Bfs

from collections import deque

def bfs(start,goal):

visited=set()

queue=deque([[start]])

if start==goal:

return [start]

while queue:

path=queue.popleft()

node=path[-1]

for neighbor in g[node]:

if neighbor in visited:

continue

new\_path=list(path)

new\_path.append(neighbor)

queue.append(new\_path)

if neighbor==goal:

return new\_path

visited.add(node)

return None

g={

1:[2,3,4],

2:[1,4,5],

3:[1,4],

4:[1,2,3,7],

5:[2,6,7],

6:[5,7],

7:[4,5,6]

}

result\_path = bfs( 1, 6)

# Print the result

if result\_path:

print(result\_path)

else:

print(f"No path found from {initial\_node} to {goal\_node}")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DFS

graph={

1:[4],

2:[3,4],

3:[4,2],

4:[1,3]

}

def dfs(start,goal,path=None):

if path==None:

path=[]

path=path+[start]

if start not in graph:#check the start in graph or not first time

return None

if start==goal:# goal is found

return path

for neighbor in graph[start]:

if neighbor not in path:

new\_path=dfs(neighbor,goal,path)

if new\_path: # its child find solution then return that one

return new\_path

return None#it return None if element does not have any neighbor

start\_node=1

goal\_node=3

paths=dfs(start\_node,goal\_node)

if paths:

print(paths)

else:

print("NO")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Remove punctuation

punctuations = '''!()-[]{};:'"\,<>./?@#$%^&\*\_~'''

my\_str = input("Enter a string: ")

no\_punct = ""

for char in my\_str:

if char not in punctuations:

no\_punct = no\_punct + char

print(no\_punct)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#tower of honai

def honai(n,A,B,C):

if n==1:

print("Move disk 1 from A to C")

return

honai(n-1,A,B,C)

print("Move disk ",n," from ",A," to ",C)

honai(n-1,B,C,A)

n=3

honai(n,'a','c','b')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#water juge problem

j1cap=4

j2cap=3

target=2

def water\_jug():

j1,j2=0,0

path=[]

path.append((j1,j2))

while j2!=target:

if j1==j1cap:

j1=0

path.append((j1,j2))

if j2==0:

j2=j2cap

path.append((j1,j2))

while j1!=j1cap and j2!=0:

j1+=1

j2-=1

path.append((j1,j2))

return path

sol\_path=water\_jug()

for j1,j2 in sol\_path:

print("Jug1: ",j1," Gallon ,Jug2: ",j2, "Gallon")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//N queen puzzle

print("Enter")

N=int(input())

board=[[0]\*N for \_ in range(N)]

def is\_attack(i,j):

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queen(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(is\_attack(i,j))) and (board[i][j]!=1):

board[i][j]=1

if N\_queen(n-1)==True:

return True

board[i][j]=0

return False

N\_queen(N)

for i in board:

print(i)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Chat bot

responses\_dict = {

"hello": "Hello! How can I help you?",

"how are you": "I'm doing well, thank you for asking.",

"what is your name": "I'm just a chatbot.",

"bye": "Goodbye! Have a great day!",

"default": "I'm sorry, I don't understand that. ask me about name, class ,college CGPA",

"name":"your name is Maroti Bhise",

"class": "your class is Fy MSC CS",

"college": "your college name is Sarhad",

"cgpa":"Your last year cgpa is 8",

}

def get\_response(user\_input):

input\_lower = user\_input.lower()

return responses\_dict.get(input\_lower, responses\_dict["default"])

def stat\_bot():

print("Welcome to the College Info Bot!")

while True:

user\_input = input("You: ")

response = get\_response(user\_input)

print("Chatbot:", response)

if user\_input.lower() == "bye":

break

stat\_bot()

Write Python program to implement crypt arithmetic problem TWO+TWO=FOUR

from itertools import permutations

# Function to check the equation

def solve\_cryptarithmetic():

for perm in permutations(range(10), 8): # We have 8 unique letters (S, E, N, D, M, O, R, Y)

