Project 2: Kropki Sudoku Solver

How to run the source file

Assuming that input files are in the same directory as solver.py, run the program in command line:

```
py solver.py
```

Problem formulation

The variables are defined as the grids on the board where each one is filled with a number. The completion state is when all of the variables have been assigned a number from 1 to 9, and all the regular sudoku row, column, and box constraints, as well as the dot constraints have been met. This is a discrete variable CSP problem where the domain is finite. The domain was determined to be {1, 2, 3, 4, 5, 6, 7, 8, 9}, as those are the values allowed in a sudoku puzzle. The constraints are that variables in the same row cannot have the same value, variables in the same column cannot have the same value, variables in the same box cannot have the same value, variables with white dots have to be within 1 of its neighbor, and variables with black dots have to be half or twice as much as the neighbor.

Extra credit

Forward checking inference was implemented in this code for extra credit (in the forward_check() function).

Source code

```
import copy
import math


# Constant variables
DOMAIN = [1, 1, 1, 1, 1, 1, 1, 1, 1]
NUM_ROWS = 9
NUM_COLS = 9


class Variable:
    """
```

```
    Variable class that is designed to contain the constraint satisfaction problem state for each
    Each Variable represents a single grid.
    """
    def __init__(self, value='0', domain=DOMAIN, up='0', down='0', left='0', right='0'):
        """
        Constructor
        value: this variable's assignment
        domain: the remaining values in the domain, where 1==available and 0==invalid for its ind
        up: contains this variable's constraint relationship with the grid above.
        down: contains this variable's constraint relationship with the grid below.
        left: contains this variable's constraint relationship with the grid to the left.
        right: contains this variable's constraint relationship with the grid to the right.
        """
        self.value = value
        self.domain = copy.copy(domain)
        self.up = up
        self.down = down
        self.left = left
        self.right = right

    def assigned(self):
        """
        Checks if the variable has already been assigned.
        """
        return True if self.value != '0' else False

    def remaining_value_count(self):
        """
        Checks the number of remaining valid value assignments.
        """
        return sum(self.domain)


def parse_file(filename):
    """
    Function to read through an input file, the initial board state into variables, and update co

    Params:
    filename: string

    Returns:
    board: nested list of all the Variables
    """
    file = open(filename, "r")
    board = []

    # reads initial board assignment
    for i in range(NUM_ROWS):
        line = file.readline()
```

```
        line = line.strip()
        linelst = line.split()
        board_row = []
        for val in linelst:
            board_row.append(Variable(val))
        board.append(board_row)

    # reads horizontal dot constraints
    line = file.readline()
    for i in range(NUM_ROWS):
        line = file.readline()
        line = line.strip()
        linelst = line.split()
        for j in range(len(linelst)):
            if linelst[j] != '0':
                board[i][j].right = linelst[j]
                board[i][j+1].left = linelst[j]

    # reads vertical dot constraints
    line = file.readline()
    for i in range(NUM_ROWS-1):
        line = file.readline()
        line = line.strip()
        linelst = line.split()
        for j in range(len(linelst)):
            if linelst[j] != '0':
                board[i][j].down = linelst[j]
                board[i+1][j].up = linelst[j]

    for i in range(NUM_ROWS):
        for j in range(NUM_COLS):
            if not board[i][j].assigned():
                continue
            update_neighbors(board, i, j)

    file.close()
    return board


def update_neighbors(board, ivar, jvar):
    """
    Function that initializes the inferences based on the assignment of the current Variable.

    Params:
    board: nested list of Variables
    ivar: int
    jvar: int
    """
    var = board[ivar][jvar]
```

```
    valint = int(var.value)

    # updates positional neighbors
    for k in range(NUM_COLS):
        if k == jvar:
            continue
        if board[ivar][k].assigned():
            continue
        board[ivar][k].domain[valint-1] = 0
    for k in range(NUM_ROWS):
        if k == ivar:
            continue
        if board[k][jvar].assigned():
            continue
        board[k][jvar].domain[valint-1] = 0
    for k, l in box_indexes(ivar, jvar):
        if k == ivar or l == jvar:
            continue
        if board[k][l].assigned():
            continue
        board[k][l].domain[valint-1] = 0

    # updates dot constraint neighbors
    up = board[ivar][jvar].up
    down = board[ivar][jvar].down
    left = board[ivar][jvar].left
    right = board[ivar][jvar].right
    if up != '0' and not board[ivar-1][jvar].assigned():
        remain = dot_remains(up, valint)
        for k in range(len(board[ivar-1][jvar].domain)):
            if k + 1 not in remain:
                board[ivar-1][jvar].domain[k] = 0
    if down != '0' and not board[ivar+1][jvar].assigned():
        remain = dot_remains(down, valint)
