Richard Messina rdm420 N13468622
CS4613 Project 2 – Sudoku Solver

INSTRUCTIONS

The included file, *solver.py*, contains the source code to my solution. It was written for Python 3.7.3, but most versions of Python 3 should work. Instructions for running this script can be found as follows:

```
> python3 solver.py --help
usage: solver.py [-h] -i INPUT -o OUTPUT -f FUNCTION

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Input file name
  -o OUTPUT, --output OUTPUT
                        Output file name
```

Therefore, to run this file against certain inputs, you can follow this example:

```
> python3 solver.py -i InputFile.txt -o OutputFile.txt
```

RESULTING OUTPUT FILES

Output for SUDUKO_Input1.txt

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

Output for SUDUKO_Input2.txt

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

Output for SUDUKO_Input3.txt

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

SOURCE CODE (contents of *solver.py*)

```
# Richard Messina rdm420 N13468622
# CS4613 Project 2 - Sudoku Solver
# Python 3.7.3

import argparse


class SudokuTile(object):
    def __init__(self, row_idx, col_idx, domain=None):
        self._domain = domain if domain else {1, 2, 3, 4, 5, 6, 7, 8, 9}
        self.row_idx = row_idx
        self.col_idx = col_idx

        self.row_neighbors = None
        self.col_neighbors = None
        self.block_neighbors = None

        self.backtracking_value = None
        self.inferred_exclusions = set()

    def __repr__(self):
        return str(self.value)

    @property
    def value(self):
        if len(self.domain) == 1:
            return tuple(self.domain)[0]
        return self.backtracking_value

    @property
    def neighbors(self):
        return self.row_neighbors + self.col_neighbors +
self.block_neighbors

    @property
    def ordered_domain(self):
        # domain values ordered from lowest to highest value
        return sorted(self.domain)

    @property
    def domain(self):
        # the working domain should include any inferences
        return self._domain.difference(self.inferred_exclusions)

    def apply_inferences(self):
        self._domain = self.domain
        self.inferred_exclusions.clear()

    def check_val_consistent(self, val):
        for neighbor_tile in self.neighbors:
            if neighbor_tile.value == val:
                return False
        return True
```

```
class SudokuBoard(object):
    def __init__(self, input_file):
        self.board_tiles_from_file(input_file)
        self.establish_neighbors()

    # represent the sudoku board as a 9x9 matrix of SudokuTiles
    def board_tiles_from_file(self, input_file):
        with open(input_file, 'r') as in_stream:
            self.board = [
                [SudokuTile(row_idx, col_idx, {int(tl)}) if tl != '0' else
SudokuTile(
                    row_idx, col_idx) for col_idx, tl in
enumerate(row.split())]
                for row_idx, row in enumerate(in_stream.readlines())
            ]

    def get_row_neighbors(self, tile):
        return self.board[tile.row_idx][:tile.col_idx] +
self.board[tile.row_idx][tile.col_idx + 1:]

    def get_col_neighbors(self, tile):
        return [row[tile.col_idx] for idx, row in enumerate(self.board) if
idx != tile.row_idx]

    def get_block_neighbors(self, tile):
        block_row = 3 * (tile.row_idx // 3)
        block_col = 3 * (tile.col_idx // 3)
        block_neighbors = self.board[block_row][block_col:block_col + 3] +
\
            self.board[block_row + 1][block_col:block_col + 3] + \
            self.board[block_row + 2][block_col:block_col + 3]
        current_tile = ((tile.row_idx % 3) * 3) + (tile.col_idx % 3)
        return block_neighbors[:current_tile] +
block_neighbors[current_tile + 1:]

    # neighbors for each tile should never change, so let's establish
these right away
    def establish_neighbors(self):
        for row in self.board:
            for tile in row:
                tile.row_neighbors = self.get_row_neighbors(tile)
                tile.col_neighbors = self.get_col_neighbors(tile)
                tile.block_neighbors = self.get_block_neighbors(tile)

    def is_complete(self):
        for row in self.board:
            for tile in row:
                if not tile.value:
                    return False
        return True

    def forward_check(self, tile):
        if not tile.value:
            for neighbor_tile in tile.neighbors:
                tile.inferred_exclusions.add(neighbor_tile.value)

        if tile.value:
            # neighbors' domains can be further reduced
            return self.forward_check_neighbors(tile)
        elif len(tile.domain) == 0:
            return False
        return True

    def forward_check_neighbors(self, tile):
        for neighbor_tile in tile.neighbors:
            if not neighbor_tile.value:
                if not self.forward_check(neighbor_tile):
                    return False
        return True

    def forward_check_all_tiles(self):
        for row in self.board:
            for tile in row:
                if not tile.value:
                    if not self.forward_check(tile):
                        return False

        # immediately apply all initially generated inferences
        for row in self.board:
            for tile in row:
                tile.apply_inferences()

        return True

    def minimum_remaining_value_heuristic(self):
        candidates = None
        smallest_domain_size = 10
        for row in self.board:
            for tile in row:
                if not tile.value:
                    if len(tile.domain) < smallest_domain_size:
                        candidates = [tile]
                        smallest_domain_size = len(tile.domain)
                    elif len(tile.domain) == smallest_domain_size:
                        candidates.append(tile)
        return candidates

    def degree_heuristic(self, candidates):
        desired_tile = None
        highest_degree = 0
        for tile in candidates:
            blank_neighbors = list(filter(lambda tl: not tl.value,
tile.neighbors))
            if len(blank_neighbors) > highest_degree:
                desired_tile = tile
                highest_degree = len(blank_neighbors)
        return desired_tile if desired_tile else candidates[0]

    # finds the tile with the minimum remaining values (smallest domain)
and highest degree
    def select_blank_tile(self):
        candidates = self.minimum_remaining_value_heuristic()
        if len(candidates) == 1:
            return candidates[0]
        return self.degree_heuristic(candidates)

    def clear_all_inferences(self):
        for row in self.board:
            for tile in row:
                tile.inferred_exclusions.clear()

    def backtrack_board(self):
        if self.is_complete():
            return True

        tile = self.select_blank_tile()

        for possible_val in tile.ordered_domain:
            if tile.check_val_consistent(possible_val):
                tile.backtracking_value = possible_val
                if self.forward_check(tile) and self.backtrack_board():
                    return True
            tile.backtracking_value = None
            self.clear_all_inferences()
        return False

    def no_solution(self):
        print('No solution exists for the given board :(')

    def solve_to_file(self, output_file):
        # reduce the domain via forward checking for each tile which does
not have a value assigned
        if not self.forward_check_all_tiles():
            self.no_solution()
            return

        # use backtracking to explore possible solutions until a complete
solution is found
        if not self.backtrack_board():
            self.no_solution()
            return

        print('Solution found! Writing to {}...'.format(output_file))
        with open(output_file, 'w') as out_stream:
            for row in self.board:
                print(*row, sep=' ', file=out_stream)
                print(*row, sep=' ')


if __name__ == '__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument('-i', '--input', help='Input file name',
required=True)
    parser.add_argument('-o', '--output', help='Output file name',
required=True)

    args = parser.parse_args()

    SudokuBoard(args.input).solve_to_file(args.output)
```