KATHMANDU UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MINI PROJECT REPORT ON

SUDOKU SOLVER USING BACKTRACKING

by:

Bimal Timilsina (62)

November 2020

1 Sudoku

Sudoku is a logic based combinatorial number-placement puzzle. Here in this project I am using a 9*9 grid sudoku. In the 9*9 Sudoku, each row and column can only contains number from 1 to 9 without repetition. There will be 9 boxes of size 3 * 3 which must contains numbers from 1 to 9 without repetition as before. An, example is given in below figure.

4	3	5	2	6	9	7	8	1
6	8	2	5	7	1	4	9	3
1	9	7	8	3	4	5	6	2
8	2	6	1	9	5	3	4	7
3	7	4	6	8	2	9	1	5
9	5	1	7	4	3	6	2	8
5	1	9	3	2	6	8	7	4
2	4	8	9	5	7	1	3	6
7	6	3	4	1	8	2	5	9

Fig: A Solved Sudoku

Here in the above sudoku, the black numbers are the initial provided numbers and red numbers are the one we have to guess and solve the puzzle. Our program fills these red numbers in the sudoku. The bold line denotes 3*3 grids.

2 Backtracking

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.

While solving sudoku using backtracking, the following steps are followed:

i. At first we find the empty places in sudoku to fill the numbers.
ii. Now, we start to fill the numbers from 1 to 9 in that position, and check if that number already exists in the same column, row and 3*3 box,
   a. If the number already exists,
      We discard that number and add another number there and again check if the sudoku is valid or not after adding that number.
   b. If the number do not exists,
      The board is valid, and we move to another empty position given by step i.
iii. We fill the numbers recursively and when the box becomes invalid, we backtrack and fill another number, i.e. we do not move forward until that position is filled.
iv. Finally, the solver completely fills the numbers in sudoku puzzle. But if the sudoku is not solvable, then the algorithm returns False.

2.1 Algorithm

```
find_empty_spaces(board):
for i in rows:
    for j in columns:
         if board[i][j] == 0:
              return (i,j)

is_valid(board, number, position):

if number in board[position[0]]:
    return False #number already exists

if number in columns:
    return False # number already exists in columns

if number in 3*3 block:
    return False # Number already exists in that block

# if all checks completed then the number at that position is
valid
Return True


Solve(board):
    Position = Find_empty_spaces(board)
```

```
For i=1 to 9:
    If is_valid(board, I, position):
        Board[position] = i

    If solve(board): # If board can be solved after filling that number
        Return True

    # Board cannot solved by that number so, reset that number and move accordingly
    Board[position] = 0

Return False # if board cannot be solved
```

2.2 Time Complexity Analysis

Here the recursion tree of sudoku problem using backtracking yields total of 9 branches since we have 9 possible options at the first position, The number of possibility decreases after each level by 1 since a number will be already filled in previous position.

Here, let the sudoku is of grid size **N** * **N** where **N** is a perfect square.

Let M = N * N then the time complexity will be: O(9^M)

Since, for every empty spaces there will be 9 possible options, the upper bound time complexity will be O(9^M) but, when using backtracking, some of the possibilities are pruned in the middle, so the actual time taken will be lesser than O(9^M).

