Sudoku Puzzle Generator and Solver

By: Jeet Padhya

Project Inspired by:

Mr. Gribbin & Dr. James Bailey

Version 1.0

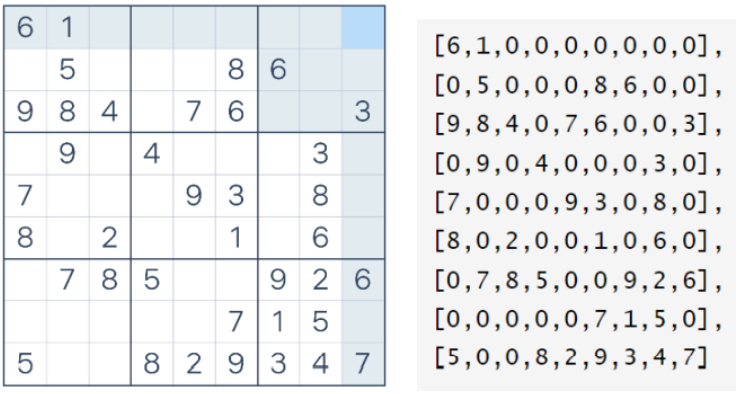
1. Introduction – Describing the Problem

A Sudoku puzzle is represented as a square grid of size n2×n2n^2 \times n^2n2×n2. Empty cells are represented by the value 0, while filled cells contain their respective numbers. The goal of the solver is to fill the empty cells with numbers from 1 to n2n^2n2 while following the rules of Sudoku.

The rules of the puzzle are:

1. Each row must contain every number from 1 to n^2 exactly once.
2. Each column must contain every number from 1 to n^2 exactly once.
3. Each n×n subgrid must contain every number from 1 to n^2 exactly once.

The difficulty of a Sudoku puzzle depends on how many values are removed from a completed grid and how constrained the remaining cells are. A valid puzzle must always have at least one solution.



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

Figure 1: Sudoku puzzle and the corresponding array representation used by the program.

2. Generating & Solving Process

2.1. User Input and Validation

Before generating or solving a Sudoku puzzle, the program must obtain valid input parameters from the user. These inputs determine the size of the grid and the difficulty level of the generated puzzle.

The subgrid size nnn is requested from the user, where the full Sudoku grid has dimensions ((n^2) x (n^2)). For example, an input of n = 3 produces a standard 9 × 9 Sudoku grid. The **`validInt`** function ensures that the input is numeric and that the value is at least 2. If the input is invalid or too small, the program displays an error message and terminates safely.

```python
def validInt(value):
    try:
        num = int(value)
        if num >= 2:
            return "ok", num
        else:
            return "too-small", None
    except ValueError:
        return "non-numeric", None
```

Listing 1: validInt Function (Python Code)

In addition to the grid size, the user is prompted to select a difficulty level. The accepted difficulty values are **easy, medium, hard, and extreme**. The **`validStr`** function checks whether the provided difficulty matches one of these options. If an invalid difficulty is entered, the program defaults to **medium**, ensuring that execution can continue without crashing.

```python
def validStr(value):
    valid_difficulties = ["easy", "medium", "hard", "extreme"]
    if value in valid_difficulties:
        return value
    else:
        return "medium"  # default difficulty
```

Listing 2: validStr Function (Python Code)

By validating both numeric and string inputs before proceeding, the program prevents runtime errors and guarantees that the puzzle generation process operates within defined and meaningful parameters.

2.2. Validity Checking

Before a number can be placed into the Sudoku grid, the program must verify that the placement follows all Sudoku rules. This is handled by the **`isValid`** function.

```python
def isValid(puzzle, n, row, col, value):
    # check row
    if value in puzzle[row]:
        return False

    # check column
    if value in puzzle[:, col]:      You, 22 hours ago • Update ma
        return False
    # check subgrid
    startRow = (row // n) * n
    startCol = (col // n) * n

    # check if the value already exists in the subgrid
    # subgrid starts at (startRow, startCol) and is n x n in size
    if value in puzzle[startRow:startRow+n, startCol:startCol+n]:
        return False  # value cannot be placed here
    # value is valid in row, column, and subgrid
    return True
```

Listing 3: isValid Function (Python Code)

For a given cell, the function checks three conditions:

1. The value does not already exist in the same row.
2. The value does not already exist in the same column.
3. The value does not already exist in the corresponding n×n subgrid.

