Assignment 2 report

Yisi Liu

March 2024

1 Introduction

While there are many algorithms for digitally solving a sudoku puzzle, some optimizing for speed, others for aesthetics, backtracking remains a classic for one crucial reasonsimplicity. For the same reason, I chose to program it for Assignment 2. In the following report, I will describe backtracking before explaining exactly how it was implemented, going over the specific purpose of each of my functions.

2 The Algorithm

Backtracking is an aptly named algorithm. To see why, and to see how it works, it's best to go over it as a series of steps.

1. place a 1 in the first available square
2. check if 1 is allowed to be there by sudoku rules
3. if yes, repeat from step 1 until the sudoku is solved. If not, increment the 1 to the next possible solution, 2, and check if 2 is allowed. Repeat until a valid number is found.
4. if no valid numbers from 1 to 9 exist for a given square, **backtrack** to the previous available square and increment the previous input there. Repeat until the sudoku is solved, or until we discover that there are no valid numbers for the first available square, in which case no solution exists.

Due to its straightforward nature, backtracking is much easier to implement than other sudoku-solving algorithms (though it's not exactly easy).

3 The Implementation

At first, I took the algorithm very literally after finding a description of it on Wikipedia. I was forced to give up my initial approach after realizing the issue that I've backed myself into solving, which was somehow differentiating between numbers that are the puzzle's parameters and ones that are temporary solutions when backtracking, was more trouble than it was worth.

After speaking with my friend, Saadia Mahmood, I understood that implementing backtracking, especially implementing the backtracking in backtracking, can be a little more implicit instead of my code micromanaging each step of the algorithm.

Please refer to Appendix 1 to see the codes that I reference from this point forward.

Two main functions are responsible for the functionality of program, valid() and solveSudoku(), with solveSudoku() being the main-main one that the count variable tracks iterations of.

The simpler one is valid(), which takes in a 2D array representing the current state of a sudoku puzzle, two integers i and j representing the position of an empty spot to be filled, and an integer num to fill that spot. It returns 1, which is the same as True under an if-statement in C, if num can fill the appointed spot with no violations to the rules of sudoku, and returns 0, the same as False, if there are any violations. In other words, valid() checks if a certain number is a valid solution at any given point. It does this by first comparing num to all the numbers already present in the same column of the given position, and then the entire same row as the given position, in the first for-loop that iterates an index variable, ind, through 0-8. Then, to compare num against all the remaining squares in its 3x3 subsection, the indexes are modulo-ed by 3 which can turn a sequence of 0 to 8 to three repeating sequences of 0 to 2, essentially being able to turn any i or j from a spot in the 9x9 grid to its relative location in any 3x3 subsection. With switch statements to handle each possible relative location, the variables r1, r2, c1, and c2 are filled with the values of the indexes of the two rows and columns in the same subsection but not the same row or column as the given position. An if-statement then checks all four remaining squares.

The function solveSudoku() is more complicated as it is a recursive function. It returns 1 if the sudoku is solved, and 0 if there is no solution. First, it increments the global variable count, which ensures that count increments each time the function is called. Then, to traverse the 9x9 grid of a sudoku, there are two for-loops with one nested within the other. The outer one that iterates through i is responsible for iterating through the rows, while the inner one with variable j does the columns within each row. Now that we're down to the level of single squares appointed by i and j within the two for-loops, we first proceed to the first empty square by using an if-statement that skips an iteration if the appointed square isn't empty. Here, the core functionality of solveSudoku() is implemented with a for-loop, filling in the square with the appropriate solution(which we'll dive into shortly). Thus, if we've executed past for-loop and the appointed square is still empty, namely, still equal to 0, then there must be no valid input, but since we're still technically on the first square, that means there's no solution to the sudoku, and thus, the function returns 0. If the square is filled in, however, it proceeds past this if-statement with no issue, and thus at the end of the function it returns 1, the sudoku having been solved.

The meat and potatoes of solveSudoku(), the innermost for-loop, is able to fill the appointed square in with the appropriate input because it's where the backtracking happens. The for-loop simply iterates through all the possible answers from 1 to 9, and the if-statement inside it employing valid() simply fills in the square with any valid number (so unless there's only one valid number we still need to search for the solution). The key is the innermost if-statement, which calls solveSudoku() recursively. With a single grid filled in, this call essentially asks if a new sudoku with the same parameters as the one we're trying to solve but the first available square with a 1 (or whatever first valid number) in it has a solution. If it doesn't then that valid number can't be the solution to sudoku. The ask works because solveSudoku() has this recursive implementation, where the inner solveSudoku() will call another solveSudoku() of its own, all the way until the puzzle is solved or there's no valid input for the first available square. Since the ask works, the if-statement with the setting the square to 0 for backtracking works (set to 0 because 0 is seen as an empty square by solveSudoku()), essentially like an eraser, allowing for backtracking.

