1. **Implement depth first search algorithm and Breadth First Search algorithm. Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure**.

from collections import defaultdict, deque

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def add\_edge(self, u, v):

        self.graph[u].append(v)

        self.graph[v].append(u)  # Since it's an undirected graph

    def dfs\_recursive(self, vertex, visited=None):

        if visited is None:

            visited = set()

        visited.add(vertex)

        print(vertex, end=' ')

        for neighbor in self.graph[vertex]:

            if neighbor not in visited:

                self.dfs\_recursive(neighbor, visited)

    def bfs(self, start):

        visited = set()

        queue = deque([start])

        visited.add(start)

        while queue:

            vertex = queue.popleft()

            print(vertex, end=' ')  # <-- Fixed this line

            for neighbor in self.graph[vertex]:

                if neighbor not in visited:

                    queue.append(neighbor)

                    visited.add(neighbor)

g = Graph()

g.add\_edge(0, 1)

g.add\_edge(0, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 3)

g.add\_edge(3, 5)

g.add\_edge(4, 5)

g.add\_edge(3, 6)

print("Depth First Search (starting from vertex 0):")

g.dfs\_recursive(0)

print("\nBreadth First Search (starting from vertex 0):")

g.bfs(0)

1. **Implement A star (A\*) Algorithm for any game search problem**

import heapq

class AStar:

    def \_\_init\_\_(self, grid, start, goal):

        self.grid = grid  # 2D grid where 0 = walkable, 1 = blocked

        self.start = start  # Start position (x, y)

        self.goal = goal  # Goal position (x, y)

        self.rows = len(grid)

        self.cols = len(grid[0])

    def heuristic(self, node):

        return abs(node[0] - self.goal[0]) + abs(node[1] - self.goal[1])

    def neighbors(self, node):

        dirs = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Directions: right, down, left, up

        result = []

        for d in dirs:

            neighbor = (node[0] + d[0], node[1] + d[1])

            if 0 <= neighbor[0] < self.rows and 0 <= neighbor[1] < self.cols and self.grid[neighbor[0]][neighbor[1]] == 0:

                result.append(neighbor)

        return result

    def a\_star\_search(self):

        open\_list = []

        heapq.heappush(open\_list, (0, self.start))

        came\_from = {}

        g\_score = {self.start: 0}

        f\_score = {self.start: self.heuristic(self.start)}

        while open\_list:

            current = heapq.heappop(open\_list)[1]

            if current == self.goal:

                return self.reconstruct\_path(came\_from, current)

            for neighbor in self.neighbors(current):

                tentative\_g\_score = g\_score[current] + 1

                if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:

                    came\_from[neighbor] = current

                    g\_score[neighbor] = tentative\_g\_score

                    f\_score[neighbor] = tentative\_g\_score + self.heuristic(neighbor)

                    heapq.heappush(open\_list, (f\_score[neighbor], neighbor))

        return []

    def reconstruct\_path(self, came\_from, current):

        total\_path = [current]

        while current in came\_from:

            current = came\_from[current]

            total\_path.append(current)

        return total\_path[::-1]

grid = [

    [0, 1, 0, 0, 0],

    [0, 1, 0, 1, 0],

    [0, 0, 0, 1, 0],

    [1, 1, 0, 0, 0],

    [0, 0, 0, 1, 0]

]

start = (1, 0)

goal = (3, 4)

a\_star = AStar(grid, start, goal)

path = a\_star.a\_star\_search()

print("Path from start to goal:", path)

**3.) Implement Greedy search for selection sort**

def selection\_sort(arr):

    n = len(arr)

    for i in range(n):

        min\_index = i

        for j in range(i+1, n):

            if arr[j] < arr[min\_index]:

                min\_index = j

        arr[i], arr[min\_index] = arr[min\_index], arr[i]

        print(f"Step {i+1}: {arr}")  # Print the array after each swap

    return arr

arr = [95,10,85,90,48,50,1]

print("Original array:", arr)

sorted\_arr = selection\_sort(arr)

print("Sorted array:", sorted\_arr)

