**DAY-2 LAB PROGRAMS**

K. Suma

192210145

CSA1757-ARTIFICIAL INTELLIGENCE FOR GOOGLE ASSISTANT

**9. TRAVELLING SALESMAN:**

import itertools

def tsp(cities, start\_city):

all\_cities = cities.copy()

all\_cities.remove(start\_city)

min\_path = None

min\_dist = float('inf')

for perm in itertools.permutations(all\_cities):

path = [start\_city] + list(perm) + [start\_city]

dist = 0

for i in range(len(path) - 1):

dist += graph[path[i]][path[i + 1]]

if dist < min\_dist:

min\_dist = dist

min\_path = path

return min\_path, min\_dist

graph = {

'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},

'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},

'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},

'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}

}

start\_city = 'A'

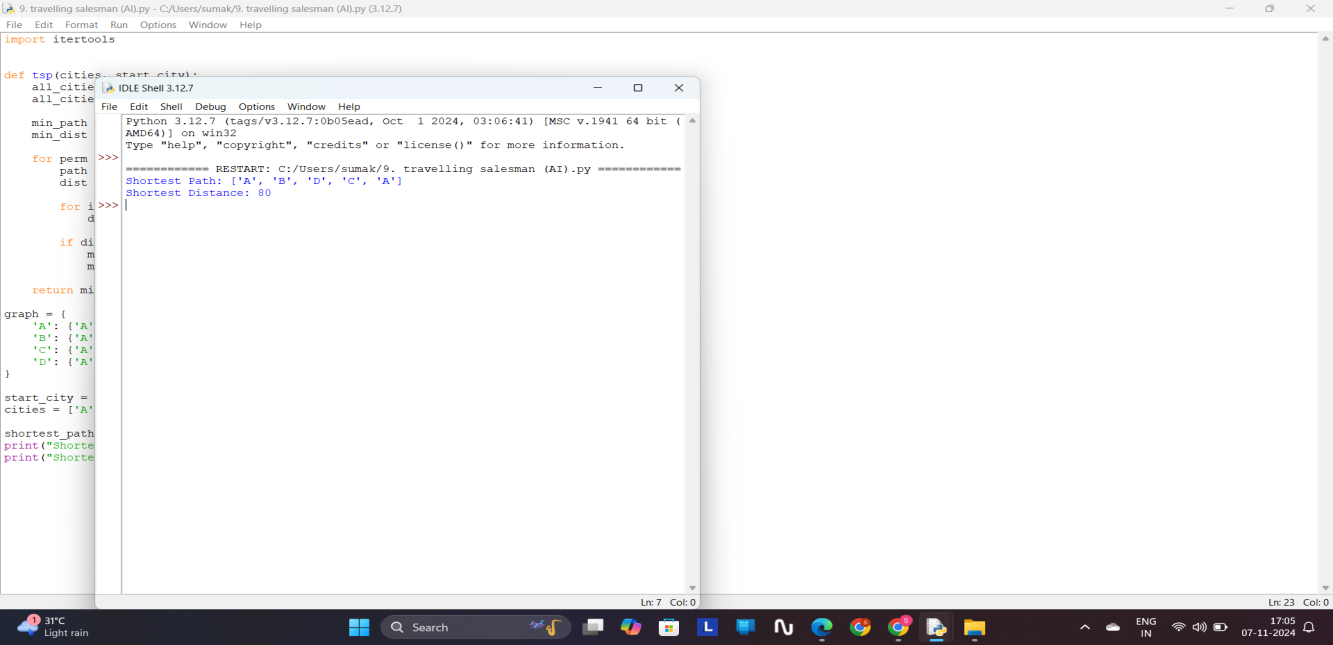
cities = ['A', 'B', 'C', 'D']

shortest\_path, shortest\_distance = tsp(cities, start\_city)

print("Shortest Path:", shortest\_path)

print("Shortest Distance:", shortest\_distance)

**OUTPUT:**



**10. STAR SEARCH ALGORITHM:**

import heapq

def a\_star\_search(graph, start, goal, heuristic):

open\_list = []

