Homework 2: Problem Solving Problems

Due February 15th at 10pm

To submit your homework, follow these steps:

1. Create a Homework 2 folder within the FIRSTNAME LASTNAME CS232 Submissions folder that you shared with me last week.

2. Upload your homework writeup and Python code to this directory.

Part 1: Sudoku

Your goal is to write a Sudoku solver, which is a problem that requires *backtracking search constraint propagation*. If you haven't played Sudoku before, you should play a few games by hand before starting. There are lots of Sudoku mobile apps and websites, including sudoku.com and websudoku.com.

Representing Sudoku

A Sudoku board is a 9-by-9 grid, where each cell is either blank or has a value in the range 1–9. For this assignment, we'll use strings of length 81 to represent Sudoku boards, where each block of nine characters represents a successive row. For example, the following string:

puzzle = "....8.3...6..7..84.3.5..2.9...1.54.8.....4.27.6...3.1..7.4.72..4..6...4.1...3"

Represents the following board:

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

Here are some more examples:

".43.8.25.6.....1.949....4.7....6.8....1.2....382.5.....5.34.9.71."
"2...8.3...6..7..84.3.5..2.9...1.54.8.....4.27.6...3.1..7.4.72..4..6...4.1...3"
"..3.2.6..9..3.5..1..18.64....81.29..7.....8..67.82....26.95..8..2.3..9..5.1.3.."
"1..92....524.1.....7..5...81.2.....4.27...9..6.....3.945....71..6"

Your first task (described in more details below) is to parse strings that represent Sudoku boards. You may assume that all strings represent solvable boards and that the string has exactly 81 characters.

Solving Sudoku puzzles is much harder, but we'll walk you through it.

Sudoku as a search problem

To solve a Sudoku puzzle, we can use **backtracking search**. The intuition is that we guess a value for a square, and see if any conflicts arise. If they do, we undo that guess and eliminate the number as a possible value for that square.

Backtracking search is a recursive algorithm that operates as follows. Given a Sudoku board B:

- If B is already filled completely, return the solution.
- If B is an invalid board (e.g., two 2s in a row), abort.
- Otherwise, select *one* cell that is not filled in, generate a list of boards that fill in that cell with a number. For each generated board, recursively apply the search function:
  - If any board produces a valid solution, return that board
  - If no board produces a solution, abort and return

This approach will work in principle. But, in practice there are too many boards to search: there are 81 squares, and each square can hold 10 values (a digit or blank). Therefore, there are 10^{81} possible boards, which is an enormous number.

Solving Sudoku with constraint propagation

To actually solve Sudoku problems, we need to combine backtracking search with *constraint propagation*. When you play Sudoku yourself, every time you fill a digit into a cell, you can eliminate that digit from several other cells.

For example, if you fill 2 into the top-left corner of the board above, you can eliminate 2 from the first row, first column, and first box. i.e., there is no point even trying to place 2 in those spots, since the 2 in the corner *constrains* those cells.

Another trick is to pick a **good search strategy**. If we start with the cell that has the most constraints (most numbers eliminated), we will have fewer possibilities to consider.

Augmenting backtracking search with constraint propagation implements this intuition. The key idea is to store the *list of values* that may be placed in a cell, instead of only storing a single value in a cell. For example, on the empty board the values 1—9 may be placed at any cell:

123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789

With this representation, when we place a value at a cell, we eliminate it from the other cells in the same row, column, and box (collectively known as the *peers* of a cell). For example, if we place 5 at the top-left corner of the empty board, we eliminate 5 from its peers as follows:

5	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789
1234 6789	1234 6789	1234 6789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	1234 6789	1234 6789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789

This procedure will significantly reduce the number of boards that need to be visited.

Programming Task

The starter file defines a class called Board that represents a Sudoku board. This class has a constructor that you **should not change** and a method that displays the board as a grid.