# Assign letters to digits

S, E, N, D, M, O, R, Y = perm

SEND = 1000\*S + 100\*E + 10\*N + D

MORE = 1000\*M + 100\*O + 10\*R + E

MONEY = 10000\*M + 1000\*O + 100\*N + 10\*E + Y

# Check if the equation SEND + MORE = MONEY holds

if SEND + MORE == MONEY:

print(f"SEND = {SEND}, MORE = {MORE}, MONEY = {MONEY}")

return

# Call the function

solve\_cryptarithmetic()

#hill climbing algorithm

import random

def func(x):

return -x\*\*2+5\*x

def hill\_climbing(start,step,max\_it):#stating point step\_size maximum iteration

for \_ in range(max\_it):

neighbors=[start+step,start-step]

next\_x=max(neighbors,key=func)

if func(next\_x)<=func(start):

break

start=next\_x

return start,func(start)

start\_p=random.uniform(-10,10)

opt\_x,opt\_value=hill\_climbing(start\_p,2,1000)

print("Optimal x= ",opt\_x," Optimal solution =",opt\_value)

Hangman

Code

import random

words=["python","gamming","hangman"]

def hangman():

word=random.choice(words)

guessed="-"\*len(word)

attempts=6

while attempts>0:

print("Word ",guessed)

guess=input("Guess latter ")

if len(guess)!=1:

print("Invalid guess Try again")

continue

if guess in word:

print("Good guess")

guessed=''.join(guess if word[i]==guess else guessed[i] for i in range(len(word)))

else:

attempts-=1

print("Wrong guess !",attempts," attempts left")

if '-' not in guessed:

print("congratulation The word is ",word)

return

else:

print("Out of attempts The word is ",word)

hangman()

Monkey and Banana Problem

# Monkey and Banana Problem

# Initial positions and status

monkey\_position = 0

banana\_position = 5

has\_banana = False

# Loop until the monkey gets the banana

while not has\_banana:

print(f"Monkey is at position {monkey\_position}")

# Move the monkey closer to the banana

if monkey\_position < banana\_position:

monkey\_position += 1

# Check if the monkey has reached the banana

if monkey\_position == banana\_position:

has\_banana = True

print("Monkey has reached the banana!")

print("Monkey got the banana!")

**Remove stop words from sentence**

from nltk.tokenize import sent\_tokenize, word\_tokenize

from nltk.corpus import stopwords

import nltk

nltk.download('stopwords')

nltk.download('punkt')

data = text = open("a.txt").read().lower()

stopWords = set(stopwords.words('english'))

words = word\_tokenize(data)

wordsFiltered = [w for w in words if w not in stopWords]

print(wordsFiltered)

**Lemmatization**

import nltk

from nltk.stem import WordNetLemmatizer

wordnet\_lemmatizer = WordNetLemmatizer()

text = "studies studying cries cry "

nltk.download('punkt')

nltk.download('wordnet')

tokenization = nltk.word\_tokenize(text)

for w in tokenization:

print("Lemma for {} is {}".format(w, wordnet\_lemmatizer.lemmatize(w)))

#calender

#calander

import calendar

year=int(input("enter a year"))

month=int(input("enter a month"))

print(calendar.month(year,month))

A\* algorithm

import heapq

# Graph with nodes and edge costs

graph = {

'A': {'B': 9, 'C': 4,'D':7},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1, 'G': 3},

'E': {},

'F': {},

'G': {}

}

# Heuristic estimates for each node

heuristics = {'A': 7, 'B': 6, 'C': 2, 'D': 1, 'G': 0}

def a\_star(start, goal):

open\_list = [(heuristics[start], 0, start)] # Priority queue with f = g + h

g\_costs = {start: 0} # Store g(n) costs

parents = {start: None}

while open\_list:

f, g, current\_node = heapq.heappop(open\_list)

if current\_node == goal:

path = []

while current\_node:

path.append(current\_node)

current\_node = parents[current\_node]

return path[::-1] # Return reversed path

for neighbor, cost in graph[current\_node].items():

new\_g = g\_costs[current\_node] + cost

if neighbor not in g\_costs or new\_g < g\_costs[neighbor]:

g\_costs[neighbor] = new\_g

parents[neighbor] = current\_node

f = new\_g + heuristics[neighbor]

heapq.heappush(open\_list, (f, new\_g, neighbor))

return None

# Example usage

path = a\_star('A', 'G')

print("Path:", path if path else "No path found")