Artificial Intelligence Homework 3
Due date: 3/10 by 11:59pm

Our final application of search is solving constraint satisfaction problems. In this assignment, you will implement the AC-3 algorithm and BackTracking search to solve Sudoku puzzles.

This is a programming assignment and you may work with a partner if you choose. Click here to download the starter code. You will notice that the code is arranged in packages:

- **graphics:** This package contains the Java classes responsible for creating the GUI, calling your solver, and graphically displaying the solved Sudoku puzzle. The constructor in SudokuPanel.java is where your code is called. *You do not need to modify anything in this package*.

- **csp:** This package contains the classes that encode the constraint satisfaction problem

- **search:** This package contains the AC-3 and BackTracking algorithms

Sudoku

If you have never played a game of Sudoku, I recommend going online (https://www.websudoku.com/) and playing a few games first. I also recommend reading section 6.2.6 in the textbook which talks specifically about solving Sudoku puzzles using CSPs.

A Sudoku board consists of a 9x9 grid subdivided into 3x3 boxes. The goal is to fill each cell in the grid with the numbers 1 through 9 such that:

- Each row contains the numbers 1 through 9
- Each column contains the numbers 1 through 9
- Each 3x3 box contains the numbers 1 through 9

Since the board has 9 rows and 9 columns, there can be no duplicate values in a row, column, or 3x3 box.

The starting Sudoku board is blank with the exception of a few cells that have been filled in. Below is an example of an initial Sudoku board and the corresponding solution.

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

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

Representing a Constraint Satisfaction Problem

A CSP consists of 3 components: the variables, the domain for each variable, and the set of constraints. In Sudoku, each cell in the 9x9 grid represents a variable with domain {1, 2,..., 8, 9}. The values for some variables have been specified by the original problem and are fixed.

As for the constraints, each pair of cells in the same row, the same column, and the same 3x3 box are involved in a difference constraint -- i.e., they must take on different values. Thus, there are 9*72 row constraints, 9*72 column constraints, and 9*72 box constraints for a total of 1,944 constraints.

The **csp** package contains the code for encoding Sudoku as a CSP. In particular,

- The `Variable` class should represent a single variable and should encode the domain of that variable (either {1, 2, ..., 9} or a single value specified by the problem). Think carefully about what data structure you will use to represent the domain -- what are the operations you need to perform on the domain? Choose a data structure that performs those operations quickly. I also recommend adding to this class lots of methods for interacting with the domain: adding a single value to the domain, adding an entire list of values to the domain, removing a value from the domain, clearing the entire domain, getting the only value from the domain, etc.

- The `BinaryDiffConstraint` class represents a binary difference constraint -- i.e., it is a constraint specifying that two variables must take on a different value. You will notice that this class has already been written for you and is nothing more than a plain old data class. All of the fields are public and its only purpose is to package together the row and column indices for both variables.

- The `Constraints` class should store a list of all 1,944 constraints. Your class should contain the logic for generating all constraints and needs nothing more than a list in which to store them.

AC-3 and BackTracking Search

The **search** package is where you will write your AC-3 and BackTracking algorithms. The `solve()` method in the `Solver` class takes in a starting Sudoku puzzle and should return the solved puzzle.

As your textbook states, the easiest Sudoku puzzles can be solved using AC-3 alone. Harder puzzles, however, require actual search. As such, your `solve()` method should:

- First try AC-3 to solve the Sudoku puzzle
- If AC-3 is not sufficient, run BackTracking search with AC-3 used during the inference step.

The AC-3 Algorithm

Pseudocode for the AC-3 algorithm can be found in your textbook and on the course webpage. There are some simplifications that can be made when applying AC-3 to Sudoku. In particular, when considering the constraint (X_i, X_j)

- If the domain of X_i is fixed (i.e. specified by the original problem), then you can move on to the next iteration of the algorithm. The domain of this variable cannot be modified.

- If there is more than 1 value in the domain of X_j, then all values in the domain of X_i are consistent. You should convince yourself of this.

- Remember that you need to put each constraint on the queue twice: once in the form (X_i, X_j) and once in the form (X_j, X_i)

BackTracking with AC-3

Pseudocode for BackTracking can be found in your textbook and on the course webpage. Since BackTracking is nothing more than DFS (with some additions), I highly recommend creating a `Node` class just as you did for the first homework assignment. Your `Node` class can contain all information needed for a node in a search tree -- e.g., `isGoal()`, `getChildren()`, etc.

Your BackTracking search should use the minimum-remaining-values heuristic to choose which variable to assign a value to next. You do **not** need to use the least-constraining-value heuristic...just go ahead and check each value in the domain from 1 to 9. Finally, you should use AC-3 during the inference step to speed up the search process.

Running Your Code

The main() method is in Sudoku.java. You can pass a puzzle to this class via the command line. This class will read in the puzzle from file, call the SudokuPanel class, which then calls your solve() method to solve the puzzle. The solved puzzle is then graphically displayed on the screen. Note that the bolded values are those specified by the puzzle. (If your solve() method returns null, then the SudokuPanel class will simply display the original, unsolved puzzle).

There are various Sudoku puzzles in the "puzzles" directory ranging from easy to evil. Your code should be able to solve all puzzles (even the evil puzzle) in a matter of seconds.

If you would like to create your own Sudoku puzzle files, feel free. Each line in the file specifies a row index, column index, and value.

Grading and Submission
Your assignment will be graded based upon correctness (i.e. I will run your code on various puzzle files), efficiency (i.e. the time/space complexity of your implementation), as well as its adherence to the Java style guide.

To submit, rename your directory hw3_FirstName_LastName. Then zip your directory and upload it to Moodle. Please remember to rename your directory *before* you zip it.

Last modified: Tue Jul 15 13:34:17 PDT 2014