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

g\_costs = {start: 0}

came\_from = {start: None}

while open\_list:

\_, current = heapq.heappop(open\_list)

if current == goal:

path = []

while current is not None:

path.append(current)

current = came\_from[current]

return path[::-1], g\_costs[goal]

for neighbor, cost in graph[current]:

new\_cost = g\_costs[current] + cost

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

g\_costs[neighbor] = new\_cost

priority = new\_cost + heuristic(neighbor, goal)

heapq.heappush(open\_list, (priority, neighbor))

came\_from[neighbor] = current

return None, float('inf')

def heuristic(node, goal):

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

graph = {

(0, 0): [((1, 0), 1), ((0, 1), 1)],

(1, 0): [((0, 0), 1), ((1, 1), 1)],

(0, 1): [((0, 0), 1), ((1, 1), 1)],

(1, 1): [((1, 0), 1), ((0, 1), 1), ((2, 1), 1)],

(2, 1): [((1, 1), 1), ((2, 2), 1)],

(2, 2): [((2, 1), 1)]

}

start = (0, 0)

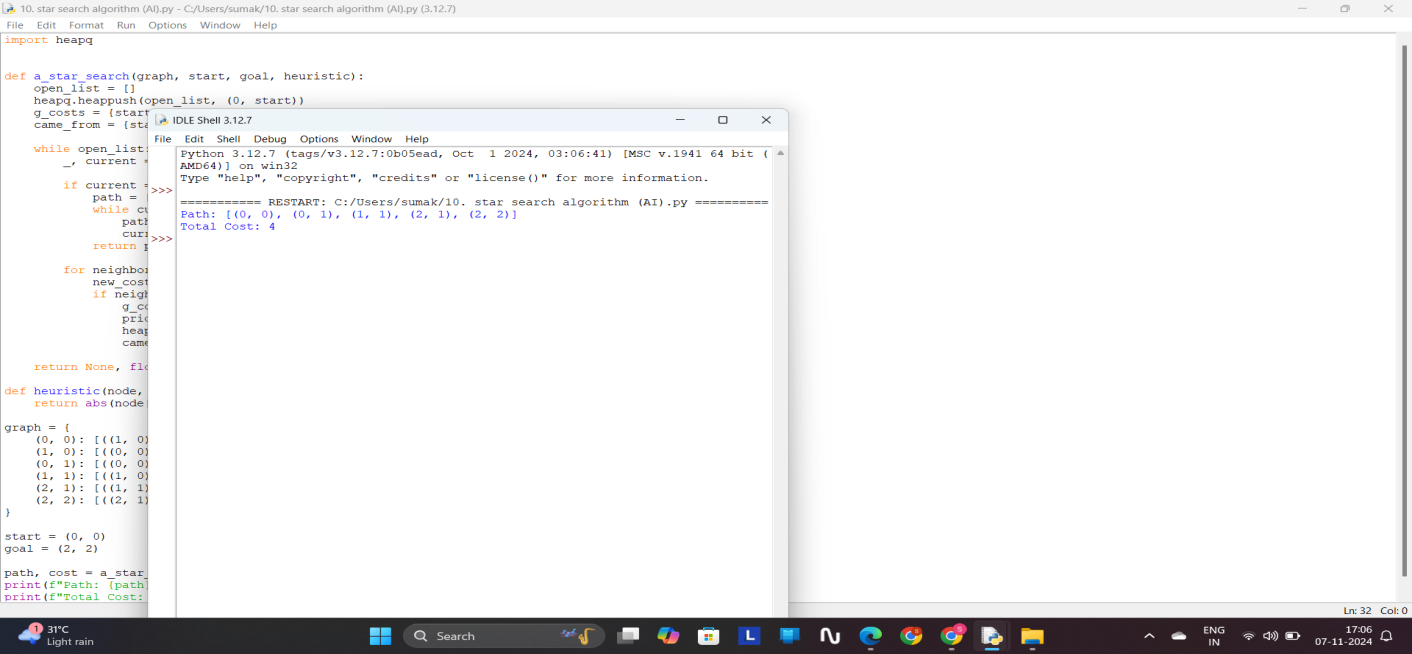
goal = (2, 2)

path, cost = a\_star\_search(graph, start, goal, heuristic)

print(f"Path: {path}")

print(f"Total Cost: {cost}")

