Introduction to Artificial Intelligence – EECS 348
Programming Assignment 3 – Sudoku (Constraint Satisfaction)

This is a GROUP assignment. You may form a group of up to 3 students total (no more than 3). You may also work alone or in a pair. You will submit only ONE assignment for the group.

In this assignment, you will write a constraint satisfaction problem solver to solve sudoku puzzles of varying size (see http://en.wikipedia.org/wiki/Sudoku for the rules if you're unfamiliar with sudoku puzzles). The goal is to obtain a performance comparison of various CSP strategies. In this project, you must implement the following strategies:

1) Backtracking
2) Forward Checking
3) MRV (minimum remaining values) Heuristic – choose the variable with the fewest values left
4) MCV (most constrained variable - degree) Heuristic – choose the variable that is involved in the largest number of constraints with unassigned variables
5) LCV (least constraining value) Heuristic – choose the value that rules out the fewest choices for other unassigned variables

The input to the solver is a Sudoku board, and the output should be a complete solution. In addition to returning a complete solution, you will be monitoring the number of consistency checks utilized (for this assignment, this is equal to the number of variable assignments your code makes).

Provided code:

We've supplied basic starter code for representing a board, and reading the puzzles from the input files you'll use for the assignment.

The starter code includes the following functions:

- To create a SudokuBoard object initialized with values from a file, use sb = init_board("filename.extension")
- to create a SudokuBoard object initialized with specific values, use the command sb = SudokuBoard(size, board_array)
- to create a new SudokuBoard which is the same as an existing board, but with one more number played, use the following sbA = sb.set_value(row,col,value) where row and col are 0 indexed
- a function to print the contents of a SudokuBoard.

Your code:

Below is a transcript of how I'll call your code. Notice the empty 'solve' function inside

the starter code. Regardless of how you implement your solver, I must be able to call it in the manner described below. In place of "SudokuStarter.py" will be your submission (netid.py).

```
>>> execfile("SudokuStarter.py")
>>> sb = init_board("input_puzzles/easy/4_4.sudoku")
>>> sb.print_board()
+--------+--------+
|      2 |        |
|        |        |
|        |      2 |
+--------+--------+
|      3 |  4     |
|        |        |
|  4     |        |
+--------+--------+
>>> fb = solve(sb, True, True, False, False)
Your code will solve the initial_board here!
Remember to return the final board (the SudokuBoard object).
I'm simply returning initial_board for demonstration purposes.
>>> fb.print_board()
+--------+--------+
|      2 |        |
|        |        |
|        |      2 |
+--------+--------+
|      3 |  4     |
|        |        |
|  4     |        |
+--------+--------+
```

The above call to solve would use forward checking and the mrv heuristic but the other two heuristics would be turned 'off'. If heuristics are not used, variables and values are just tested in the order that they appear.

Of course, your code would have returned a complete solution to a Sudoku puzzle, not the initial board that I returned above.

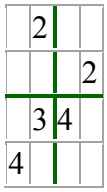
Input Puzzles:

We've supplied specific input puzzles you'll use for this assignment (thanks to Stef Schoenmackers of decide.com for these). Functions for reading the boards from files are included in the sample code, but for completeness the files have the following format:
<board_size> - (board size will be 4, 9, 16, or 25)
<num_initial_values> - (The number of set cells in the initial state)
<row1> <col1> <value1> - value of the cell at row1, col1
<row2> <col2> <value2> - value of the cell at row2, col2
<row3> <col3> <value3> - value of the cell at row3, col3

... for a total of num_initial_values lines

When testing your code, it will help to try some easy test boards, which you can find in the ‘easy’ folder inside of the ‘input puzzles’ folder.

Each row, col, and value is in the range [1, board_size]. The following image corresponds to the board ‘4_4.sudoku’ in the ‘easy’ folder:



For more complicated boards, see the ‘more’ folder inside of the ‘input_puzzles’ folder. Note that for the larger problems, without powerful heuristics your program will not terminate in a reasonable amount of time -- you should choose an upper bound to the number of consistency checks you make so that the program terminates in, say, 10 minutes.

Write up:

In addition to your code, you must submit a very brief write-up. Specifically, it need ONLY include a table comparing the performance of the heuristics (in terms of number of consistency checks, i.e. variable assignments) for these test problems: 4x4 9x9 16x16 25x25

The table should have the following format (numbers are random placeholders) where the last four columns indicated the performance of backtracking with each of those approaches only. Feel free to add more columns for other combinations of approaches. Save this table in a file titled netid.pdf (or netid.txt).

Problem	Backtracking	Forward Checking	MCV	MRV	LCV
4x4	1234	567		...	...
9x9	23456	7890		...	...
16x16	789K	12K		...	...
25x25	(> 1M)	234K		...	...

What to submit:

- netid.py: (i.e. the renamed "SudokuStarter.py" file) The file containing all the code you have written.
- netid.pdf (or netid.txt) containing your performance comparison table.

Be sure to include all of your team member's names and netid's in comments at the beginning of the files. Additionally, your code should be readable, commented and clear. You will lose points for poorly commented or poorly organized code.