Review - 3

Date : 17-08-23

Sudoku Solver

1CR21CS151 Rohan N Karnkoti
1CR21CS214 Vismitha S

Under the guidance of:
Dr. Radhakrishnan
Associate Professor

CELEBRATING 25 YEARS
CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Department of Computer Science & Engineering

TITLE

Program to solve Sudoku Problems

PROBLEM STATEMENT

Hard sudoku problems are difficult to solve. However, with a help of a computer program even hard problems can be easily solved.

DETAILED REQUIREMENTS

- Knowledge of Python
- Understanding of Sudoku solving algorithm
- Understanding of tkinter library to implement graphical user interface to take input and display output.

SOFTWARE TOOLS USED

- Visual Studio Code
- Git
- GitHub
- Python 3.11
- Tkinter library

PROPOSED SOLUTION

Rules to solve a sudoku puzzle:

- Sudoku is played on a 9x9 grid divided into 3x3 boxes called regions.
- The goal is to fill the grid with digits from 1 to 9, ensuring that each row, each column, and each region contains all the digits from 1 to 9 exactly once.
- Initially, some cells in the grid are pre-filled with numbers (clues), and the player's task is to fill in the remaining empty cells.
- The puzzle is solved when all cells are filled, and the three rules mentioned above are satisfied.
- When filling the empty cells, each number must be unique within its row, column, and region.

PROPOSED SOLUTION

Constraint Propagation:

- Constraint propagation is a deduction-based technique used to reduce the solution space by applying constraints to eliminate possibilities and reveal new values.
- It involves iteratively applying logical rules and constraints to deduce information about the puzzle.
- In the context of Sudoku, constraint propagation looks at the known numbers and uses the constraints imposed by those numbers to narrow down possible values for empty cells.
- Techniques like Naked Single (a cell has only one possible value) and Hidden Single (a number appears only once in a row, column, or box) are examples of constraint propagation strategies used in Sudoku.
- Constraint propagation aims to fill in as many cells as possible without resorting to exhaustive search

TOTAL HOUR & WORKS

This project will require around 30 days.
15 days for one person.

DESIGN

Constraint Propagation:

- Constraint propagation is a deduction-based technique used to reduce the solution space by applying constraints to eliminate possibilities and reveal new values.
- It involves iteratively applying logical rules and constraints to deduce information about the puzzle.
- In the context of Sudoku, constraint propagation looks at the known numbers and uses the constraints imposed by those numbers to narrow down possible values for empty cells.
- Techniques like Naked Single (a cell has only one possible value) and Hidden Single (a number appears only once in a row, column, or box) are examples of constraint propagation strategies used in Sudoku.
- Constraint propagation aims to fill in as many cells as possible without resorting to exhaustive search

IMPLEMENTATION

- **is_valid(board, row, col, num):**

  This function checks if placing a number (`num`) at a specific cell (`row`, `col`) on the Sudoku board is a valid move.

- **find_empty_cell(board):**

  This function finds the first empty cell (cell with value 0) on the Sudoku board.

- **solve_sudoku(board):**

  This function uses constraint propagation and a simple backtracking approach (without recursion) to solve the Sudoku puzzle.

- **constraint_propagation(board):**

  This function uses constraint propagation to iteratively solve the Sudoku puzzle without backtracking.

SOURCE CODE

```python
import tkinter as tk

def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

def find_empty_cell(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None
```

SOURCE CODE

```
23  def solve_sudoku(board):
24      empty_cell = find_empty_cell(board)
25      if not empty_cell:
26          return True
27
28      row, col = empty_cell
29      for num in range(1, 10):
30          if is_valid(board, row, col, num):
31              board[row][col] = num
32              if solve_sudoku(board):
33                  return True
34              board[row][col] = 0
35
36      return False
37
```

SOURCE CODE

```
38  def constraint_propagation(board):
39      changed = True
40      while changed:
41          changed = False
42          for row in range(9):
43              for col in range(9):
44                  if board[row][col] == 0:
45                      possibilities = set(range(1, 10))
46                      possibilities -= set(board[row][j] for j in range(9))
47                      possibilities -= set(board[i][col] for i in range(9))
48                      start_row, start_col = 3 * (row // 3), 3 * (col // 3)
49                      possibilities -= set(board[start_row + i][start_col + j] for i in range(3) for j in range(3))
50                      if len(possibilities) == 1:
51                          board[row][col] = possibilities.pop()
52                          changed = True
53      return board
54
55  def solve_button_handler():
56      input_board = []
57      for i in range(9):
58          row = []
59          for j in range(9):
60              cell_value = int(cell_entries[i][j].get())
61              row.append(cell_value)
62          input_board.append(row)
```

