

Sudoku solver

Files submitted

makefile	Creates executable sudoku
sudoku.c	Contains the main function
solver.c	Contains recursive function for completing the Sudoku grid.
auxiliary.c	Contains auxiliary functions used for displaying tables and updating state
check.c	Contains functions for updating the state of the sudoku and checking if a value is suitable for its context (row, column ,box) : e.g. updateCell()
solver.h	Header files for the respective .c files above
auxiliary.h	
check.h	
g_Variables.h	Contains global variables and structs used by all files in the program
test	Contains the main method for testing the program, which reads pairs of problem and solution files from the directories below, outputs the current state of the Sudoku for each of them and the correct solution if the solution was incomplete.
testprob	Directory containing problem description files used to test the program
testsoln	Directory containing solution files used to test the program
test_general: Probfile.txt Solnfile.txt Probfile_hard.txt Solnfile_hard.txt	Directory contains two 9 x 9 sudoku problems of medium and hard difficulty levels Can be used to test the program using the command ./sudoku <filename> <filename>

Level of completion

» **Basic requirements completed:**

○ Usage:

```
./sudoku <full path to problem file> <full path to solution file>
e.g. : ./sudoku "D:\\c2\\probfile.txt" "D:\\c2\\solnfile.txt"
```

- The program reads the description from probfile.txt and the solution from solnfile.txt
- Correctly assesses the state of the Sudoku, displaying one or more possibilities
 - » Invalid problem
 - » Invalid solution
 - » Incomplete solution
 - » Valid solution
- If solution is incomplete, will fill the table with correct values and overwrite the current solnfile with the completed solution.
- Upon command: **./sudoku test** will run the **readSolve()** method on pairs of files from the **testprob** and **testsoln** directories solving the puzzles and outputting the states of the sudoku's before and after filling in the empty cells, if any.

Input format

- The solution has to contain the dimension of the grid, followed by the values of the grid in consecutive order, row by row, separated by spaces. Empty cells are marked with 0.
- The problem description inside probfile.txt has to have the following format:

Number of cages

Cage size cage sum coordinates(row <space> column) of cells within that cage

Note: the upper left corner of the Sudoku grid has coordinates [1][1]

Example using the Sudoku grid taken from the Practical Requirements:

probfile.txt	solnfile.txt
<pre> 29 2 15 8 6 8 7 2 3 1 1 1 2 3 15 1 3 1 4 1 5 4 22 1 6 2 5 2 6 3 5 2 4 1 7 2 7 2 16 1 8 2 8 4 15 1 9 2 9 3 9 4 9 4 25 2 1 2 2 3 1 3 2 2 17 2 3 2 4 3 9 3 3 3 4 4 4 3 20 3 7 3 8 4 7 3 8 3 6 4 6 5 6 2 6 4 1 5 1 2 14 4 2 4 3 3 13 5 2 5 3 6 2 3 20 5 4 6 4 7 4 3 17 4 5 5 5 6 5 3 6 7 2 6 3 7 3 4 27 6 1 7 1 8 1 9 1 2 8 8 2 9 2 2 16 8 3 9 3 4 10 7 5 8 4 8 5 9 4 3 17 4 8 5 8 5 7 2 12 5 9 6 9 2 6 6 7 6 8 3 20 6 6 7 6 7 7 4 14 7 8 7 9 8 8 8 9 2 17 9 8 9 9 3 13 9 5 9 6 9 7 </pre>	<pre> 9 2 1 5 6 4 7 3 9 8 3 6 8 9 5 2 1 7 4 7 9 4 3 8 1 6 5 2 5 8 6 2 7 4 9 3 1 1 4 2 5 9 3 8 6 7 9 7 3 8 1 6 4 2 5 8 2 1 7 3 9 5 4 6 6 5 9 4 2 8 7 1 3 4 3 7 1 6 5 2 8 9 </pre>
	<pre> solnfile.txt - incomplete version 9 2 1 5 6 0 5 0 0 0 0 4 9 3 1 1 4 2 5 9 3 8 6 7 9 7 3 8 1 6 4 2 5 8 2 1 7 3 9 5 4 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </pre>

Design & implementation

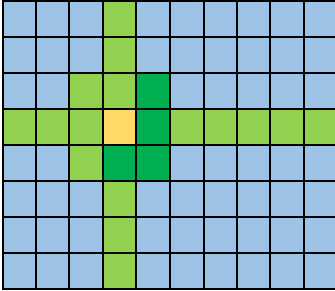
I have started implementing the program by finding a suitable input format for the data.

