**WATER JUG PROBLEM**

j1 = 0

j2 = 0

x = 4

y = 3

print("Initial state = (0,0)")

print("Capacities = (4,3)")

print("Goal state = (2,0)")

while j1 != 2:

r = int(input("Enter rule : "))

if(r == 1):

j1 = x

elif(r == 2):

j2 = y

elif(r == 3):

j1 = 0

elif(r == 4):

j2 = 0

elif(r == 5):

t = y - j2

j2 = y

j1 -= t

if j1<0:

j1=0

elif(r == 6):

t = x - j1

j1 = x

j2 -= t

if j2<0:

j2=0

elif(r == 7):

j2 += j1

j1 = 0

if j2>y:

j2=y

elif(r == 8):

j1 += j2

j2 = 0

if j1>x:

j1=x

print (j1, j2)

**8-PUZZLE PROBLEM**

from queue import Queue

import copy

import time

def printNode(node):

print(node[0],node[1],node[2])

print(node[3],node[4],node[5])

print(node[6],node[7],node[8])

global nodeNumber

print('Node:', nodeNumber)

print('Depth:', len(node[9:]))

print('Moves:', node[9:])

print('------')

nodeNumber += 1

def checkFinal(node):

if node[:9]==finalNode:

printNode(node)

return True

if node[:9] not in visitedList:

printNode(node)

queue.put(node)

visitedList.append(node[:9])

return False

if \_\_name\_\_ == '\_\_main\_\_':

startNode = [1,5,4, 3,7,2, 6,8,0]

finalNode = [0,1,2, 3,4,5, 6,7,8]

found = False

nodeNumber = 0

visitedList = []

queue = Queue()

queue.put(startNode)

visitedList.append(startNode)

printNode(startNode)

t0 = time.time()

while (not found and not queue.empty()):

currentNode = queue.get()

blankIndex = currentNode.index(0)

if blankIndex!=0 and blankIndex!=1 and blankIndex!=2:

upNode = copy.deepcopy(currentNode)

upNode[blankIndex] = upNode[blankIndex-3]

upNode[blankIndex-3] = 0

upNode.append('up')

found = checkFinal(upNode)

if blankIndex!=0 and blankIndex!=3 and blankIndex!=6 and found==False:

leftNode = copy.deepcopy(currentNode)

leftNode[blankIndex] = leftNode[blankIndex-1]

leftNode[blankIndex-1] = 0

leftNode.append('left')

found = checkFinal(leftNode)

if blankIndex!=6 and blankIndex!=7 and blankIndex!=8 and found==False:

downNode = copy.deepcopy(currentNode)

downNode[blankIndex] = downNode[blankIndex+3]

downNode[blankIndex+3] = 0

downNode.append('down')

found = checkFinal(downNode)

if blankIndex!=2 and blankIndex!=5 and blankIndex!=8 and found==False:

rightNode = copy.deepcopy(currentNode)

rightNode[blankIndex] = rightNode[blankIndex+1]

rightNode[blankIndex+1] = 0

rightNode.append('right')

found = checkFinal(rightNode)

t1 = time.time()

print('Time:', t1-t0)

print('------')

**QUEENS PROBLEM**

import random

"""function one: the parameter is the current checkerboard layout state,

and the queen logarithm of the current eight queens layout conflicts is judged according to the layout."""

def get\_numof\_conflict(status):

num = 0

for i in range(len(status)):

for j in range(i + 1,len(status)):

if status[i] == status[j]:

num += 1

offset = j - i

if abs(status[i]-status[j]) == offset:

num += 1

return num

"""function two: the parameter is the current checkerboard layout state,

using the hill climbing method to select the best layout of the neighbor state and return"""

def hill\_climbing(status):

convert = {}

length = len(status)

for col in range(length):

best\_move = status[col]

for row in range(length):

if status[col] == row:

continue

status\_copy = list(status)

status\_copy[col] = row

convert[(col,row)] = get\_numof\_conflict(status\_copy)

answers = [] #

conflict\_now = get\_numof\_conflict(status) #current queen conflict logarithm

# Store all possible successor dictionaries to find the best successor

for key,value in convert.items():

if value < conflict\_now:

conflict\_now = value

for key,value in convert.items():

if value == conflict\_now:

answers.append(key)

#If the best successor set element is greater than one randomly select one

if len(answers) > 0:

x = random.randint(0,len(answers)-1)

col = answers[x][0]

row = answers[x][1]

status[col] = row

return status

"""function three: find a solution that the eight queens satisfy the conflict number of 0,

and loop out the subsequent set of each step until there is no rush > sudden"""

def Queens():

status = [0,1,2,3,4,5,6,7] # initial state, all queens are on the diagonal

"""When the number of conflicts is greater than 0,

the loop solves the best successor until the eight queens solution is found."""

while get\_numof\_conflict(status) > 0:

status = hill\_climbing(status)

print("\n",status)

print(get\_numof\_conflict(status))

print("\n","the answer is")

print(status)

if \_\_name\_\_ == '\_\_main\_\_':

Queens()

**CHAT BOT PROBLEM**

pairs = [

[ "My name is (.\*)", ["Hello %1, How are you today ?"] ],

[ "(.\*) help", ["Yes, I can help you",] ],

[ "(.\*) your name ?", ["My name is Robo. I'm a chatbot.",] ],

[ "how are you (.\*) ?", ["i am great!"] ],

[ "sorry (.\*)", ["Its OK, never mind that",] ],

[ "(hi|hey|hello|hola|holla)(.\*)", ["Hello", "Hey there","Hi" ] ],

[ "(.\*) not feeling (.\*)", ["Ok. Tell me your problem",] ],

[ "(.\*)created(.\*)", ["Mmvc created me using Python's NLTK library ","top secret ;)",] ],

