Assignment 2

B351 / Q351

Comprehension Questions Due: January 23rd, 2018 @ 11:59PM

Programming Due: January 30th, 2018 @ 11:59PM

1 Summary

- Solve various difficulties of Sudoku puzzles
- Demonstrate your understanding of solving constraint satisfaction problems through forward checking and brute force searching

You may collaborate **with one partner** on this assignment (optional). Your files will be run against Moss to check for plagiarism within the class outside of your partnership.

2 Background

The objective of Sudoku is to fill every square in the game board (see below) such that the final game state meets the following constraints:

1. For an $n^2 \times n^2$ board, every cell must contain a number between 1 and n^2 (inclusive). For example, in the 9×9 board below, every cell must contain a value between 1 and 9.
2. Every row must contain only unique values. In other words, there can only be one of each value in a row.
3. Every column must contain only unique values.
4. Every inner $n \times n$ board delineated by bold bordering must contain only unique values.
5. You must work around the starting values in the board (see below).

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

Example Sudoku Board

For more information, you may find Wikipedia to be helpful as a primer: `https://en.wikipedia.org/wiki/Sudoku`.

3 Programming

In this assignment's programming component, you will utilize the starter framework that we have provided to built a Sudoku solver that implements the forward checking algorithm discussed in lecture and lab.

3.1 Data Structures

3.1.1 Input Sudoku Boards

Your program will take input Sudoku boards in the form of `csv` files. You can find examples of such boards in the `testBoard_*.csv` files. Note that empty spaces in the `csv` file will correspond to empty space in the Sudoku board.

You can change what file is read by changing the input string at the bottom of your `a2.py` file.

3.1.2 Board Class

This class is implemented in the `board.py` file. The class encapsulates the following data members:

1. `n2` - The length of one side of the board (must be a perfect square).
2. `n` - The length of one side of an inner square.
3. `spaces` - The total number of cells in the Sudoku board. A typical 9×9 board will contain 81.
4. `board` - A dictionary containing the mapping $(r, c) \rightarrow k$ where r and c are the **0-indexed** row and column respectively, and k is a value from 1 to n^2. If (r, c) does not exist as a key in `board`, then the cell at row r and column c does not currently contain any value.

   In the example board in Section 2, `board[(2,3)]` would return 1 and `(0,0) in board` would return `False`.
5. `unSolved` - A Python set that contains (r, c) tuples that indicate cells that currently have no value assigned. (r, c) represents the cell at row r and column c (**0-indexed**).

   In the example board in Section 2, `(2,3) in unSolved` would return `False` and `(0,0) in unSolved` would return `True`.

6. `valsInRows` - A Python list that represents a mapping of $r \rightarrow vals$ where r is the row index and $vals$ is a set of the values currently in the corresponding row on the board.

   In the example board in Section 2, `valsInRows[3]` would return {`5,7`}.

7. `valsInCols` - A Python list that represents a mapping of $c \rightarrow vals$ where c is the column index and $vals$ is a set of the values currently in the corresponding column on the board.

   In the example board in Section 2, `valsInCols[3]` would return {`1,4,2`}.

8. `valsInBoxes` - A Python list that represents a mapping of $b \rightarrow vals$ where b is the inner box index and $vals$ is a set of the values currently in the inner box on the board. Inner boxes are indexed from 0 start at the top left and progressing from left to right and then top to bottom. You will find the `rcToBox` function helpful in converting from a row and column index to an inner box index.

   In the example board in Section 2, `valsInBoxes[rcToBox(1,4)]` would return {`1,2,3`}.

In addition to these data member definitions we have given you a few starter methods:

1. `__init__` and `loadSudoku` - These contain all of the logic to load your Sudoku board `csv` files into your program and initialize all of your variables with the proper values. To create a new board, simply pass the filepath of the file that contains your starting Sudoku board into the constructor.

2. `print` - This will print a graphical representation of your Sudoku board to your command line.

3. `rcToBox` - This will conduct the conversion of a row and column index into the appropriate box index.

3.1.3 Solver Class

This class is implemented in the `a2.py` file. You will need to provide the implementation for the `solve` function (see below), but otherwise this simply serves as an entry point into your Sudoku solver application.