**4.) Implement solution for a constraint satisfaction problem. N queens**

def print\_board(board):

    for row in board:

        print(" ".join("Q" if col else "." for col in row))

    print("\n")

def is\_safe(board, row, col, n):

    for i in range(row):

        if board[i][col]:

            return False

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

        if board[i][j]:

            return False

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

        if board[i][j]:

            return False

    return True

def solve\_n\_queens\_backtracking(board, row, n):

    if row == n:

        print\_board(board)

        return True

    found\_solution = False

    for col in range(n):

        if is\_safe(board, row, col, n):

            board[row][col] = True  # Place the queen

            found\_solution = solve\_n\_queens\_backtracking(board, row + 1, n) or found\_solution

            board[row][col] = False  # Backtrack

    return found\_solution

def n\_queens\_backtracking(n):

    board = [[False] \* n for \_ in range(n)]

    if not solve\_n\_queens\_backtracking(board, 0, n):

        print("No solution exists.")

def is\_safe\_branch\_and\_bound(row, col, cols, diags1, diags2, n):

    return not (cols[col] or diags1[row + col] or diags2[row - col + (n - 1)])

def solve\_n\_queens\_branch\_and\_bound(row, n, cols, diags1, diags2, board):

    if row == n:

        print\_board(board)

        return True

    found\_solution = False

    for col in range(n):

        if is\_safe\_branch\_and\_bound(row, col, cols, diags1, diags2, n):

            board[row][col] = True

            cols[col] = True

            diags1[row + col] = True

            diags2[row - col + (n - 1)] = True

            found\_solution = solve\_n\_queens\_branch\_and\_bound(row + 1, n, cols, diags1, diags2, board) or found\_solution

            board[row][col] = False

            cols[col] = False

            diags1[row + col] = False

            diags2[row - col + (n - 1)] = False

    return found\_solution

def n\_queens\_branch\_and\_bound(n):

    board = [[False] \* n for \_ in range(n)]

    cols = [False] \* n

    diags1 = [False] \* (2 \* n - 1)

    diags2 = [False] \* (2 \* n - 1)

    if not solve\_n\_queens\_branch\_and\_bound(0, n, cols, diags1, diags2, board):

        print("No solution exists.")

n = 8

print("Solutions using Backtracking:")

n\_queens\_backtracking(n)

print("Solutions using Branch-and-Bound:")

n\_queens\_branch\_and\_bound(n)

**5.) Chat bot**

import nltk

from nltk.chat.util import Chat, reflections

import tkinter as tk

from tkinter import scrolledtext

# Define chatbot responses using pairs

pairs = [

    [r"hi|hello|hey", ["Hello! How can I assist you today?", "Hi there! How can I help you?"]],

    [r"how are you?", ["I'm just a bot, but I'm doing fine! How about you?", "I'm a chatbot, I don't have feelings, but thanks for asking!"]],

    [r"(.\*)your name?", ["I'm a chatbot, here to assist you."]],

    [r"bye|goodbye", ["Goodbye! Have a great day!", "Bye! Take care!"]],

    [r"(.\*)", ["I'm not sure how to respond to that. Could you rephrase?"]]

]

# Create chatbot

chatbot = Chat(pairs, reflections)

# Function to send message

def send\_message():

    user\_input = user\_entry.get()

    chat\_history.insert(tk.END, f"You: {user\_input}\n")

    response = chatbot.respond(user\_input)

    chat\_history.insert(tk.END, f"Bot: {response}\n\n")

    user\_entry.delete(0, tk.END)

# GUI setup

root = tk.Tk()

root.title("Simple Chatbot")

chat\_history = scrolledtext.ScrolledText(root, wrap=tk.WORD, width=50, height=15)

chat\_history.pack(padx=10, pady=10)

user\_entry = tk.Entry(root, width=40)

user\_entry.pack(padx=10, pady=5)

send\_button = tk.Button(root, text="Send", command=send\_message)

send\_button.pack(pady=5)

root.mainloop()