Assignment 2: Sudoku Solver

Anh Thieu

March 6, 2024

Contents

Chapter 1

Algorithm and Helper Functions

1.1 Algorithm

The algorithm of this automatic Sudoku solving system is based on the use of backtracking with the help of some written helper functions.

The algorithm in the main solving function, solveSudoku(), initializes variables row and col to keep track of current positions in the array. The function also keeps track of the number of iterations that have been performed by increasing the global count variable by 1 in each iteration.

First, the function emptySlots() is used to check the whole array to see if there are any empty slots or zeros in the array. If all the slots are filled, then the stop the algorithm and check the solution found, otherwise, continue the program. Then, use a for loop to go through the array's rows and columns and check if the current slot is mutable or zero. If it is mutable then have a for loop consisting of integers from 1 to 9. Call these "trial numbers" since they will be iterated through and used to check for a valid number that belongs in the current slot. In the "trial number" loop, use functions numInCol(), numInRow(), and numIn3x3() to check whether the "trial number" at that slot belongs there, or in other words, if there are any numbers in the same row, column, or respected 3x3 block that match the current "trial number". If it seems valid, change the current slot to the "trial number" and recursively call solveSudoku() (the main function) to check if it leads to a possible solution.

- If it does lead to a solution, do one final check through the entire array for any duplicates in rows, columns, and 3x3's using numInCol(), numInRow(), and numIn3x3() functions. If solution is completely valid then stop search and return true, otherwise return false. If it
- If it does not lead to a solution, reset the current slot to zero to prepare for backtracking.

If the Sudoku puzzle goes through all "trial numbers" 1 to 9 and have no possible solution, then return false to stop that iteration of recursion. This will ensure that any combinations of numbers that are invalid will not be returned as a solution. The algorithm will also return false when it has tried all possibilities of numbers in every slot but no solutions have been found.

1.2 Helper Functions

The code includes 5 helper functions to calculate and display the results.

1. void printArray(int arr[N][N])
    - This function takes in an array as input, uses for loops to print the array, and returns nothing.
    - Purpose: prints array.
2. bool numInRow(int arr[N][N], int num, int numRow, int numCol)
    - This function takes in inputs: an array, the slot number, row number, and column number
    - It uses a for loop that goes through each columns of the row, returns true if the slot number matches any of the row's numbers and false if there are no matches.
    - Purpose: searches the row to find any matching numbers to a given slot in array so Sudoku puzzle has no repetition of numbers.
3. bool numInCol(int arr[N][N], int num, int numRow, int numCol)
    - This function returns a boolean and takes in inputs: an array, the slot number, row number, and column number
    - It uses a for loop that goes through each rows of the column, returns true if the slot number (num) matches any of the column's numbers and false if there are no matches.
    - Purpose: searches the column to find any matching numbers to a given slot in array so Sudoku puzzle has no repetition of numbers.
4. bool numIn3x3(int arr[N][N], int num, int numRow, int numCol)
    - This function returns a boolean and takes in inputs: an array, the slot number, row number, and column number
    - It uses switch cases to identify which block of 3x3 to check, then loops through that designated block to return true if the slot number (num) matches and false if there are no matches.
    - Purpose: searches respected 3x3 block so Sudoku puzzle has no repetition of numbers.
5. bool emptySlots(int arr[N][N])
    - This function returns a boolean and takes in an array as an input.
    - It uses for loops to go through the entire array and checks if there are any empty slots, returns true if there is and false if there isn't.
    - Purpose: checks the progress. If all cells are filled then possible solution found, stop recursion.