3.2 Objective

Your goal is to provide the implementations to the following functions:

1. 15% - `Board.makeMove`

2. 15% - `Board.removeMove`
3. 15% - `Board.isValidMove`
4. 15% - `Board.getMostConstrainedUnsolvedSpace`
5. 40% - `Solver.solve`

to the following specifications:

3.2.1 Board.makeMove

This function should take a space tuple (r, c) where r and c are a row and column index respectively and a valid value assignment for the space. It should:

1. Save the value in `board` at the appropriate location
2. Record that the value is in now in the appropriate row, column, and box
3. Remove the space from `unSolved`

3.2.2 Board.removeMove

This function should take a space tuple (r, c) where r and c are a row and column index respectively and a valid value assignment for the space. It should:

1. Remove the value from `board` at the appropriate location
2. Record that the value is no longer in the appropriate row, column, and box
3. Add the space to `unSolved`

3.2.3 Board.isValidMove

This function should take a space tuple (r, c) where r and c are a row and column index respectively and a valid value assignment for the space. It should return `True` iff placing the value in the indicated space does not conflict with any of the constraints of Sudoku.

3.2.4 Board.getMostConstrainedSpace

This function should return a space tuple (r, c) where r and c are a row and column index respectively. This space tuple should represent the unsolved space on the board with the smallest domain of valid value assignments. If there exists a tie, you may choose an arbitrary method to break it.

3.2.5 Solver.solve

This should implement a brute force search with forward checking (constraint propagation) to assign values to empty spaces in the Sudoku board. At the termination of the initial call to `Solver.solve`, `Solver.board` should be left in a solved state if there exists a solution and in its original state if otherwise.

As a point of reference, it should take no longer than 1-2 seconds to solve any of the given Sudoku boards on even the slowest of machines. If your program hangs, you may want to check if you are using `getMostContstrainedUnsolvedSpace` effectively.

4 Grading

100% of your grade on this assignment will depend on the correctness of the solutions to the programming problems. Partial credit will be given for attempts proportionate to the progress you make and it's overall correctness. There are several ways that you can maximize the credit that you receive:

1. Create verbose comments that explain your thought process and how your code is intended to work. The better we understand that you were on the right track, the more points we can assign to incomplete or incorrect solutions;
2. Abstracting the logical components of your code is not only the ideal way to isolate common errors, but it allows to better assign partial credit. If all of your code is in one large function and that function contains an error, you will likely receive far less credit than if your code was broken into multiple components and only one component contained an error;

5 Bonus

5.1 Counting Guesses (10 %)

In selecting values for cells in the Sudoku board, your program will run into three scenarios. Either it will

1. have no valid value assignments and thus be forced to backtrack;
2. have a single valid value assignment and thus be forced to place that value; or
3. have a variety of valid values to choose from for the particular cell.

The program is logically constrained in options (1) and (2), but in option (3) the board will be forced to make a "best guess". For this bonus section, we are interested in how guessing affects the time complexity of your solver algorithm.

Objectives:

1. Augment your completed solver code to record the following:
    (a) the number of guesses that your program makes when solving the board
    (b) the total number of moves and removes that your program makes when solving the board
2. Answer the following questions in a separate `txt` file that you include in your repository:
    (a) Does the difficulty of the Sudoku puzzle affect the number of guesses?
    (b) How are the number of guesses and the number of total moves/removes related?

As a baseline reference, the "singletonsOnly" board should require no guessing to solve.

5.2 Alternative Solvers (20 %)

Using forward checking is only one method to solve Sudoku problems. It also happens to be slowest and there are a variety of alternative methods for solving such problems that are available to choose from. Two possible methods are using an "exact cover" algorithm or using an evolutionary algorithm.

Objectives:

1. Implement an alternative algorithm for solving Sudoku puzzles. Store your algorithm in a separate file called `a2_bonus.py` (do NOT delete your original solution).

2. Answer the following questions in a separate `txt` file that you include in your repository:
    (a) What algorithm did you choose? How does it work?
    (b) How does the efficiency of this algorithm compare to forward checking? Provide some empirical benchmarks.