FINAL PROJECT

SUBMISSION DUE DATE: 20/09/2019 23:00 (FINAL DEADLINE! NO EXTENSIONS!)

INTRODUCTION

In the final project you will create an interactive **Sudoku** program with an ILP (Integer Programming) solver component. You will use ILP to both solve Sudoku puzzles and check possible solutions to it. ILP uses mathematical optimization methods to speed up the solution time, compared to the exponential backtracking algorithm from HW3.

The Sudoku program should work in a similar way to the program you implemented in HW3, with user interaction performed through console commands.

Important! Tutorial 3 and the forums are an integral part of the project, and you should follow both closely for additions, clarifications, and/or changes.

HIGH-LEVEL DESCRIPTION

The Sudoku project should have the following functionalities:

- Solving Sudoku puzzles
- Editing existing and creating new Sudoku puzzles
- Supporting puzzles of any size
- Validating the board
- Saving and loading puzzles from files
- Randomly generating new puzzles
- Finding the number of solutions for a given board
- Auto-filling option for obvious cells (where only a single value is valid)

This document will detail the requirements for your project, and the rest is up to you and part of your grade. There are no specific requirements for output, modules, etc., unless otherwise specified.

The program will have three main modes: ***Init***, ***Edit*** and ***Solve***. In the Solve mode, the user attempts to solve a pre-existing puzzle. In the Edit mode, the user edits existing puzzles or creates new ones, and saves these for opening in the *Solve* mode later. In the *Init* mode, the user loads a file to enter either *Edit* or *Solve* mode. The program starts in *Init* mode.

The project consists of 5 main parts:

- Sudoku game logic
- Console user interface
- ILP solver algorithm
- Random puzzle generator
- Exhaustive backtracking solver (for determining the number of different solutions)

The executable will be named "*sudoku-console*". It receives no command-line arguments.

SUDOKU

Sudoku is a number-placement puzzle where the objective is to fill a NxN grid (usually 9x9) with digits so that each column, each row, and each block contain all digits from 1 to N.

		3	10	7				6	
8	6					5			
	10	9	4		2	6		1	
				5	7			4	
4	5	1						2	6
	8				10				1
							1	3	
	3	5					6		7
		4					9		
1						10	5		

The grid consists n x m blocks of size m x n each, for a total of N=mn rows and columns. For example, a 10x10 grid can consist of 5x2 blocks, each of size 2x5 cells (see figure above, where n=5 and m=2).

For each cell in the grid, we denote by its neighbors the other cells in its row, column, and block. The main rule is that the value of each cell is different from the values of its neighbors.

A Sudoku puzzle is completed successfully once all N^2 cells are filled with legal values.

Note that the Sudoku puzzle described here is different from the one described in HW3, as its size varies. We will limit both *n* and *m* to 5; however, this is for practical use only and your implementation should not rely on this limitation *in any way*.

DETAILS

The guidelines below should help you understand how your program should work. The implementation details (such as data structures, modules, etc.) are up to you and will be considered for grading.

OVERVIEW

The Sudoku program operates in either "Edit mode" or "Solve mode" (or "Init mode" at the beginning of the program).

When the user enters the Solve mode (via the "solve" command), it is given a puzzle to solve. The user may save the puzzle at any time to resume it later.

When the user enters the Edit mode (via the "edit" command), it may freely edit a puzzle and save it to a file. Files can be loaded in either the Edit mode ("edit" command) for updates, or the Solve mode ("solve" command) for solving them.

When saving a file in the Edit mode, the board is validated (i.e., make sure that a solution for this puzzle exists), and all filled cells are marked as "fixed". Such cells cannot be edited by the user in the Solve mode.

At any point in the program the user may use the "exit" command to exit the program. Before exiting, make sure all resources are freed, all files closed, etc. You do not have to exit cleanly on errors.

GAME PARAMETERS

Here we describe two game parameters that are central to the operation of the program.

MARK ERRORS

The program will keep a "mark errors" parameter with a value of either 1 (True) or 0 (False). This parameter determines whether errors in the Sudoku puzzle are displayed.

When editing or solving a puzzle, the user is free to enter any input, even erroneous one (according to the rules of Sudoku). An erroneous board is a board with cells which contain illegal values, i.e., the same value for two or more neighbors.

When this parameter is set to 1, erroneous values are displayed followed by an asterisk. All cells that participate in an error should be marked. The default value is 1.

