Sudoku using Backtracking algorithm

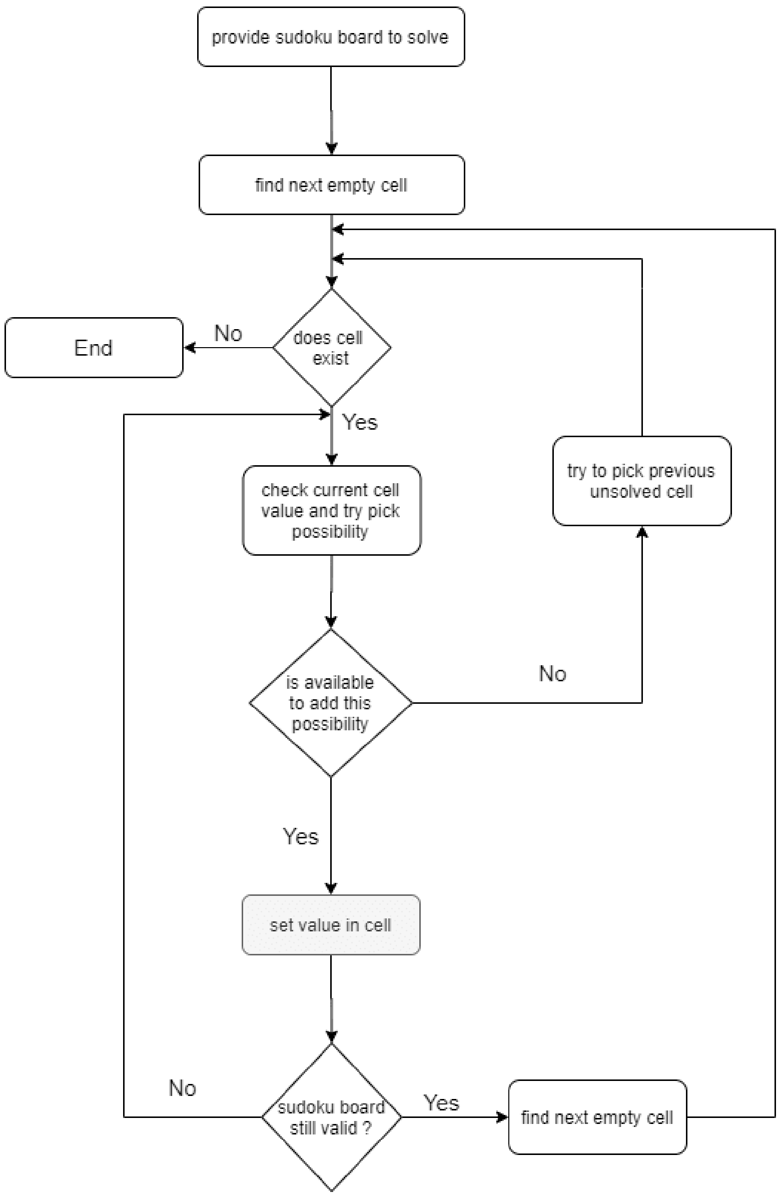
Introduction

- Backtracking is a kind of brute-force approach which comes into picture when solving a problem requires considering multiple choices as we don't know which choice is correct and we try to solve the problem using trail and error method considering one choice at a time until required answer is obtained.

Idea

- *This algorithm visits all the empty cells in some order, filling the digits incrementally, backtracks when a number is not found to be valid. The program starts by filling '1' in the first cell if there are no violations, then advances to the second cell and places '1' incase violation occurs then increments and places '2'.It checks incrementally in this way and when none of the '1' to '9' numbers fit in this cell then the algorithm leaves the cell blank and goes back to the previous cell. The value in that cell is incremented and this process continues until all the cells are filled.*

Algorithm flowchart


provide sudoku board to solve
find next empty cell
End
No
does cell exist
Yes
check current cell value and try pick possibility
try to pick previous unsolved cell
is available to add this possibility
No
Yes
set value in cell
No
sudoku board still valid ?
Yes
find next empty cell

STEPS:

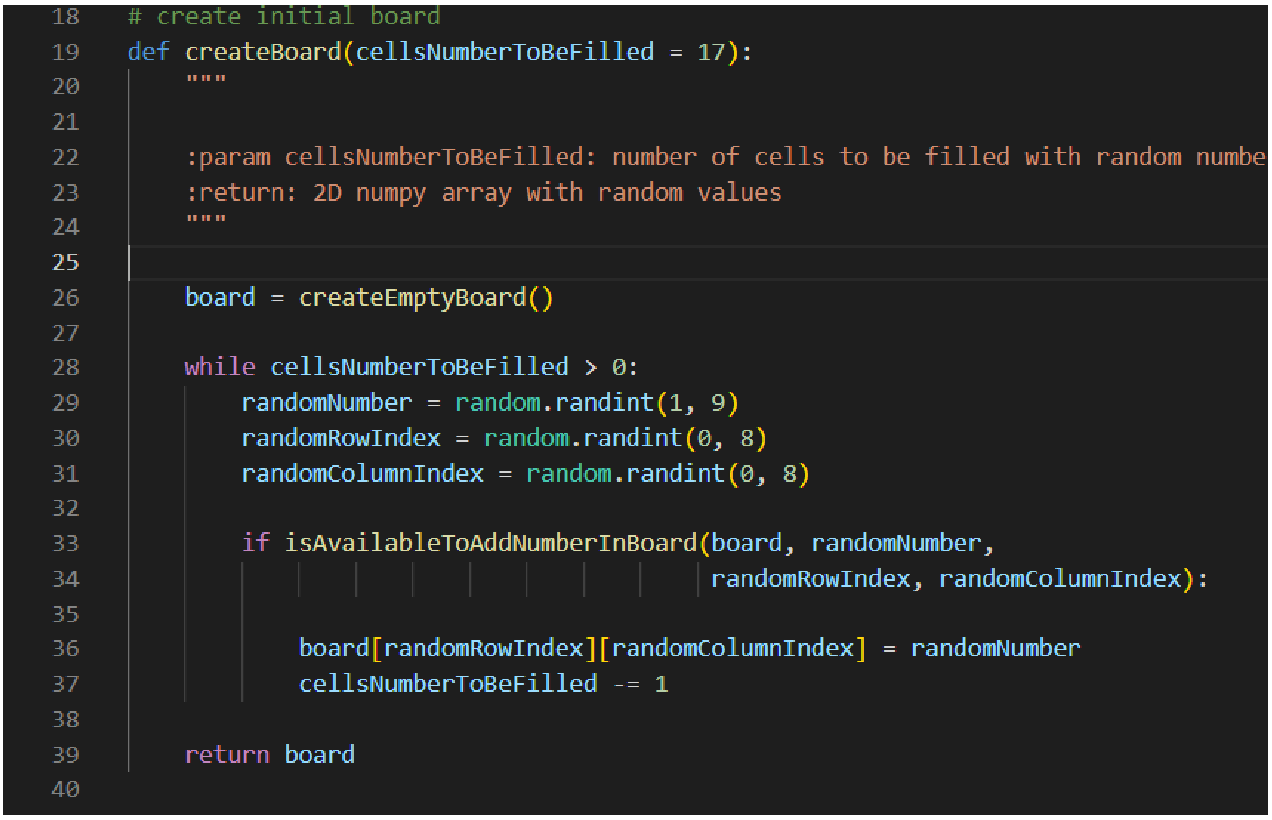
-We start by finding an unfilled cell(i,j).
-If all the cells are filled then a valid sudoku is obtained.
-We try to place every number between 1 to 9 in the unfilled cell.

-Before placing we check for the constraints by checking the current row,current column and current 3×3 submatrix.
-If any of the any of the constraint becomes false we'll not place that number at (i,j).
-If all the constraints are true then we'll place that number at (i,j) and then repeat the process by finding an unfilled cell.
-If at any point none of the numbers can be placed at (i,j) then we've to backtrack and change the values for already visited cells

first create sudoku board

1. create 2d empty array with numpy -> array of zeros zero means cell is empty
2. fill the created board with random values to start
3. save locations that have starting values to make them never change

