**Overview:**

This Python code is a graphical user interface (GUI) application that visually demonstrates the backtracking algoritm of solving a Sudoku puzzle step-by-step. The application is built using the tkinter library, and it allows users to interactively move through the steps of the puzzle being solved.

**Classes and Functions:**

**1. SudokuVisualizer Class:**

This class handles the entire visualization of the Sudoku-solving process. It takes care of the UI elements, drawing the puzzle grid, and providing controls to navigate through the solving steps.

**Attributes:**

* **root**: The main window for the application.
* **original\_puzzle**: A copy of the initial unsolved puzzle to differentiate between the original and filled-in cells.
* **solution\_steps**: A list containing the states of the puzzle at each step.
* **current\_step**: The index that tracks the current step being displayed.
* **last\_changed\_indexes**: A list that stores the indexes of cells that were last modified at each step.
* **size**: The size of each cell in the grid (50x50 pixels).

**Methods:**

* **\_\_init\_\_(self, root, puzzle, solution\_steps, last\_changed\_indexes)**:
  + Initializes the application, sets up the UI elements (buttons, canvas), and displays the first step of the solution.
* **draw\_puzzle(self, puzzle, l\_index)**:
  + Draws the current state of the puzzle on the canvas.
  + Uses different colors for original and newly filled cells.
  + Highlights the last changed cell in red.
* **show\_prev\_step(self)**:
  + Moves to the previous step in the solution and updates the displayed puzzle.
* **show\_next\_step(self)**:
  + Advances to the next step and updates the puzzle accordingly.
* **jump\_to\_step(self)**:
  + Jumps to a specific step entered by the user.
* **show\_final(self)**:
  + Directly shows the final solved puzzle by skipping to the last step.
* **get\_solution\_steps(puzzle, solver\_function, last\_changed\_indexes):**
* A helper function that runs the Sudoku solver function and collects each step of the solution, storing them in steps.
* **sudoku(puzzle, steps, last\_changed\_indexes):**
* This function implements a backtracking Sudoku solver algorithm. It attempts to fill each cell with possible values and uses backtracking to find a valid solution.
* It saves each step of the process (as the puzzle evolves) and records the indexes of the cells that were changed at each step.

**User Interaction:**

* **Previous Button (< Previous)**: Moves to the previous step in the solving process.
* **Next Button (Next >)**: Moves forward to the next step.
* **Jump to Step (Go to step:)**: Allows the user to input a specific step number to jump to.
* **Show Final (Show Final)**: Skips directly to the final, solved state of the puzzle.

**Example Usage:**

When the user runs the application, they will see the initial state of a Sudoku puzzle. They can press "Next" to move step-by-step through the process of solving it. The cells that were changed in the last step will be highlighted in red, allowing users to clearly see how the puzzle progresses.

The "Show Final" button allows the user to skip to the end and see the fully solved puzzle instantly.

### Explanation of the Sudoku Solver Algorithm

Backtracking is a common algorithmic technique that explores possible solutions by building them incrementally and discarding partial solutions that do not satisfy the problem's constraints. In Sudoku, the constraints are that each number must be unique in its row, column, and 3x3 subgrid.

Let's break down the algorithm into its components and analyze its working in detail.

### 1. ****Understanding the Sudoku Puzzle Structure****:

* **9x9 Grid**: A Sudoku puzzle is a 9x9 grid where each row, column, and 3x3 subgrid must contain the digits 1 through 9 exactly once.
* **Empty Cells (0s)**: In the input puzzle, the empty cells are represented by the number 0. These are the cells the algorithm tries to fill.

### 2. ****Steps of the Backtracking Algorithm****:

#### Step 1: ****Initialization****:

* The algorithm starts by identifying all the empty cells (0s) in the puzzle. These cells are stored in the empty\_cells\_indx list as pairs of coordinates[i,j].
* It also prepares a list may (short for "maybes") which holds possible candidates for each empty cell. Initially, this list contains the numbers 1 through 9 for each empty cell, meaning that any number is a potential candidate at first.

#### Step 2: ****Iterative Filling (Backtracking Core)****:

* The algorithm iterates over each empty cell in the empty\_cells\_indx list, starting with the first empty cell.
* For each cell, the algorithm checks the list of candidates (may[count]) and tries to place the first candidate that satisfies the Sudoku constraints.

#### Step 3: ****Constraint Checking****:

For each empty cell, the algorithm checks if the current candidate number violates any of the following constraints:

* **Row Constraint**: The number should not already exist in the current row.
* **Column Constraint**: The number should not already exist in the current column.
* **Subgrid Constraint**: The number should not already exist in the 3x3 subgrid that contains the current cell.