**OUTPUT:**



**11. MAP COLORING:**

class MapColoringCSP:

def \_\_init\_\_(self, variables, domains, neighbors):

self.variables = variables

self.domains = domains

self.neighbors = neighbors

self.assignment = {}

def is\_consistent(self, variable, color):

"""Check if assigning 'color' to 'variable' is consistent with current assignments."""

for neighbor in self.neighbors[variable]:

if neighbor in self.assignment and self.assignment[neighbor] == color:

return False

return True

def select\_unassigned\_variable(self):

"""Select a variable with MRV heuristic (Minimum Remaining Values)."""

unassigned\_variables = [v for v in self.variables if v not in self.assignment]

return min(unassigned\_variables, key=lambda var: len(self.domains[var]))

def order\_domain\_values(self, variable):

"""Order values with Least Constraining Value (LCV) heuristic."""

return sorted(self.domains[variable], key=lambda color: self.conflicts(variable, color))

def conflicts(self, variable, color):

"""Count the number of conflicts assigning 'color' to 'variable' would produce."""

return sum(1 for neighbor in self.neighbors[variable]

if neighbor not in self.assignment and color in self.domains[neighbor])

def backtrack(self):

"""Backtracking search algorithm to solve the CSP."""

if len(self.assignment) == len(self.variables):

return self.assignment

variable = self.select\_unassigned\_variable()

for color in self.order\_domain\_values(variable):

if self.is\_consistent(variable, color):

self.assignment[variable] = color

result = self.backtrack()

if result:

return result

del self.assignment[variable]

return None

def solve(self):

"""Solve the CSP problem."""

return self.backtrack()

variables = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']

domains = {var: ['Red', 'Green', 'Blue'] for var in variables}

neighbors = {

'WA': ['NT', 'SA'],

'NT': ['WA', 'SA', 'Q'],

'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],

'Q': ['NT', 'SA', 'NSW'],

'NSW': ['Q', 'SA', 'V'],

'V': ['SA', 'NSW'],

'T': []

}

csp = MapColoringCSP(variables, domains, neighbors)

solution = csp.solve()

if solution:

print("Solution found:")