Implementation



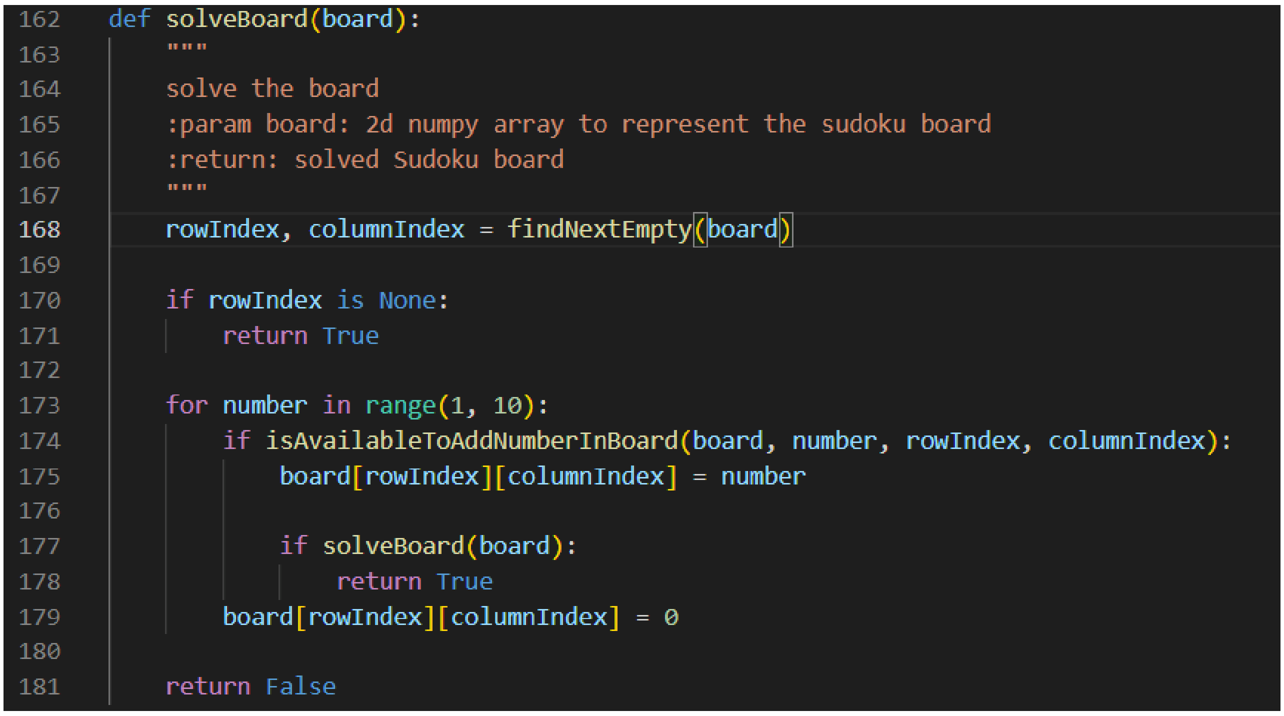
```python
# create initial board
def createBoard(cellsNumberToBeFilled = 17):
    """


    :param cellsNumberToBeFilled: number of cells to be filled with random numbe
    :return: 2D numpy array with random values
    """

    board = createEmptyBoard()

    while cellsNumberToBeFilled > 0:
        randomNumber = random.randint(1, 9)
        randomRowIndex = random.randint(0, 8)
        randomColumnIndex = random.randint(0, 8)

        if isAvailableToAddNumberInBoard(board, randomNumber,
                                        randomRowIndex, randomColumnIndex):

            board[randomRowIndex][randomColumnIndex] = randomNumber
            cellsNumberToBeFilled -= 1

    return board

```

Output

```
0  0  3  |0  2  0  |6  0  0
9  0  0  |3  0  5  |0  0  1
0  0  1  |8  0  6  |4  0  0
------+------+------
0  0  8  |1  0  2  |9  0  0
7  0  0  |0  0  0  |0  0  8
0  0  6  |7  0  8  |2  0  0
------+------+------
0  0  2  |6  0  9  |5  0  0
8  0  0  |2  0  3  |0  0  9
0  0  5  |0  1  0  |3  0  0
```

second solve board

Implementation



```python
def solveBoard(board):
    """
    solve the board
    :param board: 2d numpy array to represent the sudoku board
    :return: solved Sudoku board
    """
    rowIndex, columnIndex = findNextEmpty(board)

    if rowIndex is None:
        return True

    for number in range(1, 10):
        if isAvailableToAddNumberInBoard(board, number, rowIndex, columnIndex):
            board[rowIndex][columnIndex] = number

            if solveBoard(board):
                return True
        board[rowIndex][columnIndex] = 0

    return False
```

Output

```
9 6 0 | 5 0 7 | 9 5 2
1 9 3 | 9 3 4 | 5 4 2
4 9 7 | 2 3 0 | 1 3 1
------+-------+------
3 0 1 | 6 7 3 | 9 8 3
2 4 5 | 7 8 7 | 8 0 8
0 1 4 | 9 3 9 | 3 9 6
------+-------+------
6 1 2 | 8 7 6 | 5 0 1
4 3 9 | 3 0 8 | 5 6 6
4 1 7 | 5 9 9 | 3 1 7
```

Time and Space Complexity

Consider there are m unfilled cells in the sudoku

- Worst Case Time Complexity: **O(9m)**
- Average Case Time Complexity: **O(9m)**
- Best Case Time Complexity: **O(m2)** [when the number of backtracking steps are minimized]
- the space complexity would be **O(m)** [there can be **m** function calls in the recursion stack]

Shared folder

https://github.com/Boodyahmedhamdy/SudokuSolver

Development platform

- developed using **https://jupyter.org/**
- language **python**
- libraries **numpy** and **random**