**Bfs and dfs**

**Task 1:**

Apply BFS and DFS on trees and graphs.

**For trees:**

from collections import deque

class TreeNode:

    def \_\_init\_\_(self, val):

        self.val = val

        self.children = []

def bfs\_tree(root):

    """Breadth First Search (BFS) for a tree."""

    if not root:

        return

    # Create a queue for BFS

    queue = deque([root])

    while queue:

        # Dequeue the node from the queue

        node = queue.popleft()

        # Visit the current node

        print(node.val)

        # Enqueue children of the current node

        for child in node.children:

            queue.append(child)

def dfs\_tree(root):

    """Depth First Search (DFS) for a tree."""

    if not root:

        return

    # Create a stack for DFS

    stack = [root]

    while stack:

        # Pop the node from the stack

        node = stack.pop()

        # Visit the current node

        print(node.val)

        # Push children of the current node onto the stack

        stack.extend(reversed(node.children))

# Example tree:

#       1

#     / | \

#    2  3  4

#   / \

#  5   6

# /

# 7

# Construct the tree

root = TreeNode(1)

root.children = [TreeNode(2), TreeNode(3), TreeNode(4)]

root.children[0].children = [TreeNode(5), TreeNode(6)]

root.children[0].children[0].children = [TreeNode(7)]

# Perform BFS and DFS traversals

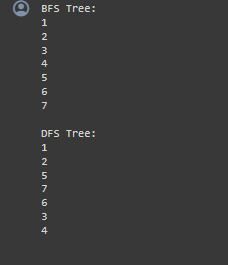
print("BFS Tree:")

bfs\_tree(root)

print("\nDFS Tree:")

dfs\_tree(root)

**OUTPUT:**



**For graph:**

from collections import deque

class Graph:

    def \_\_init\_\_(self):

        self.adj\_list = {}

    def add\_edge(self, u, v):

        """Add an edge between nodes u and v."""

        # Add nodes to the adjacency list if not already present

        if u not in self.adj\_list:

            self.adj\_list[u] = []

        if v not in self.adj\_list:

            self.adj\_list[v] = []

        # Undirected graph, so add edges in both directions

        self.adj\_list[u].append(v)

        self.adj\_list[v].append(u)

def bfs\_graph(graph, start):

    """Breadth First Search (BFS) for a graph."""

    visited = set()

    queue = deque([start])

    while queue:

        # Dequeue the vertex from the queue

        vertex = queue.popleft()

        if vertex not in visited:

            # Visit the current vertex

            print(vertex)

            visited.add(vertex)

            # Enqueue neighbors of the current vertex

            for neighbor in graph.adj\_list[vertex]:

                if neighbor not in visited:

                    queue.append(neighbor)

def dfs\_graph(graph, start):

    """Depth First Search (DFS) for a graph."""

    visited = set()

    stack = [start]

    while stack:

        # Pop the vertex from the stack

        vertex = stack.pop()

        if vertex not in visited:

            # Visit the current vertex

            print(vertex)

            visited.add(vertex)

            # Push unvisited neighbors onto the stack

            for neighbor in graph.adj\_list[vertex]:

                if neighbor not in visited:

                    stack.append(neighbor)

# Example graph:

# 1 -- 2 -- 3

# |    |    |

# 4 -- 5 -- 6

# Construct the graph

graph = Graph()

graph.add\_edge(1, 2)

graph.add\_edge(2, 3)

graph.add\_edge(1, 4)

graph.add\_edge(2, 5)

graph.add\_edge(3, 6)

graph.add\_edge(4, 5)

# Perform BFS and DFS traversals

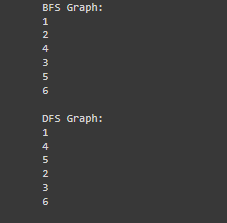
print("BFS Graph:")

bfs\_graph(graph, 1)

print("\nDFS Graph:")

dfs\_graph(graph, 1)

**OUTPUT:**



**2) tic tac toe game :**

**Write a Program to Implement Tic-Tac-Toe game using Python.**

def print\_board(board):

    """

    Prints the Tic-Tac-Toe board.

    """

    for row in board:

        print(" | ".join(row))

        print("-" \* 5)

def check\_winner(board, player):

    """

    Checks if the given player has won the game.

    """

    # Check rows

    for row in board:

        if all(cell == player for cell in row):

            return True

    # Check columns

    for col in range(3):

        if all(board[row][col] == player for row in range(3)):

            return True

    # Check diagonals

    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):

        return True

    return False

def is\_board\_full(board):

    """

    Checks if the board is full (no empty cells).

    """

    for row in board:

        for cell in row:

            if cell == " ":

                return False

    return True

def main():

    """

    Main function to run the Tic-Tac-Toe game.

