140012056 Ian 3rd Thursday 2015

CS2002 Practical03

Overview:

For this practical we were asked to write a solver for the killer Sudoku problem. There were two main steps in which we had to implement. One being to write a killer Sudoku checker and the other to write a killer Sudoku solver.

Design:

I have tackled the step 1 and step 2 problems separately. Each with a design for themselves. In my solution they are two different applications.

I first thought about the best way to design the file format to expressing killer Sudokus. The end file format I used was in the following way: Side length followed by cage sum, cage size and then the co-ordinates of the cage. I used this because the killer sudoku relies heavily on cages to work and this seemed the best way in order to format it. It allowed one to figure out exactly all the elements each cage needed.

Step 1:

The Sudoku checker uses 4 files. The file names being sudoku.c, readfiles.c, mkaechecks.c and finally checkerHeader.h. checkerHead.h is where the header files is where I write the structs in order to define the data structure to hold the information about the cages. Sudoku.c is where the main method lies. This is where the output of INVALIDPROBLEM, INVALIDSOL, INCOMPLETE and SOLVED are printed out. Readfiles.c is where I read in the two files, one that contains the solution and the other that holds the problem file. Makechecks.c is where all the checks are made for the killer sudoku. 6 main checks are made. The checks are: checkCages() this checks if all cages have summed up to the correct total. This is used to check if the squares are part of the cage. checkProblemValid() checks if the cells add up to the total size of the cage. This is used to check if a problem is invalid. CheckNoDuplicatesCages() checks if there are no duplicates in a cage. CheckSudokuGrid checks if the columns and rows of the Sudoku grid have any duplicates. CheckRowColSums() checks if the rows and columns add up to the correct total they are supposed to. ValidBoxes() checks if the boxes are valid and that each one contains unique numbers.

Step 2:

The solver is very similar to that of step 1. The main difference is that it only takes in one file and solves it instead of checking for only validity. I first read the file in and store it into the same data structure previously done. I then have a recursive backtracking algorithm in order to solve the Sudoku problem. The solver calls four checks within it. These check to see if the element being placed if valid or not. The checks are similar to those in step 1 in the sense that they check the same things. However they do not check them as thoroughly as they do not need to. It also makes checking more efficient this way.

Testing:

I have taken quite a few Screen shots of my program working and hence have included an extra pdf that contains the testing. The pdf name is called “testing”.

Extension:

I have made my program so that it is able to take in multiple file sizes. I did this by using malloc as I did not know the size of the array I would need until I read the file in. As I use the malloc to the very end I did not see the need to free it as once the program ended it would free it anyways.

Evaluation:

I think I've finished the requirements of this practical quite well. Testing shows that the code runs as it is supposed to. The two solutions are quite different. However the solver draws heavily from the checker as the solver needs to do its own checking as well. I think I could have merged the two solutions together into one application. However at the beginning of the planning of the practical out I had chosen to write two applications as this made more sense with out hindsight.

Conclusion:

I found this practical to be enjoyable. It was quite challenging to do however. The most difficult parts I found were when trying to debug the solver. Finding out where a bug lied whilst recursively calling so many checks was a challenging task. I find using malloc and structs in c easier then I had done before.