

Project 1: Constrained Satisfaction & Backtracking

Introduction

In classic sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

	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

Sudoku has 81 variables, i.e., 81 tiles. The variables are named by row and column and are valued from 1 to 9 subject to the constraints that two cells in the same row, column, or box may be the same.

Frame your problem in terms of variables, domains, and constraints. We will represent a Sudoku using a Python dictionary or hash map, where each key is a variable name based on location, and value of the tile placed there. For instance, for the Sudoku above, we have the following:

- `sudoku_dict["B3"] = 7`
- `sudoku_dict["F1"] = 7`
- `sudoku_dict["D1"] = 0` (we assign 0 to empty cells)

Running your program

Your program needs to execute as follows:

```
python3 sudoku.py <input_string>
```

You will be provided with a file named `sudoku_boards.txt` that contains samples of unsolved Sudoku boards, and `sudoku_boards_solved.txt` with their corresponding solutions. Each board is represented as a single line of text, starting from the top-left corner of the board, and listed left-to-right, top-to-bottom.

For instance, the string

```
003020600900305001001806400008102900700000008006708200002609500800203009005010300
```

is equivalent to the following Sudoku board:

```
0 0 3 0 2 0 6 0 0
9 0 0 3 0 5 0 0 1
0 0 1 8 0 6 4 0 0
0 0 8 1 0 2 9 0 0
7 0 0 0 0 0 0 0 8
0 0 6 7 0 8 2 0 0
0 0 2 6 0 9 5 0 0
8 0 0 2 0 3 0 0 9
0 0 5 0 1 0 3 0 0
```

Your program will generate `output.txt`, containing a single line of text representing the finished Sudoku board. E.g.:

```
483921657967345821251876493548132976729564138136798245372689514814253769695417382
```

Test your program using `sudoku_boards_solved.txt`, which contains the solved versions of all of the same puzzles.

Backtracking Algorithm

Implement backtracking search using the minimum remaining value heuristic. Pick your own order of values to try for each variable and apply forward checking to reduce variables domains.

- Test your program on `sudoku_boards.txt`.

Grading Submissions

We test your final program on 20 boards. Each board is worth 5 points if solved, and zero otherwise. These boards are similar to those in `sudoku_boards.txt`, so if you solve all those, you'll get full credit.

No brute-force please! Your program should solve puzzles in well under a minute per board. Programs with much longer running times will be killed.

Deliverables

1. Your `sudoku.py` file (and any other python code dependency)

2. A `README.txt` with your results, including the:

- number of boards you could solve from `sudoku_boards.txt`,

- Ensure that your file is named `sudoku.py`. You should build on top of the `sudoku.py` file provided which contains some helper functions.

- Ensure that your file compiles and runs.

Important: We encourage group discussion, but the submission is individual.