2.3 SOURCE CODE

sudoku_solver.py

```
class SudokuSolver:
    def __init__(self, board):
        self.board = board
        self.n = len(self.board[0])

    def find_empty_position(self):
        """
            Returns the positions(row, col) of empty cells i.e cells to be filled.
        """
        for row in range(self.n):
            for col in range(self.n):
                if self.board[row][col] == 0:
                    return (row, col)
        return None
    def is_valid(self, num, pos: tuple):
        """
        Checks if the number we are about to fill is valid or not i
n that position.
        """
        row, col = pos
        # Check Row
        if num in self.board[row]:
            return False

        # Check column
        colum_vals = [self.board[idx][col] for idx in range(self.n)
]
        if num in colum_vals:
            return False

        # Check 3*3 Grid
        start_x = (row // 3) * 3
        start_y = (col // 3) * 3

        box_vals = [(self.board[row][col] for col in range(
            start_y, start_y+3)) for row in range(start_x, start_y+
3)]

        if num in box_vals:
            return False

        return True

    def solve(self):
        """
        Solves the Sudoku recusively, if sudoku cannot be solved re
turns False
        """
        pos = self.find_empty_position()
        if not pos:
            return True  # Board already solved
        else:
            row, col = pos

        for num in range(1, 10):
            if self.is_valid(num, pos):
                self.board[row][col] = num

                # if the newly added value helps to solve the probl
em, solve it
                if self.solve():
                    return True

                # if after adding the board is not solved, reset th
at value and go back
                self.board[row][col] = 0
        return False

    def print_board(self):
        """
        Prints the board.
        """
        print("+" + "---+" * self.n)
        for i, row in enumerate(board):
            print(("|" + " {}   {}   {} |" * 3).format(*
                                                       [x for x in
row]))
            if i % 3 == 2:
                print("+" + "---+" * self.n)
            else:
                print("+" + "   +" * self.n)

if __name__ == "__main__":
    board = [[0, 0, 0, 2, 6, 0, 7, 0, 1],
             [6, 8, 0, 0, 7, 0, 0, 9, 0],
             [1, 9, 0, 0, 0, 4, 5, 0, 0],
             [8, 2, 0, 1, 0, 0, 0, 4, 0],
             [3, 0, 4, 6, 0, 2, 9, 0, 0],
             [0, 5, 0, 0, 0, 3, 0, 2, 8],
             [0, 0, 9, 3, 0, 0, 0, 7, 4],
             [0, 4, 0, 0, 5, 0, 0, 3, 6],
             [7, 0, 3, 0, 1, 8, 0, 0, 0]]
    solver = SudokuSolver(board)
    solver.solve()
    solver.print_board()
```

test_sudoku.py

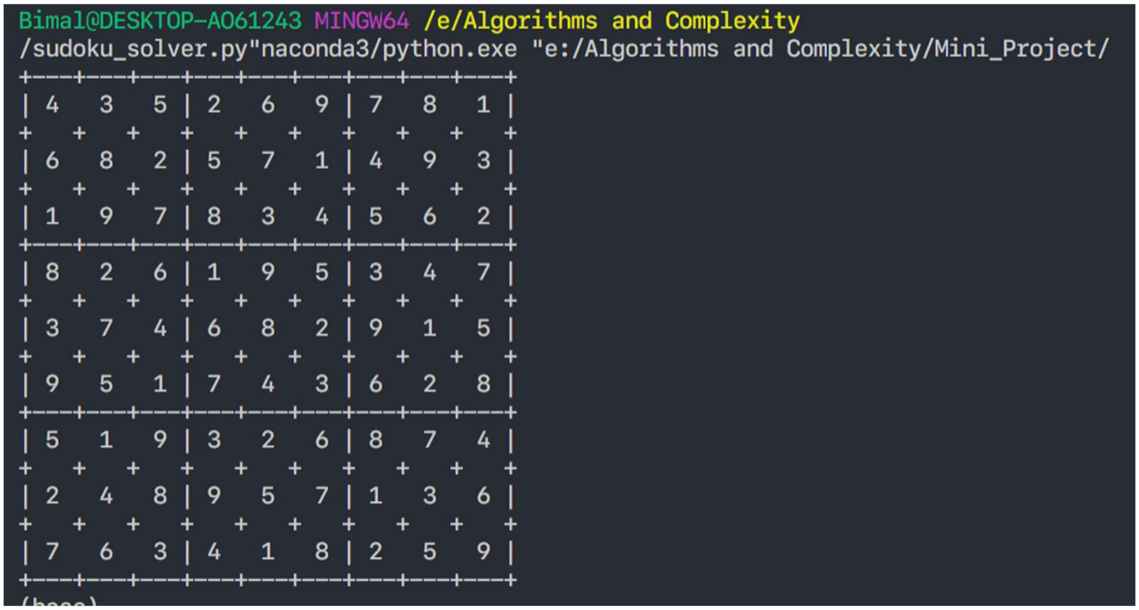
```
from sudoku_solver import SudokuSolver
import unittest

class TestSudoku(unittest.TestCase):
    def setUp(self):
        self.board = [[0, 0, 0, 2, 6, 0, 7, 0, 1],
                      [6, 8, 0, 0, 7, 0, 0, 9, 0],
                      [1, 9, 0, 0, 0, 4, 5, 0, 0],
                      [8, 2, 0, 1, 0, 0, 0, 4, 0],
                      [3, 0, 4, 6, 0, 2, 9, 0, 0],
                      [0, 5, 0, 0, 0, 3, 0, 2, 8],
                      [0, 0, 9, 3, 0, 0, 0, 7, 4],
                      [0, 4, 0, 0, 5, 0, 0, 3, 6],
                      [7, 0, 3, 0, 1, 8, 0, 0, 0]]
        self.solver = SudokuSolver(self.board)
        self.solver.solve()
        self.result = [i for i in range(1, 10)]
    def test_row(self):
        """
        Checks if every row contains value from 1 to 9 or not. Also
 checks if there are any duplicate items or not.
        """
        for i in range(len(self.solver.board[0])):
            self.assertListEqual(sorted(self.solver.board[i]), self
.result)

    def test_column(self):
        """
        Checks if every column contains value from 1 to 9 or not. A
lso checks if there are any duplicate items or not.
        """
        for i in range(len(self.solver.board[0])):
            res = [self.solver.board[j][i] for j in range(9)]
            self.assertListEqual(sorted(res), self.result)

    def test_box(self):
        """
        Checks if every 3*3 box contains value from 1 to 9 or not.
Also checks if there are any duplicate items or not.
        """
        for row in range(3):
            for col in range(3):
                start_x = row * 3
                start_y = col * 3
                vals = [self.solver.board[row][col] for col in rang
e(
                    start_y, start_y+3) for row in range(start_x, s
tart_x+3)]
                self.assertListEqual(sorted(vals), self.result)

if __name__ == "__main__":
    unittest.main()
```