These constraints are checked as follows:

* The algorithm examines the current row (row\_values), the current column (columns), and the appropriate 3x3 subgrid (subgrids).
* If the candidate number satisfies all these constraints, it is placed in the current cell, and the algorithm moves to the next empty cell.

#### Step 4: ****Backtracking****:

* If none of the candidates for the current cell satisfy the constraints, the algorithm backtracks. This means:
  + It moves to the previous empty cell, removes the number that was placed there, and tries the next candidate from its list.
  + The list of candidates for all future cells (may[count] where count > current) is reset to contain numbers 1 through 9 again.
* This process of trial and error continues until the algorithm finds a valid number for the current cell.

#### Step 5: ****Solution Storage****:

* After each valid placement of a number in an empty cell, the algorithm saves the current state of the puzzle into the steps list. This list is used to visualize the solution step-by-step.
* The coordinates of the last modified cell are stored in the last\_changed\_indexes list, allowing for easy identification of which cell was changed in each step.

#### Step 6: ****Termination****:

* The algorithm terminates when all empty cells have been filled with valid numbers. At this point, the Sudoku puzzle is solved.
* The algorithm then returns the step-by-step solution (steps), which can be visualized in the SudokuVisualizer.

### 3. ****Detailed Code Breakdown(python)****:

for i in range(9):

for j in range(9):

if puzzle[i][j] == 0:

empty\_cells\_indx.append([i, j])

may.append([1, 2, 3, 4, 5, 6, 7, 8, 9])

* This loop identifies all empty cells (0s) and stores their coordinates in the empty\_cells\_indx list. It also initializes the may list with all possible candidates for each empty cell.

columns = []

for i in range(9):

p = []

for j in range(9):

p.append(puzzle[j][i])

syun.append(p)

* This part creates columns, which is the transpose of the puzzle. Each list in columns represents a column in the puzzle, which makes it easier to check for column constraints later.

while i != 9:

j = 0

while j != 9:

* These nested loops iterate through each row (i) and each column (j) of the puzzle.

if puzzle[i][j] == 0:

ind = 0

while ((may[count][ind] in row\_values) or

(may[count][ind] in columns[j]) or

(may[count][ind] in subgrids[k][0]) or

(may[count][ind] in subgrids[k][1]) or

(may[count][ind] in subgrids[k][2])):

ind += 1

* This block checks whether the candidate number (may[count][ind]) satisfies the row, column, and subgrid constraints. If the candidate number is found in any of these, the algorithm moves to the next candidate by incrementing ind.

if ind >= len(may[count]):

i, j=empty\_cells\_indx[count-1][0],empty\_cells\_indx[count-1][1]

may[count-1].remove(puzzle[i][j])

count -= 1

* If all candidates have been tried and none satisfy the constraints, the algorithm backtracks to the previous cell.

for o in range(count+1, len(may)):

may[o] = [1, 2, 3, 4, 5, 6, 7, 8, 9]

#code below finds in which subgrid is the element

if 0 <= i <= 2:

h = j // 3

elif 3 <= i <= 5:

h = 3 + j // 3

else:

h = 6 + j // 3

for u in range(3):

if puzzle[i][j] in subgrids[h][u]:

subgrids[h][u][subgrids[h][u].index(puzzle[i][j])] = 0

puzzle[i][j] = 0

columns[j][i] = 0

success\_flag = 0

break

* As a valid candidate is not found the list of candidates for all future cells is reset, 0 is placed in the cell, the subgrid (subgrids), column structures ( columns) are updated

if success\_flag:

for u in range(3):

if puzzle[i][j] in subgrids[k][u]:

subgrids[k][u][subgrids[k][u]. index(puzzle[i][j])]= may[count][ind]

puzzle[i][j] = may[count][ind]

last\_changed\_indexes.append([i,j])

columns[j][i] = puzzle[i][j]

count += 1

steps.append([row[:] for row in puzzle])

* If a valid candidate is found, it is placed in the cell, the subgrid (subgrids) and column structures (columns) are updated, and the solution step is saved.

### 4. ****Time Complexity****:

The time complexity of backtracking for Sudoku can be very high in the worst case. For an empty grid (81 empty cells), the algorithm may need to try all 9 possibilities for each cell, leading to a time complexity of O(9^81). However, since the algorithm rejects invalid candidates early (constraint checking), the actual number of recursive calls is much smaller.

### *Conclusion:*

*The backtracking approach works efficiently for solving standard Sudoku puzzles because it systematically explores all possible placements and backtracks when constraints are violated. The visualization provided by SudokuVisualizer offers an intuitive way to observe this problem-solving process in action.*