In Edit mode we always mark errors, regardless of the parameter's value.

UNDO/REDO LIST

Throughout a single game (or puzzle) the program maintains a *doubly-linked* list of the user's moves, and a pointer marking the current move.

Whenever the user makes a move (via "set,", "autofill", "generate", or "guess"), the redo part of the list is cleared, i.e., all items beyond the current pointer are removed, the new move is added to the end of the list and marked as the current move, i.e., the pointer is updated to point to it.

The user may freely traverse this list (move the pointer) using the "undo" and "redo" commands, which also update the board according to the move to undo or redo.

Note that when starting a new puzzle (by "solve" or "edit") the undo/redo list is cleared and starts **empty**, even if certain cells already contain values. These initial values are not considered moves and cannot be undone.

PROGRAM FLOW

When the program starts, it prints the title (your choice) and awaits user commands. The initial game mode is *Init*.

The user enters commands and, whenever loading a new board (via the "edit" or "solve" commands), or successfully updating the current board (via "set", "autofill", "undo", "redo", "generate", "guess", or "reset") the board is printed to the user **after** the command's output. We consider the board updated whenever such a command executes successfully, even if the board was not changed (e.g., setting a cell value to the same value it already contains). If the board is printed following a command, printing the board is always the last output of the command, after which we read the next user command.

We assume ***nothing*** regarding user input, and the program should continue operating no matter what the user inputs. If the user enters an invalid command, you should print an error that no such command exists. If the user enters a command with extra parameters, print an error that too many parameters were entered (likewise for a command with not enough parameters). In both cases, instruct the user of the correct syntax of the command it was trying to execute. If a command was entered with the correct number of parameters but with the incorrect value, inform of an error in the first parameter that is incorrect, and instruct the user of the legal range of values for that parameter. In general, make sure any erroneous input is treated with a proper, descriptive error message that instructs the user how to correct it.

Treat each line of input as a new command, i.e., no command may span more than one line and no line may contain more than one command. Recall that, unlike HW3, a command with extra parameters is invalid! Additionally, you should ignore blank lines, i.e., lines that contain only white-space characters are ignored and no output is provided.

While editing or solving a board, the user enters values into cells. The user always has the possibility to enter erroneous values (i.e., a value that already exists in one of the cell's neighbors); however, these cells will be clearly marked if the "Mark errors" parameter is set to 1, or the program is in *Edit* mode in which case errors are always marked, regardless of the actual value of the parameter. Additionally, puzzles with erroneous values may never be saved in **Edit** mode (but *can* be saved in **Solve** mode).

Additionally, in **Solve** mode the user may only enter (or edit) values into cells that are **not** *fixed*, i.e., they were not part of the original puzzle. In **Edit** mode, the user may enter values into any cell, as no cell is considered (or marked as) fixed. When saving a puzzle to a file in **Edit** mode, all cells containing values are immediately considered *fixed* and saved as such.

While no specific outputs are required, you need to make sure that the output is consistent. That is, similar errors, in the same or different commands, should result in the same error messages, formats should be similar, and the output of the program should be consistent throughout any execution.

BOARD PRINT FORMAT

The only output which you must follow exactly is when outputting the board. Your board output should match this format exactly.

Recall that the board consists of NxN cells, arranged in n-by-m blocks (n rows of blocks, m columns of blocks) of size m-by-n (m rows of cells, n columns of cells).

The board printing format should consist of N+n+1 rows, consisting of two types:

1. Separator row – a series of 4N+m+1 dashes.
2. Cells row – a pipe character '|' starts and ends the row, and separates each set of n cells (i.e., each block). Unlike in HW3, no space separates cells and pipes beyond the space characters defined here. Each cell is represented as 4 characters:
    a. A space character.
    b. Two digits for the cell value (printf format "%2d"). Use two spaces instead of a value for a blank cell.
    c. A dot '.' for a fixed cell, an asterisk '*' for an erroneous cell when in **Edit** mode or the "Mark errors" parameter is set to 1, or space otherwise.
    A fixed cell can never be erroneous!

