CS2002:Practical3-C2 Sudoku Solver 150015227 7th March 2017

Overview:

This practical required writing a Sudoku solver in C. This involved successfully reading in a sudoku puzzle through standard input and representing it using a struct. We were to have a program called sudoku_check which used functions to check the sudoku, providing output as to whether it was invalid, incomplete or complete. Furthermore, we were to have a program called sudoku_solve which would solve a given sudoku recursively, printing out the solution, or printing out 'MULTIPLE' if the sudoku had more than one solution or 'UNSOLVABLE' if the sudoku was unsolvable. Finally, the extension for this practical was to extend the solver by making a new program called sudoku_advanced, which used more complex techniques to solve a sudoku faster. A makefile was to be submitted to be able to make the programmes sudoku_check, sudoku_solve and sudoku_advanced.

Design & Extensions:

I decided to start off by having a 'sudoku.h' file, containing function/struct declarations to be used by all the files. In this file, I included:

- Sudoku struct declaration, which just contains an integer 'n' and a 2d array called puzzle, as I thought a 2d array would be the most sensible and easy to understand/use representation of a sudoku puzzle
- 'enum' check_output, the 3 possible outputs from the sudoku_check program and functions
- 'enum' solve_output, the 3 possible outputs from the sudoku_solve program and functions
- a number of functions to free a sudoku, print a sudoku, copy a sudoku, check if a number in a given position is valid, check if a sudoku is complete/invalid/incomplete, get a sudoku from standard input, and solve a sudoku.

sudoku_io.c:

This file contains the implementations to print a sudoku, free a sudoku and copy a sudoku. They all take in the 2d array representing a puzzle, and an integer 'n' to represent the size. All of my functions take in this integer which represents the size, as I thought it made more sense to do this rather than having the same global variable to represent the size in each file which would always have to be checked. The print function contains extra spaces to be printed out before each value if the value is less than 10, in order to fit the output from stacscheck. The freeSudoku function simply assigns all the values of a sudoku to 0 (although I never actually use this function, I decided to include it as it is required in the specifications), and the copySudoku function simply creates a new array and copies into it the array passed, before returning the new array.

sudoku_checker.c:

This file contains the functions used to check a sudoku. I decided to have a function called check_num rather than check_list as the specification suggests, as I thought it would be faster and more efficient to do it this way. check_num checks that a number is valid in the given position in the sudoku, returning incomplete if the number is 0, invalid if the number can't be in that position, and complete if it can. Then, in check_sudoku, I go through all the positions in the sudoku, using check_num on each value. If any value is invalid, it returns invalid. At the end, if any value was a 0, it returns incomplete, otherwise it returns complete. Finally, this file contains the function getSudoku, which is used to read in a sudoku from standard input, and returns the new sudoku struct, with 'n' and 'puzzle' assigned.

sudoku_checker_main.c:

This is the file that contains the 'main' function to check a sudoku. It simply gets the sudoku, prints it out and checks the sudoku using the above functions, and then prints out what the check function returned (invalid, complete or incomplete).

sudoku_solver.c:

This file contains 1 function: solve, which solves a sudoku recursively. It takes in the sudoku to be checked, its size and returns solve_output (unsolvable, multiple or solved). It firstly checks if a sudoku is invalid or complete (in which case it returns), otherwise (incomplete) it goes on. It then finds the position of the first zero in the sudoku, and loops through all values (1 to the square of 'n') that could fill that position. It uses check_num to only fill that position with a valid number. Once the position has been filled with a valid value, a recursive call is made to fill in the rest of the sudoku. If this eventually returns solved, a flag is raised to show a solution has been found, and the solved sudoku is assigned to a new 2d array correctPuzzle, and then it keeps going. If another solution is found, 'multiple' is returned, otherwise if the end is reached without a different solution, the correctPuzzle is copied into the original sudoku's puzzle, and solved is returned. Otherwise, if correctPuzzle is empty, unsolvable is returned.