The third function I wrote is print(), which prints the sudoku with some for-loops.

4 Appendix 1

The following is my complete code file(contains some messy commented-out past attempts too ignore those).

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

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

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

void print(int grid[N][N]);

int valid(int grid[N][N], int i, int j, int num)
{
    for (int ind = 0; ind < N; ind++)
    {
        if (grid[ind][j] == num || grid[i][ind] == num)
        {
            return 0;
        }
    }
    /*for (int row = 0; row < 9; row++)
    {
        if (row == i)
            continue;

        if (grid[row][j] == num)
        {
```

```
            return 0;
        }
    }

    for (int col = 0; col < 9; col++)
    {
        if (col == j)
            continue;

        if (grid[i][col] == num)
        {
            return 0;
        }
    }*/

    int r1, r2, c1, c2;
    switch (i % 3)
    {
    case 0:
        r1 = i + 1;
        r2 = i + 2;
        break;
    case 1:
        r1 = i - 1;
        r2 = i + 1;
        break;
    case 2:
        r1 = i - 2;
        r2 = i - 1;
        break;
    }
    switch (j % 3)
    {
    case 0:
        c1 = j + 1;
        c2 = j + 2;
        break;
    case 1:
        c1 = j - 1;
        c2 = j + 1;
        break;
    case 2:
        c1 = j - 2;
        c2 = j - 1;
        break;
    }

    if (grid[r1][c1] == num || grid[r2][c1] == num || grid[r1][c2] == num || grid[r2][c2] == num)
    {
        return 0;
    }

    return 1;
}

int solveSudoku(int grid[N][N])
```

```
{
    count++;

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (grid[i][j] != 0)
            {
                continue;
            }

            for (int ans = 1; ans <= 9; ans++)
            {
                if (valid(grid, i, j, ans))
                {
                    grid[i][j] = ans;
                    if (solveSudoku(grid) == 0)
                    {
                        grid[i][j] = 0;
                    }
                    /*if (solveSudoku(grid))
                    {
                        break;
                    } else
                    {
                        grid[i][j] = 0;
                    }*/
                }
            }

            if (grid[i][j] == 0) //if still zero, ie no numbers fit
            {
                return 0;
            }
        }
    }
    return 1;
}

int main()
{
    // This is hard coding to receive the    g r i d
    int grid[N][N] = {
        {1, 0, 0, 4, 8, 9, 0, 0, 6},
        {7, 3, 0, 0, 5, 0, 0, 4, 0},
        {4, 6, 0, 0, 0, 1, 2, 9, 5},
        {3, 8, 7, 1, 2, 0, 6, 0, 0},
        {5, 0, 1, 7, 0, 3, 0, 0, 8},
        {0, 4, 6, 0, 9, 5, 7, 1, 0},
        {9, 1, 4, 6, 0, 0, 0, 8, 0},
        {0, 2, 0, 0, 4, 0, 0, 3, 7},
        {8, 0, 3, 5, 1, 2, 0, 0, 4}};

    // 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");
    //   p r i n t   is a function we define to print the   g r i d
    print(grid);

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

/*Code : If you have functions that are declared but not
    implemented they,
here write the implementation. */

void print(int grid[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%d  ", grid[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}


/*{
    count += 1;
    // Code: count+1, the number of times the function was called.
    // Code: here write the implementation of solveSudoku

//base cases
    if (i > 8)
    {
        return 1;
    }

    if (i < 0)
    {
        return 0;
    }

//fill the square at (i, j)
    do
    {
        if (grid[i][j] < 9)
        {
```

```c
            grid[i][j] += 1;
        }
        else
        {
            grid[i][j] = 0;
        }
    } while (valid(grid, grid[i][j]) == 0 && grid[i][j] != 0);

//change squares
    if (valid(grid, grid[i][j]))
    {
        do
        {
            if (j < 8)
            {
                j += 1;
            }
            else // j == 8
            {
                i += 1;
                j = 0;
            }
        } while (grid[i][j] != 0);
    }
    else if (grid[i][j] == 0)
    {
        do
        {
            if (j > 0)
            {
                j -= 1;
            } else // j == 0
            {
                i -= 1;
                j = 8;
            }
        } while (grid[i][j] != 0); //oh no can't differentiate between puzzle and entry
    }

    if (solveSudoku(grid))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}*/
```

Listing 1: Sudoku Solver.c