CMPSC 442: Homework 3 Constraint Satisfaction Problems

Follow these steps exactly, so the Gradescope autgrader can grade your homework. Failure to do so will result in a zero grade:

1. You *must* download the homework template file `homework3_cmpsc442.py` from Canvas. Each template file is a python file that gives you a headstart in creating your homework python script with the correct function names for autograding.

2. You *must* rename the file by replacing `cmpsc442` with your PSU id from your official PSU. For example, if your PSU email id is abcd1234, you would rename your file as `homework3_abcd1234.py` to submit to Canvas, and to Gradescope.

3. You *must* download the zip file with the puzzles to practice on, sudoku.zip.

4. Upload your *py file to Canvas by the due date, and to Gradescope by its due date. The Gradescope due date is five minutes later, but it is your responsibility to upload on time. If the submission on Canvas or Gradescope closes before you upload, your homework will be counted late, and you might get a zero grade.

5. Make sure your file can import before you submit; the autograder imports your file. If it won't import, you will get a zero.

Instructions

In this assignment, you will implement three inference algorithms for the popular puzzle game Sudoku.

A skeleton *.py file `homework3_cmpsc442.py` containing empty definitions for each question has been provided (see above). Since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in case you find yourself reinventing the wheel. If you are unsure where to start, consider taking a look at the data structures and functions defined in the `collections`, `copy`, and `itertools` modules.

You will find that in addition to a problem specification, most programming questions also include one or two examples from the Python interpreter. These are meant to illustrate typical use cases to clarify the assignment, and are not comprehensive test suites. In addition to performing your own testing, you are strongly encouraged to verify that your code gives the expected output for these examples before submitting.

It is highly recommended that you follow the Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. However, your code will not be graded for style.

1. Sudoku

In the game of Sudoku, you are given a partially-filled 9 x 9 grid, grouped into a 3 x 3 grid of 3 x 3 blocks. The objective is to fill each square with a digit from 1 to 9, subject to the requirement that each row, column, and block must contain each digit exactly once.

In this section, you will implement the AC-3 constraint satisfaction algorithm for Sudoku, along with two extensions that will combine to form a complete and efficient solver.

A number of puzzles have been made available in the sudoku.zip file, including:

- An easy-difficulty puzzle: `hw3-easy.txt`.
- Four medium-difficulty puzzles: `hw3-medium1.txt`, `hw3-medium2.txt`, `hw3-medium3.txt`, and `hw3-medium4.txt`.
- Two hard-difficulty puzzles: `hw3-hard1.txt` and `hw3-hard2.txt`.

The examples in this section assume that these puzzle files have been placed in a folder named `sudoku` located in the same directory as the homework file.

An example puzzle from the Daily Pennsylvanian, available as `hw3-medium1.txt`, is depicted below.

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

Textual Representation

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

Initial Configuration

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

Solved Configuration

1. **[3 points]** In this section, we will view a Sudoku puzzle not from the perspective of its grid layout, but more abstractly as a collection of cells. Accordingly, we will represent it internally as a dictionary mapping from cells, i.e. (row, column) pairs, to sets of possible values.

   In the `Sudoku` class, write an initialization method `__init__(self, board)` that stores such a mapping for future use. Also write a method `get_values(self, cell)` that returns the set of values currently available at a particular cell.

In addition, write a function `read_board(path)` that reads the board specified by the file at the given path and returns it as a dictionary. Sudoku puzzles will be represented textually as 9 lines of 9 characters each, corresponding to the rows of the board, where a digit between `"1"` and `"9"` denotes a cell containing a fixed value, and an asterisk `"*"` denotes a blank cell that could contain any digit.

```
>>> b = read_board("sudoku/hw3-
medium1.txt") >>> Sudoku(b).get_values((0,
0))
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> b = read_board("sudoku/hw3-
medium1.txt") >>> Sudoku(b).get_values((0,
1))
set([1])
```

2. **[2 points]** Write a function `sudoku_cells()` that returns the list of all cells in a Sudoku puzzle as (row, column) pairs. The line `CELLS = sudoku_cells()` in the `Sudoku` class then creates a class-level constant `Sudoku.CELLS` that can be used wherever the full list of cells is needed. Although the function `sudoku_cells()` could still be called each time in its place, that approach results in a large amount of repeated computation and is therefore highly inefficient. The ordering of the cells within the list is not important, as long as they are all present.

```
>>> sudoku_cells()
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), ..., (8, 5), (8, 6), (8, 7), (8, 8)]
```

