JunHao Chen
Artificial Intelligence Project - Cryptarithmetic Problem
December 7, 2019

Extra Credit - I've implemented inference function with forward checking

Instruction for Running Program

The driver() function takes in an input file to generate an output file that contains the solution for the cryptarithmetic problem. The driver() function will automatically generate an output file that contains the same name as the input file. For example, Input1.txt will generate Output1.txt. The program will automatically assume that Input1.txt already exists in the file directory and will output solutions for Input1.txt To test new input files, simply change the parameter of the driver function to the name of the desired input file. For example, to test input4.txt, change function to driver("input4.txt"). The program currently only supports 1 input file per compile due to the use of global variables.

Output1.txt

```
9567
1085
10652
```

Output2.txt

```
7483
7455
14938
```

Source Code

```
import copy

# mapping relationships between letters to variables
lettersRelation = {}
# mapping variables to letters
varRelation = {}

# dict of neighbors constraints for each var
neighbors = {}


def setup(filename):
    # function to parse through input file

    # maintain how the input is structured to simplify output
    lines = []
    # # possible domains of each variables
    varDomains = {}
    # mapping relationships between letters to variables
    # lettersRelation = {}
    # # mapping variables to letters
    # varRelation = {}
    # dict of value assignments
    assignments = {}
    assignments['C1'] = None
    assignments['C2'] = None
    assignments['C3'] = None
    assignments['C4'] = 1



    # set up initial constraint neighbors
    neighbors ['X1'] = ['X5', 'X10', 'C4', 'C3' ]
    neighbors ['X2'] = ['X6', 'X11', 'C3', 'C2']
    neighbors ['X3'] = ['X7', 'X12', 'C2', 'C1']
    neighbors ['X4'] = ['X8', 'X13', 'C1' ]

    neighbors ['X5'] = ['X1', 'X10', 'C4', 'C3' ]
    neighbors ['X6'] = ['X2', 'X11', 'C3', 'C2' ]
    neighbors ['X7'] = ['X3', 'X12', 'C2', 'C1']
    neighbors ['X8'] = ['X4', 'X13', 'C1' ]

    neighbors['X10'] = ['X1', 'X5', 'C4', 'C3' ]
    neighbors['X11'] = ['X2', 'X6', 'C3', 'C2']
    neighbors['X12'] = ['X3', 'X7', 'C2', 'C1']
    neighbors['X13'] = ['X4', 'X8', 'C1' ]

    neighbors['C3'] = ['X2', 'X11', 'X6', 'X1', 'X5', 'X10']
    neighbors['C2'] = ['X3', 'X12', 'X7', 'X2', 'X11', 'X6']
    neighbors['C1'] = ['X4', 'X13', 'X8', 'X3', 'X12', 'X7']


    varDomains['C1'] = [0,1]
    varDomains['C2'] = [0,1]
    varDomains['C3'] = [0,1]
    varDomains['C4'] = 1

    count = 1

    with open(filename) as file:
        for line in file:
            chars= []
            line = line.strip()
            for i in line:
                chars.append(i)

                var = 'X' + str(count)

                # domain reduction by analysis
                # X9 = 1
                if var == 'X9':
                    varDomains[var] = [1]
                # X1 and X5 cant be 0
```

```
                elif var == 'X1' or var =='X5':
                    varDomains[var] = [1, 2, 3, 4, 5, 6, 7, 8, 9]
                else:
                    varDomains[var] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
                assignments[var] =  None

                if i not in lettersRelation:
                    lettersRelation[i] = [var]

                else:
                    lettersRelation[i].append(var)

                varRelation[var] = i
                count+=1

            lines.append(chars)

    # # since X9 is 1, we can assign 1 to X9

    varDomains['X9'] = 1
    assignments['X9'] = 1

    return lines, lettersRelation, varDomains, assignments, varRelation

def checkComplete(assignments):
    # print(assignments)
    for var in assignments:
        if assignments[var] is None:
            # return false if there's an unassigned
            return False
    return True

def checkNeighbors(assignments, possible):
    # keep count of unassigned neighbors for each variable
    count = {}

    for var in possible:
        listsOfNeigh =neighbors[var]
        localCount = 0
        for neighVar in listsOfNeigh:
            if assignments[neighVar] is None:
                localCount += 1
        count[var] = localCount

    # find the var with the most unassigned neighbors
    max = -1
    select = None
    for var in count:
        if count[var] > max:
            select = var
            max = count[var]

    return select

def selectVar(assignments,varDomains):
    min = 100
    # possible variables to choose
    possible = []

    # choose least remaining value
    for var in varDomains:
        if assignments[var] is None:
            # checks if there's two var that has the same minimum remaining values
            if type(varDomains[var]) == list and len(varDomains[var]) == min:
                possible.append(var)
            # finds a new min, reset possible vars
            elif type(varDomains[var]) == list and len(varDomains[var]) < min:
                possible = []
                possible.append(var)
                min = len(varDomains[var])

    # print("Possible: {}".format(possible))

    if len(possible) == 1 :
        return possible [0]

    else:
        #Degree heuristic: choose var with the most unassigned neighbors
        return checkNeighbors(assignments,possible)

def letterDupsCheck(assignments):
    letterAssignments = {}

    for var in assignments:
        if var[0] != 'C' and assignments[var] is not None:
            letter = varRelation[var]
            # check each variable and their corresponding letter to see if there's a duplicate
            if assignments[var] in letterAssignments:
                if letter != letterAssignments[ assignments[var] ]:
                    return False
            else:
                letterAssignments[ assignments[var] ] = letter
    return True


def checkConsistent(assignments):
    # make sure no duplicate values for unique letters