The board will be printed in the following format:

1. Separator row
2. Repeat n times for each row of blocks:
    a. Cell row
        i. Repeat m times for each of the row of cells
    b. Separator row

The board example from HW3 is repeated here in the new format:

```
 -------------------------------------
 |            |            |      3.  |
 |         6  |      1.    |  8   4   |
 |      5.  3 |  6.  4   7.|  9.      |
 -------------------------------------
 |  5       8.|      6   3.|      9   4.|
 |  3.      9*|          8.|  2   7.    |
 |  4       9*|  1.  2.    |  5   8     |
 -------------------------------------
 |      8.    |  5   3.    |            |
 |  1   7.    |          4 |  3   6     |
 |      3   2.|  9.      1.|          8 |
 -------------------------------------
```

INTEGER LINEAR PROGRAMMING (ILP)

Linear programming is a method to achieve the best outcome (maximum or minimum) in a mathematical model whose requirements are represented by linear relationships.

An integer linear program is a mathematical optimization program in which the variables are restricted to be integers, and the objective function and constraints are linear.

An integer linear program in canonical form is expressed as:

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \\ \text{and} & \mathbf{x} \in \mathbb{Z}^n, \end{array}$$

When given a problem, we can use an ILP solver to compute an optimum solution, i.e., assign values to the program variables to optimize the objective function while satisfying all the constraints.

In this project, you will use ILP to solve Sudoku puzzles for the user, either for validation or puzzle generation.

Further details on ILP and first steps for expressing Sudoku are provided in Tutorial 3.

The ILP solver we will use is the Gurobi Optimizer, a commercial optimization solver for linear programming. Gurobi supports a variety of programming and modeling languages, including a matrix-oriented interface for C which your program should use.

To use Gurobi, you should first register for an academic license using your TAU email: https://user.gurobi.com/download/licenses/free-academic

Gurobi is installed in *nova* and its code should be executed there. The current version installed in *nova* is Gurobi 5.6.3. You may also install Gurobi on your personal computer; however, do so at your own risk as there may be differences.

Our recommendation is to use Gurobi only on the *nova* server.

Gurobi guidelines and example code are provided in moodle.

SUDOKU GUIDELINES

Details for expressing Sudoku using ILP were given in **Tutorial 3**:

- Define a binary variable *Xijk* **only** for each *legal* value *k* of cell <i,j>.
- Set constraints that ensure a single value in each cell.
- For each row, column, block and cell, set constraints that ensure all available values appear exactly once.

Remember, it is important to minimize the number of variables and constraints we pass to Gurobi. This is the emphasis of this module. Do not allocate a huge array or transfer all constraints regardless of the current board state, do it smartly so that both Gurobi utilizes minimum parameters, and your code allocates and uses minimum memory.

For guessing values (the "guess" and "guess_hint" commands) we will be using linear programming, rather than ILP. In linear programming we do not limit the variables to be integers (or binary), thus a range of values are available, and solutions can be found in linear time (instead of exponential time in the worst case for ILP).

Linear programming will thus provide faster but (possibly) inaccurate solutions to the user.

Initial details for expressing Sudoku using LP were given in **Tutorial 3**:

- Use variables that are GRB_CONTINUOUS instead of GRB_BINARY.
- Constrain variables to the desired range.
- Determine a good objective function (trial & error!)
    - Using random weights in the objective function

EXHAUSTIVE BACKTRACKING

The exhaustive backtracking algorithm is another variant of the backtracking algorithm that we can use to count the number of solutions for a given Sudoku board.

The exhaustive backtracking doesn't terminate when the last cell is filled with a legal value. Instead, it marks the board as solved, incrementing a counter, and then continues to increment the value of that cell, or backtrack if necessary, in order to retrieve other potential solutions.

Thus, the algorithm exhausts all possible values for all empty cells of the board, counting all solutions found in this process. Once the algorithm backtracks from the 1st empty cell, the algorithm is finished, and the counter contains the number of different solutions available for the current board.

Note that for some boards (according to size and values), the exhaustive backtracking algorithm can be very slow – that is OK. The *nova* server terminates processes that take longer than 5-10 minutes, thus it is recommended to check your code on your computer, and only use small boards when testing the exhaustive backtracking algorithm on *nova*.

Important: for the project you are required to implement the exhaustive backtracking algorithm with an explicit stack for simulating recursion, rather than through recursive calls. You may not implement it with recursion! You must implement a stack data structure, and then use it to implement exhaustive backtracking.

Any recursive algorithm can be replaced with a stack, by replacing recursive calls with operations on the stack. Whenever a recursive call should be made, we instead add the relevant parameters to the stack (as a single item). Our code then iterates removing an item from the stack and performing the logic for it (which usually adds items to the stack) until the stack is empty. A modular stack implementation is part of this requirement.

