System maintenance is scheduled for Thursday, March 29, 2018 from 14:00-16:00 UTC. Courses might not be available during this time.

edX

Week 9 Project: 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.
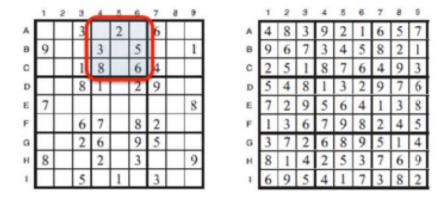
INSTRUCTIONS

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 visit **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
> **II.** What You Need To Submit
> **III.** AC-3 Algorithm
> **IV.** Backtracking Algorithm
> **V.** Important Information
> **VI.** Before You Submit

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.

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. However, you can choose other data structures if you prefer.

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. In this assignment, we will use the number **zero** to indicate tiles that have not yet been filled. For example, the Sudoku board in the diagram shown above is represented as the string:

003020600900300050010018064400... (and so on)

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 and the algorithm name (**AC3** or **BTS**, explained later) which solved the Sudoku board. You must use **a single white space** as a delimiter between the board and the algorithm name. For example, output.txt looks like:

```
1675238499841765233254896717983152646427981355316427984768319522139574868
59264317 BTS
```

(single line, separated by a single white space)

Since this board is solved, the string representation will contain no zeros. 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 as follows:

```
$ python3 driver_3.py <input_string>
```

III. AC-3 Algorithm (`AC3`)

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. As shown in `sudokus_finish.txt`, there are only 2/400 Sudoku boards which can be solved AC3 alone. Is this expected or unexpected?

IV. Backtracking Algorithm (`BTS`)

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. Precedence over BTS

To check how powerful BTS is compared to AC3, **you must execute AC-3 algorithm before Backtracking Search algorithm**. That is, your program looks like this:

```
assignment = AC3(given_sudoku_board)
if (solved(assignment))
        return "<filled sudoku board>" + " AC3"
assignment = BTS(given_sudoku_board)
        return "<filled sudoku board>" + " BTS"
```

2. 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 board and the algorithm name is separated by a single white space.

3. Grading Submissions

We will test your final program on **20 test cases**. You can assume all test cases can be solved at least by BTS. Some of test cases might be solved by AC3 alone. Each input test case will be rated **5 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 100 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.

4. 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.t) (External resource)

(100.0 points possible)

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

Launch Project ⧉

Learn About Verified Certificates