The backtracking algorithm is implemented in the **SudokuSolver.java** file. The backtracking algorithm is implemented in the following lines of the SudokuSolver.java file:

The algorithm starts at line 29:

private static boolean solve(int[][] puzzle, int row, int col) {

First, it checks if we've reached the end of the puzzle:

if (row == 9) {

// All rows have been filled, puzzle is solved

return true;

}

If we've reached the end of a row, it moves to the next row:

if (col == 9) {

// Move to next row

return solve(puzzle, row + 1, 0);

}

If the current cell is already filled, it moves to the next cell:

if (puzzle[row][col] != 0) {

// Cell is already filled, move to next cell

return solve(puzzle, row, col + 1);

}

The core of the algorithm is in the following loop:

for (int value = 1; value <= 9; value++) {

if (isValid(puzzle, row, col, value)) {

puzzle[row][col] = value;

if (solve(puzzle, row, col + 1)) {

return true;

}

puzzle[row][col] = 0; // Backtrack

}

}

This loop tries each value from 1 to 9 in the current cell. If a value is valid (doesn't violate Sudoku rules), it:

* Places the value in the cell
* Recursively tries to solve the rest of the puzzle
* If the recursive call returns true, it means a solution is found
* If not, it backtracks by setting the cell back to 0 and tries the next value

If no value works for this cell, it returns false, which triggers backtracking in the previous recursive call:

return false;

The `isValid` method (lines 64-82) is used to check if placing a value in a specific cell violates any Sudoku rules by checking the row, column, and 3x3 sub-grid.

**Algorithm Mathematical Expression**

Given a 9×9 Sudoku grid 𝑃 where some cells are pre-filled with numbers (from 1 to 9) and others are empty (denoted as 0), the goal is to fill in all the empty cells such that each row, each column, and each 3×3 sub-grid contains all the numbers from 1 to 9 without repetition.

**Variables and Functions**

**Puzzle Grid:** 𝑃 is a 9×9 grid where 𝑃[𝑖][𝑗] represents the value at row 𝑖 and column 𝑗.

**Row and Column:** The algorithm iterates over the grid using row and col indices.

**Valid Placement Function:** isValid(𝑃,𝑖,𝑗,𝑣) checks if placing a value 𝑣 at 𝑃[𝑖][𝑗] follows Sudoku rules.

**Backtracking Algorithm**

The backtracking algorithm can be expressed mathematically as follows:

**Base Case:**

**Recursive Step:**

**Validation Function:** The isValid function ensures that placing a value 𝑣 at 𝑃[𝑖][𝑗] is consistent with Sudoku rules:

**Sudoku Generator**

Another part is `SudokuGenerator.java` which contains an algorithm for generating Sudoku puzzles. The core of the generation algorithm is contained in the `solve` method, which uses a recursive backtracking approach to generate a complete Sudoku solution.

The algorithm starts at line 76:

private boolean solve(int row, int col) {

if (col == 9) {

col = 0;

row++;

if (row == 9) {

return true;

}

}

if (solution[row][col] != 0) {

return solve(row, col + 1);

}

List<Integer> numbers = generateRandomNumbers();

for (int num : numbers) {

if (isSafe(row, col, num)) {

solution[row][col] = num;

if (solve(row, col + 1)) {

return true;

}

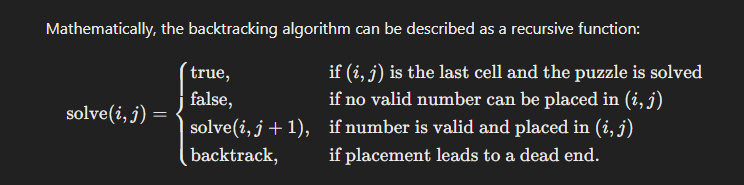
solution[row][col] = 0;

}

}

return false;

}



Explaining the Algorithm Code:

private boolean solve(int row, int col) {

if (col == 9) {

col = 0;

row++;

if (row == 9) {

return true;

}

}

This part handles moving to the next row when we reach the end of a column. If we've filled the entire grid (row == 9), we return true as we've found a solution.

if (solution[row][col] != 0) {

return solve(row, col + 1);

}

If the current cell is already filled (not 0), we move to the next cell.

List<Integer> numbers = generateRandomNumbers();

for (int num : numbers) {

if (isSafe(row, col, num)) {

solution[row][col] = num;

if (solve(row, col + 1)) {

return true;

}

solution[row][col] = 0;

}

}

This is the core of the backtracking algorithm:

1. We generate a list of numbers 1-9 in random order.
2. For each number, we check if it's safe to place in the current cell.
3. If it's safe, we place the number and recursively try to solve the rest of the grid.
4. If the recursive call returns true, we've found a solution.
5. If not, we remove the number (backtrack) and try the next number.

return false;

If we've tried all numbers and none work, we return false to trigger backtracking.

The isSafe method (lines 157-159) checks if a number can be placed in a cell without violating Sudoku rules:

private boolean isSafe(int row, int col, int num) {

return !usedInRow(row, num) && !usedInColumn(col, num) && !usedInBox(row - row % 3, col - col % 3, num);

}

After generating a complete solution, the algorithm removes cells to create the puzzle:

private void removeCells(int difficulty) {

Random rand = new Random();

int cellsToRemove = 81 - difficulty;

while (cellsToRemove > 0) {

int row = rand.nextInt(9);

int col = rand.nextInt(9);

if (puzzle[row][col] != 0) {

int temp = puzzle[row][col];

puzzle[row][col] = 0;

if (!hasUniqueSolution()) {

puzzle[row][col] = temp;

} else {

cellsToRemove--;

}

}

}

}

This method:

1. Randomly selects cells to remove.
2. Removes a cell's value temporarily.
3. Checks if the puzzle still has a unique solution.
4. If not, it restores the value; if yes, it keeps the cell empty.
5. Continues until the desired number of cells are removed.