COMMANDS

The interaction should be done via console commands. The user will interact with the program by typing commands from a list provided below, along with desired parameters. We describe below which commands are available in which mode of the program, as well as the specific details and arguments of each command.

The program repeatedly prompts for user commands. Before any command, the program prints a prompt to the user to enter a command, and then awaits a user command. The prompt is repeated after every user input (valid, invalid, blank line, etc.), after the output of the previous command is fully printed.

The list of commands is provided below. When several errors exist for the same command, follow this order:

1. Command name is correct (otherwise it is an invalid command)
2. Command has the correct number of parameters
3. Parameters are correct. Report any error of parameter 1 before parameter 2, etc. For each parameter:
    a. It is in the correct range for the command
    b. Its value is legal for the current board state
4. The board is valid for the command (e.g., the board is not erroneous for "autofill").
5. The command is executed successfully (e.g., "save" command fails writing to the file)

Note that commands are case-sensitive, i.e., "sAve" is an invalid command.

Note that throughout the project, your program should be able to handle any user input properly and execute the corresponding command or output the correct error message. However, you may treat any input line beyond 256 characters as an invalid command, with a proper error message instructing the user that too many characters were entered in a single line. This does not mean that the user may not enter a line with more than 256 characters, only that the entire line may be ignored and a proper error message should be printed.

In the list of commands below, parameters are marked with letters X, Y, Z, etc. A parameter in brackets [X] is an optional parameter. The brackets are **not** part of the command or parameter, i.e., a valid input from the user does not include the brackets.

A command that is unavailable in the current mode (e.g., "print_board" is not available in *Init* mode) is considered an invalid command, and an error message should instruct the user that the command is unavailable in the current mode, and which modes it is available in. This case is considered as part of step 1 for the order of errors described above (i.e., we report it right after reading the command name, regardless of the number and values of the rest of the command parameters).

Whenever the board is completed in *Solve* mode, follow the same guidelines as step **f** of the "set" command.

1. solve X
    a. Starts a puzzle in ***Solve*** mode, loaded from a file with the name "X", where X includes a full or relative path to the file.
    b. Note that this command is always available, in ***Solve***, ***Edit***, and ***Init*** modes. Any unsaved work on the current game board is lost.

2. edit [X]
    a. Starts a puzzle in ***Edit*** mode, loaded from a file with the name "X", where X includes a full or relative path to the file.
    b. The parameter X is optional. If no parameter is supplied, the program should enter ***Edit*** mode with an empty 9x9 board. There is no command to create a board of different size (the user must edit a file for that, explained later).
    c. Recall that in ***Edit*** mode we always mark errors, regardless of the value of the "mark errors" parameter (which remains whatever value it contained previously).
    d. Note that this command is always available, in ***Solve***, ***Edit***, and ***Init*** modes. Any unsaved work on the current game board is lost.

3. mark_errors X
    a. Sets the "mark errors" setting to X, where X is either 0 or 1.
    b. This command is only available in ***Solve*** mode.

4. print_board
    a. Prints the board to the user.
    b. This command is only available in ***Edit*** and ***Solve*** modes.

5. set X Y Z
    a. Sets the value of cell <X,Y> to Z.
    b. This command is only available in ***Edit*** and ***Solve*** modes.
    c. Indices and values are 1-based. The user may empty a cell by setting Z=0.
    d. In solve mode, fixed cells may not be updated.
    e. Recall that an erroneous input (such as entering 5 when column X already contains 5) **is allowed** (but might be marked when printing the board if in *Solve* mode and the "mark errors" parameter is set to 1).
    f. In ***Solve*** mode, after the game board is printed, if this is the last cell to be filled then the board is immediately checked. If the check fails, we remain in *Solve* mode (the user can undo the move to continue solving) and we notify the user that the solution is erroneous. Otherwise, we notify the user that the puzzle was solved successfully, and the game mode is set to ***Init***.

6. validate
    a. Validates the current board using ILP, ensuring it is solvable.
    b. This command is only available in ***Edit*** and ***Solve*** modes.
    c. If the board is erroneous, the program prints an error message and the command is not executed.
    d. The command prints whether the board is found to be solvable, or not.