sudoku_advanced.c:

This file contains the other solve function, for the extension. I have an extra "checker" function in this file which is used to find the single valid number for a cell. If there is more than one, -1 is returned, if there are none, 0 is returned (sudoku is invalid), otherwise the value itself is returned. I then have multiple functions which attempt different ways of solving the sudoku. These all take in the pointer to the sudoku being solved, so it can be edited directly. solve_cells simply goes through all the cells of the sudoku and fills a cell in if it only has 1 possible value, using findValidNums. solve_box goes through each 'box' in the sudoku, finds the numbers that don't appear in the box (which they all must), finds the possible places for these numbers within the box, and if there is only 1 place, fills in the value. Similarly, solve_row does the same but instead of going through the values in a box, it goes through the values of a row, and solve_col does the same but goes through the values of a column. All of these functions return the number of edits they made to the sudoku. The isSolved method is simpler than check_sudoku and so I included it for speed reasons, as it only checks if the puzzle has a 0 in it, returning 0 if it does and 1 if it doesn’t. The solve_recursive method I use is the exact same as the solve method in the sudoku_solver.c, except instead of recursively calling itself, it recursively calls the solve function. The solve function is iterative as well as having some recursion. It firstly checks the sudoku, and if incomplete, goes into a while loop until the puzzle has no more zeros in it. It calls all the above functions to try and complete the sudoku, calling each function twice in the opposite order, so each function has the chance to make edits after the other functions have made their edits. If no edits were found, it resorts to calling the recursive function to try and put a random possible value in a cell, and tries solving again.

sudoku_solver_main.c:

This is the file that contains the main function to solve a sudoku. It simply gets a sudoku, uses a clock_t variable (from time.h) to get the start time, calls the solve function (different if compiled with sudoku_solver.c and sudoku_advanced.c), and prints out the output from the solve function. If it was solved, it writes the time it took it in seconds in a new file called “times.txt” (adds to file, doesn’t overwrite), and prints out the sudoku.

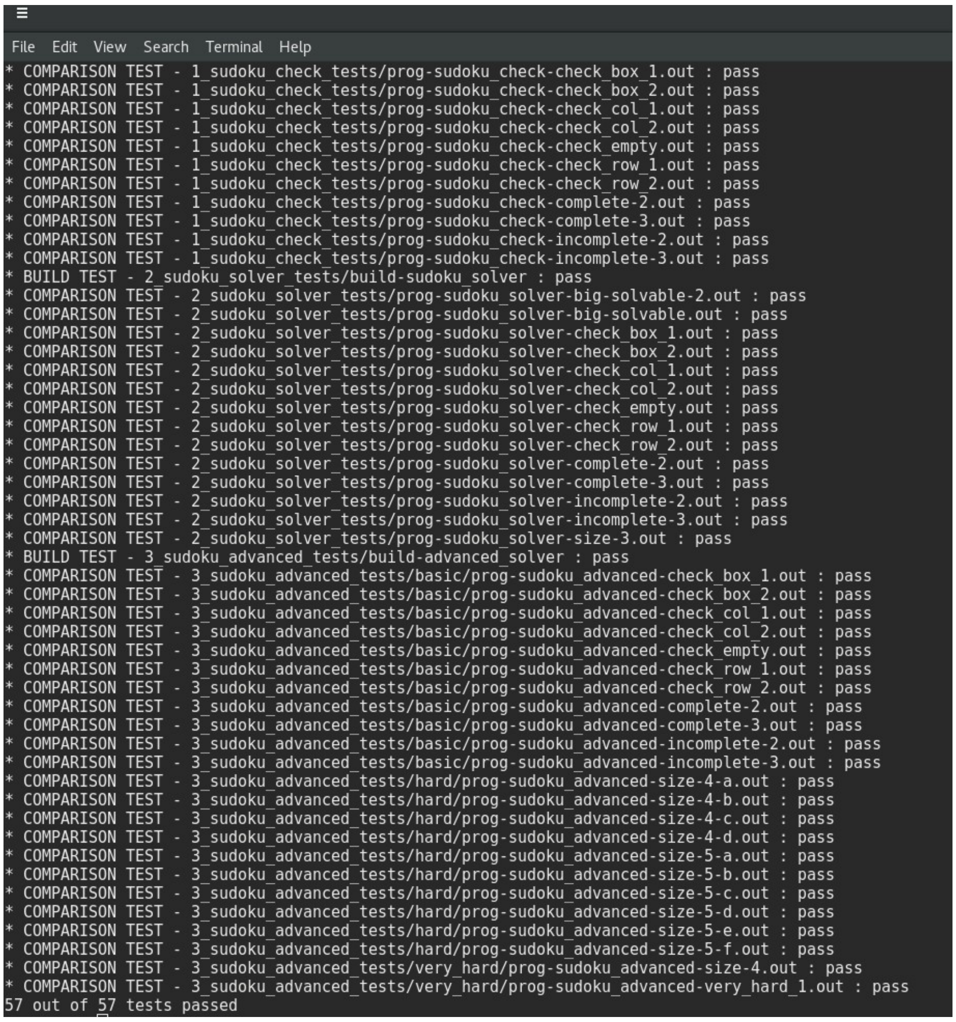
Makefile:

Using the makefile, one can make all, make clean, make sudoku_check, make sudoku_solver, and make sudoku_advanced.

Testing:

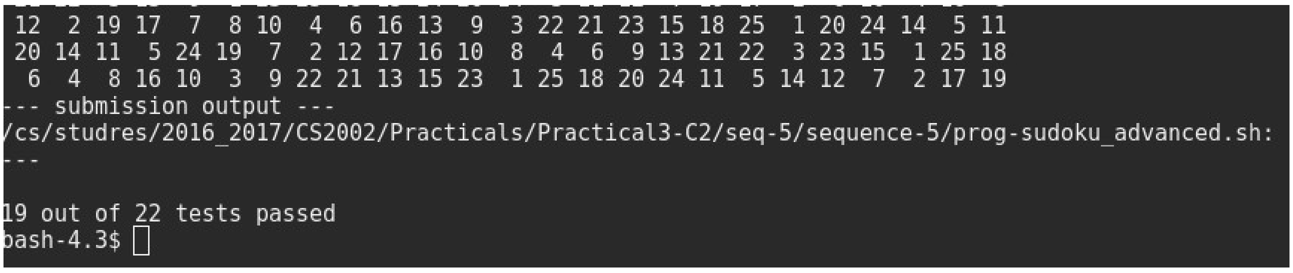
To test my program, I mainly used stacscheck through terminal, and I also would copy/paste sudokus directly from stacscheck to my console. The following screenshots prove that my program passes ALL the stacscheck tests, 19 out of the 22 ‘seq-5’ tests and 14 out of the 15 ‘seq-9’ tests. The ones that fail are because of the fact that my programme took too long, so stacscheck stopped. For further testing, I used the ‘times.txt’ file to compare the 2 solve functions.

Stacscheck:



```
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_box_1.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_box_2.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_col_1.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_col_2.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_empty.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_row_1.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-check_row_2.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-complete-2.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-complete-3.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-incomplete-2.out : pass
* COMPARISON TEST - 1_sudoku_check_tests/prog-sudoku_check-incomplete-3.out : pass
* BUILD TEST - 2_sudoku_solver_tests/build-sudoku_solver : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-big-solvable-2.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-big-solvable.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_box_1.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_box_2.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_col_1.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_col_2.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_empty.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_row_1.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-check_row_2.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-complete-2.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-complete-3.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-incomplete-2.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-incomplete-3.out : pass
* COMPARISON TEST - 2_sudoku_solver_tests/prog-sudoku_solver-size-3.out : pass
* BUILD TEST - 3_sudoku_advanced_tests/build-advanced_solver : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_box_1.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_box_2.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_col_1.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_col_2.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_empty.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_row_1.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-check_row_2.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-complete-2.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-complete-3.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-incomplete-2.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/basic/prog-sudoku_advanced-incomplete-3.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-4-a.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-4-b.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-4-c.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-4-d.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-5-a.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-5-b.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-5-c.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-5-d.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-5-e.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/hard/prog-sudoku_advanced-size-5-f.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/very_hard/prog-sudoku_advanced-size-4.out : pass
* COMPARISON TEST - 3_sudoku_advanced_tests/very_hard/prog-sudoku_advanced-very_hard_1.out : pass
57 out of 57 tests passed
```