I decided to store the cage descriptions and the values of the Sudoku grid into two separate arrays:


- `arrCages[]` - one dimensional array of `Cage` structs
- `arrTable[][]` - two dimensional array of integers starting at [1][1]

The **dimension** of the grid and the number of cages (**nrOfCages**) are stored as global variables in `g_variables.h`. Cage descriptions are represented by structs, making it easy to access their individual attributes: **size**, **actualSum** and **expectedSum**.

Design & implementation continued



Thinking about complexity

The next step was to decide how to assess the state of the Sudoku and complete the solution. The specification suggested implementing a function that can be repeatedly called by within the recursive method for filling in the empty cells. This function would check the entire Sudoku grid each time we tried a possible value for that empty cell. This seemed highly inefficient as we only need to check the **context** of the empty cell we are trying to fill in. For example, filling in  would only require checking its row, column, box and cage.

Checking whether the value fits into the cage means checking whether it is smaller than **delta** = **expected** - **actual sum**. By repeatedly updating the **actualSum** attribute of that cage we can obtain the delta in constant time, without the need to sum all the values in that cage each time we try a new possible value for that empty cell.

Checking each row, column and box can also be done in constant time if we update the state of each of these structures as we are trying different values for each empty cell.

I decided to implement this using two dimensional arrays: **rows**, **cols**, **boxes** (= state arrays) where:

rows[i].values[x] == 1 means value **x** is found in row **i**.

Each of these arrays is actually a one dimensional array of a *Linear* struct (1d array of values)

*This is just for readability: avoids accessing elements with [i*total_columns+j]*

Thus, by updating the state arrays and the **actualSum** of the respective cages, we can decide whether a value is suitable for its context in constant time.

Assessing the state of the sudoku

Assessing the general state of the Sudoku can be done only once: while reading the solution and the problem files. After this, we are only concerned with the number of remaining empty cells in the incomplete Sudoku. Thus, we first read the solution file, filling in the **table[][]** array and we decide whether the solution:

- is **complete** - all cells have a valid value : in the range **[0,dimension]**
- **contains duplicates** or could be valid (we don't know the cage sums yet)
- has a **valid size** (dimension must be a perfect square)

If the grid has a valid size and all cells have a valid value, we can continue by reading the problem description and can now decide whether the sudoku problem-solution combination is:

- **valid**
 - × all cages were described
 - × all cells are uniquely included in a cage
 - × sum within completed cages is correct
- **solved** - all cells of **table[][]** are complete & their values satisfy the cage descriptions
- **invalid** - all cells complete but values don't satisfy cage descriptions

>> As we read the cage descriptions, we fill in the **arrCages[]** array and check the values of the cells corresponding to each cage. When we find an empty cell, we allocate it an **EmptyCell** struct which we then add to the **emptyCells[]** array. We pass into the struct the pointer to cell's cage, so that we can update the actual sum in that cage as we try different values for that empty cell.

>> We also **store pointers** to cell's row, column and box in the **EmptyCell** struct, in order to be able to instantly check whether the value we are trying for the empty cell is suitable for its context.

Solving the sudoku

In order to fill in the incomplete **table[][]** we recursively traverse the **emptyCells[]** array by passing the index of the empty cell within this array to the **fillEmpty()** function.

We use this function to try all the possible values for each empty cell. The **range** of possible values is determined by **delta** in cell's cage. We only try out the values that can satisfy cage's expected sum.

>> If, after checking the state of cell's row, column and box we find that the value is suitable, we update the **table[][]** array and the **actualSum** in that cage and call the **fillEmpty()** method on the next empty cell.
 >> If the value is not suitable we undo the changes and try the next possible value.
 >> If none of the possible values are suitable, we backtrack, returning -1: the control passes to the recursive call made by the previous empty cell and another combination is tried. If we have no more cells to fill, we return 1. Thus, if, after finding a suitable value for an empty cell and initiating a recursive call on the following empty cell, the function returns one, we have found a solution, which includes the current value of the respective empty cell.

