MP6

MP 6 - Sudoku Solver

Introduction

Your task this week is to implement a C program that solves the Sudoku puzzle using recursive backtracking. A standard Sudoku puzzle contains 81 cells, in a 9 by 9 grid, and has 9 zones. Each zone is the intersection of 3 rows and 3 columns (e.g. size 3x3). Each cell may contain a number from 1 to 9 and **each number can only occur once in each 3x3 zone, row, and column of the grid**. At the beginning of the game, several cells begin with numbers, and the goal is to fill in the remaining cells with numbers satisfying the puzzle rules. Unfilled locations will be set to 0. A Sudoku example is shown below:

The pseudocode of a typical backtracking algorithm to solve the Sudoku puzzle is given as follows:

```
bool solve_sudoku(int sudoku[9][9])

{

  int i, j;

  if (all cells are assigned by numbers) {

    return true;

  }

  else {

    find a non-filled (0) cell (i, j)

  }


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

    if (cell (i, j) can be filled with num) {

      sudoku[i][j] <- num;

      if (solve_sudoku(sudoku)) {

        return true;

      }

      sudoku[i][j] <- non-filled (0);

    }
  }

  return false;
```

The Pieces

In this MP, you are given a set of files:

`main.c` - The C source file that contains the `main` function to run the Sudoku solver.

`sudoku.c` - The main source file for your code. Details can be found in the next section.

`sudoku.h/sudoku_golden.h` - The header definition of the program.

Details

You should put your code only in `sudoku.c` and complete all fields marked by `TODO`. The functions we provide follow the pseudocode of the backtracking algorithm given in the introduction section.

```
int is_val_in_row(const int val, const int i, const int sudoku[9][9]);

int is_val_in_col(const int val, const int j, const int sudoku[9][9]);

int is_val_in_3x3_zone(const int val, const int i, const int j, const int sudoku[9][9])
```

You should begin with the above three functions `is_val_in_row`, `is_val_in_col`, and `is_val_in_3x3_zone`, that check if the given number `val` has already been filled in the row `i`, column `j`, and the 3x3 zone corresponding to the cell `(i, j)`. Return `1` (`true`) if the number exists and `0` (`false`) otherwise. Based on these three functions, you should use another function `is_val_valid` that checks the legality of filling a cell `(i, j)` with a given number `val`.

```
int is_val_valid(const int val, const int i, const int j, const int sudoku[9][9]);
```

Given the helper function `is_val_valid`, which you have to code, your final job is to complete the function `solve_sudoku` which implements the recursive backtracking algorithm as shown in the first section. We suggest drawing a diagram of the recursive backtracking by hand if you don't understand it.

```
int solve_sudoku(int sudoku[9][9]);
```

Building and Testing

We suggest that you begin by developing your code using on the EWS Linux machine. For better flexibility, we have offered a make file that wraps the compilation process and program execution. In order to compile the whole program, type the following command under mp6 folder:

```
make
```

If successful, the compiler produces an executable called `mp6`. You can then run the program by the following commands:

```
make sudoku1

make sudoku2

make sudoku3
```

The command make `sudoku1` runs your program with a given sudoku instance specified in `sudoku1.txt` and so on. We give you three Sudoku instances and you are free to make any change from the given files. By default, the program will invoke a hidden golden checker to test your solution and display the solution message every time you make a Sudoku instance.

```
------------------- Begin Verifying MP6 ---------------------

[+10]: Your <is_val_in_row> is correct.

[+10]: Your <is_val_in_col> is correct.

[+30]: Your <is_val_in_3x3_zone> is correct.

[+40]: Your <solve_sudoku> is correct.

Your final score for this MP is 90

-------------------- End Verifying MP6 ----------------------
```

Note that we will use another Sudoku instance (hidden) to test your program during the final grading. All Sudoku instances to test your program will be legal (i.e., a solution always exists).

Debugging with GDB

Use the following command to debug your code using GDB with input file `sudoku1.txt`.

```
gdb --args mp6 sudoku1.txt
```

Grading Rubric

Functionality (totally 90 points)

- +10 - `is_val_in_row` function works correctly
- +10 - `is_val_in_col` function works correctly
- +30 - `is_val_in_3x3_zone` function works correctly
- +40 - `solve_sudoku` function works correctly

Style (totally 5 points)

- +5 - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Comments, clarity, and write-up (totally 5 points)

- +5 - introductory paragraph explaining what you did (even if it's just the required work)

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.