Constraint Satisfaction Problems

ACADEMIC HONESTY

As usual, the standard honor code and academic honesty policy applies. We will be using automated **plagiarism detection** software to ensure that only original work is given credit. Submissions isomorphic to (1) those that exist anywhere online, (2) those submitted by your classmates, or (3) those submitted by students in prior semesters, will be detected and considered plagiarism.

INSTRUCTIONS

In this assignment you will focus on constraint satisfaction problems. You will be implementing the AC-3 and backtracking algorithms to solve Sudoku puzzles. The objective of the game is just to fill a 9 x 9 grid with numerical digits so that each column, each row, and each of the nine 3 x 3 sub-grids (also called boxes) contains one of all of the digits 1 through 9. If you have not played the game before, you may do so at **sudoku.com** to get a sense of how the game works. Please read all sections of the instructions carefully.

I. Introduction

	1	2	3	4	5	6	7	8	9
A	8		9	5		1	7	3	6
B	2		7		6	3			
C	1	6							
D					9		4		7
E		9		3		7		2	
F	7		6		8				
G								6	3
H				9	3		5		2
I	5	3	2	6		4	8		9

	1	2	3	4	5	6	7	8	9
A	8	4	9	5	2	1	7	3	6
B	2	5	7	8	6	3	9	1	4
C	1	6	3	7	4	9	2	5	8
D	3	2	5	1	9	6	4	8	7
E	4	9	8	3	5	7	6	2	1
F	7	1	6	4	8	2	3	9	5
G	9	8	4	2	7	5	1	6	3
H	6	7	1	9	3	8	5	4	2
I	5	3	2	6	1	4	8	7	9

Consider the Sudoku puzzle as pictured above. There are 81 **variables** in total, i.e. the tiles to be filled with digits. Each variable is named by its **row** and its **column**, and must

be assigned a **value** from 1 to 9, subject to the constraint that no two cells in the same row, column, or box may contain the same value.

In designing your classes, you may find it helpful to represent a Sudoku board with a Python dictionary. The keys of the dictionary will be the variable names, each of which corresponds directly to a location on the board. In other words, we use the variable names **Al** through **A9** for the top row (left to right), down to **I1** through **I9** for the bottom row. For example, in the example board above, we would have *sudoku*["**B1**"] = **9**, and *sudoku*["**E9**"] = **8**. This is the highly suggested representation, since it is easiest to frame the problem in terms of **variables**, **domains**, and **constraints** if you start this way. In this assignment, we will use the number **zero** to indicate tiles that have not yet been filled.

II. What You Need To Submit

Your job in this assignment is to write driver.py, which intelligently solves Sudoku puzzles. Your program will be executed as follows:

$ python driver.py <input_string>

In the starter code folder, you will find the file sudokus_start.txt, containing hundreds of sample Sudoku puzzles to be solved. Each Sudoku puzzle is represented as a single line of text, which starts from the top-left corner of the board, and enumerates the digits in each tile, row by row. For example, the first Sudoku board in sudokus_start.txt is represented as the string:

003020600900305001001806400008102900700000008006708200002609500800203009005010300

When executed as above, replacing "<input_string>" with any valid string representation of a Sudoku board (for instance, taking any Sudoku board from sudokus_start.txt), your program will generate a file called output.txt, containing a single line of text representing the finished Sudoku board. Since this board is solved, the string representation will contain no zeros. Here is an example of an output:
483921657967345821251876493548132976729564138136798245372689514814253769695417382

You may test your program extensively by using sudokus_finish.txt, which contains the solved versions of all of the same puzzles.

Note on Python 3

As usual, if you choose to use Python 3, then name your program driver_3.py. In that case, the grader will automatically run your program using the python3 binary instead. Please only submit one version. If you submit both versions, the grader will only grade one of them, which probably not what you would want. To test your algorithm in Python 3, execute the game manager like so:

$ python3 driver_3.py <input_string>

Besides your code (your driver and any other python code dependency), submit a file **hw_sudoku_UNI.txt** with you results and observations (including the number of sudoku you could solve and which one were solved), running time, and all relevant information.

III. AC-3 Algorithm

First, implement the **AC-3 algorithm**. Test your code on the provided set of puzzles in sudokus_start.txt. To make things easier, you can write a separate wrapper script (bash, or python) to loop through all the puzzles to see if your program can solve them. How many of the puzzles you can solve? Is this expected or unexpected? Report the puzzles you can solve with AC-3 alone in hw_sudoku_UNI.txt. Reporting the index of the puzzles you can solve with Sudoku is enough. Report also your observations, running time, and any relevant information about your implementation.

IV. Backtracking Algorithm

Now, implement **backtracking** search using the **minimum remaining value** heuristic. The order of values to be attempted for each variable is up to you. When a variable is assigned, apply **forward checking** to reduce variables domains.
Test your code on the provided set of puzzles in sudokus_start.txt. Use your **AC-3** implementation to reduce the domains of variable before you perform backtracking search.
Can you solve all puzzles now? Report the puzzles you can solve now using AC-3 followed by backtracking search. Report your observations, running time, difference in running time between backtracking alone and backtracking preceded with AC-3 and any relevant information about your implementation (no specific format of the write-up will be provided, use your best judgment).

V. Important Information

Please read the following information carefully. Before you post a clarifying question on piazza, make sure that your question is not already answered in the following sections.

1. Test-Run Your Code

Please test your code and make sure it successfully produces an output file with the correct format. Make sure the format is the same the example provided above.

2. Grading Submissions

We will test your final program on **20 test cases**. Each input test case will be rated **5 points** for a successfully solved board, and zero for any other resultant output. The test cases are no different in nature than the hundreds of test cases already provided in your starter code folder, for which the solutions are also available. If you can solve all of those, your program will most likely get full credit. In addition, you will also get 10 extra points for a clear concise write-up. In sum, your submission will be assessed out of a total of 110 points. The 10 extra points are bonus.

3. Time Limit

By now, we expect that you have a good sense of appropriate data structures and object representations. Naive brute-force approaches to solving Sudoku puzzles may take minutes, or even hours, to [possibly never] terminate. However, a correctly implemented backtracking approach as specified above should take **well under a minute** per puzzle. The grader will provide some breathing room, but programs with much longer running times will be killed.

4. Just for fun

Try your code on the world's hardest Sudoku and see if you can solve them! There is nothing to submit here, just for fun. Here is an example:

Sudoku:

800000000003600000070090200050007000000045700000100030001000068008500010090000400

Solution:

812753649943682175675491283154237896369845721287169534521974368438526917796318452