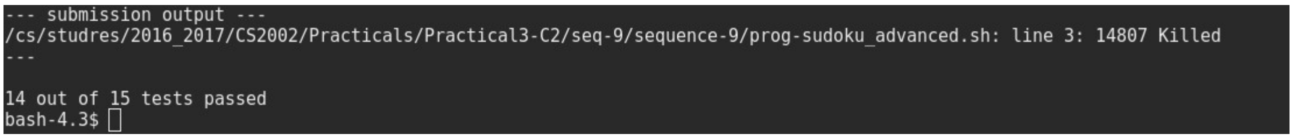
Seq-5:



```
 12  2 19 17  7  8 10  4  6 16 13  9  3 22 21 23 15 18 25  1 20 24 14  5 11
 20 14 11  5 24 19  7  2 12 17 16 10  8  4  6  9 13 21 22  3 23 15  1 25 18
  6  4  8 16 10  3  9 22 21 13 15 23  1 25 18 20 24 11  5 14 12  7  2 17 19
--- submission output ---
/cs/studres/2016_2017/CS2002/Practicals/Practical3-C2/seq-5/sequence-5/prog-sudoku_advanced.sh:
---

19 out of 22 tests passed
bash-4.3$
```

Seq-9:



```
--- submission output ---
/cs/studres/2016_2017/CS2002/Practicals/Practical3-C2/seq-9/sequence-9/prog-sudoku_advanced.sh: line 3: 14807 Killed
---

14 out of 15 tests passed
bash-4.3$ 
```

I included sudoku_solver_times.txt and sudoku_advanced_times.txt files. To generate the first, I ran all the stacscheck tests using JUST the sudoku_solve solve function(made edits to the makefile), and then did the same using JUST the sudoku_advanced solve function. Therefore, you can compare the speed of the 2 solve functions over the same sudokus, which shows how much faster sudoku_advanced is. Furthermore, sudoku_advanced could solve all the puzzles before stacscheck timed out, whereas sudoku_solver could only solve the first 13 (not counting the unsolvable/multiple cases). As you can see, the sudoku_advanced function only took 92 seconds to complete the very hard puzzle.

Running any of the solve functions/running stacscheck itself will edit the times.txt file and add the times that it took for each puzzle that was solved. If you want to test a puzzle, empty the contents of times.txt, and then run only the puzzle, so that it is simpler to know the time for that specific puzzle.

Evaluation & Conclusion:

Overall, I greatly enjoyed this practical, as it made me properly familiarise myself with the C language, which I now feel much more comfortable with, as I got to explore functions like malloc, free, fgets, writing to files, and getting execution time. I found it very interesting to try and write a sudoku solver, as it is something I always had fun solving, but needed to translate my methods to a computer. The sudoku_check and sudoku_solve programmes were not too complicated, but the methods I used in sudoku_advanced were more challenging to code, but really make the solver much quicker. The areas I had trouble with were the maths behind solving the values within a box, and even more so in actually reading in the puzzle from standard input. Using the demonstrators’ help, I managed to understand how to properly use malloc on a 2d array (which I was having trouble with), as well as correctly using the fgets method. In conclusion, I believe the fact that my solver can solve almost all the puzzles, including the very hard one in 2 minutes, proves that it is very quick, effective and clever in its solving techniques.