The starting position of the subgrid is computed using integer division. By slicing the NumPy array to isolate the subgrid, the program can efficiently determine whether the value violates any Sudoku constraints. If any rule is broken, the function returns **`False`**; otherwise, the placement is valid.

2.3. Solving Algorithm (Backtracking)

The Sudoku solver uses a recursive backtracking algorithm implemented in the **`solveSudoku`** function. The algorithm searches the grid for an empty cell and attempts to fill it with a valid number from 1 to n^2.

```python
def solveSudoku(puzzle, n):
    size = n * n

    # loop through the grid to find an empty cell
    for row in range(size):
        for col in range(size):
            if puzzle[row, col] == 0:

                # try numbers in random order so puzzles differ
                numbers = np.arange(1, size + 1)
                np.random.shuffle(numbers)

                for value in numbers:
                    # check if the value is valid in this position
                    if isValid(puzzle, n, row, col, value):
                        puzzle[row, col] = value

                        # recursively attempt to solve the rest of the grid
                        if solveSudoku(puzzle, n):
                            return True
                        # undo placement if it leads to a dead end
                        puzzle[row, col] = 0

                # no valid number fits here, trigger backtracking
                return False
    # no empty cells left, puzzle is solved
    return True
```

Listing 4: solveSudoku Function (Python Code)

For each empty cell, candidate values are tested in random order to ensure variation between solutions. A placement is considered valid only if it does not violate Sudoku rules for rows, columns, and n×n subgrids, which is checked using the **`isValid`** function.

If a valid value is placed, the solver recursively continues to the next empty cell. If no valid continuation exists, the algorithm backtracks by resetting the cell and trying the next value. The puzzle is considered solved when no empty cells remain.

2.4. Generating a Complete Sudoku Grid

Before a playable Sudoku puzzle can be created, a fully solved grid must be generated. This is handled by the **`generateFullGrid`** function.

```python
def generateFullGrid(n):
    size = n * n  # total grid size
    puzzle = np.zeros((size, size), dtype=int)  # start with an empty grid
    # fill the grid using the Sudoku solver
    solveSudoku(puzzle, n)
    return puzzle  # return a complete valid Sudoku grid
```

Listing 5: generateFullGrid Function (Python Code)

The function initializes an empty (n^2) x (n^2) grid filled with zeros and applies the backtracking solver to fill it completely. Since the solver attempts values in a randomized order, the resulting completed grid differs each time the program is executed. This randomness ensures that the base solution used for puzzle generation is not repetitive and provides a diverse set of valid Sudoku grids.

2.5. Puzzle Generation and Difficulty Control

After generating a complete grid, values are removed to create a playable puzzle. This process is implemented in the **`generatePuzzle`** function and is controlled by a user-selected difficulty level.

```python
def generatePuzzle(n, difficulty):
    puzzle = generateFullGrid(n)
    size = n * n

    # remove about half of the values to create a puzzle
    if difficulty == "easy":
        removals = (size * size) // 3
    elif difficulty == "medium":
        removals = (size * size) // 2
    elif difficulty == "hard":
        removals = (size * size) * 2 // 3
    elif difficulty == "extreme":
        removals = (size * size) * 3 // 4

    attempts = removals * 5

    # randomly remove values while keeping the puzzle solvable
    while removals > 0 and attempts > 0:
        # pick a random cell in the grid
        row = np.random.randint(0, size)
        col = np.random.randint(0, size)

        # only remove a number if the cell is not already empty
        if puzzle[row, col] != 0:
            backup = puzzle[row, col]
            puzzle[row, col] = 0

            # make a copy and try solving it
            # this checks whether the puzzle is still solvable after removal
            test = puzzle.copy()
            if solveSudoku(test, n):
                removals -= 1
            else:
                # restore the value if removal makes the puzzle unsolvable
                puzzle[row, col] = backup
        attempts -= 1
    return puzzle
```

Listing 6: generatePuzzle Function (Python Code)

The difficulty setting determines how many cells are removed from the grid. Easier difficulties remove fewer values, while harder and extreme difficulties remove a larger proportion of the grid. Cells are selected randomly for removal to avoid predictable patterns.

