms is O(X*S) where ‘X’ is the amount of blank spaces in the puzzle and ‘S’ is the amount of solutions. In both cases this is trivial as finding enough solutions to cause concern for storage space would take a long time.

Discussion

Overall this algorithm is very effective in finding multiple solutions to puzzles. A good sudoku puzzle will only have one solution, therefore, a possible application to this program could be verifying that puzzles for this condition. Possible improvements could be made to the GUI to allow for faster puzzle entry. A file interface that has the potential to export solutions to a text file would also benefit the program. The algorithm could be improved by starting with the most populated area of puzzles. This would keep the backtracking down to a minimum and allow for a quicker solution.

This project was a great conclusion to the Data Structures and Algorithms class and tied together many of the skills learned throughout the course.