    """

    board = [[" " for \_ in range(3)] for \_ in range(3)]

    players = ["X", "O"]

    turn = 0

    while True:

        print\_board(board)

        player = players[turn % 2]

        print(f"Player {player}'s turn")

        row = int(input("Enter row (0, 1, 2): "))

        col = int(input("Enter column (0, 1, 2): "))

        if board[row][col] == " ":

            board[row][col] = player

            if check\_winner(board, player):

                print\_board(board)

                print(f"Player {player} wins!")

                break

            if is\_board\_full(board):

                print\_board(board)

                print("It's a tie!")

                break

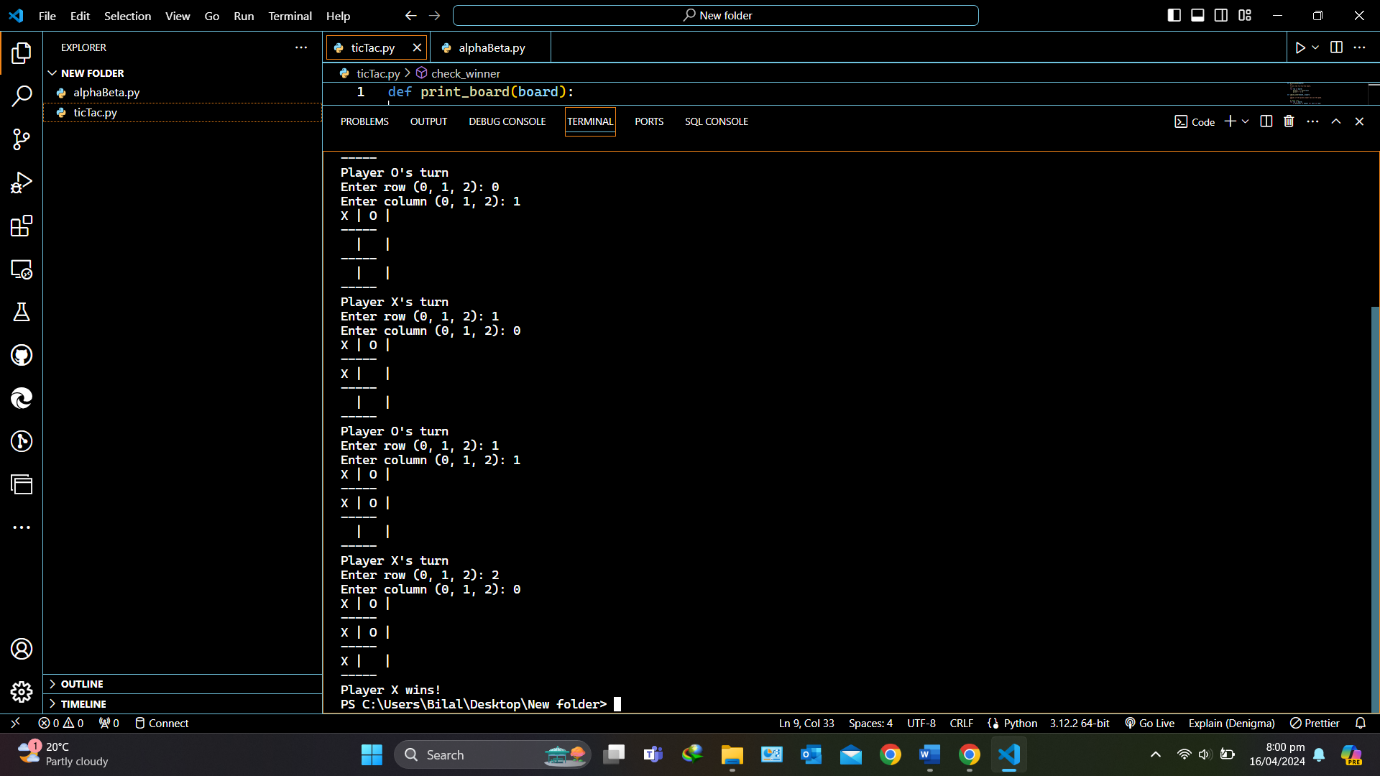
            turn += 1

        else:

            print("That spot is already taken, try again.")

