Introduction to Computer Science II

Project 1 – Sudoku

1 Objectives

The purpose of this assignment is to give you an opportunity to:

- Implement a large C program;
- Work with two-dimensional arrays in C;
- Practice recursion;
- Practice automated testing of C programs.

Please read the entire description before starting working on the project.

2 Background

Have you heard of recent victory of a computer program over a human in the game of Go?

http://www.theverge.com/2016/3/9/11185030/google-deepmind-alphago-go-artificial-intelligence-impact

To celebrate this achievement, in this project we will work on a computer implementation of a single-player game, Sudoku. Even though it is a much simpler task than two-player Go, it will possess the features of state-of-the art games - solution verification and look-ahead evaluation.

2.1 Background on Sudoku

A Sudoku grid is a special kind of Latin square. Latin squares, which were so named by the 18th-century mathematician Leonhard Euler, are *n x n* matrices that are filled with n symbols in such a way that the same symbol never appears twice in the same row or column. Two examples are shown. The standard completed Sudoku grid (also known as a solution grid) is a *9x9* Latin square that meets the additional constraint of having each of its nine sub-grids contain the digits 1 to 9.

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Figure 1: Small Latin square (n = 4)

5	8	6	4	2	1	3	7	9
3	2	7	9	6	5	4	8	1
9	1	4	3	7	8	6	2	5
1	6	3	5	8	4	7	9	2
2	4	5	1	9	7	8	6	3
8	7	9	6	3	2	5	1	4
7	5	8	2	1	3	9	4	6

6	3	1	7	4	9	2	5	8
4	9	2	8	5	6	1	3	7

Figure 2: Latin square that is also a completed Sudoku grid (n = 9)

For more background information about Sudoku, please read the sources provided at the end of project description.

2.2 A Sudoku Example

One of the basic techniques for solving Sudoku by hand is to produce a list of *candidates* for each empty cell in the grid. [2] A candidate is a digit between 1 and 9 that you can put into an empty cell without invalidating the Sudoku grid. The following example shows how you can use candidates to help you solve a puzzle.

Consider the Sudoku grid in Figure 3. The candidates for the square in the upper left corner are 7, 8, and 9. That is, if you insert any of these digits into the square, the grid is still a valid Sudoku grid. 1, 2, 4, and 5 won't work because they are already in the first row. Similarly, you can disqualify 3 and 6 because they appear in the first column.

0	1	0	4	2	0	0	0	5
0	0	2	0	7	1	0	3	9
0	0	0	0	0	0	0	4	0
2	0	7	1	0	0	0	0	6
0	0	0	0	4	0	0	0	0
6	0	0	0	0	7	4	0	3
0	7	0	0	0	0	0	0	0
1	2	0	7	3	0	5	0	0
3	0	0	0	8	2	0	7	0

Figure 3: An incomplete Sudoku grid

Now consider the sub-grid (also called a *box*) in the upper right. The candidates for each of the blank squares are shown in Figure 4. Notice that two of the squares have 6 and 8 as their only candidates. This is called a *naked pair*. [4] One of them must be 6 and the other must be 8. Therefore, no other squares in the box can have a 6 or 8. You could then conclude that the square in the upper left of the box must be 7.

6, 7, 8	6, 8	
6, 8		
1, 2, 6, 7, 8		1, 2, 7, 8

Figure 4: Candidates for the *box* in the upper right corner

3 First step, solution verification

The first step in solving any problem is figuring out how the solution would be. I.e. if someone hands you a 9*9 array and claims it is a Sudoku solution, you would like to be sure that it is indeed a solution by checking if it satisfies all of these properties.

