Min Kim (msk723)

Instructions:

1) Ensure python 3 is installed so you can run file in command prompt
2) Open your computer's command prompt and go to the directory in which the python file is stored using "cd"
3) Input files must also be in the same directory in order for this to work (Input files will be txt files that contain 9 lines, with 9 values each separated by spaces. This represents the game board. The goal is to solve the sudoku puzzle so each row and column have numbers 1-9, as well as each of the 9 3x3 blocks as shown in the example below. 0's represent blank tiles)

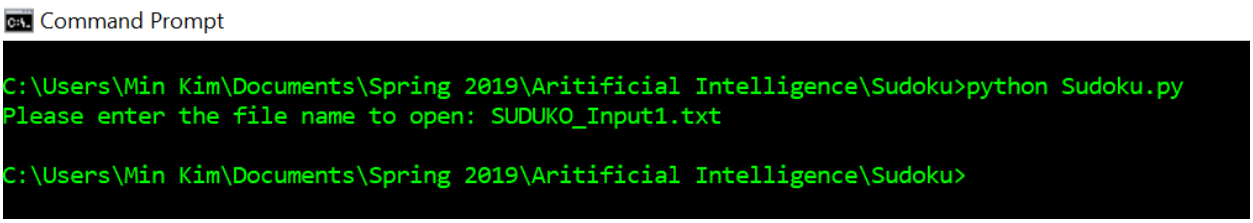
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1. Initial cell values.

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

Figure 2. Solution.

4) Run the command: "python Sudoku.py" without quotes
5) Note that this is assuming file name was not changed, please type in the file name that reflects the current name in directory
6) By doing this the program will run and you will be prompted for the file name, enter the name of the file ex: "SUDUKO_Input1.txt" …Note, for correct formatting of output file name please keep input as "input" or "Input" without random capital letters like "inPUT"
7) Here is an example of a correct input:



8) This will produce a new file in the same directory replacing "input" with "Output" in the file name, creating an example output file name such as "SUDUKO_Output1.txt"
9) The output file should have 9 lines also with 9 values each, replacing the 0 blank tiles with the solution values***Note: Example output files and source code on next pages**