It also has several methods for you to fill out. I suggest implementing them in the following order:

1. Implement the peers function. peers(r, c) produces the coordinates of all cells in the same row as r, same column as c, and same block as (r,c). Note that (r,c) is *not* a peer of itself.
2. Implement parse.You should assume that the input string has exactly 81 characters, where each character is a digit or a period. Each successively block of nine characters represents a row (i.e., row-major order).
   As the starter code suggests, you need to store the set of available values at each cell (**NOT** the value of the cell). The empty board is a dictionary with all values available in all cells.
3. Implement Board.valueAt. You should produce the digit stored at the given row and column or None *if it is blank or has multiple available values*.
4. Implement Board.place. The call board.place(row, col, value) produces a new board with value placed at (row, col) of board.
   To produce the new board, you'll have to:

   (a) Remove value from set of available values of each peer of (row, col).
   (b) While doing (1), if a peer is constrained to exactly 1 value, then recursively apply place to that peer.
   (c) If there is already only 1 value at (row, col), there is no new information to propagate, so you can just return.
5. Implement next_boards, which returns a list of possible next boards. To do so, (1) pick a cell that has a minimal number of available values, and (2) place each available value in that cell. **IMPORTANT: Use Board.copy so that each next board is independent!**
6. Implement Board.is_solved and Board.is_unsolvable. A board is solved if every cell is constrained to exactly one value. Similarly, a board is unsolvable if any cell is constrained to the empty set of values (i.e., nothing can be placed in that cell).
7. Implement board.solve. If board.is_solved is true, then return board. If it is unsolvable, return None. Otherwise, iterate through the list of next boards, recursively applying solve, and return the first board that is solved.

The starter code also includes some tests with boards that your solver should be able to solve. These tests only check that a solution is found. **They do not check that the solution is correct.** You should write some tests for yourself that check this. I recommend playing around and creating your own puzzles to test your solver as well. Your solver may not be able to solve arbitrary Sudoku boards, but it will do pretty well.

Part 2: Reading Reflections

These questions ask you to reflect on Chapters 1 and 2 of *You Look Like A Thing And I Love You*.

Question 1

In Chapter 1, Shane discusses four signs of AI doom: indications that a problem is not amenable to AI. But right now, whether we like it or not, many, many people are trying to solve problems with AI.

Consider three plausible contemporary scenarios of automating grading in a university:

(1) The computer science department is currently overwhelmed with the number of students who would like to take CS courses. In order to increase the number of students who can take Intro CS, the instructor team decides to use autograding for all assignments. The autograder uses the same test suite that the human instructors used to use to assess whether a program executes correctly. However, it cannot give feedback on programming style.

(2) The philosophy department is concerned about implicit bias in grading essays. Perhaps instructors are swayed by their own preferences or beliefs when assessing student work. Moreover, philosophy, like CS and many STEM fields, is a discipline that is heavily white and male. To combat the possibility of implicit bias in grading, the philosophy department trains an algorithm to grade essays, using samples of student work from previous years.

(3) The psychology department would like to offer a more flexible Intro to Statistics course. Currently, the course requires students to program in R, annoying students who prefer Python or Mat-

lab. To relax this requirement, the department builds a system that translates code from these languages into R so that they can grade all submissions using their R test suite. To train this system, they scrape programs from Github, which may or may not include statistical analysis.

Compare and contrast the appropriateness of the use of technology in these scenarios. Discuss which of Shane's four signs of AI doom apply to each scenario.

Question 2

Given what you learned in Chapter 2 about the limitations of AI models, discuss how difficult you think the following problems would be:

- Generate image captions for an album of family vacation photos
- Given several episodes of an animated TV show, identify fan art of various characters online
- Given an unfinished novel by a prolific author, generate an ending for the work

Question 3

Pick a game that you enjoy playing that we have not discussed in class or in the reading. Research whether an AI player has been built for it. How well does the AI player do? What kind of techniques does it use? If the AI player has surpassed human performance, when did this happen?

You can pick either a board game or a video game.