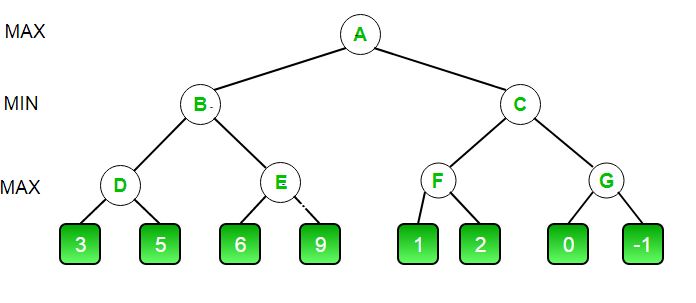
if \_\_name\_\_ == "\_\_main\_\_":

    main()



**3)Alpha beta purinng:**

**Write a Program to Implement Alpha-Beta Pruning using Python.**



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

    if depth == 3:

        return values[nodeIndex]

    if maximizingPlayer:

        best = float("-inf")

        for i in range(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 = float("inf")

        for i in range(2):

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

            best = min(best, val)

            beta = min(beta, best)

            if beta <= alpha:

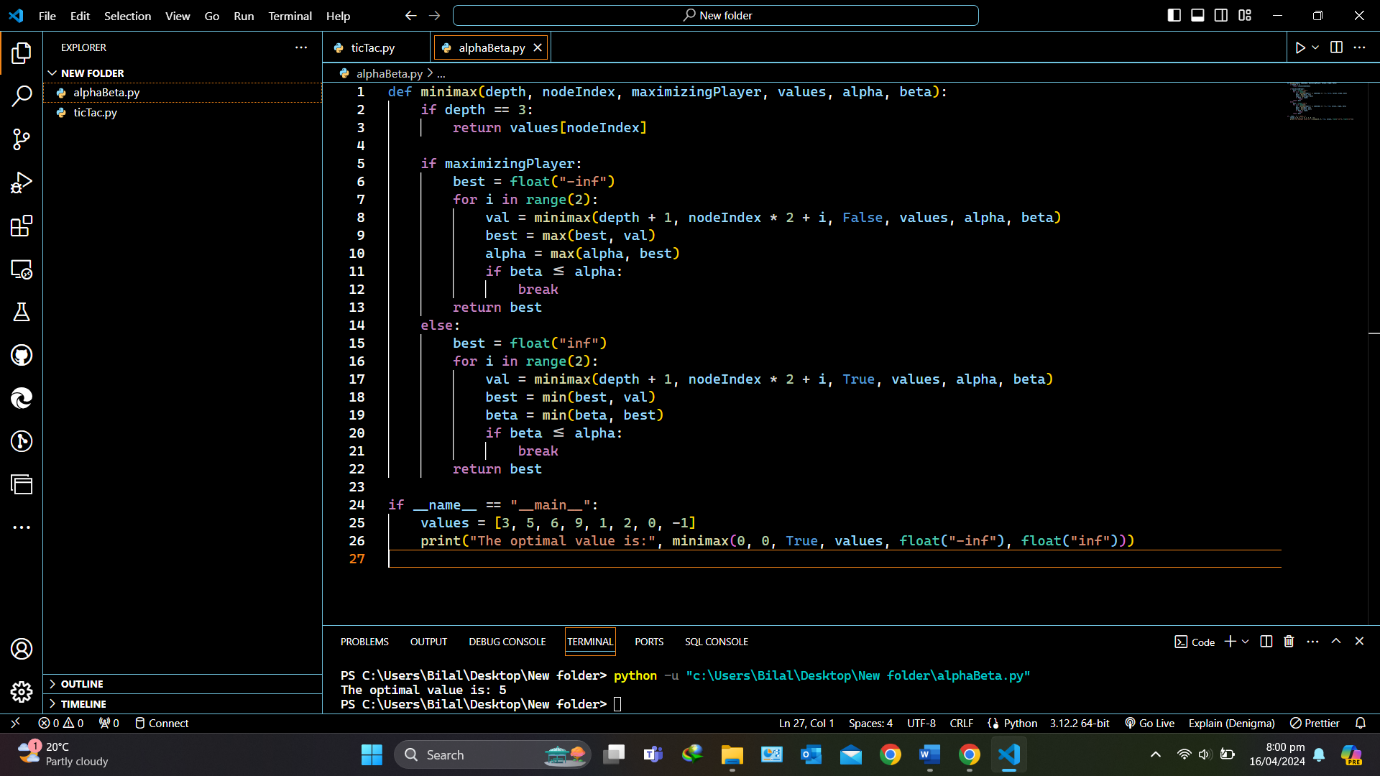
                break

        return best

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

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

    print("The optimal value is:", minimax(0, 0, True, values, float("-inf"), float("inf")))



**4)Hill climb:**

import random

def calculate\_attacks(board):

    return sum(board[i] == board[j] or abs(i - j) == abs(board[i] - board[j]) for i in range(len(board)) for j in range(i + 1, len(board)))

def print\_board(board):

    for row in range(len(board)):

        print(" ".join("Q" if board[row] == col else "-" for col in range(len(board))))

def random\_board(n):

    return random.sample(range(n), n)

def hill\_climbing(n):

    board = random\_board(n)

    current\_attacks = calculate\_attacks(board)

    while True:

        neighbor = board.copy()

        i, j = random.sample(range(n), 2)

