Assignment 2

B351 / Q351

Due: September 15th, 2020 @ 11:59PM

1 Summary

- Solve various difficulties of Sudoku puzzles
- Demonstrate your understanding of solving constraint satisfaction problems through forward checking and backtracking search.

We will be using Python 3 so be sure you aren't using older versions. Code that will compile in Python 2.7 may not compile in Python 3. See our installation guide for help removing old versions and installing Python 3.

Please submit your completed files to your private Github repository for this class. You may not make further revisions to your files beyond the deadline without incurring a late penalty.
No collaboration is allowed on this assignment.

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 `tests` directory such as `example.csv`. 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 `a2.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. `unsolvedSpaces` - A Python set that contains (r, c) tuples that indicate spaces on the board 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 unsolvedSpaces` would return `False` and `(0,0) in unsolvedSpaces` 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 `spaceToBox` function helpful in converting from a row and column index to an inner box index.

   In the example board in Section 2, `valsInBoxes[spaceToBox(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. `spaceToBox` - 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 `solveBoard` 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. `Board.makeMove`
2. `Board.undoMove`
3. `Board.isValidMove`
4. `Board.getMostConstrainedUnsolvedSpace`
5. `Solver.solveBoard`

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 `unsolvedSpaces`

3.2.2 Board.undoMove

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 `unsolvedSpaces`

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. Returns `False` when the space is not on the board or when the space is not empty.

3.2.4 Board.getMostConstrainedUnsolvedSpace

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. This should also return `None` if `unsolvedSpaces` has nothing in it.

3.2.5 OPTIONAL: Board.evaluateSpace

You are encouraged, but not required to write this helper function to evaluate spaces based on the number of constraints or possible values that you have.

3.2.6 Solver.solveBoard

This function takes in a `Board` object, and should implement a backtracking search with forward checking (constraint propagation) to assign values to empty spaces in the given Sudoku board. At the termination of the initial call to `Solver.solveBoard`, the given `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 minute to solve any of the given 9×9 Sudoku boards on even the slowest of machines. Larger boards may take longer. If your program hangs, you may want to check if you are using `getMostContstrainedUnsolvedSpace` effectively.

4 Grading

Problems are weighted according to the percentage assigned in the problem title. For each problem, points will be given based on the criteria met in the corresponding table on the next page and whether your solutions pass the test cases on the online grader.

Finally, remember that this is an individual assignment.

4.1 makeMove (10%)

Criteria	Points
Assigns space to value in the board	3
Records that value has already occurred in the row, column, and box of the space	5
Removes space from the set of unsolved spaces	2
Total	10

4.2 undoMove (10%)

Criteria	Points
Deletes space from the board	3
Records that value no longer appears in the row, column, and box of space	5
Adds space to the set of unsolved spaces	2
Total	10

4.3 isValidMove (10%)

Criteria	Points
Returns False when the space is not on the board	2
Returns False when the space is not empty	2
Checks that the value has not already occurred in the column, row, and box of space	5
Returns True whenever the assignment is valid	1
Total	10

4.4 getMostConstrainedUnsolvedSpace (20%)

Criteria	Points
Returns None when the board has been fully solved	2
Runs the evaluation function on unsolved spaces	2
Either minimizes or maximizes with respect to the evaluation function	5
Always returns the same value given the same board	1
Returns one of the most constrained spaces for every unsolved board	10
Total	20

4.5 solveBoard (30%)

Criteria	Points
Attempts to make moves in a single space, the most constrained unsolved space	5
Does not attempt any invalid values, and attempts each valid value exactly once (until a solution is found)	5
Recursively calls solveBoard exactly once for each move made, in order to test whether it leads to a solution	5
Clears every move that does not lead to a solution	5
Does not undo any correct moves or attempt any other moves after a solution is found	2
Never overwrites or clears any of the squares of the original board	2
Returns True in the case where the entire board is already solved	2
Returns True when a solution is found, indicating that the current state leads to a solution	2
Returns False when no solution exists, because the most constrained cell cannot take any value	2
Total	30

4.6 Competency (20%)

Criteria	Points
Leaves the board in its original state if no solution exists	5
After running on a solvable board, the board's final state is the same as the solution	15
Total	20

5 Bonus

5.1 Sudoku Competition

Participants compete in a series of contests that test their optimized implementations of Sudoku solvers. There are fifteen total contests made up of six test suites of 20-100 boards each, and nine named boards that test the different features of your solver. The test suites range in dimension from 9×9 to 36×36.

5.1.1 Suggestions for Optimizations

- Try using different heuristics for choosing which cell to assign next.
- Try restricting the domain for each cell by using what you know about the logic of the Sudoku puzzle (be careful with backtracking!).

- Try using different data structures if copy operations or other similar processes are slowing down your code.

If none of this gives you the speed you're wanting, consider implementing a fully different algorithm such as an evolutionary or exact cover solution.

To work out which parts of your code need optimization, trying importing cProfile and then running it like this on boards that are giving you trouble:
`‘cProfile.run(‘‘Solver(‘‘tests/example.csv")", sort=‘‘tottime")’`

5.1.2 Contest Rules

The six test suites together contribute to half of each contestant's final ranking and the nine named boards together contribute to the other half.

Participants are eligible for any contest where they can complete every board correctly, under two minutes each.

The top-ranked participant overall will receive at least 25 points of extra credit on this assignment, and the second-ranked participant will receive 15 points. Overall, credit will be allocated based on the impressiveness of your performance.

Overall rankings will be calculated by averaging the number of times the lowest time a person's program took to complete each contest. For example, if a person got first place in one contest and took 3 seconds in the other one, where the first place finisher took 2 seconds, that person's average score will be 1.25. Those who do not compete in a certain contest will receive score of the slowest finisher plus one.

5.1.3 Other Restrictions

- All code must be in standard Python 3.
- Submitted code must be your own work.
- Must cite all sources for ideas and algorithms.
- Cannot use features of any particular board in your code/algorithm.
- Cannot change behavior of any Python libraries.
- Your code must be entirely contained within your a2.py file or a2_bonus.py file.
- Solver.board.board must be accessible in the format used in the homework (Its internal representation can be different if you'd like).

5.1.4 Submissions

The contest will be run in two rounds, with the first round due at:
11:59PM Monday September 14th,
and the second due at:
11:59PM Wednesday September 16th.
You are only allowed to participate in the second round if you have participated in the first round, however, only the second round performance will be considered for extra credit.

The results for each round will be presented in class. A submission consists of your a2.py or a2_bonus.py file and a nickname you would like to be used publicly to present your results. Your nickname cannot contain your name or any personally identifiable information. We will generate a new nickname for you if your chosen nickname is not acceptable.

Please email all submissions to: **abrleite@iu.edu**