Homework 4 - Serious Search Seeking the Solution to Some Studious Sudokus

Due Date: December 20

Assignment Overview

In this assignment you will implement a Sudoku solver. Sudoku is a puzzle game based around filling in a 9 by 9 grid to satisfy a series of constraints. For those who don't know the rules of sudoku, use this Wikipedia article to use as reference
.
At the C level you will simply implement a class that understands the basic rules of Sudoku. At the B level you will extend your class to solve Sudoku by trying every possible value in a brute force style At the A level you will extend your class to solve Sudoku using a more intelligent search.
Because Sudoku is a relatively hard AI problem (too hard for brute force enumeration, not hard at all with the right tricks) we will be asking you to solve two sizes off Sudoku. Size "3" which you are all familiar is a 9 by 9 grid that is composed of nine 3 by 3 subgrids (arranged in a 3 by 3 grid).

```
53 | 7 |
6  |195|
 98|   | 6
---+---+---
8  | 6 |  3
4  |8 3|  1
7  | 2 |  6
---+---+---
 6 |   |28
   |419|  5
   | 8 | 79
```

Additionally we will ask you to support size "2" Sudoku which have a 4 by 4 grid of numbers split up into four 2 by 2 sections

```
1 |42
 2|
--+--
3 | 1
  |
```

Write your own code!

For this assignment to be an effective learning experience, you must write your own code! I

emphasize this point because you will be able to find Python implementations of most or perhaps even all of the required functions on the web. Please do not look for or at any such code!
Here's why:

- The most obvious reason is that it will be a huge temptation to cheat: if you include code written by anyone else in your solution to the assignment, you will be cheating. As mentioned in the syllabus, this is a very serious offense, and may lead to you failing the class.
- However, even if you do not directly include any code you look at in your solution, it surely will influence your coding. Put another way, it will short-circuit the process of you figuring out how to solve the problem, and will thus decrease how much you learn.

So, just don't look on the web for any code relevant to these problem: don't do it.

Homework requirements.

You are required to turn in two files sudoku.py, containing the required functions, and README.txt, which must specify which level you are attempting/ claiming to have completed. As always, if you do the homework with a partner, both of you must include your names in the README.txt file.
To help you judge your progress we will be giving you Sudokus with known unique solutions. You can find them here:

- 2sudoku.txt This file contains size 2 sudoku and their solutions.
- 3sudoku.txt This file contains easy size 3 sudoku and their solutions.

Both files contain a list of Sudoku. Each line contains a Sudoku with blanks, a single comma and then the solved sudoku, both in the "compact" representation introduced below.
You will find it helpful to write your own tests in addition to trying the test Sudoku. While the tests Sudoku can tell you whether your code is right or wrong, your own tests can help tell you what part of your code is right or wrong.

Grade level descriptions

C-level

At the C-level you will create a class Sudoku. The Sudoku class is responsible for understanding the rules of Sudoku. This class will have several functions:

- **__init__**

  This should have two arguments:

  1. size - This should be 2 or 3 (or maybe 4 if you want a real challenge) and represents the "size" of the Sudoku board. A Size n Sudoku should be n*n wide on each side.
  2. data - This is a *string*. The format of this is a dense packing of the Sudoku, by row and column so that row i, column j is stored in position i*width+j So the Sudoku:

```
1 |42
 2|
--+--
3 | 1
  |
```

Would be encoded as

`"1 42 2  3 1      ",` where spaces represent blanks.

Note that while a string will be passed in, your data representation of the sudoku puzzle need not necessarily be a string, too (More on that below).

- **__str__**

This function doesn't need to be fancy, but it should display the data as a grid. The *minimum* acceptable output for

```
13|42
42|13
--+--
34|21
21|34
```

Would be

```
1342
4213
3421
2134
```

Although you are free to do fancier things like drawing the borders.

- **get(row, col)**

This function must return the value at the given row and column, or a space if nothing is currently at that place. You are *not* required to use this function internally, but you must have it for inspection of your Sudoku to be possible.