SUDUKO_Output1.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

SUDUKO_Output2.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

SUDUKO_Output3.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:

```
from copy import deepcopy

#function opens user_inputted file and reads the state of the Sudoku board
def readFile():
    #input for file
    cells = []

    #enter filename of puzzle
    fileName = input("Please enter the file name to open: ")

    #open file with inputed text
    with open(fileName) as f:
        line = f.readline()
        #for each line (getting rid of white space and new line) get the data for the puzzle
        while line:
            for num in line:
```

```
            if(num != ' ' and num != '\n'):
                cells.append(num)
        line = f.readline()
    return cells, fileName

#represents each cell in the sudoku, both filled and nonfilled
class Cell:
    #cells have their value, location, and domain
    def __init__(self, number, row, col, block):
        self.number = number
        self.row = row
        self.col = col
        self.block = block
        if number == 0:
            self.domain = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        else:
            self.domain = [number]


#represents the puzzle as a whole and its funcitonality to be solved
class Sudoku:
    def __init__(self, myCells):
        #checks if puzzle is unsolvable
        self.failed = False


        #fill cells, unassigned, assigend, and blocks list to keep track of cell status and location
```

```
        self.cells, self.unassignedCells, self.assignedCells, self.blocks =
self.organizeCells(myCells)


        #forward checking on all assignedCells to decrease possible domains
        for i in range(len(self.assignedCells)):
            if self.failed ==  False:
                if not self.forwardChecking(self.assignedCells[i]):
                    self.failed = True


    #organizes all cells into appropriate lists for the construction of the class, takes a list of
cells in order of the puzzle
    def organizeCells(self, sequentialCells):
        # stores cells as a 2d array for rows and columns and 2d array for blocks
        cells = []
        blocks = []
        #create the 2d arrays
        for i in range(9):
            cells.append([])
            blocks.append([])


        unassignedCells = []
        assignedCells = []
        #sort cells into their blocks
        for row in range(9):
            for col in range(9):
                if row < 3:
                    if col < 3:
```

```
            block = 0
        elif col < 6:
            block = 1
        elif col < 9:
            block = 2
    elif row < 6:
        if col < 3:
            block = 3
        elif col < 6:
            block = 4
        elif col < 9:
            block = 5
    elif row < 9:
        if col < 3:
            block = 6
        elif col < 6:
            block = 7
        elif col < 9:
            block = 8
    #create each cell and put them in the appropriate locations
    currCell = Cell(int(sequentialCells[row * 9 + col]), row, col, block)
    cells[row].append(currCell)
    blocks[block].append(currCell)
    if currCell.number == 0:
        unassignedCells.append(currCell)
    else:
        assignedCells.append(currCell)
```

```
        return cells, unassignedCells, assignedCells, blocks


    #does forward checking for the currCell, programmer must handle using forward checking properly for all given values
    def forwardChecking(self, currCell):
        #check blocks
        block = currCell.block
        for index in range(9):
            #for every item in block, decreease domains approprately
            if currCell is not self.blocks[block][index] and currCell.number in self.blocks[block][index].domain:
                self.blocks[block][index].domain.remove(currCell.number)
                #if domain is less than one, cannot be satsifed reuturn false, the puzzle can't be solved
                if len(self.blocks[block][index].domain) < 1:
                    return False
        #check rows
        row = currCell.row
        for col in range(9):
            if currCell is not self.cells[row][col] and currCell.number in self.cells[row][col].domain:
                #for every row decrease domain
                self.cells[row][col].domain.remove(currCell.number)
                if len(self.cells[row][col].domain) < 1:
                    return False
        #check col
        col = currCell.col
        for row in range(9):
```

```
            if currCell is not self.cells[row][col] and currCell.number in
self.cells[row][col].domain:
                #for every col decrease domains
                self.cells[row][col].domain.remove(currCell.number)
                if len(self.cells[row][col].domain) < 1:
                    return False
        return True


    #checks how many unassigned neighbirs their are to pick between ties in the MRV,
take a cell and check neighbors
    def degreeHeuristic(self, currCell):
        sum = 0
        #blocks / do not inlcude row and columns, already counted in next loops
        for index in range(9):
            if self.blocks[currCell.block][index].number == 0 and currCell is not
self.blocks[currCell.block][index]:
                if currCell.row != self.blocks[currCell.block][index].row and currCell.col !=
self.blocks[currCell.block][index].col:
                    sum += 1
        #row
        for col in range(9):
            if self.cells[currCell.row][col].number == 0 and currCell is not
self.cells[currCell.row][col]:
                sum += 1
        #col
        for row in range(9):
            if self.cells[row][currCell.col].number == 0 and currCell is not
self.cells[row][currCell.col]:
                sum += 1
```

```
        return sum


    #find the cell with lowest amount of values in domain
    def MRV(self):
        currMinIndex = 0
        currMinLen = len(self.unassignedCells[0].domain)
        #compares every unassinged cell for the MRV
        for i in range(1, len(self.unassignedCells)):
            #replaces MRV if new lowest one is found
            if len(self.unassignedCells[i].domain) < currMinLen:
                currMinIndex = i
                currMinLen = len(self.unassignedCells[i].domain)
            #break ties with heuristic, number of unassigned neighbors
            elif len(self.unassignedCells[i].domain) == currMinLen:
                currDegree = self.degreeHeuristic(self.unassignedCells[currMinIndex])
                potentialDegree = self.degreeHeuristic(self.unassignedCells[i])
                if currDegree <= potentialDegree:
                    currMinIndex = i
        return self.unassignedCells[currMinIndex]


    #checks that the assignment of myNumber will be consistent with the puzzle
    def consistencyCheck(self, currCell, myNumber):
        #blocks
        for i in range(9):
            if currCell is not self.blocks[currCell.block][i] and myNumber ==
self.blocks[currCell.block][i].number:
```

```
            return False
        #rows
        for col in range(9):
            if currCell is not self.cells[currCell.row][col] and myNumber == self.cells[currCell.row][col].number:
                return False
        #col
        for row in range(9):
            if currCell is not self.cells[row][currCell.col] and myNumber == self.cells[row][currCell.col].number:
                return False
        return True


    #reverse assigns made when backtracking finds that possible solution is a failure
    def reverseAssigns(self, currCell, myNumber):
        #make cell empty again and put back as unassigned
        currCell.number = 0
        self.unassignedCells.append(currCell)
        #blocks
        for i in range(9):
            if currCell is not self.blocks[currCell.block][i] and myNumber not in self.blocks[currCell.block][i].domain:
                self.blocks[currCell.block][i].domain.append(myNumber)
        #Row
        for col in range(9):
            if currCell is not self.cells[currCell.row][col] and myNumber not in self.cells[currCell.row][col].domain:
                self.cells[currCell.row][col].domain.append(myNumber)
```

```
        #col
        for row in range(9):
            if currCell is not self.cells[row][currCell.col] and myNumber not in
self.cells[row][currCell.col].domain:
                self.cells[row][currCell.col].domain.append(myNumber)
        return True


    #backtrcking search, calls backtracking to work and returns false if no solution is
found
    def backtrackingSearch(self):
        if not self.failed:
            return self.backtracking()


    #recursively assigns elements from domains and checks consisteny until a solution or
no solution is found
    def backtracking(self):
        #if the puzzle is finished, return true
        if len(self.unassignedCells) == 0:
            return True
        #pick MRV for next unassigned cell
        nextCell = self.MRV()


        #deepcopy the domain so we can use it for assigns without ruining the ability to go
back
        cellDomain = deepcopy(nextCell.domain)


        #check if assignment of each value in domain is consistent
```

```
        for item in cellDomain:
            if self.consistencyCheck(nextCell, item):
                #assign but do not change domain yet in case assignment is not ok later
                nextCell.number = item
                #for now though we need to remove it from unassigned to check if it works
                self.unassignedCells.remove(nextCell)
                #forward checking to find failures faster, do not return false yet, in case other domain values work(works without this)
                if self.forwardChecking(nextCell):
                    #if forward checking and recursion work, we return true for a solution of assigned values
                    if self.backtracking():
                        return True
                #otherwise have to go backtrack through recursion and unassign everythin that doesnt work
                self.reverseAssigns(nextCell, item)
        self.failed = True
        return False


def main():
    cells, filename = readFile()
    #create puzzle with cells
    myPuzzle = Sudoku(cells)
    #if early failure inform user
    if myPuzzle.failed:
        print("Sudoku puzzle has no solution, no output file created")
        return None
    #otherwise attempt to solve
```

```
    elif myPuzzle.backtrackingSearch():
        filename = filename.replace("input", "Output")
        filename = filename.replace("Input", "Output")
        file = open(filename, "w")
        first = True
        for row in range(9):
            if not first:
                file.write('\n')
            first = False
            for col in range(9):
                if col < 8:
                    file.write(str(myPuzzle.cells[row][col].number) + " ")
                else:
                    file.write(str(myPuzzle.cells[row][col].number))
        return None
    else:
        print("Sudoku puzzle has no solution, no output file created")
        return None


main()
```