**TASK-1A VTU24978**

**CODE:**

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:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

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

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

**OUTPUT:**

A screen shot of a computer

AI-generated content may be incorrect.

**TASK-1B**

**CODE:**

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)

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

dfs(visited, graph, '5')

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**TASK-2 VTU24978**

**CODE:**

from sys import maxsize

from itertools import permutations

V = 4

def travellingSalesmanProblem(graph, s):

vertex = []

for i in range(V):

if i != s:

vertex.append(i)

min\_path = maxsize

next\_permutation = permutations(vertex)

for i in next\_permutation:

current\_pathweight = 0

k = s

for j in i: # Fixed capitalization of 'for'

current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][s]

min\_path = min(min\_path, current\_pathweight)

return min\_path # Changed capitalization of 'return'

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

graph = [[0, 10, 15, 20], [10, 0, 35, 25],

[15, 35, 0, 30], [20, 25, 30, 0]]

s = 0

print(travellingSalesmanProblem(graph, s))

**OUTPUT:**

**A blue square with white text

AI-generated content may be incorrect.**

**TASK-3A VTU24978**

**CODE:**

def aStarAlgo(start\_node, stop\_node):

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

closed\_set = set()

g = {} # store distance from starting node

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or n is None or Graph\_nodes[n] is None:

break

else:

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

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

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

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

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

open\_set.remove(n)

closed\_set.add(n)

if n is None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found:', path)

return path

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

h\_dist = {

'A': 11,

'B': 6,

'C': 5,

'D': 7,

'E': 3,

'F': 6,

'G': 5,

'H': 3,

'I': 1,

'J': 0

}

return h\_dist[n]

Graph\_nodes = {

'A': [('B', 6), ('F', 3)],

'B': [('A', 6), ('C', 3), ('D', 2)],

'C': [('B', 3), ('D', 1), ('E', 5)],

'D': [('B', 2), ('C', 1), ('E', 8)],

'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

'F': [('A', 3), ('G', 1), ('H', 7)],

'G': [('F', 1), ('I', 3)],

'H': [('F', 7), ('I', 2)],

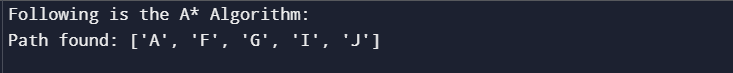
'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

print("Following is the A\* Algorithm:")

aStarAlgo('A', 'J')

**OUTPUT:**



**TASK-3B VTU24978**

**CODE:**

def aStarAlgo(start\_node, stop\_node):

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

closed\_set = set()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or n is None or n not in Graph\_nodes:

break

else:

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

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

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

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

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

open\_set.remove(n)

closed\_set.add(n)

if n is None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found:', path)

return path

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

h\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0

}

return h\_dist[n]

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('A', 2), ('C', 1), ('G', 9)],

'C': [('B', 1)],

'D': [('E', 6), ('G', 1)],

'E': [('A', 3), ('D', 6)],

'G': [('B', 9), ('D', 1)]

}

print("Following is the A\* Algorithm:")

aStarAlgo('A', 'G')

**OUTPUT:**

**A blue background with white text

AI-generated content may be incorrect.**

**TASK-4 VTU24978**

**CODE:**

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = MAX

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

break

return best

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

values = [3, 5, 6, 9, 1, 2, 0, -1]

print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))

**OUTPUT:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**TASK-5 VTU24978**

**CODE**:

import numpy as np

from numpy import inf

d = np.array([[0,10,12,11,14]

,[10,0,13,15,8]

,[12,13,0,9,14]

,[11,15,9,0,16]

,[14,8,14,16,0]])

iteration = 100

n\_ants = 5

n\_citys = 5

m = n\_ants

n = n\_citys

e = .5

alpha = 1

beta = 2

visibility = 1/d

visibility[visibility == inf ] = 0

pheromne = .1\*np.ones((m,n))

rute = np.ones((m,n+1)

for ite in range(iteration)

rute[:,0] = 1

for i in range(m)

temp\_visibility = np.array(visibility)

for j in range(n-1)

combine\_feature = np.zeros(5)

cum\_prob = np.zeros(5)

cur\_loc = int(rute[i,j]-1)

temp\_visibility[:,cur\_loc] = 0

p\_feature = np.power(pheromne[cur\_loc,:],beta)

v\_feature = np.power(temp\_visibility[cur\_loc,:],alpha)

p\_feature = p\_feature[:,np.newaxis]

v\_feature = v\_feature[:,np.newaxis]

combine\_feature = np.multiply(p\_feature,v\_feature)

total = np.sum(combine\_feature)

probs = combine\_feature/total

cum\_prob = np.cumsum(probs)

r = np.random.random\_sample()

city = np.nonzero(cum\_prob>r)[0][0]+1

rute[i,j+1] = city

left = list(set([i for i in range(1,n+1)])-set(rute[i,:-2]))[0]

rute[i,-2] = left

rute\_opt = np.array(rute)

dist\_cost = np.zeros((m,1))

for i in range(m)

s = 0

for j in range(n-1)

s = s + d[int(rute\_opt[i,j])-1,int(rute\_opt[i,j+1])-1]

dist\_cost[i]=s

dist\_min\_loc = np.argmin(dist\_cost)

dist\_min\_cost = dist\_cost[dist\_min\_loc]

best\_route = rute[dist\_min\_loc,:]

pheromne = (1-e)\*pheromne

for i in range(m):

for j in range(n-1):

dt = 1/dist\_cost[i]

pheromne[int(rute\_opt[i,j])-1,int(rute\_opt[i,j+1])-1] = pheromne[int(rute\_opt[i,j])-1,int(rute\_opt[i,j+1])-1] + dt

peromne

print('route of all the ants at the end :')

print(rute\_opt)

print()

print('best path :',best\_route)

print('cost of the best path',int(dist\_min\_cost[0]) + d[int(best\_route[-2])-1,0])

