Project 2: Hyper Sudoku

Helen Xu and Tatyana Graesser

Part A: Compilation instructions

Any Python compiler (idle etc.) should work -- just make sure test files are in the same directory as the code

After running the code, input the name of the Input text file, and then after the problem is solved input the name of the Output file.

Part B: Hyper Sudoku as a constraint satisfaction problem

Variables: Each cell in a 27x27 matrix (rows and columns are numbered from 0 to 8)
Domain for variables: {1 2 3 4 5 6 7 8 9}
Constraints:

- Each value in a row must be different
- Each value in a column must be different
- Each value in a designated 3x3 square (+ hyper squares) must be different

Part C: Code and Output Files

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

```
"""
CS 4613 - AI Project 2
Authors: Tatyana Graesser (tg1625), Helen Xu (hjx201)
"""
import random
from copy import deepcopy

#globals
width = 9
height = 9

domains = [[[] for j in range(width)] for i in range(height)] #create data structure for domains. it is a 3d array also this was the only way to do it w/o pythons weird pointer system being weird
assignment = [[0 for j in range(width)] for i in range(height)] #2d matrix for assignments. A 0 is unassigned, any other # is assigned

'''
Functions for translating txt file to matrix, and translating matrix back to txt files
- createStates()
- printOutput()
'''
def createStates(filepath): #create 2D matrices for initial state and goal state from file
  state = [[]] #initial state matrix
  with open(filepath, 'r') as fp:
    char = '' #keeping track of number
    row = 0 #keeping track of matrix row
    for c in fp.read():
      if c.isnumeric():
        char += c
      elif c == ' ':
        state[row].append(int(char))
        char = ''
      elif c == '\n':
        if (char != ''): #add any found numbers
          state[row].append(int(char))
          char = ''
```

```
        row += 1 #move to a new row
        state.append([ ])
    if char != '':
      state[row].append(int(char)) #getting last character we may have missed
  return state[:height]

def printOutput(initial): #print final output to file
    printed = False
    while not printed:
      filepath = input("Enter output file name: ")
      try:
        with open(filepath, "w") as fp:
          #printing out initial state
          for row in initial:
            for char in row:
              fp.write(str(char) + " ")
            fp.write("\n")
          fp.write("\n")
      except:
        print("An error occurred, please try again")
      else:
        print("Output printed to", filepath)
        printed = True


'''
Functions for setting up the CSP with forward checking
- initCSP()
- forwardCheck()
- getConstrainingNeighbors()
- removeValue()
'''

# methods used for forward checking
def initCSP(root): #initialize domain of problem using forward checking
  assignment = deepcopy(root)
  for rowi in range(height):
    for colj in range(width):
      domains[rowi][colj] = [1, 2, 3, 4, 5, 6, 7, 8, 9].copy()
```

```
  for rowi in range(height):
    for colj in range(width):
      if (root[rowi][colj] != 0):
        domains[rowi][colj] = [ int(root[rowi][colj])].copy()
        # assignment[rowi][colj] = int(root[rowi][colj])
        forwardCheck(rowi, colj)
        #print(domains)
  return domains, assignment


def forwardCheck(row, col):
  num = domains[row][col][0]
  neighbors = getConstrainingNeighbors(row, col)

  for cell in neighbors:
    removeValue(cell[0], cell[1], num)


#gets list of neighbors for a cell
def getConstrainingNeighbors(row, col):
  neighbors = []

  for rowi in range(height): # same row
    if(rowi != row and (rowi, col) not in neighbors):
      neighbors.append((rowi, col))

  for coli in range(width): #same column
    if(coli != col and (row, coli) not in neighbors):
      neighbors.append((row, coli))

  for rowi in range(row//3*3, row//3*3+3):  #same square
    for coli in range(col//3*3, col//3*3+3):
      if((coli != col or rowi != row) and (rowi, coli) not in neighbors):
        neighbors.append((rowi, coli))

  if(row%4 != 0 and col%4 != 0):  #same hyper square
    leftbound = (row-row//3)//3*4 + 1 # weird maths to get upper left corner of
square
```

```
    upbound = (col-col//3)//3*4 + 1

    for rowi in range(leftbound, leftbound+3):
      for coli in range(upbound, upbound+3):
        if((coli != col or rowi != row) and (rowi, coli) not in neighbors):
          neighbors.append((rowi, coli))
  # print("For (", row, "," , col, ") Neighbors are", neighbors)
  return neighbors


#remove value from domain, forward checking ver.
def removeValue(row, col, num):
    #print("removing value in (" + str(row) + ", " + str(col) + ")")