3. **[5 points]** Write a function `sudoku_arcs()` that returns the list of all arcs between cells in a Sudoku puzzle corresponding to inequality constraints. In other words, each arc should be a pair of cells whose values cannot be equal in a solved puzzle. The arcs should be represented a two-tuples of cells, where cells themselves are (row, column) pairs. The line `ARCS = sudoku_arcs()` in the `Sudoku` class then creates a class-level constant `Sudoku.ARCS` that can be used wherever the full list of arcs is needed. The ordering of the arcs within the list is not important, as long as they are all present.

```
>>> ((0, 0), (0, 8)) in sudoku_arcs()
True
>>> ((0, 0), (8, 0)) in sudoku_arcs()
True
>>> ((0, 8), (0, 0)) in sudoku_arcs()
True
```

```
>>> ((0, 0), (2, 1)) in sudoku_arcs()
True
>>> ((2, 2), (0, 0)) in sudoku_arcs()
True
>>> ((2, 3), (0, 0)) in sudoku_arcs()
False
```

4. **[10 points]** In the `Sudoku` class, write a method `remove_inconsistent_values(self, cell1, cell2)` that removes any value in the set of possibilities for `cell1` for which there are no values in the set of possibilities for `cell2` satisfying the corresponding inequality constraint. Each cell argument will be a (row, column)

pair. If any values were removed, return `True`; otherwise, return `False`.

Hint: Think carefully about what this exercise is asking you to implement. How many values can be removed during a single invocation of the function?

```
>>> sudoku = Sudoku(read_board("sudoku/hw3-easy.txt")) # See below for a picture.
>>> sudoku.get_values((0, 3))
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> for col in [0, 1, 4]:
...     removed = sudoku.remove_inconsistent_values((0, 3), (0, col))
...     print removed, sudoku.get_values((0, 3))
...
True set([1, 2, 3, 4, 5, 6, 7, 9])
True set([1, 3, 4, 5, 6, 7, 9])
False set([1, 3, 4, 5, 6, 7, 9])
```

5. **[15 points]** In the `Sudoku` class, write a method `infer_ac3(self)` that runs the AC-3 algorithm on the current board to narrow down each cell's set of values as much as possible. Although this will not be powerful enough to solve all Sudoku problems, it will produce a solution for easy-difficulty puzzles such as the one shown below. By "solution", we mean that there will be exactly one element in each cell's set of possible values, and that no inequality constraints will be violated.

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

Initial Configuration

`hw3-easy.txt`

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

Result of Running AC-3

6. **[30 points]** Consider the outcome of running AC-3 on the medium-difficulty puzzle shown below. Although it is able to determine the values of some cells, it is unable to make significant headway on the rest.

				8	3	4		

				8	3	4	7	

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

Initial Configuration

`hw3-medium2.txt`

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

Inference Beyond AC-3

However, if we consider the possible placements of the digit 7 in the upper-right block, we observe that the 7 in the third row and the 7 in the final column rule out all but one square, meaning we can safely place a 7 in the indicated cell despite AC-3 being unable to make such an inference.

In the `Sudoku` class, write a method `infer_improved(self)` that runs this improved version of AC-3, using `infer_ac3(self)` as a subroutine (perhaps multiple times). You should consider what deductions can be made about a specific cell by examining the possible values for other cells in the same row, column, or block. Using this technique, you should be able to solve all of the medium-difficulty puzzles.

7. **[30 points]** Although the previous inference algorithm is an improvement over the ordinary AC-3 algorithm, it is still not powerful enough to solve all Sudoku puzzles. In the `Sudoku` class, write a method `infer_with_guessing(self)` that calls `infer_improved(self)` as a subroutine, picks an arbitrary value for a cell with multiple possibilities if one remains, and repeats. You should implement a backtracking search which reverts erroneous decisions if they result in unsolvable puzzles. For efficiency, the improved inference algorithm should be called once after each guess is made. This method should be able to solve all of the hard-difficulty puzzles, such as the one shown below.

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

Initial Configuration

`hw3-hard1.txt`

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

Result of Inference with Guessing