Manxueying Li (ml5719)

How to run my program

Things to do manually:

In main() function, set the input and out put file by using

```
infile = "SUDUKO_Input1.txt" #input file
outf = open("output1.txt", "w")#output file
```

Output 1:

```
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 2:

```
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 3:

```
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

digits = cols = "123456789" #dight in each cell are numbers/ col number of each cell is also
represent by numbers (increasing top down)
rows = "ABCDEFGHI" #using letters to represent row from left to right

#representing every cell
def cross(A, B):
        return [a + b for a in A for b in B] #using (letter dight)represent each cell, eg. the
upper left is A1

#getting the representation of the whole board
squares = cross(rows, cols)

class csp:

        #initializing csp
        def __init__ (self, domain = digits, grid = ""):
                self.variables = squares #representing every cell
                self.domain = self.getDict(grid) #domain of each cell
                self.values = self.getDict(grid) #possible values of each cell

        #27 lists of peers
                self.unitlist = ([cross(rows, c) for c in cols] +
                                 [cross(r, cols) for r in rows] +
                                 [cross(rs, cs) for rs in ('ABC','DEF','GHI') for cs in
('123','456','789')])
        #dictionary of the cells and the corresponding lists of peers
                self.units = dict((s, [u for u in self.unitlist if s in u]) for s in squares)
        #dictionary of the all cells(81 cells on the board) and the corresponding set of 20 peers
(8+8+4=20)
        #define peers as the row and col and the block it envolved
                self.peers = dict((s, set(sum(self.units[s],[]))- set([s])) for s in squares)

    #getting input str and return the dic
        def getDict(self, grid=""):
                i = 0
                values = dict()
                for cell in self.variables:
                        if grid[i]!='0': #if not blank
                                values[cell] = grid[i] #set the cell to the number given
                        else: #if blank
                                values[cell] = digits #all possible digit of the cell
                        i = i + 1
                return values


#implementing backtrack search algo her:
def Backtracking_Search(assignment,csp):
    return Backtrack(assignment, csp) #call backtrack function with the initiallized assignment


#recursive function using backtrack
def Backtrack(assignment, csp):
    if isComplete(assignment):
        return assignment #if done, return the assignment

    var = Select_Unassigned_Variables(assignment, csp) #getting the cell with smallest domain
    domain = deepcopy(csp.values) #deep copy domain

    for value in csp.values[var]: #iterate through the curr domain
        if isConsistent(var, value, assignment, csp): #if the new assignment is consistant
            assignment[var] = value #try to assign this number to the cell
            inferences = {}
            inferences = Inference(assignment, inferences, csp, var, value)
            if inferences != "FAILURE": # forward checking is failure is reached
                result = Backtrack(assignment, csp) #continue backtrack search
                if result != "FAILURE": # if cannot find assignment, return failure
                    return result
```

```
            del assignment[var]
            csp.values.update(domain)

    return "FAILURE" #if cannt find assignment, return failure


# using forward checking to detect earily failures
def Inference(assignment, inferences, csp, var, value):
    inferences[var] = value
    # print(value,inferences)
    for neighbor in csp.peers[var]:
        if neighbor not in assignment and value in csp.values[neighbor]:
            if len(csp.values[neighbor]) == 1:
                return "FAILURE"

            remaining = csp.values[neighbor].replace(value, "") #reduing the domain

            if len(remaining) == 1: #if domain is reduced to 1
                flag = Inference(assignment, inferences, csp, neighbor, remaining) #inference
again to check failure
                if flag == "FAILURE":
                    return "FAILURE"
    return inferences


# is the assignment compelte?
def isComplete(assignment):
    return set(assignment.keys()) == set(squares) #if all cells are being assigned


# choose the next varible to assign
def Select_Unassigned_Variables(assignment, csp):
    #make a dic of all unassigned vars and find the one with min domain to be choosen next
    unassigned_variables = dict(
        (squares, len(csp.values[squares])) for squares in csp.values if squares not in
assignment.keys())
    mrv = min(unassigned_variables, key=unassigned_variables.get) # getting the var with min
domian
    return mrv


# check if the assignment is consistant
def isConsistent(var, value, assignment, csp):
    for neighbor in csp.peers[var]:
        if neighbor in assignment.keys() and assignment[neighbor] == value: #if the numebr is
already taken by its neighbor
            return False
    return True


# forward checking
def forward_check(csp, assignment):
    domain = deepcopy(csp.values) #deep copy domain
    for key,val in domain.items():
        if len(val) == 1: #if it is already being assigned with number
            inferences = {}
            Inference(assignment, inferences, csp, key, val)#recursivly check if its neighbor's
domain can be reduced

# return the solved suduko as a str
def write(values):
    output = ""
    ind = 1
    for variable in squares:
        output += values[variable]+" "
        if (ind % 9 == 0): # formating the string, 9 values in a line
            output += "\n"
        ind += 1
    # print(output)
    return output
```

```python
def main():
    str = '' #a single str of digits of all cells
    infile = "SUDUKO_Input3.txt" #input file
    print("input file is: ",infile)
    with open(infile, "r") as inf:
        for line in inf:
            line = line.rstrip() #cut the new line char
            for i in line:
                if(i != " "): #dismiss space
                    str += i

    # print(str)
    inf.close()

    outf = open("output3.txt", "w")#output file
    print("Solved Suduko: ")

    sudoku = csp(grid=str) #create csp representing sudoku
    assignment = {}
    forward_check(sudoku,assignment)#forward checking
    solved  = Backtracking_Search(assignment,sudoku)#using backtracking search solve sudoku
    if solved!="FAILURE":#if can be solved
        outf.write(write(solved)+"\n")
        print(write(solved))
    outf.close()

main()
```