7. guess X
    a. Guesses a solution to the current board using LP (*not ILP!*), with threshold X.
    b. This command is only available in ***Solve*** mode.
    c. The parameter X is a float and represents the threshold we use for the LP solution.
    d. If the board is erroneous, the program prints an error message and the command is not executed.
    e. The command fills all cell values with a score of X or greater. If several values hold for the same cell, randomly choose one according to the score (i.e., a score of 0.6 has double the chance of 0.3).
    f. Make sure **not** to fill illegal values created along the way – ignore invalid cell values regardless of their score.

8. generate X Y
    a. Generates a puzzle by randomly filling X empty cells with legal values, running ILP to solve the board, and then clearing all but Y random cells.
    b. This command is only available in ***Edit*** mode.
    c. If the board does not contain X empty cells, then that is an error and the user should be notified.
    d. To generate a board: randomly choose X cells and fill each selected cell with a random legal value. Then, run ILP to solve the resulting board. After the board is solved, randomly choose Y cells out of the entire board and clear the values of all other cells (i.e., except the Y chosen cells).
    e. Note that the X chosen cells are unrelated to the Y chosen cells – we choose X empty cells from the current board state and, after solving the entire board with ILP, choose *any* Y cells from the board to keep.
    f. If one of the X randomly-chosen cells has no legal value available, or the resulting board has no solution (i.e., the ILP solver fails), reset the board back to its original state (before executing the command) and repeat the previous step. After 1000 such iterations, treat this as an error in the puzzle generator.

9. undo
    a. Undo a previous move done by the user.
    b. This command is only available in ***Edit*** and ***Solve*** modes.

c. Set the current move pointer to the previous move and update the board accordingly. This does not add or remove any item to/from the list, only updates our pointer into the list.
d. If there are no moves to undo, it is an error.
e. If there was a move to undo, clearly print to the user the change that was made. The value of empty cells (with a value of 0) can be printed as "0".

10. redo
    a. Redo a move previously undone by the user.
    b. This command is only available in ***Edit*** and ***Solve*** modes.
    c. Set the current move pointer to the next move and update the board accordingly. This does not add or remove any item to/from the list, only updates our pointer into the list.
    d. If there are no moves to redo, it is an error.
    e. If there was a move to redo, clearly print to the user the change that was made. The value of empty cells (with a value of 0) can be printed as "0".

11. save X
    a. Saves the current game board to the specified file, where X includes a full or relative path to the file.
    b. This command is only available in ***Edit*** and ***Solve*** modes.
    c. In ***Edit*** mode, erroneous boards may not be saved.
    d. In ***Edit*** mode, boards without a solution may not be saved. Validate the board before saving and notify accordingly.
    e. File formats for save files are described below.
    f. Saving the puzzle to a file *does not* modify the Redo/Undo list in any way, and a *reset* command (described below) will still revert to the state of the originally loaded file.
    g. In ***Edit*** mode, cells containing values are marked as "fixed" in the saved file.

12. hint X Y
    a. Give a hint to the user by showing the solution of a single cell X,Y.
    b. This command is only available in ***Solve*** mode.
    c. If the board is erroneous, cell <X,Y> is fixed, or cell <X,Y> already contains a value, it is an error (in that order).
    d. Run ILP to solve the board. If the board is unsolvable, it is an error. Otherwise, print to the user the value of cell <X,Y> found by the ILP solution.

13. guess_hint X Y
    a. Show a guess to the user for a single cell X,Y.
    b. This command is only available in ***Solve*** mode.
    c. If the board is erroneous, cell <X,Y> is fixed, or cell <X,Y> already contains a value, it is an error (in that order).
    d. Run LP (*not ILP!*) for the current board. If the board is unsolvable, it is an error. Otherwise, print to the user all legal values of cell <X,Y> and their

scores. You should only print values with a score greater than 0.

14. num_solutions
    a. Print the number of solutions for the current board.
    b. This command is only available in ***Edit*** and ***Solve*** modes.
    c. If the board is erroneous it is an error.
    d. Run an exhaustive backtracking for the current board. The exhaustive backtracking algorithm exhausts all options for the current board, instead of terminating when a solution is found (as in HW3). Once done, the program prints the number of solutions found by the algorithm.
    e. The exhaustive backtracking algorithm is described below.

15. autofill
    a. Automatically fill "obvious" values – cells which contain a single legal value.
    b. This command is only available in ***Solve*** mode.
    c. If the board is erroneous it is an error.
    d. Go over the board and check the legal values of each empty cell. If a cell <X,Y> has a single legal value, fill it with the value and notify the user of the update. Repeat for all cells in the board.
    e. **Note** that only cells that had a single legal value *before* the command was executed should be filled, e.g., if cell <5,5> has two legal values, but by updating cell <4,5> we eliminate one of the legal values of cell <5,5>, then cell <5,5> should **not** be auto-filled! Some cells may be filled by values that are erroneous (two neighbors with the same legal value) – that is OK!
    f. After performing this command, additional cells may contain obvious values. However, this command performs only a single iteration and these cells should not be filled.