Testing

In the sections dedicated to normal & exceptional input I have used the problem from the Practical Specification and repeatedly modified it to suit various test cases.

Normal input:

Valid problem. Correct solution.

```
Current solution is:
2 1 5 6 4 7 3 9 8
3 6 8 9 5 2 1 7 4
7 9 4 3 8 1 6 5 2
5 8 6 2 7 4 9 3 1
1 4 2 5 9 3 8 6 7
9 7 3 8 1 6 4 2 5
8 2 1 7 3 9 5 4 6
6 5 9 4 2 8 7 1 3
4 3 7 1 6 5 2 8 9

State of the sudoku:
Solved :      Solution is valid.
```

Valid problem. Incorrect solution.

```
Current solution is:
2 1 5 6 4 7 3 9 8
3 6 8 9 5 2 1 7 4
7 9 4 3 8 1 6 5 2
5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5
8 2 1 7 3 9 5 4 6
5 5 5 5 5 5 5 5 5
4 3 7 1 6 5 2 8 9

State of the sudoku:
Invalid Solution :      Wrong sum in one or more cages.
```

Valid problem. Incomplete solution.

```
Current solution is:
2 1 5 6 4 7 3 9 8
3 6 8 9 5 2 1 7 4
7 9 4 3 8 1 6 5 2
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
8 2 1 7 3 9 5 4 6
0 0 0 0 0 0 0 0 0
4 3 7 1 6 5 2 8 9

State of the sudoku:
Incomplete solution :   Some cells are incomplete.

Please wait... Solving the sudoku...

Successfully completed sudoku.
Table is:
2 1 5 6 4 7 3 9 8
3 6 8 9 5 2 1 7 4
7 9 4 3 8 1 6 5 2
5 8 6 2 7 4 9 3 1
1 4 2 5 9 3 8 6 7
9 7 3 8 1 6 4 2 5
8 2 1 7 3 9 5 4 6
6 5 9 4 2 8 7 1 3
4 3 7 1 6 5 2 8 9

Written solution to file: D:\c2\solnfile9.txt
```

Testing the `solnfile9.txt` again, to check that in the test case on the left we have actually written the correct solution after we solved the puzzle.

```
Current solution is:
2 1 5 6 4 7 3 9 8
3 6 8 9 5 2 1 7 4
7 9 4 3 8 1 6 5 2
5 8 6 2 7 4 9 3 1
1 4 2 5 9 3 8 6 7
9 7 3 8 1 6 4 2 5
8 2 1 7 3 9 5 4 6
6 5 9 4 2 8 7 1 3
4 3 7 1 6 5 2 8 9

State of the sudoku:
Solved :      Solution is valid.
```

Exceptional input:

Invalid problem - not all cages described.

Removed a few lines of cage descriptions from the probfile9.txt

```
$ ./sudoku "D:\\c2\\probfile9.txt" "D:\\c2\\solnfile9.txt"
State of the sudoku:
Invalid Problem :      Not all cages were described.
```

Invalid problem - some cells are in more than one cage.

Duplicated coordinates in the description of cages

```
State of the sudoku:
Invalid Problem :      Some cells are in more than one cage.
Invalid Solution :      Wrong sum in one or more cages.
```

Invalid problem - some cells not included in a cage.

Removed one cage description and decremented the nrOfCages, so that all cages are described

```
State of the sudoku:
Invalid Problem :      Not all cells were included in a cage.
```

Invalid solution - wrong sum in a cage.

Replaced a few sum values with random numbers

```
State of the sudoku:
Invalid Solution :      Wrong sum in one or more cages.
```

Invalid problem - invalid sum in a cage.

Replaced the sum in one cage with a negative number, without changing the actual values in the grid

```
State of the sudoku:
Invalid Problem :      Invalid sum in one or more cages.
Invalid Solution :      Wrong sum in one or more cages.
```

Invalid solution - cell value without acceptable range.

