Assignment : 2
Programming report

Lucas Machado

Problem Solving and Algorithm Design: Sudoku Solver Program

1 Problem description

Write a program to solve Sudoku puzzles. Sudoku is a logic-based combinatorial number-placement puzzle. The objective is to fill a 9x9 grid with digits so that each column, row, and nine 3x3 subgrids that compose the grid contain all of the digits from 1 to 9.

2 Problem analysis

We are considering two options, either brute force or backtracking. We can generate every possible sudoku board when considering the brute force method. This is an exponential amount of boards, and the boards do not have to be valid; they are just all possibilities. Then, we have to validate all the boards we generated, try to validate the single solution if it exists for the board, and then return the board. This solution is unreasonable as an exponential amount of boards would have to be generated. To be precise, are 6,670,903,752,021,072,936,960 boards possible, which is not computationally feasible. Thus, we can use a recursive backtracking algorithm that is more efficient with fewer computations.

3 Recursive backtracking algorithm

We will solve the sudoku board using a traditional recursive backtracking algorithm. The backtracking algorithm is constructed from three significant parts. The first condition is our choices, followed by our constraints and the algorithm's goal. We choose a number from 1-9 as the board can only have numbers from 1-9. We have three main constraints. We first have to verify the first constraint. That is, we must check if the number we are trying to place already exists in the given row; thus, if we have a board and we are looking at row one and row one has the numbers 1,2 and 9, then we cannot place the numbers 1,2 and 9 in any blocks in that row. We then check if the number is already present within the target column. For instance, consider being positioned at the intersection of row 1 and column 1, where the column comprises the numbers 1, 2, and 8, and the row contains the numbers 8, 3, and 7. Under these circumstances, we cannot use the numbers 1, 2, or 8 due to their presence in the column. Similarly, the number 7 is also not allowed to be used as it exists in the row. The third constraint involves the subgrid: we cannot place a number that already exists within the subgrid we are targeting. To identify the subgrid, we compute the starting indices using $3 \times \left\lfloor \frac{\text{col}}{3} \right\rfloor$ for columns and $3 \times \left\lfloor \frac{\text{row}}{3} \right\rfloor$ for rows, where *row* and *col* represent the current row and column within the function, respectively. The board is split into nine sub-grids, and the first sub-grid goes from row 1 block 1 to row 1 block three and from column 1 block 1 to column 1 block 3. This is the subgrid that is formed. If we say that we have 7, 4,5 in that subgrid, then we cannot place either 7,4 or 5. Now, we must check the final condition: that our board is complete; that is, the 9*9 board is full with valid numbers. If all of the conditions pass and the board is complete, it is implied that the board is valid. If the board is incomplete, it implies it does not have a solution.

4 C-style pseudo-code for IsValid function

Algorithm 1 Check Validity

```
1: function IsValid(grid[N][N], row, col, choice)
2:     for i = 0 to N − 1 do
3:         if grid[row][i] = choice then
4:             return false
5:         end if
6:     end for
7:     for i = 0 to N − 1 do
8:         if grid[i][col] = choice then
9:             return false
10:        end if
11:    end for
12:    startCol = 3 × ⌊col/3⌋
13:    startRow = 3 × ⌊row/3⌋
14:    for i = 0 to 2 do
15:        for j = 0 to 2 do
16:            if grid[startRow + i][startCol + j] = choice then
17:                return false
18:            end if
19:        end for
20:    end for
21:    return true
22: end function
```

5 C-style pseudo-code for SolveSudoku function

Algorithm 2 Solve Sudoku

```
1: function SolveSudoku(grid[N][N], row, col)
2:     if row = 9 then
3:         return true
4:     end if
5:     if col = 9 then
6:         return SolveSudoku(grid, row + 1, 0)
7:     end if
8:     if grid[row][col] ≠ 0 then
9:         return SolveSudoku(grid, row, col + 1)
10:    end if
11:    for choice = 1 to 9 do
12:        if IsValid(grid, row, col, choice) then
13:            grid[row][col] = choice
14:            if SolveSudoku(grid, row, col + 1) then
15:                return true
16:            end if
17:            grid[row][col] = 0
18:        end if
19:    end for
20:    return false
21: end function
```