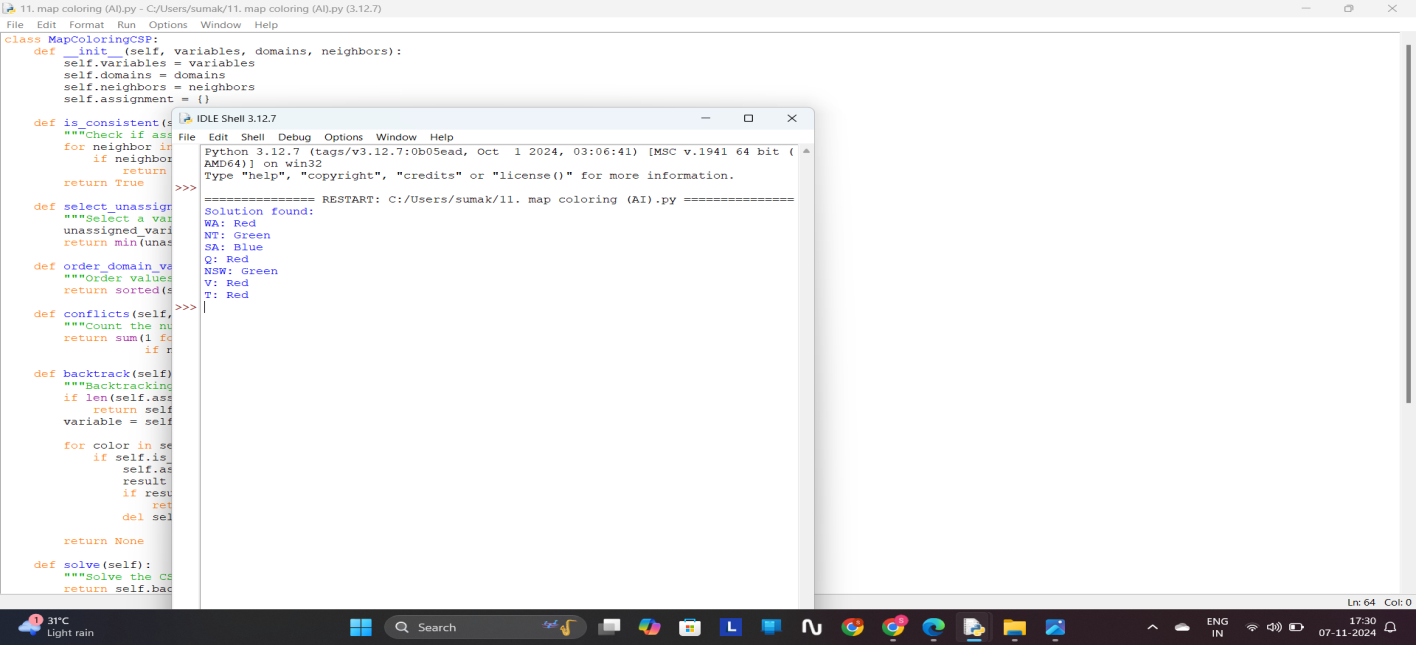
for region, color in solution.items():

print(f"{region}: {color}")

else:

print("No solution found.")

**OUTPUT:**



**12. TIC TAC TOE:**

def print\_board(board):

"""Prints the Tic Tac Toe board."""

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

"""Checks if the specified player has won."""

for i in range(3):

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

return True

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 check\_draw(board):

"""Checks if the game is a draw."""

for row in board:

for cell in row:

if cell == ' ':

return False

return True

def main():

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

players = ['X', 'O']

current\_player = 0

while True:

print\_board(board)

print(f"Player {players[current\_player]}'s turn.")

while True:

try:

row, col = map(int, input("Enter row and column (0, 1, or 2) separated by space: ").split())

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

break

else:

print("That cell is already occupied. Try again.")

except ValueError:

print("Invalid input. Please enter two integers separated by space.")

except IndexError:

print("Invalid input. Please enter row and column numbers within 0 to 2.")

board[row][col] = players[current\_player]

if check\_winner(board, players[current\_player]):

print\_board(board)

print(f"Congratulations! Player {players[current\_player]} wins!")

break

elif check\_draw(board):

print\_board(board)

print("It's a draw!")

