Week 9 Project: Constraint Satisfaction Problems

courses.edx.org/courses/course-v1:ColumbiaX+CSMM.101x+1T2017/courseware/70f2dbe84e944ba485121acdd0a1fe47/fd682db

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.

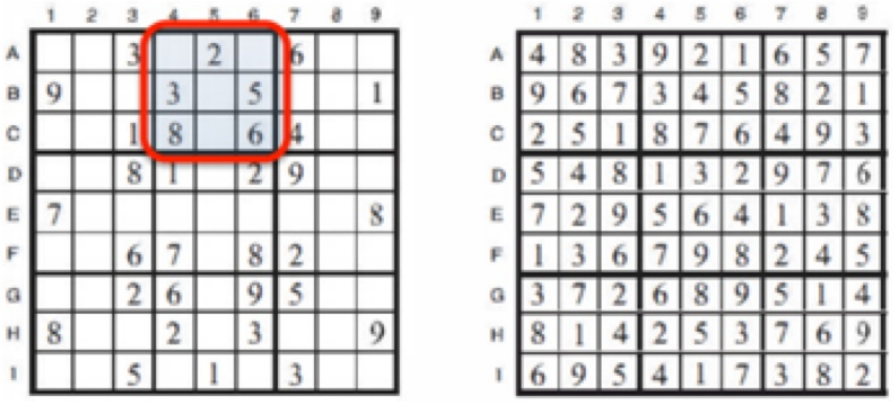
INSTRUCTIONS

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. In particular, note that you have a total of **5 submission** attempts.

I. Introduction

Consider the Sudoku puzzle as pictured below. 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.



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

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

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 **A1** 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

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 Sudoku board in the diagram shown above is represented as the string:

003020600900300050010018064 00... (and so on)

When executed as above, replacing "" 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 file. 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

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?

IV. Backtracking Algorithm

Now, implement **backtracking** 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. Can you solve all puzzles now?

V. Important Information

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

1. Test-Run Your Code

To avoid wasting submission attempts, please test-run your code on Vocareum, and make sure it successfully produces an output file with the correct format. You can do this by hitting the **RUN** button, which simply executes your program with a sample input string containing a valid starting Sudoku board. After you hit **RUN**, when your program terminates, you should locate the output file within your working directory. Make sure the format is the same as the **example**.

2. Grading Submissions

We will test your final program on **20 test cases**. Each input test case will be rated **10 points** for a successfully solved board, and zero for any other resultant output. In sum, your submission will be assessed out of a total of 200 points. 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.

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.

USE OF VOCAREUM

This assignment uses Vocareum for submission and grading. Vocareum comes equipped with an editing environment that you may use to do your development work. You are **NOT** required to use the editor. In particular, you are free to choose your favorite editor / IDE to do your development work on. When you are done with your work, you can simply upload your files onto Vocareum for submission and grading.

However, your assignments will be graded on the platform, so you **MUST** make sure that your code executes without error on the platform. In particular, do not use any additional third-party libraries and packages. We do not guarantee that they will work on the platform, even if they work on your personal computer. For the purposes of this project, everything that comes with the standard Python library should be more than sufficient.

Constraint Satisfaction (AI.a) (External resource)

(200.0 points possible)

Your email address will be used to identify your submission entry.