To ensure that the puzzle remains solvable, each removal is validated by creating a copy of the puzzle and attempting to solve it using the backtracking solver. If the solver successfully finds a solution, the removal is kept. If the puzzle becomes unsolvable, the removed value is restored. This process continues until the desired number of removals is reached or a predefined number of attempts is exceeded, preventing infinite loops.

2.6. Visualization of Results

Once the puzzle and its solution are generated, the program produces visual representations using Matplotlib. The `drawSudoku` function renders the Sudoku grid as an image, using thicker lines to separate subgrids and thinner lines for individual cells.

```python
def drawSudoku(puzzle, n, filename):
    size = n * n

    # create a blank background for the grid
    plt.imshow(np.zeros((size, size)), cmap="gray_r")

    # every n lines, we reach the edge of a subgrid
    # for example, in a 9x9 board (n = 3), lines at 0, 3, 6, 9 separate the 3x3 boxes
    for i in range(size + 1):
        if i % n == 0:
            lineWidth = 2      # thicker line for subgrid boundary
        else:
            lineWidth = 0.5    # regular cell line

        # offset by 0.5 so lines appear between cells
        plt.axhline(i - 0.5, linewidth=lineWidth, color="black")
        plt.axvline(i - 0.5, linewidth=lineWidth, color="black")

    # draw the numbers in each non-empty cell
    for row in range(size):
        for col in range(size):
            if puzzle[row, col] != 0:
                # place number in the center of the cell
                plt.text(col, row, str(puzzle[row, col]),
                         ha="center", va="center", fontsize=14)

    # remove axis labels for a clean Sudoku look
    plt.xticks([])
    plt.yticks([])

    # save the image and close the figure
    plt.savefig(filename)
    plt.close()
```

Listing 7: drawSudoku Function (Python Code)

Each non-empty cell is displayed with its corresponding number centered within the cell, resulting in a clean and readable layout. Axis labels are removed to maintain a traditional Sudoku appearance. Separate images are generated for the puzzle and its solution, allowing for easy comparison and verification of correctness.

---

3. Conclusions

In this project, a Sudoku generator and solver was designed using Python and NumPy. The program validates user input, generates complete Sudoku grids, and solves puzzles using a recursive backtracking algorithm that enforces row, column, and subgrid constraints.

Randomized value selection allows for varied puzzle generation, and Matplotlib is used to visualize both the puzzle and its solution. This demonstrates the effectiveness of backtracking as a general approach to solving constraint satisfaction problems such as Sudoku.

3.1. Future Improvements

While the current implementation successfully generates and solves Sudoku puzzles, several improvements could further enhance its efficiency, robustness, and flexibility. Possible future developments include:

- **Full Vectorization:** Refactoring portions of the validation and solving logic to rely more heavily on NumPy vectorized operations, reducing reliance on nested loops and improving performance for larger grids.

- **Improved Difficulty Metrics:** Replacing the current difficulty model, which is based primarily on the number of removed cells, with a more meaningful metric. This could include analyzing the number of backtracking steps, recursion depth, or constraint violations required to solve the puzzle.

- **Heuristic-Based Backtracking:** Incorporating heuristics such as selecting the cell with the fewest valid candidates (minimum remaining values) to significantly reduce the search space and improve solving speed.

- **Unique Solution Enforcement:** Adding checks to ensure that each generated puzzle has exactly one valid solution, increasing puzzle quality and preventing ambiguous boards.