        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]

        neighbor\_attacks = calculate\_attacks(neighbor)

        if neighbor\_attacks < current\_attacks:

            board, current\_attacks = neighbor, neighbor\_attacks

        else:

            break

    return board

solution = hill\_climbing(8)

print("Solution (queen positions):", solution)

print("Chessboard:")

print\_board(solution)

**5)A\* :**

def manhattan\_distance(point1, point2):

    # Calculate the Manhattan distance between two points

    x1, y1 = point1

    x2, y2 = point2

    return abs(x1 - x2) + abs(y1 - y2)

def a\_star\_search(maze, start, goal):

    # Get the number of rows and columns in the maze

    rows, cols = len(maze), len(maze[0])

    # Initialize scores and parent pointers for all nodes

    g\_score = {(x, y): float('inf') for x in range(rows) for y in range(cols)}

    g\_score[start] = 0

    f\_score = {(x, y): float('inf') for x in range(rows) for y in range(cols)}

    f\_score[start] = manhattan\_distance(start, goal)

    parent = {(x, y): None for x in range(rows) for y in range(cols)}

    # Priority queue for open list

    open\_list = [(f\_score[start], start)]

    # Closed list to store explored nodes

    closed\_list = set()

    while open\_list:

        # Get the node with the lowest f\_score

        current\_f\_score, current\_node = min(open\_list)

        # If goal is reached, reconstruct path

        if current\_node == goal:

            path = []

            while current\_node:

                path.append(current\_node)

                current\_node = parent[current\_node]

            return path[::-1]  # Reverse for start to goal order

        open\_list.remove((current\_f\_score, current\_node))

        closed\_list.add(current\_node)

        # Explore neighbors

        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

            x, y = current\_node[0] + dx, current\_node[1] + dy

            # Check if the neighbor is within the maze boundaries and is not a wall

            if 0 <= x < rows and 0 <= y < cols and maze[x][y] != 'W':

                neighbor = (x, y)

                tentative\_g\_score = g\_score[current\_node] + 1  # Assuming uniform movement cost

                # If neighbor not yet explored or has better path through current node

                if neighbor not in closed\_list and tentative\_g\_score < g\_score.get(neighbor, float('inf')):

                    parent[neighbor] = current\_node

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

                    f\_score[neighbor] = g\_score[neighbor] + manhattan\_distance(neighbor, goal)

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

    return None  # No path found

# Example maze (replace with your actual maze layout)

maze = [

    [' ', ' ', 'W', ' ', 'X', 'Y'],

    ['R', 'S', 'T', 'U', ' ', 'V'],

    ['M', 'N', ' ', 'O', 'P', 'Q'],

    ['H', 'I', 'J', ' ', 'K', 'L'],

    ['F', ' ', 'G', ' ', ' ', ' '],

    ['A', ' ', 'B', 'C', 'D', 'E'],

]

start = (5, 0)  # Replace with your actual start position (A)

goal = (0, 5)   # Replace with your actual goal position (Y)

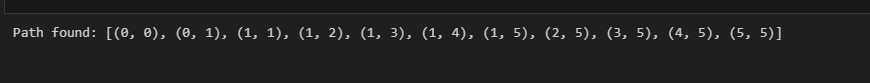
path = a\_star\_search(maze, start, goal)

if path:

    print("Path found:", path)

else:

    print("No path found")



**6)Min max Algorithm:**

import math

def min\_max(board, depth, is\_maximizing):

    if check\_winner(board) != None:

        if check\_winner(board) == "X":

            return -1

        elif check\_winner(board) == "O":

            return 1

        else:

            return 0

    if is\_maximizing:

        best\_score = -math.inf

        for row in range(3):

            for col in range(3):

                if board[row][col] == "":

                    board[row][col] = "O"

                    score = min\_max(board, depth + 1, False)

                    board[row][col] = ""

                    best\_score = max(score, best\_score)

        return best\_score

    else:

        best\_score = math.inf

        for row in range(3):

            for col in range(3):

                if board[row][col] == "":

                    board[row][col] = "X"

                    score = min\_max(board, depth + 1, True)

                    board[row][col] = ""

                    best\_score = min(score, best\_score)

        return best\_score

def check\_winner(board):

    # Code to check for winner

    pass

# Example usage

board = [["X", "O", "X"],

         ["", "O", ""],

         ["", "", ""]]

print("Optimal score for player O:", min\_max(board, 0, True))