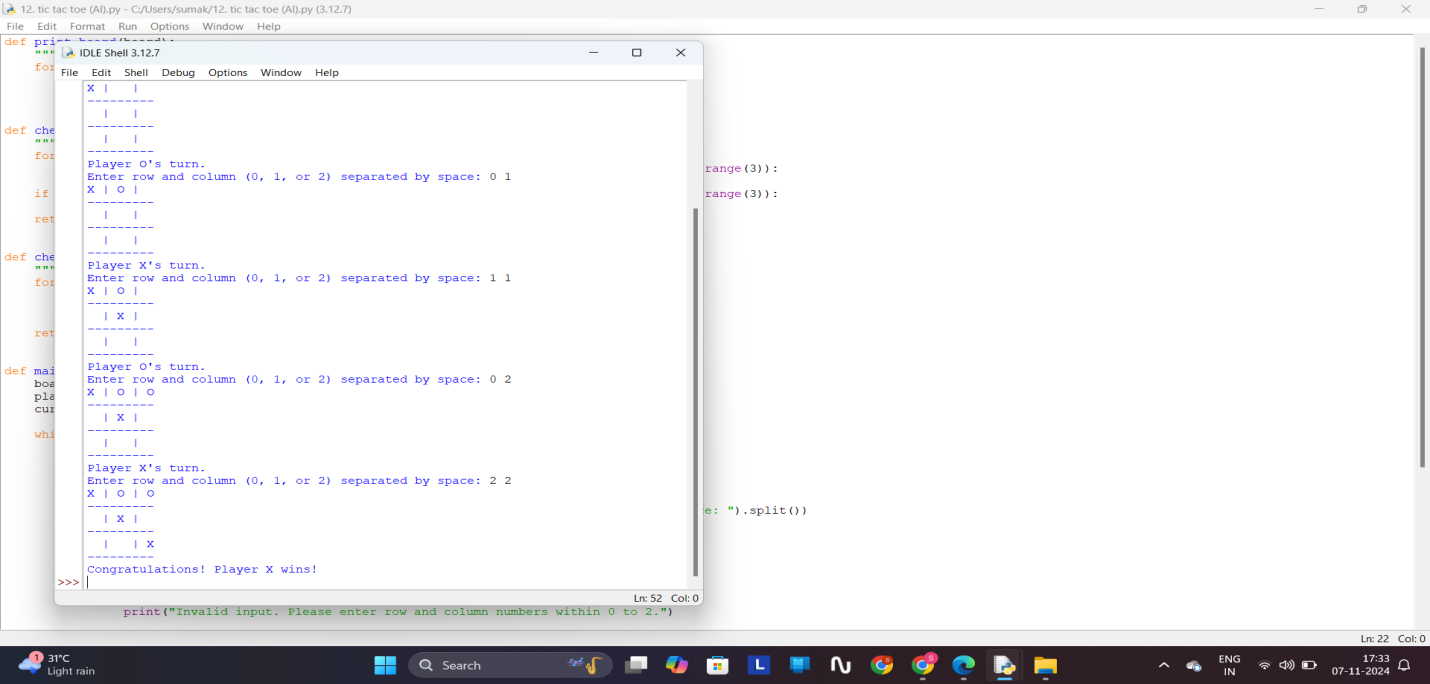
break

current\_player = (current\_player + 1) % 2

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

main()

**OUTPUT:**



**13. MIN AND MAX ALGORITHM:**

import math

PLAYER\_X = 'X'

PLAYER\_O = 'O'

EMPTY = ' '

def print\_board(board):

"""Prints the Tic-Tac-Toe board."""

for row in board:

print("| " + " | ".join(row) + " |")

print("-------------")

print()

def evaluate(board):

"""Evaluate the current state of the board."""

for row in board:

if all(cell == PLAYER\_X for cell in row):

return 10

elif all(cell == PLAYER\_O for cell in row):

return -10

for col in range(3):

if all(board[row][col] == PLAYER\_X for row in range(3)):

return 10

elif all(board[row][col] == PLAYER\_O for row in range(3)):

return -10

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

return 10

elif all(board[i][i] == PLAYER\_O for i in range(3)) or all(board[i][2 - i] == PLAYER\_O for i in range(3)):

return -10

return 0

def is\_moves\_left(board):

"""Check if there are any empty cells left."""

for row in board:

if EMPTY in row:

return True

return False

def minimax(board, depth, is\_maximizing):

"""Minimax algorithm implementation."""

score = evaluate(board)

if score == 10:

return score - depth

elif score == -10:

return score + depth

elif not is\_moves\_left(board):

return 0

if is\_maximizing:

best = -math.inf

for i in range(3):

for j in range(3):

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

board[i][j] = PLAYER\_X

best = max(best, minimax(board, depth + 1, not is\_maximizing))

board[i][j] = EMPTY

return best

else:

best = math.inf

for i in range(3):

for j in range(3):

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

board[i][j] = PLAYER\_O

best = min(best, minimax(board, depth + 1, not is\_maximizing))

board[i][j] = EMPTY

return best

def find\_best\_move(board):

"""Find the best move using Minimax."""

best\_val = -math.inf

best\_move = (-1, -1)

for i in range(3):

for j in range(3):

