Sudoku Solver

Usage

Requires `python 3.6+`

```
python3 sudoku/solver.py sudoku.txt
more log.info
```

Log output: `log.info`

Optional Docker Usage

```
docker build -t sudoku:1.0 .
docker run -it sudoku:1.0 bash
```

And once in the container:

```
python3 sudoku/solver.py sudoku.txt
more log.info
```

Tests

Requires `pytest`

```
pytest -s -v tests/tester.py
```

Approaches

In the large space of potential approaches for Sudoku solvers, I started with the simplest, non-exhaustive approach, and measured the performance. Then, by applying two Sudoku-specific heuristics to this approach, I was able to reach a satisfactory level of performance.

Brute-force wasn't an option as a first approach since the complexity is `O(n^m)`, where `n` is the number of possible values per cell (9) and `m` is the number of unsolved cells.

1. Depth first search with backtracking

Note: code not included

Algorithm:

1. Find the first unsolved cell (in raster-scan order)
   a. If we cannot find an unsolved cell, the board is solved (base case)
2. For each possible candidate (1-9)
   a. If candidate is valid (does not break Sudoku rules), assign that cell with candidate, and return (recurse) to step 1.
3. If no candidates in step 2) are valid, undo the candidate assignment, backtrack up the tree and try a new candidate for the previous node

Pseudocode:

```
func solve(board):
    pos = find_unsolved_cell(board)
    if no pos exists:
        return True

    for each candidate:
        if is_valid(board, pos, candidate):
            board[pos] = candidate
            if solve(board):
                return True
        backtrack(board[pos])
    return False
```

Results

Total time for the 50 Sudoku puzzles: 25s
Hardware: 2015 Intel Core i7

2. DFS with backtracking + heuristics

DFS can be interpreted as traversal through an N-ary tree. In our case, the first unsolved cell is the root node, and each candidate can be represented as a child node.

Heuristic:

In approach 1), we made two arbitrary decisions:

1. Evaluate 1-9 as candidates for each unsolved cell
2. Solve cells in raster-scan order

We instead, can

1. Eliminate candidates of unknown cells based on the initial known cell values
2. Solve cells in order of increasing # of candidates

By eliminating candidates and choosing the cells with the least # of candidates to solve first, we significantly constrain the search space.

For example, if we start by choosing an unsolved cell with 6 candidates, there are 6 child nodes for DFS to visit, whereas if we choose an unsolved cell with 2 candidates, there are only 2 different DFS paths available.

This is known as the branching factor of the tree, and a reduction in the branching factor can reduce the real-world execution time almost exponentially.

Algorithm:

0. For each unsolved cell, compute the set of candidates and store this in a `map`. A cell's existence in this `map` indicates that it is still unsolved.

Then, the algorithm is the same as in Approach 1 except for a couple details (bolded):

1. Find the first unsolved cell **(with the least # of candidates, precomputed in step 0.)**
   a. If we cannot find an unsolved cell, the board is solved
2. For each candidate **(from the precomputed `map`)**
   a. If candidate is valid (**does not cause peers to have 0 candidates***), assign that cell with that value, and return (recurse) to step 1.
3. If no candidates in step 3) are valid, backtrack** up the tree and try a new candidate for the previous node

* This is the backtracking trigger. As we assign a candidate to a cell, we can remove that candidate from the cell's peers. If this removal causes the peer to not have any candidates left, we've made an error and must backtrack.

** We'll need to keep track of the deleted candidates for backtracking purposes

Pseudocode:

```
candidates = precompute_candidates(board)           <-

func solve(board, candidates):
    pos = find_unsolved_cell(board)                 <-
    if no pos exists:
        return True

    for each candidate:
        if is_valid(board, pos, candidate):         <-
            board[pos] = candidate
            if solve(board):
                return True
        backtrack(board[pos])                       <-
    return False
```

The overall structure of the algorithm remains the same. `<-` indicates modifications to how this method is implemented compared to approach 1)

Results

Stats for the 50 Sudoku puzzles:

	Time (s)
Total	0.339
Mean	0.006
Std	0.003
Min	0.004
Max	0.022

Hardware: 2015 Intel Core i7

Assumptions

Input file format of `sudoku.txt`:

```
Grid 01
003020600
900305001
001806400
008102900
700000008
006708200
002609500
800203009
005010300
Grid 02
200080300
060070084
030500209
000105408
000000000
402706000
301007040
720040060
004010003
```

- 9x9 grids
- Each row a consecutive set of characters
- Delimited by "Grid XX\n", where X is an integer [0-9]