S. Islam, E. Reamer
Professor Bankston
CMPS 1100
09 Sept 2024

Detailed Project Proposal

Description:

We will create a program to solve Sudoku-style puzzles of varying size. Classic Sudoku puzzles are 9x9 grids, partially filled by numbers ranging 1-9. The objective of the solver is to fill each square on the grid with a number (usually range 1-9) unique to its row, column, and (typically 3x3) box. While the difficulty of a classic Sudoku puzzle typically varies inversely with the number of squares that come pre-filled, the same rules can be implemented in any square of squares. Our program should be able to return a solved Sudoku-style matrix of any size or difficulty.

As of now, the user will input the Sudoku puzzle as a nested list, with zeros representing the assigned squares. Example: [[3, 0, 6, 5, 0, 8, 4, 0, 0], [5, 2, 0, 0, 0, 0, 0, 0, 0], [0, 8, 7, 0, 0, 0, 0, 3, 1], [0, 0, 3, 0, 1, 0, 0, 8, 0], [9, 0, 0, 8, 6, 3, 0, 0, 5], [0, 5, 0, 0, 9, 0, 6, 0, 0], [1, 3, 0, 0, 0, 0, 2, 5, 0], [0, 0, 0, 0, 0, 0, 0, 7, 4], [0, 0, 5, 2, 0, 6, 3, 0, 0]]. The program will first check that the lists given create a valid Sudoku puzzle. To do this, it must first evaluate the size of the matrix. If the matrix is not a square (#of rows =/= #of columns), the program will return unsolvable. The number of columns/rows will also be the range of numbers allowed (9x9 is range 9, 16x16 is range 16, etc.) The square root of the range will give it the dimensions of the boxes with in it (9x9 is 3x3, 16x16 is 4x4, etc.) If the frequency of each existing number in each row, column, and box is more than 1, the program will return unsolvable. The program will then use a recursive function that goes to the each unassigned square, assigns it a number within the range and then, similar to the first function, check that it is still valid. The function should continue this mechanism until all unassigned squares are assigned a number unique to its row, column and square.

Useful Libraries:

https://www.geeksforgeeks.org/sudoku-backtracking-7/

https://www.askpython.com/python/examples/sudoku-solver-in-python

https://github.com/dhhruv/Sudoku-Solver

GitHub Repository:

https://github.com/CMPS1100-Sania-Ella-Final-Project/CMPS1100_FinalProject.git

LLM investigation:

I prompted Chat 3.5: Make a Sudoku Solver for any size sudoku puzzle. It returned the following code (also in repository). It seems to work well. I will have to look further into a simpler input than the nested lists.

```
import math

class SudokuSolver:

    def __init__(self, board):

        self.board = board

        self.size = len(board)  # Size of the board (n x n)

        self.subgrid_size = int(math.sqrt(self.size))  # Size of the subgrids (k x k)

    def is_valid(self, row, col, num):

        # Check if 'num' is not in the current row

        if num in self.board[row]:

            return False

        # Check if 'num' is not in the current column

        for i in range(self.size):

            if self.board[i][col] == num:

                return False

        # Check if 'num' is not in the current subgrid

        subgrid_row_start = (row // self.subgrid_size) * self.subgrid_size

        subgrid_col_start = (col // self.subgrid_size) * self.subgrid_size


        for i in range(subgrid_row_start, subgrid_row_start + self.subgrid_size):

            for j in range(subgrid_col_start, subgrid_col_start + self.subgrid_size):

                if self.board[i][j] == num:

                    return False

        return True

    def find_empty(self):

        # Find an empty cell in the Sudoku grid (denoted by 0)

        for i in range(self.size):

            for j in range(self.size):

                if self.board[i][j] == 0:

                    return i, j

        return None

    def solve(self):

        empty = self.find_empty()
        if not empty:
            return True  # Puzzle solved
        row, col = empty
        for num in range(1, self.size + 1):  # Try numbers 1 through n
            if self.is_valid(row, col, num):
                self.board[row][col] = num
                if self.solve():
                    return True
                self.board[row][col] = 0  # Reset the cell
        return False
    def print_board(self):
        for row in self.board:
            print(" ".join(str(num) if num != 0 else "." for num in row))
# Example usage:
# Define a 9x9 Sudoku puzzle (0 denotes an empty cell)
board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]
solver = SudokuSolver(board)
if solver.solve():
    solver.print_board()
else:
    print("No solution exists.")
```