        for k in range(len(board[ivar+1][jvar].domain)):
            if k + 1 not in remain:
                board[ivar+1][jvar].domain[k] = 0
    if left != '0' and not board[ivar][jvar-1].assigned():
        remain = dot_remains(left, valint)
        for k in range(len(board[ivar][jvar-1].domain)):
            if k + 1 not in remain:
                board[ivar][jvar-1].domain[k] = 0
    if right != '0' and not board[ivar][jvar+1].assigned():
        remain = dot_remains(right, valint)
        for k in range(len(board[ivar][jvar+1].domain)):
            if k + 1 not in remain:
                board[ivar][jvar+1].domain[k] = 0
```

```python
def forward_check(board, ivar, jvar):
    """
    Algorithm that updates the inferences of the first degree neighbor of the newly assigned vari

    Params:
    board: nested list
    ivar: int
    jvar: int

    Returns:
    consistent: bool, true if no problems found, false if an inconsistency is found
    original: nested list, contains the original values of the inferences before they were change
    """
    # saves original inferences
    original = []
    for k in range(NUM_COLS):
        if k == jvar:
            continue
        if board[ivar][k].assigned():
            continue
        original.append([(ivar, k), copy.copy(board[ivar][k].domain)])
    for k in range(NUM_ROWS):
        if k == ivar:
            continue
        if board[k][jvar].assigned():
            continue
        original.append([(k, jvar), copy.copy(board[k][jvar].domain)])
    for k, l in box_indexes(ivar, jvar):
        if k == ivar or l == jvar:
            continue
        if board[k][l].assigned():
            continue
        original.append([(k, l), copy.copy(board[k][l].domain)])

    # updates positional neighbors
    var = board[ivar][jvar]
    valint = int(var.value)
    for k in range(NUM_COLS):
        if k == jvar:
            continue
        if board[ivar][k].assigned():
            continue
        board[ivar][k].domain[valint-1] = 0
        if board[ivar][k].remaining_value_count() == 0:
            return False, original
    for k in range(NUM_ROWS):
        if k == ivar:
            continue
        if board[k][jvar].assigned():
```

```
                continue
            board[k][jvar].domain[valint-1] = 0
            if board[k][jvar].remaining_value_count() == 0:
                return False, original
        for k, l in box_indexes(ivar, jvar):
            if k == ivar or l == jvar:
                continue
            if board[k][l].assigned():
                continue
            board[k][l].domain[valint-1] = 0
            if board[k][l].remaining_value_count() == 0:
                return False, original

        # updates dot constraint neighbors
        up = board[ivar][jvar].up
        down = board[ivar][jvar].down
        left = board[ivar][jvar].left
        right = board[ivar][jvar].right

        if up != '0' and not board[ivar-1][jvar].assigned():
            remain = dot_remains(up, valint)
            for k in range(len(board[ivar-1][jvar].domain)):
                if k + 1 not in remain:
                    board[ivar-1][jvar].domain[k] = 0
            if board[ivar-1][jvar].remaining_value_count() == 0:
                return False, original
        if down != '0' and not board[ivar+1][jvar].assigned():
            remain = dot_remains(down, valint)
            for k in range(len(board[ivar+1][jvar].domain)):
                if k + 1 not in remain:
                    board[ivar+1][jvar].domain[k] = 0
            if board[ivar+1][jvar].remaining_value_count() == 0:
                return False, original
        if left != '0' and not board[ivar][jvar-1].assigned():
            remain = dot_remains(left, valint)
            for k in range(len(board[ivar][jvar-1].domain)):
                if k + 1 not in remain:
                    board[ivar][jvar-1].domain[k] = 0
            if board[ivar][jvar-1].remaining_value_count() == 0:
                return False, original
        if right != '0' and not board[ivar][jvar+1].assigned():
            remain = dot_remains(right, valint)
            for k in range(len(board[ivar][jvar+1].domain)):
                if k + 1 not in remain:
                    board[ivar][jvar+1].domain[k] = 0
            if board[ivar][jvar+1].remaining_value_count() == 0:
                return False, original
        return True, original
```

```python
def dot_remains(dot_type, valint):
    """
    Returns a list of possible values that a variable can have given its dot neighbor's value and

    Params:
    dot_type: str, '1' for white dot, '2' for black
    valint: int, assigned value of neighbor variable

    Returns:
    remain: list of values
    """
    if dot_type == '1':
        if valint == 1:
            remain = [valint+1]
        elif valint == 9:
            remain = [valint-1]
        else:
            remain = [valint-1, valint+1]
    elif dot_type == '2':
        if valint % 2 == 0 and valint <= 4:
            remain = [valint // 2, valint * 2]
        elif valint % 2 == 0 and valint > 4:
            remain = [valint // 2]
        elif valint % 2 == 1 and valint <= 3:
            remain = [valint * 2]
        elif valint % 2 == 1 and valint > 3:
            remain = []
    return remain


def box_indexes(i, j):
    """
    Returns the positions of the box of a particular grid.

    Params:
    i: int
    j: int

    Returns