1.3 solveSudoku.c Code

```
// Code : Here include your necessary library(s)
#include <stdio.h>
#include <stdbool.h>

// Code : Write your global variables here, like:
#define N 9
#define EMPTY 0
int count = 0;

/*Code : write your functions here, or the declaration of the function/
For example write the recursive function solveSudoku(), like:*/

//function to print array
void printArray(int arr[N][N]) {
    //for loop go through all rows
    for (int i=0;i<N;i++) {
        printf("\n");
        //for loop go through all cols
        for (int j=0;j<N;j++) {
            //print number at row & col
            printf("%d ",arr[i][j]);
            if (j == 2 || j == 5) {
                printf ("| ");
            }
        }
        if (i == 2 || i == 5) {
            printf("\n---------------------");
        }
    }
    printf("\n");
}

//function to check if a number is in that row
bool numInRow(int arr[N][N], int num, int numRow, int numCol){
    //for loop go through each col
```

```
    for (int i=0; i<N; i++) {
        //if any number exists in that row -> return true
        if ((arr[numRow][i] == num) && (i != numCol)) {
            return true;
        }
    }
    //if no number in that row -> return false
    return false;
}

//function to check if a number is in that col
bool numInCol(int arr[N][N], int num, int numRow, int numCol){
    //for loop go through each row
    for (int i=0; i<N; i++) {
        //if any number exists in that col -> return true
        if ((arr[i][numCol] == num) && (i != numRow)) {
            return true;
        }
    }
    //if no number in that col -> return false
    return false;
}

//function to check if a number is in respected 3x3 cell
bool numIn3x3(int arr[N][N], int num, int numRow, int numCol) {
    //initialize variables
    int rCheck = 0; // respected starting row for 3x3
    int cCheck = 0; // respected starting col for 3x3

    //switch cases to determine which 3x3 block (out of 9 3x3's) to check
    switch (numRow) {
        case 0 ... 2: // row is from 0 to 2
            rCheck = 0;
        break;
        case 3 ... 5: // row is from 3 to 5
            rCheck = 3;
        break;
        case 6 ... 8: // row is from 6 to 8
```

```
            rCheck = 6;
        break;
    }
    switch (numCol) {
        case 0 ... 2: // col is from 0 to 2
            cCheck = 0;
        break;
        case 3 ... 5: // col is from 3 to 5
            cCheck = 3;
        break;
        case 6 ... 8: // col is from 6 to 8
            cCheck = 6;
        break;
    }

    //for loop go from row to col + 3 (for 3x3 blocks)
    for (int i = rCheck; i<rCheck+3; i++) {
        //for loop go from col to col + 3 (for 3x3 blocks)
        for (int j = cCheck; j<cCheck+3; j++) {
            //if number exists in 3x3 block -> return true
            if ((num == arr[i][j]) && (i != numRow) && (j != numCol)) {
                return true;
            }
        }
    }
    //if number isn't in 3x3 block -> return false
    return false;
}

//function to check if all the slots in the sudoku puzzle is filled
bool emptySlots(int arr[N][N]) {
    //for loop go through rows of sudoku puzzle
    for (int i=0; i<N; i++) {
        //for loop go through cols of sudoku puzzle
        for (int j=0; j<N; j++) {
            //if that slot is EMPTY (0) -> return true
            if (arr[i][j] == EMPTY) {
                return true;
```

```
            }
        }
    }
    //if no slots are EMPTY (0) -> return false
    return false;
}

//function to solve sudoku puzzle using all helper functions
bool solveSudoku(int arr[N][N]) {
    //initialize variables to keep track of position
    int row;
    int col;

    //increase count of iteration
    count += 1;

    //use emptySlots() function to check if there progress
    if (emptySlots(arr) == false) {
        return true; // if all slots are filled -> solution found! -> return true
    }

    //for loop go through rows of sudoku
    for (row=0; row<N; row++) {
        //for loop go through cols of sudoku
        for (col=0; col<N; col++) {
            //checking if the current slot is an empty/mutable slot
            if (arr[row][col] == EMPTY) {
                //for loop go through numbers (1 to 9) -- call it a "trial #"
                for (int i=1; i<=N; i++) {
                    /*using numInCol(), numInRow(), & numIn3x3() functions
                    to check if the "trial #" is valid in that position
                    (if no other numbers in row/col/3x3 = "trial #")*/
                    //if "trial #" is valid
                    if ((numInCol(arr,i,row,col) == false)
                    && (numInRow(arr,i,row,col) == false)
                    && (numIn3x3(arr,i,row,col) == false)) {
                        arr[row][col] = i; // change array at position to "trial #"
```

```
                    /*recursively call solveSudoku() function to check if
                      the changed "trial #" will lead to a solution */
                    /*if solveSudoku(a) == true -> leads to a solution ->
                                                    return true & stop */
                    if (solveSudoku(arr) == true) {

                        //final check the whole puzzle
                        for (int c=0; c<N;c++) {
                            for (int b=0; b<N; b++) {
                                if ((numInCol(arr,arr[c][b],c,b) == true)
                                || (numInRow(arr,arr[c][b],c,b) == true)
                                || (numIn3x3(arr,arr[c][b],c,b) == true)) {
                                    return false;
                                }
                            }
                        }

                        return true; //puzzle is solved
                    } /*if solve Sudoku(a) == false -> won't lead to solution
                        -> keep going */

                    /*no solution found with "trial #" so reset slot to
                        EMPTY (0) & backtrack (undo) */
                    arr[row][col] = EMPTY;
                }
            }
            return false; /*if no trial #'s (1 to 9) lead to a solution ->
                            stop recursion*/
        }
    }
    return false; /*if all possibilites have been tried but no
                solution -> return false & stop recrusion */
}

//main code block
int main()
{
```

```
    // This is hard coding to receive the "arr"
    int grid[N][N] = {
        {0, 2, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 6, 0, 0, 0, 0, 3},
        {0, 7, 4, 0, 8, 0, 0, 0, 0},
        //---------------------------
        {0, 0, 0, 0, 0, 3, 0, 0, 2},
        {0, 8, 0, 0, 4, 0, 0, 1, 0},
        {6, 0, 0, 5, 0, 0, 0, 0, 0},
        //---------------------------
        {0, 0, 0, 0, 1, 0, 7, 8, 0},
        {5, 0, 0, 0, 0, 9, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 4, 0}};

    // For more samples to check your program, google for solved samples, or
    // check https://sandiway.arizona.edu/sudoku/examples.html

    printf("The input Sudoku puzzle:\n");
    // "print" is a function we define to print the "arr"
    printArray(grid);
    if (solveSudoku(grid) == true)
    {
        // If the puzzle is solved then:
        printf("\nSolution found after %d iterations:", count);
        printArray(grid);
    }
    else
    {
        printf("\nNo solution exists.\n");
    }
    return 0;
}
```