Replaced one of the cell values with
a) 88
b) -75
and modified the sum in that cage accordingly

```
State of the sudoku:
Invalid Solution :      Found an invalid value in the solution.
```

Solution is in wrong format: dimension not a perfect square
Replaced dimension with -18

```
State of the sudoku:
Invalid Problem :      Dimension of the grid must be a perfect square.
```

Automated testing : `test()` method

The `test()` method reads pairs of files from the `testsoln` and `testprob` folders and uses the `readSolve()` method to output the state of each Sudoku problem, solve the puzzles with incomplete solutions and write the correct solution to the corresponding files inside `testsoln` directory.

I ran the program on Sudoku puzzles of size **4 x 4**: files `100,200,300,400` → e.g. `prob100.txt` and **9 x 9** : `101,102,201,202,301,302,401,402` → e.g. `prob101.txt`

File number	Test case	Result ; Green = passed
100,101,102	All solutions are incomplete.	Solved all, although Sudoku 902 took a very long time to solve.
200,201,202	All solutions are partially complete	Correctly completed all Sudoku grids
300,301,302	Valid solution Invalid problem 300 – not all cages described 301 – some cells are in more than one cage 302 – not all cells have a cage	Appropriate message displayed in each case.
400,401,402	Valid problem Solution complete but invalid 400 – duplicate values in a row 401 – duplicate values in a column Both of the above -> also wrong sum in one+ cages 402 – invalid value in one+ cells	Appropriate message displayed in each case.

```
$ ./sudoku test
Will test sudoku:

soln_prob100.txt      prob100.txt
-----
State of the sudoku:
Incomplete solution :   Some cells are incomplete.

...Solving sudoku...
test:      Successfully solved sudoku:

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

soln_prob101.txt      prob101.txt
-----
State of the sudoku:
Incomplete solution :   Some cells are incomplete.

...Solving sudoku...
test:      Successfully solved sudoku:

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

```
soln_prob202.txt      prob202.txt
-----
State of the sudoku:
Incomplete solution :   Some cells are incomplete.

...Solving sudoku...
test:      Successfully solved sudoku:

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

Below is the program output to the command
`./sudoku test`
with files `100 101 102 200 201 202`

```
soln_prob102.txt      prob102.txt
-----
State of the sudoku:
Incomplete solution :   Some cells are incomplete.

...Solving sudoku...
test:      Successfully solved sudoku:

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

soln_prob200.txt      prob200.txt
-----
State of the sudoku:
Incomplete solution :   Some cells are incomplete.

...Solving sudoku...
test:      Successfully solved sudoku:

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

soln_prob201.txt      prob201.txt
-----
State of the sudoku:
Incomplete solution :   Some cells are incomplete.

...Solving sudoku...
test:      Successfully solved sudoku:

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

Below are the results for files 300 - 402

```
soln_prob300.txt          prob300.txt
-----
State of the sudoku:
Invalid Problem :      Not all cages were described.
test:                Invalid problem or solution

soln_prob301.txt          prob301.txt
-----
State of the sudoku:
Invalid Solution :     Wrong sum in one or more cages.
test:                Invalid problem or solution

soln_prob302.txt          prob302.txt
-----
State of the sudoku:
Invalid Problem :      Not all cells were included in a cage.
test:                Invalid problem or solution

soln_prob400.txt          prob400.txt
-----
State of the sudoku:
Invalid Solution :     Grid contains duplicates.
test:                Invalid problem or solution

soln_prob401.txt          prob401.txt
-----
State of the sudoku:
Invalid Solution :     Grid contains duplicates.
test:                Invalid problem or solution

soln_prob402.txt          prob402.txt
-----
State of the sudoku:
Invalid Solution :     Found an invalid value in the solution.
test:                Invalid problem or solution
```

Note on performance

The program can solve an easy - medium Sudoku problem in less than 5 seconds, as shown by tests using the two 9 x 9 empty grids.

A Sudoku problem such as the one named **prob_hard.txt** took 3 minutes to solve when 72 out of 81 cells were incomplete and more than 10 minutes to solve from scratch.

For all partially incomplete problems, where less than 70% of the grid is empty, the problem is solved instantly.