- Every row contains each of the numbers 1-9 exactly once
- Every column contains each of the numbers 1-9 exactly once
- The 4 3x3 boxes (top right and left, bottom right and left) contain each of the numbers 1-9 exactly once

3.1 Input

You will need to read a grid description from a file. To make the input file more readable, it contains a blank between the boxes. For example, the following file describes the Sudoku grid in Figure 3.

```
010 420 005
002 071 039
000 000 040

207 100 006
000 040 000
600 007 403

070 000 000
120 730 500
300 082 070
```

Note that blank squares (if any) are represented with a zero. There is also a blank line between boxes. The numbers in each box are listed next to each other. For example the first box in the upper-left corner contains 010 in the first row, 002 in the second row, and 000 in the third row. You can read one line of input as a string and then extract one digit at a time using a scheme of your choice.

Write your program so that it asks the user for the input file name.

3.2 Output

You also need to write your grid as a string. This string can then be used to print the grid when needed. For example, if we take the grid in Figure 3, convert it to a string and print, it would look like this.

```
gentle.txt
-------------------------
| . 1 . | 4 2 . | . . 5 |
| . . 2 | . 7 1 | . 3 9 |
| . . . | . . . | . 4 . |
|-------+-------+-------|
| 2 . 7 | 1 . . | . . 6 |
| . . . | . 4 . | . . . |
| 6 . . | . . 7 | 4 . 3 |
|-------+-------+-------|
| . 7 . | . . . | . . . |
| 1 2 . | 7 3 . | 5 . . |
```

```
| 3 . . | . 8 2 | . 7 . |
-------------------------
```

There are several things to note about this string representation.

- The name of the input file is printed first. This makes it easier to identify the output.
- Empty cells print with a period making them easy to distinguish from the others.
- Boxes have lines around them making them easy to recognize.
- There is a box around the entire grid.

Use info from ch. 21.11-21.14 for help in working with strings.

3.3 Verification

Now that you have a 2-D grid at hand, add functions for verification of three conditions:
is_row_valid()
is_col_valid()
is_block_valid()
Then use these three functions for rows and columns and blocks of the grid to verify the entire grid. Verification function should returns 1 (in place of true) if all conditions are satisfied and the provided board is complete and is indeed a Sudoku solution.

3.4 Putting it together

Now construct a C program that asks the user for the input file name, reads completed Sudoku grid from that file, and prints a message indicating whether the input file is a valid Sudoku solution.

Important: organize your code so that input of the grid, output from the grid, and verifications are three separate functions in your code: input(), verify(), output(). You can do this organization from the start, or you can run one long program, and then when everything is working fine break it into separate functions. See section Implementation Levels for more details on passing the grid to those functions.

4 Second step, solver program

To solve Sudoku, the program will read a partially completed grid and determine the candidates for each empty cell.

4.1 Modified verification

Although the basic scheme to check Sudoku is simply to compare numbers within all rows and within all columns that they don't contain any duplicated numbers, and to make sure each of the boxes are also Sudoku, now you are asked to look at the problem from each cell's point of view. That is, for a given cell (a cell can be represented by the struct Point (given) or by a pair of numbers *(x, y)*, one would check to see if the box, and the row and the column to which this cell belongs meet the rules of a Sudoku puzzle. That is,

1. Every number in this column is different from the value at the current cell;
2. Every number in this row is different from the value at the current cell;
3. Every number in the box around this cell is different from the value at the current cell.

Checking whether a row or a column contains numbers different from the one in the current cell is not difficult. But how does one identify the box containing the current cell? The key is to identify the starting

row and column of the box that contains this cell. If the size of the box is *k x k* and the index of the current cell is *(x, y)*, then

startingRow = x div k * k;
startingCol = y div k * k;

Think about this, use a few real numbers to test the idea and make sure you understand it.

4.2 Recursive Algorithm to Solve the Puzzle

Here is a recursive algorithm for solving a Sudoku grid, if you call the function *canSolve()*.

1. Find the next empty cell in the grid (have a separate functions for this).
2. If there is no empty cell, the grid is solved.
3. Try successive integers between 1 and 9 in the empty cell using the *canSolve()* function (recursively) until one is found that results in a valid grid and the remainder of the grid can be solved.
4. If no integer is found, the current grid is not solvable.

