Python 3.7.4

**How To Run:**

1. Create folder with python file (backtrack.py) and input file as a txt inside
2. Open python file and rename the string assigned to variable txt in line 14 to the name of the input file.
3. Run the python file.
4. An output file will be created in the folder and the python shell will display the output as well.

**Code:**   
import copy

'''

Daniel Tse

Professor E.K. Wong

CS 4613

Backtracking Algorithm for CSP's

Solving: Cryptarithmetic problem

function BACKTRACKING-SEARCH(csp) returns solution or failure

return BACKTRACK({ }, CSP )

function BACKTRACK(assignment, csp) returns a solution or failure

if assignment is complete then return assignment

var  SELECT-UNASSIGNED-VARIABLE(csp)

for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do

if value is consistent with assignment then

add {var = value} to assignment

inferences  INFERENCE(csp, var, value)

if inferences != failure then

add inferences to assignment

result BACKTRACK(assignment, csp)

if result != failure then

return result

remove {var=value} and inferences from assignment

return failure

'''

#assign txt to file name of input file

txt = "Input1.txt"

file = open("Output" + txt[5] + ".txt", "w")

#list containing original input

inp = []

#first character

first = ''

#dictionary with values of letters

letter = {}

def backtrack(assignment, domain):

if is\_complete(assignment):

return assignment

#var is a letter with possible domain

var = select\_unassigned\_variable(domain)

#parse through domain of letter to assign value

for potential in var[1]:

#check if consistent

if is\_consistent(assignment, var[0], potential):

local\_domain = copy.deepcopy(domain)

for i in local\_domain:

if i[0] == var[0]:

i[1].remove(potential)

assignment[var[0]] = potential

result = backtrack(assignment, local\_domain)

if result is not False:

return result

#inferences

assignment[var[0]] = -1

return False

def select\_unassigned\_variable(domain):

low = ('temp', [0] \* 11)

for let in domain:

if letter[let[0]] == -1:

if len(let[1]) < len(low[1]):

low = let

#list with most constrained variable, contains multiple if tie

potential = []

for let in domain:

if len(low[1]) == len(let[1]):

potential.append(let)

flipped = []

for word in inp:

flipped.append(word[::-1])

#find most constraining variable

num\_unassigned = [0] \* len(potential)

for ch in range(len(potential)):

for word in flipped:

for i in range (len(word)):

if i > 3:

break

if word[i] == potential[ch][0]:

#check if letters have values

if letter[inp[0][i]] == -1:

num\_unassigned[ch] += 1

if letter[inp[1][i]] == -1:

num\_unassigned[ch] += 1

if letter[inp[2][i]] == -1:

num\_unassigned[ch] += 1

var = potential[num\_unassigned.index(max(num\_unassigned))]

return var

def is\_complete(assignment):

for ch in assignment:

if assignment[ch] == -1:

return False

return True

#checks constraints

def is\_consistent(assignment, let, val):

assignment[let] = val

#check that no other letter shares value

for ch in letter:

if letter[ch] == val and ch != let:

if let[0] == 'x' or ch[0] == 'x':

pass

else:

return False

#prepare variables for calculation

flipped = []

for word in inp:

flipped.append(word[::-1])

#perform a check for auxilary variables

if let[0] == 'x':

#check x1 - only need to check ones and tens place

if let[1] == '1':

if empty\_place(flipped, 0) or empty\_place(flipped, 1):

pass

else:

if (not is\_valid\_addition(assignment, 0, flipped) or not is\_valid\_addition(assignment, 1, flipped)):

return False

#check x2 -

elif let[1] == '2':

if empty\_place(flipped, 1) or empty\_place(flipped, 2):

pass

else:

if (not is\_valid\_addition(assignment, 1, flipped) or not is\_valid\_addition(assignment, 2, flipped)):

return False

#check x3 - need to check thousands place

elif let[1] == '3':

if empty\_place(flipped, 2) or empty\_place(flipped, 3): #letter[flipped[0][3]] == -1 or letter[flipped[1][3]] == -1 or letter[flipped[2][3]] == -1:

pass

else:

if (not is\_valid\_addition(assignment, 2, flipped)) or (not is\_valid\_addition(assignment, 3, flipped)):

return False

#perform a check for standard letters

for word in flipped:

for i in range (len(word)):

if i > 3:

break

if word[i] == let:

#check if letters in same column dont add up

if empty\_place(flipped, i): #letter[flipped[0][i]] == -1 or letter[flipped[1][i]] == -1 or letter[flipped[2][i]] == -1:

pass

else:

if not is\_valid\_addition(assignment, i, flipped):

return False

return True

#returns true if the place value has any empty values or false if all values are filled

def empty\_place(flipped, place):

if letter[flipped[0][place]] == -1 or letter[flipped[1][place]] == -1 or letter[flipped[2][place]] == -1:

return True

return False

def is\_valid\_addition(assignment, place, flipped):

#checking ones place

if place == 0:

if assignment['x1'] == -1:

if (assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] or

assignment[flipped[2][place]] == assignment[flipped[0][place]] + assignment[flipped[1][place]]):

return True

else:

if assignment[flipped[2][place]] + (assignment['x1'] \* 10) == assignment[flipped[0][place]] + assignment[flipped[1][place]]:

return True

#checking thousands place

elif place == 3:

if assignment['x3'] == -1:

if (assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] + 1

or assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] + 0):

return True

else:

if assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] + assignment['x3']:

return True

#checking tens and hundreds place

else:

aux = 'x' + str(place)

aux2 = 'x' + str(place + 1)

if assignment[aux] == -1 and assignment[aux2] == -1:

if (assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] + 1 or

assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] or

assignment[flipped[2][place]] == assignment[flipped[0][place]] + assignment[flipped[1][place]] + 1 or

assignment[flipped[2][place]] == assignment[flipped[0][place]] + assignment[flipped[1][place]]):

return True

elif assignment[aux] != -1 and assignment[aux2] == -1:

if (assignment[flipped[2][place]] + 10 == assignment[flipped[0][place]] + assignment[flipped[1][place]] + assignment[aux] or

assignment[flipped[2][place]] == assignment[flipped[0][place]] + assignment[flipped[1][place]] + assignment[aux]):

return True

elif assignment[aux] == -1 and assignment[aux2] != -1:

if (assignment[flipped[2][place]] + (assignment[aux2] \* 10) == assignment[flipped[0][place]] + assignment[flipped[1][place]] + 1 or

assignment[flipped[2][place]] + (assignment[aux2] \* 10) == assignment[flipped[0][place]] + assignment[flipped[1][place]]):

return True

else: #if assignment[aux] == -1 and assignment[aux2] == -1:

if assignment[flipped[2][place]] + (assignment[aux2] \* 10) == assignment[flipped[0][place]] + assignment[flipped[1][place]] + assignment[aux]:

return True

return False

#create output file given the final values

def create\_output\_file(values, input):

for word in input:

for ch in word:

file.write(str(values[ch]))

file.write("\n")

def main():

#domain will be a nested list that contains the domain of each letter

domain = []

#open file and append input to list

with open(txt,'r') as f:

for line in f:

for word in line.split():

inp.append(word)

#append individual letters to dictionary with initial value -1

for word in inp:

for ch in word:

letter.update({ch: -1})

#follow constraint and add auxilary variables

letter[inp[2][0]] = 1

#create nested list to contain variables and associated domains and constraints

for ch in letter:

if ch == inp[2][0]:

pass

elif ch == inp[0][0] or ch == inp[1][0]:

domain.append((ch, [1,2,3,4,5,6,7,8,9]))

first = ch

else:

domain.append((ch, [0,1,2,3,4,5,6,7,8,9]))

letter.update({'x1': -1})

letter.update({'x2': -1})

letter.update({'x3': -1})

domain.append(('x1', [0,1]))

domain.append(('x2', [0,1]))

domain.append(('x3', [0,1]))

values = backtrack(letter, domain)

#create output file

create\_output\_file(values, inp)

file.close()

main()

**Output 1**

9567

1085

10652

**Output 2**

7483

7455

14938