    zip of positions
    """
    istart = (i // 3) * 3
    jstart = (j // 3) * 3
    ilst = [istart, istart, istart, istart+1, istart+1, istart+1, istart+2, istart+2, istart+2]
    jlst = [jstart, jstart+1, jstart+2, jstart, jstart+1, jstart+2, jstart, jstart+1, jstart+2]
    return zip(ilst, jlst)
```

```
def print_board(board):
    """
    Helper function for code testing and debugging, prints out the current assignment state.
    """
    for i in range(len(board)):
        if i == 3 or i == 6:
            print('---------------------')
        for j in range(len(board[0])):
            if j == 3 or j == 6:
                print('|', end=" ")
            print(board[i][j].value, end=" ")
        print()
    print()
    print()


def assignment_complete(board):
    """
    Checks if all variables have been assigned.
    """
    for i in range(NUM_ROWS):
        for j in range(NUM_COLS):
            if not board[i][j].assigned():
                return False
    return True


def unassigned_neighbors_count(board, i, j):
    """
    Counts how many constraint neighbors of the current variable are unassigned, used in degree h

    Params:
    board: nested list
    i: int
    j: int

    Returns:
    cons: int (number of constraints)
    """
    cons = 0
    for k in range(NUM_COLS):
        if k == j:
            continue
        if not board[i][k].assigned():
            cons += 1
    for k in range(NUM_ROWS):
        if k == i:
            continue
        if not board[k][j].assigned():
```

```
            cons += 1
    for k, l in box_indexes(i, j):
        if k == i or l == j:
            continue
        if not board[k][l].assigned():
            cons += 1
    return cons


def remove_inferences(board, original):
    """
    Goes through the original position and inference values to reinstate them.

    Params:
    board: nested list
    original: nested list
    """
    for var_pos, og_dom in original:
        i, j = var_pos
        board[i][j].domain = og_dom


def select_variable(board):
    """
    Selects the next variable to be assigned using Minimum Remaining Value heuristic and Degree h

    Params:
    board: nested list

    Returns:
    selected: tuple of i and j values of the selected variable
    """
    # MRV heuristic
    mrv = []
    curr_mrv = math.inf
    for i in range(NUM_ROWS):
        for j in range(NUM_COLS):
            var = board[i][j]
            if var.assigned():
                continue
            remain_val = var.remaining_value_count()
            if remain_val > curr_mrv:
                continue
            if remain_val == curr_mrv:
                mrv.append((i, j))
            if remain_val < curr_mrv:
                curr_mrv = remain_val
                mrv.clear()
                mrv.append((i, j))
```

```python
    # Degree heuristic
    selected = None
    selected_deg = 0
    for i, j in mrv:
        degree = unassigned_neighbors_count(board, i, j)
        if not selected:
            selected = (i, j)
            selected_deg = degree
            continue
        if degree > selected_deg:
            selected = (i, j)
            selected_deg = degree
        elif degree < selected_deg:
            continue
    return selected


def backtracking_search(board):
    """
    Main algorithm. Hosts the recursive backtrack algorithm that solves the Kropki Sudoku puzzle.
    """
    def backtrack(board):
        """
        Nested recursive function.
        """
        if assignment_complete(board):
            return board
        ivar, jvar = select_variable(board) # variable selection heuristic
        var = board[ivar][jvar]
        for val in range(1, len(var.domain) + 1):
            if not var.domain[val-1]:
                continue
            var.value = str(val)
            consistent, original = forward_check(board, ivar, jvar) # inference
            if consistent:
                result = backtrack(board)
                if result: return result
            remove_inferences(board, original) # inference removal
            var.value = '0'
        return False

    return backtrack(board)


def write_output(filename, board):
    """
    Writes the solution to the designated output file.
    """
```

```
    file = open(filename, "w")
    for i in range(len(board)):
        for j in range(len(board[0])):
            file.write(board[i][j].value + ' ')
        file.write('\n')
    file.close()


def main():
    """
    Main function. Controls which files to parse, generate, and write to.
    """
    filelist = ["Input1.txt", "Input2.txt", "Input3.txt"]

    for i in range(len(filelist)):
        file = filelist[i]
        board = parse_file(file)
        if not backtracking_search(board):
            raise Exception("No solution found for {}".format(filelist[i]))
        output_file = "Output{file_num}.txt".format(file_num=i+1)
        write_output(output_file, board)
        print("File {input}'s solution generated at {output}".format(input=filelist[i], output=ou


if __name__ == "__main__":
```

Output files

File Input1.txt's solution generated at Output1.txt

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

File Input2.txt's solution generated at Output2.txt

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

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

File Input3.txt's solution generated at Output3.txt

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