- **isFull()**

This function must return true only if a value is selected for every square in the Sudoku.

- **getNeighbors(row, col)**

This function should return a list of tuples of (row, col) for every row and column that cannot have the same value as the square at the given position. Remember, there are three ways a square can get in this list, same row, same column, or same subgrid (three by three box in a normal Sudoku)

- **isValid()**

This function should return false if there cannot exist a solution to the Sudoku because two neighbors (as defined above) share the same value.

You are allowed to return False in other situations where you can prove that the Sudoku is unsolvable (such as having a row where no square can take some value), so long as it returns True for all solvable Sudoku, and False if two conflicting cells have the same value, this function is correct.

There are several decisions you are in charge of in this implementation.

1. You should decide how to represent a Sudoku in python. While the string solution is serviceable (even preferred by a leading AI expert), it is clumsy, awkward, and not recommended. You should think through how you want to represent a Sudoku and transform the Sudoku string, into your preferred representation.
2. Do you want the Sudoku to be *mutable*? This is a very important question when you consider how to implement the B and A levels. If you allow a Sudoku to be changed in place you will have to be careful to avoid changing the value of squares that were given to you. This is important because you are not allowed to change these values when solving the Sudoku. You can do this either with careful algorithm design, or by storing extra information.
   Alternatively, if you choose to make your Sudoku immutable you should provide a function that will take a row, column, and value and return a Sudoku object in which the given row and value has been set to the given value, and all other squares are the same.
   We personally recommend creating an immutable Sudoku object over mutable Sudoku objects. The immutable design helps you avoid accidentally changing incorrect values and makes the "record keeping" involved in search problems much more straightforward. That said, the ultimate decision is yours.

B-level

At this level you will implement a brute force solver for Sudoku. This will be capable of solving a 2 x 2 Sudoku in under two minutes on reasonably modern machines.
To do this, make a function bruteForce() which recursively tries every combination of values in every (unoccupied) square of the Sudoku. It should either return a Sudoku object that has the solution in it (this can be a new object, or if you modify the one Sudoku object in place, it can be this object, either way, return it), or False, if there is no solution (I.E. you were given an unsolvable Sudoku to solve).
You should realize that because this function is recursive, returning false does not necessarily mean that the original Sudoku is unsolvable. A return value of false should serve as a signal to earlier recursive calls that their current guess at a value is wrong, leading them to try different values which can lead you to the solution.
This shouldn't be able to solve a full sized Sudoku in reasonable time. This is fine.
Your solution is required to be recursive.

A-level

At this level you will write a smarter algorithm capable of solving normal Sudokus in reasonable time. In particular it should only take a few seconds on a modern machine to solve simple puzzles, (it may take a long amount of time to solve "hard" puzzles).
To do this make a function solve() which recursively searches for solutions. This should work similarly to bruteForce with a few modifications:

- As soon as a board is obviously invalid (I.E. cannot have a solution because some value is used in too many places) it should return False.
- It should only try "valid" options for squares.

  (These two of course both give you the same end-goal, avoiding "dead-end" work on obviously unsolvable Sudoku)

Once you have these two fixes test your code. It should be quite fast on "easy" Sudoku.

Again, your function is required to be recursive.

Extra-credit

To get better performance on Sudoku you will need a smarter algorithm. One type of algorithm is a "constraint satisfaction" algorithm which tracks 2 things:

- What possible values remain for each square
- What values are still needed for each row, column, and subgrid.

If either of these "force" a value then this value is selected without searching. For example, if a square can have only one value, that value is immediately set and the stored data is updated. Likewise, If there is a row (or column, or subgrid) that has only one square that can take a value then that square must have that value, and it is set to that value without searching.
Write a function that implements these rules and only uses searching when these faster rules cannot progress further. The idea is that these rules are quicker to apply than the "random" guessing used by searching.
A related idea is to try to solve the square with the least possible values first. This ensures that the number of options we need to try at any depth is as small as possible.
Implement these two heuristics and write a paragraph comparing the solutions we present above with these in terms of code, complexity, and solver efficiency.