# Python3 Program to print BFS traversal

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue: # Creating loop to visit each node

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search")

bfs(visited, graph, '5') # function calling

dfs

# Using a Python dictionary to act as an adjacency list

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

from collections import deque

class Graph:

# example of adjacency list (or rather map)

# adjacency\_list = {

# 'A': [('B', 1), ('C', 3), ('D', 7)],

# 'B': [('D', 5)],

# 'C': [('D', 12)]

# }

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

# heuristic function with equal values for all nodes

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

# open\_list is a list of nodes which have been visited, but who's neighbors

# haven't all been inspected, starts off with the start node

# closed\_list is a list of nodes which have been visited

# and who's neighbors have been inspected

open\_list = set([start\_node])

closed\_list = set([])

# g contains current distances from start\_node to all other nodes

# the default value (if it's not found in the map) is +infinity

g = {}

g[start\_node] = 0

# parents contains an adjacency map of all nodes

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

# find a node with the lowest value of f() - evaluation function

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

# for all neighbors of the current node do

for (m, weight) in self.get\_neighbors(n):

# if the current node isn't in both open\_list and closed\_list

# add it to open\_list and note n as it's parent

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

# otherwise, check if it's quicker to first visit n, then m

# and if it is, update parent data and g data

# and if the node was in the closed\_list, move it to open\_list

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')

# prims algorithm

INF = 9999999

# number of vertices in graph

N = 5

#creating graph by adjacency matrix method

G = [[0, 19, 5, 0, 0],

[19, 0, 5, 9, 2],

[5, 5, 0, 1, 6],

[0, 9, 1, 0, 1],

[0, 2, 6, 1, 0]]

selected\_node = [0, 0, 0, 0, 0]

no\_edge = 0

selected\_node[0] = True

# printing for edge and weight

print("Edge : Weight\n")

m=0

while (no\_edge < N - 1):

minimum = INF

a = 0

b = 0

for n in range(N):

if ((not selected\_node[n]) and G[m][n]):

if minimum > G[m][n]:

minimum = G[m][n]

a = m

b = n

# m=b

print(str(a) + "-" + str(b) + ":" + str(G[a][b]))

selected\_node[b] = True

print(selected\_node)

no\_edge += 1

# nqueen

global N

N = 8

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j], end=' ')

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

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

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

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

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0]

]

if solveNQUtil(board, 0) == False:

print("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()

chatbot

import random

response = {

'hi': ["hello!", "hi there!", "hi!"],

'how are you': ["i am good ,thanks", "i am doing well,thank you", "i am fine,thanks"],

"what's your name": ["i am chatbot", 'my name is chatbot', "myself chatbot", "i am just a chatbot,i dont have a name"],

'bye': ['goodbye', 'bye', 'see you there!'],

'thank you': ['you are welcome', 'no problem', 'anytime'],

'default': ["i am sorry,i dont understant", 'can you please rephrase that?', "i am not sure what you mean"]

}

def chatbot():

print(random.choice(response['hi']))

while True:

message = input(">")

if "hi" in message.lower():

print(random.choice(response['hi']))

elif "how are you" in message.lower():

print(random.choice(response['how are you']))

elif "bye" in message.lower():

print(random.choice(response["bye"]))

elif "thank you" in message.lower():

print(random.choice(response['thank you']))

else:

print(random.choice(response['default']))

chatbot()