This algorithm is guaranteed to find a solution if one exists.

Here are some suggestions that should make your job easier.

1. You should implement this algorithm using a recursive function. You will probably need some helper functions too.
2. Have your function accept a cell location as an input value. The location of this cell will tell your method where to start looking for the next empty cell.
3. Have your function return an int value that indicates whether it was successful. It should return 1 if it was able to find a solution, 0 otherwise.
4. If you discover that no integer works in an empty cell, be sure to zero that cell when you are finished. Otherwise, the backtracking will not work.

5 Test Your Program

For each puzzle that your program processes, it should print out the file name, the unsolved puzzle and the solved puzzle.

Test your program on four of the sample problems provided first, leaving the worst-case.txt as the very last test case. The data file wrong.txt contains a setting that is not solvable.

The time needed to solve the puzzles varies for each file greatly. Notice the times taken by different provided files (you can also search the web for some existing Sudoku puzzles and try to solve them too. Let me know of interesting cases). I have a Python implementation of this project. When you're done, we'll compare Python and C programs side-by-side to see how much more efficient (if at all) C implementation is over a Python implementation.

6 Code quality

Your code should be self-explanatory for any CS literate user (not just your TAs or professors who know what the project is about). It should be free of legacy code (commented-out sections of code), should contain a reasonable number of comments, including the header comments. There should not be

explicit numbers in the code except 0 and 1 (occasional 2s are fine, avoid using 9 for the dimensions and other explicit constants. Use constants instead (ch. 18.6)).
Ideally, you should have the following functions input(), output(), verify(), is_row_valid(), is_col_valid(), is_block_valid(), [from step 1], verifyCell(), canSolve(), find_empty() [from step 2](and possibly more). Main() function should not be long (one screen or less is ideal).

The flow of this program is linear in almost all places (except the recursive canSolve(), that backtracks). The Sudoku grid is stored in a two-dimensional array.

7 Implementation levels

There are several levels of implementing this project.

Got-it-running: project implements the required functionality with code divided into functions but without passing Sudoku grid to functions. I.e. the grid is the "global" variable, that is visible to all functions in the file.

Nicely modular: grid array has fixed size. It is passed to the functions as an array of fixed size. There is no "global" grid variable; the grid is first defined in main() function.

```
void myfunc(int arr[M][N]) { // M is optional, but N is required
  ..
}

int main() {
  int somearr[M][N];
  ...
  myfunc(somearr);
  ...
}
```

Pointer guru: memory for the array is allocated dynamically using malloc(). When array is passed to functions you pass a pointer or a pointer to pointer.

Nicely Modular is the standard way. Got It Running is a less preferred way that comes with a 10% grade deduction for the coding part. It is a good compromise if you are having troubles implementing the program in Nicely Modular way (even after grade deduction your overall grade for a working Got It Running will likely be higher than that for a non-working Nicely Modular). Pointer Guru is (just slightly) more advanced, and comes with a 10% bonus grade for the coding part.

8 Submit Your Work

Include files for the first and second steps, and a readme.doc file that describes the problems that you had (if any). Mention anything that is not working properly too and any special features that you implemented. Finally, include in readme.doc screenshots of your solutions for all sample files provided. See the course schedule for 2 project deadlines (Step 1 and Step 2).

References

[1] J.-P. Delahaye, “The science behind sudoku,” Scientific American, pp. 81–87, Jun 2006.
[2] A. Johnson. Solving sudoku. [Online]. Available: http://angusj.com/sudoku/hints.php
[3] M. Mepham. Solving sudoku. [Online]. Available: http://www.sudoku.org.uk/pdf/solving_sudoku.pdf
[4] Wikipedia. List of sudoku terms and jargon. [Online]. Available: http://en.wikipedia.org/wiki/List_of_Sudoku_terms_and_jargon