2.4 OUTPUT

2.4.1 QUESTION

```
board =     [[0, 0, 0, 2, 6, 0, 7, 0, 1],
             [6, 8, 0, 0, 7, 0, 0, 9, 0],
             [1, 9, 0, 0, 0, 4, 5, 0, 0],
             [8, 2, 0, 1, 0, 0, 0, 4, 0],
             [3, 0, 4, 6, 0, 2, 9, 0, 0],
             [0, 5, 0, 0, 0, 3, 0, 2, 8],
             [0, 0, 9, 3, 0, 0, 0, 7, 4],
             [0, 4, 0, 0, 5, 0, 0, 3, 6],
             [7, 0, 3, 0, 1, 8, 0, 0, 0]]
```

2.4.2 SOLUTION:



```
Bimal@DESKTOP-AO61243 MINGW64 /e/Algorithms and Complexity
/sudoku_solver.py"naconda3/python.exe "e:/Algorithms and Complexity/Mini_Project/
+---+---+---+---+---+---+---+---+---+
| 4   3   5 | 2   6   9 | 7   8   1 |
+   +   +   +   +   +   +   +   +   +
| 6   8   2 | 5   7   1 | 4   9   3 |
+   +   +   +   +   +   +   +   +   +
| 1   9   7 | 8   3   4 | 5   6   2 |
+---+---+---+---+---+---+---+---+---+
| 8   2   6 | 1   9   5 | 3   4   7 |
+   +   +   +   +   +   +   +   +   +
| 3   7   4 | 6   8   2 | 9   1   5 |
+   +   +   +   +   +   +   +   +   +
| 9   5   1 | 7   4   3 | 6   2   8 |
+---+---+---+---+---+---+---+---+---+
| 5   1   9 | 3   2   6 | 8   7   4 |
+   +   +   +   +   +   +   +   +   +
| 2   4   8 | 9   5   7 | 1   3   6 |
+   +   +   +   +   +   +   +   +   +
| 7   6   3 | 4   1   8 | 2   5   9 |
+---+---+---+---+---+---+---+---+---+
(base)
```