SOURCE CODE

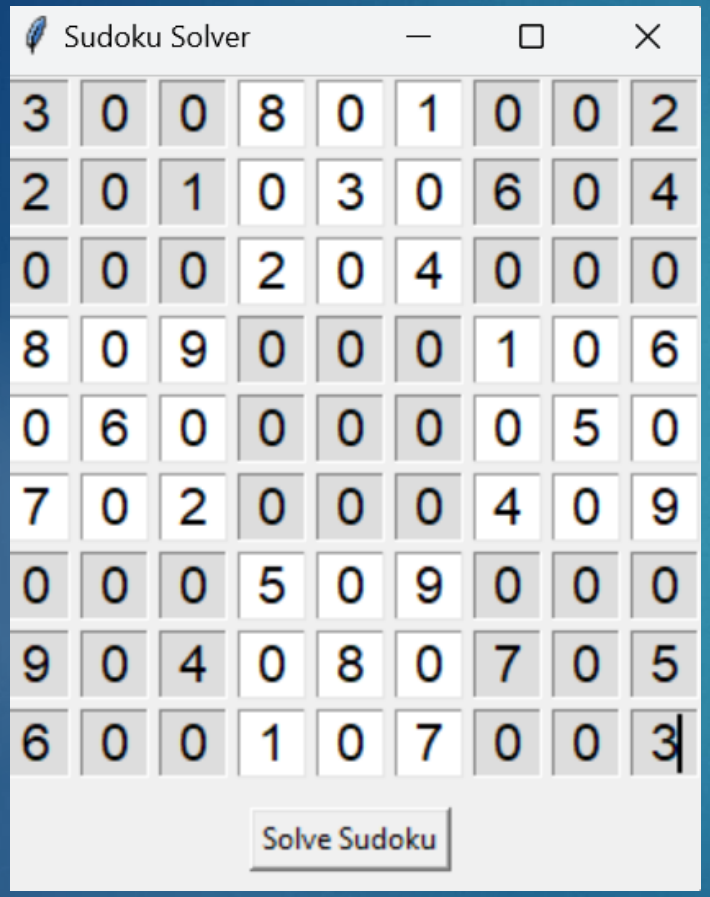
```
    constraint_propagation(input_board)
    if solve_sudoku(input_board):
        for i in range(9):
            for j in range(9):
                cell_entries[i][j].delete(0, tk.END)
                cell_entries[i][j].insert(0, str(input_board[i][j]))

def create_entry_cells(root):
    cell_entries = []
    for i in range(9):
        row_entries = []
        for j in range(9):
            entry = tk.Entry(root, width=2, font=("Arial", 16), justify="center")
            entry.grid(row=i, column=j, padx=2, pady=2)
            row_entries.append(entry)
        cell_entries.append(row_entries)
    return cell_entries
```

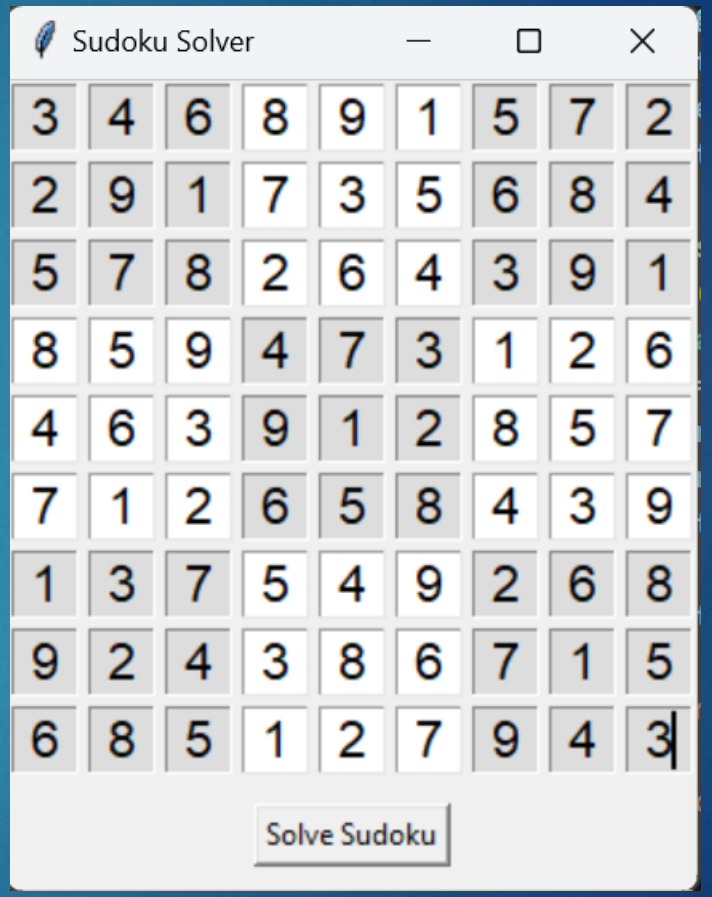
SOURCE CODE

```
81
82  def highlight_boxes(cell_entries):
83      for i in range(9):
84          for j in range(9):
85              entry = cell_entries[i][j]
86              entry_row, entry_col = i // 3, j // 3
87              if entry_row % 2 == entry_col % 2:
88                  entry.config(bg="#DDDDDD")
89              else:
90                  entry.config(bg="white")
91
92  if __name__ == "__main__":
93      root = tk.Tk()
94      root.title("Sudoku Solver")
95
96      cell_entries = create_entry_cells(root)
97      highlight_boxes(cell_entries)
98
99      solve_button = tk.Button(root, text="Solve Sudoku", command=solve_button_handler)
100     solve_button.grid(row=9, column=0, columnspan=9, padx=10, pady=10)
101
102     root.mainloop()
103
```

OUTPUT SCREENS



INPUT



OUTPUT

Thank you