6 Sudoku Solver Program

```c
#include <stdio.h>
#include <stdbool.h>

#define N 9    // Size of the Sudoku puzzle
```

```c
int count = 0; // Global count that will be used to store the number of iterations

// Function that will print out the board given an 9*9 2-D array as input
void print(int grid[N][N])
{
    printf("\n");
    for (int row = 0; row < N; row++)
    {
        if (row % 3 == 0 && row != 0)
        {
            printf(" ------------------------------\n");
        }

        for (int col = 0; col < N; col++)
        {

            if (col % 3 == 0 && col != 0)
            {
                printf(" | ");
            }

            printf(" %d ", grid[row][col]);
        }
        printf("\n");
    }
    printf("\n");
}

/*
 Function that will check if a given number from 1-9 is valid to be placed in a given block.
 This function checks three main conditions: whether the choice is in any row, column, or subgrid.
*/
bool isValid(int grid[N][N], int row, int col, int choice)
{
    // Check row for the choice
    for (int i = 0; i < N; i++)
    {
        if (grid[row][i] == choice)
        {
            return false; // If the choice already exists in the row it is not valid
        }
    }

    // Check column for the choice
    for (int i = 0; i < N; i++)
    {
        if (grid[i][col] == choice)
        {
            return false; // If the choice already exists in the column, it is not valid
        }
    }

    // Check 3x3 subgrid for the choice
    int startRow = row - row % 3; // Determine the start row index of the subgrid
```

```
59      int startCol = col - col % 3; // Determine the start column index of the
            subgrid
60      for (int i = 0; i < 3; i++)
61      {
62          for (int j = 0; j < 3; j++)
63          {
64              if (grid[i + startRow][j + startCol] == choice)
65              {
66                  return false; // If the choice already exists in the subgrid
                        , it is not valid
67              }
68          }
69      }
70
71      return true; // If the choice passes all checks, it is valid
72  }
73
74  bool solveSudoku(int grid[N][N], int row, int col)
75  {
76      count++; // Increment the count every time that the function is called
77
78      // Base case that is implying that the end of the grid has been reached
            thus the puzzle is solved
79      if (row == 9)
80      {
81          return true;
82      }
83
84      // If the current column is 9 then we move to the next row and start
            from the first column
85      if (col == 9)
86      {
87          return solveSudoku(grid, row + 1, 0); // Move to the next row
88      }
89
90      // If the cell is already filled move to the next column
91      if (grid[row][col] != 0)
92      {
93          return solveSudoku(grid, row, col + 1); // Skip the blocks that are
                filled with a non-zero
94      }
95
96      // Try all possible numbers for the a given cell
97      for (int choice = 1; choice <= 9; choice++)
98      {
99          // Check if the current choice is valid for the current cell
100         if (isValid(grid, row, col, choice))
101         {
102             grid[row][col] = choice; // Temporarily assign the cell with the
                    choice
103
104             // Recursively try to solve the Sudoku starting from the next
                    column
105             if (solveSudoku(grid, row, col + 1))
106             {
107                 return true; // Found a valid solution
108             }
109
110             grid[row][col] = 0; // Backtrack if the current choice leads to
                    a dead end, reset the cell to 0
111         }
```

```
    }

    return false; // There was not a valid number found for this cell thus
        backtrack further
}

int main()
{
    int grid[N][N] = {
        {2, 7, 0, 0, 0, 0, 0, 9, 3},
        {0, 0, 6, 0, 3, 9, 0, 0, 0},
        {3, 0, 0, 0, 0, 0, 1, 5, 0},
        {0, 3, 0, 2, 0, 4, 0, 0, 7},
        {9, 2, 5, 0, 0, 0, 4, 0, 8},
        {4, 0, 0, 6, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 7, 5},
        {5, 0, 0, 0, 0, 8, 0, 0, 1},
        {0, 0, 4, 0, 0, 3, 9, 0, 0}};

    printf("The input Sudoku puzzle:\n");
    // Assuming 'print' is a function defined to print the grid

    print(grid);
    if (solveSudoku(grid, 0, 0))
    {
        // If the puzzle is solved:
        printf("Solution found after %d iterations:\n", count);
        print(grid);
    }
    else
    {
        printf("No solution exists.");
    }

    return 0;
}
```

7 The purpose of each function

The purpose of the function print is to print the game board. It uses a nested for loop to achieve this by printing the rows and columns and filling blocks with the values from the input 2D array.

We must use a helper function, isValid. This function checks our three conditions to verify if we can place a number from 1 to 9 in a given block. It first checks if the number we are trying to place is already in the given row, and if found in the row, then the function will return false, which implies that the number cannot be placed in that block. Then we check if the number we are trying to place is in the given column, and if it is, then we will turn it false again; hence, we cannot place the number in that block. We must then check if the number we are trying to place is in the given subgrid. We first compute the bounds on the subgrid. Then, we traverse the subgrid, and if the number is found, we will return false; thus, the number we are trying to place is invalid. Moreover, if none of these checks return false, we return true, implying that the choice is valid.

The solveSudoku implements a recursive backtracking algorithm using the help of the isValid function. The recursive base case checks if the row equals nine, then it implies that we have reached the end of the grid thus we have solved it and we return true. We then check if the column equals 9. If this is the case, we will recursively call solveSudoku, moving to the next column. We also must check if the current grid block is not equal to zero because if it is equal to zero, then we should skip it; thus, we recursively call solveSudoku again with the column incremented by 1. Then, we must try all the possibilities for each block, so we will iterate through 1 to 9 and call our valid function to validate the cell. If it returns true, we know that we can return true, and the number is valid; otherwise, we set it back to zero and backtrack. At the end of the function, we return false as this implies no valid number for the cell.

The final function is the main function. This function is used to print the Sodoku grid. We first try to solve the grid; if it cannot be solved, we will print that no solution can be exists.