    if letterDupsCheck(assignments) == False :
        return False

    # check mathematical constraints
    #
    try:
        if int(assignments['C4']) != int(assignments['X9']):
            return False
    except:
        pass

    # X4 + X8 = X13
    try:
        if (int(assignments['X4']) + int(assignments['X8'])) %10 != int(assignments['X13']):
            return False
    except:
        pass

    # (X4 + X8 ) // 10 = C1
    try:
        if (int(assignments['X4']) + int(assignments['X8'])) // 10  != int(assignments['C1'])  :
            return False
    except:
        pass


    # X3 + X7 + C1 = X12
    try:
        if (int(assignments['X3']) + int(assignments['X7'])) %10 + int(assignments['C1']) !=
int(assignments['X12'])  :
            return False
    except:
        pass

    # (X3 + X7 ) // 10 = C2
    try:
        if (int(assignments['X3']) + int(assignments['X7'])) // 10  != int(assignments['C2'])  :
            return False
    except:
        pass


    # X2 + X6 + C2 = X11
    try:
        if (int(assignments['X2']) + int(assignments['X6'])) % 10 + int(assignments['C2']) !=
int(assignments['X11']):
            return False
    except:
        pass

    # (X2 + X6 ) // 10 = C3
    try:
        if (int(assignments['X2']) + int(assignments['X6'])) // 10  != int(assignments['C3'])  :
            return False
    except:
        pass

    # X1 + X5 + C3 = X10
    try:
        if (int(assignments['X1']) + int(assignments['X5'])) %10 + int(assignments['C3']) !=
int(assignments['X10']) :
            return False
    except:
        pass

    # (X1 + X5 ) // 10 = C4
    try:
        if (int(assignments['X1']) + int(assignments['X5'])) // 10  != int(assignments['C4'])  :
            return False
    except:
        pass

    # vars representing same letter must equal to each other
    for letter in lettersRelation:
        valCheck = None
        count = 0
        vars = lettersRelation[letter]
        for i in range(len(vars)):
            if assignments[ vars[i] ] is not None:
                if count == 0 :
                    valCheck = assignments[ vars[i] ]
                    count+=1
                else:
                    if assignments[ vars[i] ] != valCheck:
                        return False


    return True


def forwardChecking(var, assignments, domains):
    # print("IN FC")
    # print(assignments)
    # print()
    # print(domains)
    neighborsList = (neighbors[var])
    val = assignments[var]
    if var[0] != 'C':
    # loop through all the neighbors
        for i in neighborsList:
            if i[0] != 'C':
                letter = varRelation[i]

                # if they represent the same letter, simply reduce domain to just that one val
                if  varRelation[var] == letter :
                    domains[i] = [val]

                # else start reducing domain based on constraints
                else:
                    selectedDomain = domains[i]
                    # since all var has to be unique, remove domain val cooresponding to
assignment
                    try:
                        selectedDomain.remove(val)
                    except:
                        pass


                    # check if val is consistent with our constraints. If not, remove val

                    #
                    for domainVal in copy.deepcopy(selectedDomain):
                        tempAssignments = copy.deepcopy(assignments)
                        tempAssignments[i] = domainVal

                        # if an domain value is inconsistent with constraint, remove it
                        if  checkConsistent(tempAssignments) == False:
                            selectedDomain.remove(domainVal)

                    # if domain is 0, return false
                    if len(selectedDomain) == 0:
                        return False
    return True




def backtrackingSearch(assignments, varDomains):
    if checkComplete(assignments):
        return assignments

    var = selectVar(assignments,varDomains)
    # print("Selected Var:" + var )
    # print(varDomains)
    # print()
    # print(assignments)
    for val in varDomains[var]:
        # print("VAR: " + str(var) + " VAL:" + str(val))
        # print("Assigning {} to {}".format(val, var))
        tempAssignments = copy.deepcopy(assignments)
        tempAssignments[var] = val
        if checkConsistent(tempAssignments):
            tempDomains = copy.deepcopy(varDomains)
            # extra credit with forward checking
            inferences = forwardChecking(var, tempAssignments, tempDomains)
            if inferences == True:
                result = backtrackingSearch(tempAssignments, tempDomains)
                if result is not None:
                    return result

    return None



def convertOutput(assignments, varRelation, lines, file):
    letterAssignments = {}

    for var in assignments:
        if var[0] != 'C':
            letter = varRelation[var]
            if letter not in letterAssignments:
                letterAssignments[letter] = assignments[var]

    print(letterAssignments)

    for line in lines:
        for char in line:
            print(letterAssignments[char],end='',file=file)
        print("", file=file)

def driver(filename):
    lines, lettersRelation, varDomains, assignments, varRelation = setup(filename)
    result = backtrackingSearch(assignments,varDomains)

    output = filename.split(".")
    # output file name = input file name count
    outputFile = "Output" + output[0][-1] + ".txt"
    file = open(outputFile, "w")
    convertOutput(result, varRelation, lines, file)
    file.close()

    # test = {}
    # test ['C1'] = 1
    # test ['C2'] = 1
    # test ['C3'] = 0
    # test ['C4'] = 1
    # test ['X1'] = 9
    # test ['X2'] = 5
    # test ['X3'] = 6
    # test ['X4'] = 7
    # test ['X5'] = 1
    # test ['X6'] = 0
    # test ['X7'] = 8
    # test ['X8'] = 5
    # test ['X9'] = 1
    # test ['X10'] = 0
    # test ['X11'] = 6
    # test ['X12'] = 5
    # test ['X13'] = 2
    # print(checkConsistent(test))

driver("Input1.txt")
```