16. reset
    a. Undo all moves, reverting the board to its original loaded state.
    b. This command is only available in ***Edit*** and ***Solve*** modes.
    c. The command goes over the entire undo/redo list and reverts all moves (no output is provided).
    d. The undo/redo list is not cleared, we just move our pointer to the head of the list, as if the user entered multiple "undo" commands.
    e. This command has no output (except printing the board once done). We do not output every move that was undone.

17. exit
    a. Terminates the program.
    b. All memory resources must be freed, and all open files must be closed.
    c. Print an exit message.
    d. Note that this command is always available, even in ***Solve*** and ***Edit*** modes. Any unsaved work is lost.

e. EOF should be treated the same as if the user input the "exit" command.

SAVED GAMES

Each puzzle can be saved at any point, and then later loaded to solve or edit it. To load a puzzle, the user enters the "solve" or "edit" command. To save a puzzle, the user enters the "save" command.

Recall that when saving a puzzle in ***Edit*** mode, all cells with values are marked as "fixed".

FILES FORMAT

The format of the file is as follows (a simple text file):

- The first line contains the block size *m n*. These are two integers, separated by a single space. Recall that the board alphabet size is *N=m x n*.
- Each line contains a single row of the board.
- Each cell contains the cell's value separated with single spaces. Recall that the value in empty cells is 0. If the cell is "fixed", follow its value with a dot '.'.
- An example of a row with 4 values is: *"4 0 1. 0"*.

The save command should follow the above format precisely. An executable that allows creating save files will be posted on moodle.

When saving a file, the board size and each board row are separated by a newline, and cell values are separated by a single space. However, the program should support loading files with values separated by any number and type of whitespace characters. For example, files may contain a single line of values separated by three tabs, be broken into lines incorrectly, etc. These are all considered valid files which should be successfully loaded!

ERROR HANDLING

Your code should handle all possible errors that may occur (memory allocation problems, erroneous user input, ILP errors, etc.).

File names can be either relative or absolute and can be invalid (the file/path might not exist or could not be opened).

Do not forget to check the following elements:

- The return value of a function. You may excuse yourself from checking the return value of any of the following I/O functions: $printf$, $fprints$, $sprint$, and $perror$.
- Any additional checks, to avoid any kind of segmentation faults, memory leaks, and misleading messages. Catch and handle properly any fault, even if it happened inside a library you use.

You do not need to exit cleanly on errors. Always issue an appropriate descriptive message before terminating.

SUBMISSION GUIDELINES

The project will be submitted via Moodle. The grading may also involve testing knowledge of the code in a frontal meeting, to be set as necessary. Both students should be familiar with a significant part of the code.

You should submit a zip file named ***id1_id2_finalproject.zip***, where *id1* and *id2* are the IDs of both partners. The zipped file will contain:

- All project-related files (sources, headers, images).
- Your makefile.

CODING

Your code should be partitioned into files (modules) and functions. The design of the program, including interfaces, functions' declarations, and partition into modules – is entirely up to you, and is graded. You should aim for modularity, reuse of code, clarity, and logical partition.

In your implementation, pay careful attention to the use of constant values and proper use of memory. Do not forget to free any memory you allocated. You should especially aim to allocate only *necessary* memory and free objects (memory and files) as soon as possible.

Header and source files should contain a main comment that describes their purpose, implementation, and interface. Source files should be commented at critical places and at function declarations. Please avoid long lines of code.

Recall to properly document each function. Properly separate public functions (declared in the header and available to other modules) from private (static) functions (declared in the source file only and unavailable to other modules).

You should follow all coding guidelines from previous assignments, tutorials and lectures.

COMPILATION

You should create your own makefile, which compiles all relevant parts of your code and creates an executable file named '`sudoku-console`'. Use the flags provided in tutorial 3 and moodle to link Gurobi properly. Your project should pass the compilation test with no errors or warnings, which will be performed by running the **make all** command in a UNIX terminal on **nova**. You should also write a **make clean** command, and construct the makefile as learned in class.

GOOD LUCK