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**TASK-6 VTU24978**

**CODE:**

class Graph:

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

self.v = vertices

self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

def is\_safe(self, v, color, c):

for i in range(self.v):

if self.graph[v][i] == 1 and color[i] == c:

return False

return True

def graph\_color\_util(self, m, color, v):

if v == self.v:

return True

for c in range(1, m+1):

if self.is\_safe(v, color, c):

color[v] = c

if self.graph\_color\_util(m, color, v+1):

return True

color[v] = 0

def graph\_coloring(self, m):

color = [0] \* self.v

if not self.graph\_color\_util(m, color, 0):

return False

print("Solution exists and following are the assigned colors:")

for c in color:

print(c, end=" ")

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

g = Graph(4)

g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]

m = 3

g.graph\_coloring(m)

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**TASK-7 VTU24978**

**CODE:**

def move(subject, x1, x2):

return f"Move {subject} from {x1} to {x2}"

def push\_box(x1, x2):

return f"Push box from {x1} to {x2}"

def climb\_box(x, direction):

return f"Climb box at {x} {direction}"

def have\_banana(x):

return f"Have banana at {x}"

initial\_state = {

'monkeyAt0': True,

'monkeyLevel': 'Down',

'bananaAt1': True,

'boxAt2': True

}

goal\_state = {

'GetBanana': True,

'at': 1

}

def plan\_actions(initial\_state, goal\_state):

actions = []

if initial\_state['monkeyAt0'] and initial\_state['bananaAt1']:

actions.append(move('Monkey', 0, 1))

actions.append(climb\_box(1, 'Up'))

actions.append(have\_banana(1))

return actions

actions = plan\_actions(initial\_state, goal\_state)

print("Plan:")

for action in actions:

print(action)

**OUTPUT:**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**TASK-8 VTU24978**

**CODE:**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

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

print("Q",end=" ")

else:

print(".",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):

# Place this queen in 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]]

if solveNQUtil(board, 0) == False:

print("Solution does not exist")

return False

printSolution(board)

return True

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

solveNQ()

**OUTPUT:**

A blue rectangular object with white text

AI-generated content may be incorrect.

**TASK-9A VTU24978**

**CODE:**

Pip install openai

import openai

openai.api\_key = "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"

completion = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=[{"role": "user", "content": "Give me 3 ideas that i could build using openai apis"}])

print(completion.choices[0].message.content)

**OUTPUT:**

**1. Personalized Content Recommendation System:** Develop an AI-powered content recommendation system that suggests personalized content to users based on their interests and search history. Use OpenAI's language generation APIs to generate relevant content descriptions and summaries, and employ their natural language processing (NLP) APIs to understand user preferences and interests.

**2. Intelligent Chatbot:** Build a conversational AI-enabled chatbot that can answer customer queries, provide helpful recommendations, and complete transactions seamlessly. Use OpenAI's language processing APIs to train the chatbot to understand user inputs and respond in natural language. Integration with other APIs such as payment gateways and customer databases can make the chatbot efficient and effective.

**3. Fraud Detection System:** Develop a machine learning model that can identify and prevent fraudulent activities using OpenAI's anomaly detection and classification APIs. Train the model using historical data of fraudulent transactions, and use the APIs to continuously scan for and identify suspicious activities. Such a system can be deployed in a range of applications such as finance or e-commerce platforms.

**TASK-9B**

**CODE:**

import openai

openai.api\_key = "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"

messages = []

system\_msg = input("What type of chatbot would you like to create?\n")

messages.append({"role": "system", "content": system\_msg})

print("Your new assistant is ready! Type your query")

while input != "quit()":

message = input()

messages.append({"role": "user", "content": message})

response = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=messages)

reply = response["choices"][0]["message"]["content"]

messages.append({"role": "assistant", "content": reply})

print("\n" + reply + "\n")

**OUTPUT:**

What type of chatbot would you like to create?

Nila’s personal chatbot

(ctrl enter)

Your new assistant is ready!

**TASK-9C**

**CODE:**

import openai

import gradio

openai.api\_key = "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"

messages = [{"role": "system", "content": "You are a financial experts that specializes in real estate investment and negotiation"}]

def CustomChatGPT(user\_input):

messages.append({"role": "user", "content": user\_input})

response = openai.ChatCompletion.create(

model = "gpt-3.5-turbo",

messages = messages

)

ChatGPT\_reply = response["choices"][0]["message"]["content"]

messages.append({"role": "assistant", "content": ChatGPT\_reply})

return ChatGPT\_reply

demo = gradio.Interface(fn=CustomChatGPT, inputs = "text", outputs = "text", title = "INTELLIGENT CHATBOT")

demo.launch(share=True)

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**TASK-10 VTU24978**

**CODE:**

facts = [

"john\_is\_cold.",

"raining.",

"john\_Forgot\_His\_Raincoat.",

"fred\_lost\_his\_car\_keys.",

"peter\_footballer."

]

def verify\_fact(fact):

# Remove the trailing period

fact = fact.rstrip(".")

if fact == "john\_Forgot\_His\_Raincoat":

return True

elif fact == "raining":

return True

elif fact == "foggy":

return True

elif fact == "Cloudy":

return False # Assume it's not cloudy

else:

return False

for fact in facts:

if verify\_fact(fact):

print(f"{fact} - Yes")

else:

print(f"{fact} - No")

**OUTPUT:**

**A blue background with white text

AI-generated content may be incorrect.**