[ "(.\*)cough(.\*)", ["Take Ginger tea and take cough syrup",] ],

[ "(.\*)(headache|fever)(.\*)", ["please take paracetamol tablet after food",] ],

[ "(.\*)stomach(.\*)", ["please take lightweight food",] ],

[ "i am (good|ok)", ["Ok good. Happy to hear from you",] ],

[ "quit", ["Bye for now. See you soon. It was nice talking to you.. :)"] ],

]

from nltk.chat.util import Chat, reflections

print("Hi, I am Robo and I am a chatbot \nPlease type lowercase English language to start a conversation. \n Type quit to leave ")

#Create Chat Bot

chat = Chat(pairs, reflections)

#Start conversation

chat.converse()

**CRYPT ARTHIMETIC PROBLEM**

import itertools

import re

def solve(formula):

"""Given a formula like 'SEND + MORE = MONEY', fill in digits to solve it.

Generate all valid digit-filled-in strings."""

return filter(valid, letter\_replacements(formula))

def letter\_replacements(formula):

"""All possible replacements of letters with digits in formula."""

formula = formula.replace(' = ', ' == ') # Allow = or ==

letters = cat(set(re.findall('[A-Z]', formula)))

for digits in itertools.permutations('1234567890', len(letters)):

yield formula.translate(str.maketrans(letters, cat(digits)))

def valid(exp):

"""Expression is valid iff it has no leading zero, and evaluates to true."""

try:

return not leading\_zero(exp) and eval(exp) is True

except ArithmeticError:

return False

cat = ''.join # Function to concatenate strings

leading\_zero = re.compile(r'\b0[0-9]').search # Function to check for illegal number

print(next(solve('SEND + MORE = MONEY')))

**A\* STAR PROBLEM**

import heapq

class priorityQueue:

def \_\_init\_\_(self):

self.cities = []

def push(self, city, cost):

heapq.heappush(self.cities, (cost, city))

def pop(self):

return heapq.heappop(self.cities)[1]

def isEmpty(self):

if (self.cities == []):

return True

else:

return False

def check(self):

print(self.cities)

class ctNode:

def \_\_init\_\_(self, city, distance):

self.city = str(city)

self.distance = str(distance)

romania = {}

def makedict():

file = open("romania.txt", 'r')

for string in file:

line = string.split(',')

ct1 = line[0]

ct2 = line[1]

dist = int(line[2])

romania.setdefault(ct1, []).append(ctNode(ct2, dist))

romania.setdefault(ct2, []).append(ctNode(ct1, dist))

def makehuristikdict():

h = {}

with open("romania\_sld.txt", 'r') as file:

for line in file:

line = line.strip().split(",")

node = line[0].strip()

sld = int(line[1].strip())

h[node] = sld

return h

def heuristic(node, values):

return values[node]

def astar(start, end):

path = {}

distance = {}

q = priorityQueue()

h = makehuristikdict()

q.push(start, 0)

distance[start] = 0

path[start] = None

expandedList = []

while (q.isEmpty() == False):

current = q.pop()

expandedList.append(current)

if (current == end):

break

for new in romania[current]:

g\_cost = distance[current] + int(new.distance)

#print(new.city, new.distance, "now : " + str(distance[current]), g\_cost)

if (new.city not in distance or g\_cost < distance[new.city]):

distance[new.city] = g\_cost

f\_cost = g\_cost + heuristic(new.city, h)

#print(f\_cost)

q.push(new.city, f\_cost)

path[new.city] = current

printoutput(start, end, path, distance, expandedList)

def printoutput(start, end, path, distance, expandedlist):

finalpath = []

i = end

while (path.get(i) != None):

finalpath.append(i)

i = path[i]

finalpath.append(start)

finalpath.reverse()

print("A-star Agorithm for Romania Map")

print("\tArad => Bucharest")

print("=======================================================")

print("List of Cities that are Expanded : " + str(expandedlist))

print("Total Number of Cities that are Expanded : " + str(len(expandedlist)))

print("=======================================================")

print("Cities in Final path : " + str(finalpath))

print("Total Number of cities in final path are : " + str(len(finalpath)))

print("Total Cost : " + str(distance[end]))

def main():

src = "Arad"

dst = "Bucharest"

makedict()

astar(src, dst)

if \_\_name\_\_ == "\_\_main\_\_" :

main()

**Romania.txt**

Arad,Zerind, 75

Arad,Sibiu, 140

Arad,Timisoara, 118

Zerind,Oradea, 71

Oradea,Sibiu, 151

Timisoara,Lugoj, 111

Sibiu,Fagaras, 99

Sibiu,Rimnicu Vilcea, 80

Lugoj,Mehadia, 70

Fagaras,Bucharest, 211

Rimnicu Vilcea,Pitesti, 97

Rimnicu Vilcea,Craiova, 146

Mehadia,Dobreta, 75

Bucharest,Pitesti, 101

Bucharest,Urziceni, 85

Bucharest,Giurglu, 90

Pitesti,Craiova, 138

Craiova,Dobreta, 120

Urziceni,Hirsova, 98

Urziceni,Vaslui, 142

Hirsova,Eforie, 86

Vaslui,Lasi, 92

Lasi,Neamt, 87

**Romania.sld**

Arad, 366

Bucharest, 0

Craiova, 160

Dobreta, 242

Eforie, 161

Fagaras, 176

Giurgiu, 77

Hirsowa, 151

Lasi, 226

Lugoj, 244

Mehadia, 241

Neamt, 234

Oradea, 380

Pitesti, 100

Rimnicu Vilcea, 193

Sibiu, 253

Timisoara, 329

Urziceni, 80

Vaslui, 199

Zerind, 374