    if (num in domains[row][col]):
      domains[row][col].remove(num)

    if (len(domains[row][col]) == 0): #check for empty domain
      print("Empty domain at (" + str(row) + ", " + str(col) + "). Puzzle has no
solution")
      exit(0)

'''
Class for doing all the backtracking
- findSolution() === backtrack() in the pseudocode
'''
#all of backtracking is done in here cuz python globals are stoopid
class Backtracking:
  def __init__(self, domain, assignment):
    self.domain = domain
    self.assignment = assignment
    self.isSolved = False

  def findSolution(self):
    #check if assignment complete
    if self.isComplete():
      return self.assignment

    currVar = self.selectUnassignedVariable() #get next unassigned variable
    # print(currVar[0], ",", currVar[1], end="\r", flush=True)
```

```
    for val in self.domain[currVar[0]][currVar[1]]:#for each value in that
domain; don't need to sort domain values as they are already sorted
      if(self.isConsistent(currVar[0], currVar[1], val)): #check current domain
value for consistency w/ assignment
        self.assignment[currVar[0]][currVar[1]] = val #if consistent, assign
variable
        self.findSolution() #result = backtrack(csp, assignment)
        if self.isComplete():
          return self.assignment
        self.assignment[currVar[0]][currVar[1]] = 0

    # print("Backtracking at", currVar[0], ",", currVar[1])
    return self.assignment


  def isConsistent(self, row, col, num): #function to check for consistency
    neighbors = getConstrainingNeighbors(row, col)
    for rowi, colj in neighbors:
        if(self.assignment[rowi][colj] == num):
          return False
    return True


  def isComplete(self):   #check for completed assignment
    if(self.isSolved):
      return True
    for rowi in range(height):
      for coli in range(width):
        if(self.assignment[rowi][coli] == 0): #if anything is unassigned, not
complete
          return False
    self.isSolved = True #else is complete
    return True


  def selectUnassignedVariable(self):#use minimum remaining value and then degree
to find next cell to try
    unassigned = self.getUnassignedVars() #first get a list of the unassigned
variables
```

```
    MRVs = {} #minimum remaining vals of all unassinged vars
    mrv = 100
    for row, col in unassigned:
      rv = len(self.domain[row][col])
      if rv in MRVs:
        if rv < mrv: #don't bother adding var if not in current mininum list
          MRVs[rv].append((row, col))
      else:
        MRVs[rv] = [(row, col)]

      mrv =  min(MRVs.keys()) #get MRV

    if(len(MRVs[mrv]) == 1):  #return tuple coords of next var
      return MRVs[mrv][0]
    else: # if there is a tie, calculate degree
      unassigned2 = MRVs[mrv]
      degrees = {}
      maxdeg = -1
      for row, col in unassigned2:
        deg = self.getDegree(row, col)
        if deg in degrees:
          if maxdeg > deg: #don't bother adding var if not in current max list
            degrees[deg].append((row, col))
        else:
          degrees[deg] = [(row, col)]
        maxdeg = max(degrees.keys()) #get highest degree
      return degrees[maxdeg][0] #if there is yet another tie, just pick the first
one idk


  def getUnassignedVars(self): #returns list of unassigned variables (for MRV
heuristic)
    unassigned = []
    for rowi in range(height):
      for colj in range(width):
        if(self.assignment[rowi][colj] == 0):
          unassigned.append((rowi, colj))
    return unassigned
```

```
  def getDegree(self, row, col): #returns # of unassigned neighbors (for degree heuristic)
    degree = 0
    neighbors = getConstrainingNeighbors(row, col)
    for (r, c) in neighbors:
      if(self.assignment[r][c] == 0): #if neighbor unassigned, increase degree
        degree += 1
    return degree


def main():
  root = None

  #user input for filenames
  filepath = ""
  while True:
    filepath = input("Enter filepath for input file. Enter EXIT to end code: ")
    if filepath == "EXIT":
      print("Goodbye :)")
      exit(0)
    try:
      root = createStates(filepath) #read in initial and goal states from starting node
    except:
      print("Incorrect filepath")
    else:
      domains, assignment = initCSP(root)

      csp = Backtracking(domains, assignment)
      csp.findSolution() #gives potential solution (runs backtracking)

      #check if solution is like....actually good or not. If it's not it means its a failure
      if not csp.isSolved:
        print("No solution :(")
      else:
        print("Solution:", csp.assignment)
```

```
        printOutput(csp.assignment)  #final solution held in the assignment
attribute



if __name__ == "__main__":
  main()
```