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

board[i][j] = PLAYER\_X

move\_val = minimax(board, 0, False)

board[i][j] = EMPTY

if move\_val > best\_val:

best\_val = move\_val

best\_move = (i, j)

return best\_move

def main():

board = [

[EMPTY, EMPTY, EMPTY],

[EMPTY, EMPTY, EMPTY],

[EMPTY, EMPTY, EMPTY]

]

print("Initial Board:")

print\_board(board)

while is\_moves\_left(board):

print("Player X's turn (Human)")

while True:

x = int(input("Enter row number (0-2): "))

y = int(input("Enter column number (0-2): "))

if board[x][y] == EMPTY:

board[x][y] = PLAYER\_X

break

else:

print("Invalid move. Cell is not empty. Try again.")

print\_board(board)

if evaluate(board) == 10:

print("Player X wins!")

break

if not is\_moves\_left(board):

print("It's a draw!")

break

print("Player O's turn (Computer)")

x\_o, y\_o = find\_best\_move(board)

board[x\_o][y\_o] = PLAYER\_O

print\_board(board)

if evaluate(board) == -10:

print("Player O wins!")

break

if not is\_moves\_left(board):

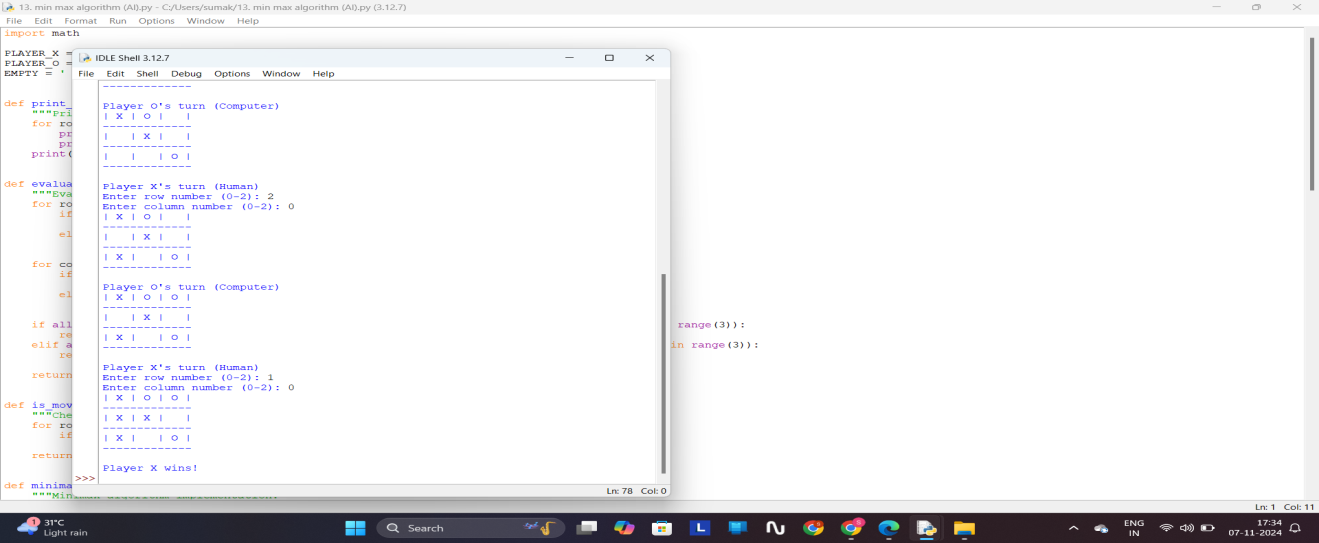
print("It's a draw!")

break

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

main()

**OUTPUT:**



**14. ALPHA BETA PRUNNING:**

import math

def alpha\_beta\_pruning(depth, node\_index, maximizing\_player, values, alpha, beta):

if depth == math.log2(len(values)):

return values[node\_index]

if maximizing\_player:

max\_eval = -math.inf

for i in range(2):

eval = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, False, values, alpha, beta)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = math.inf

for i in range(2):

eval = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, True, values, alpha, beta)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

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

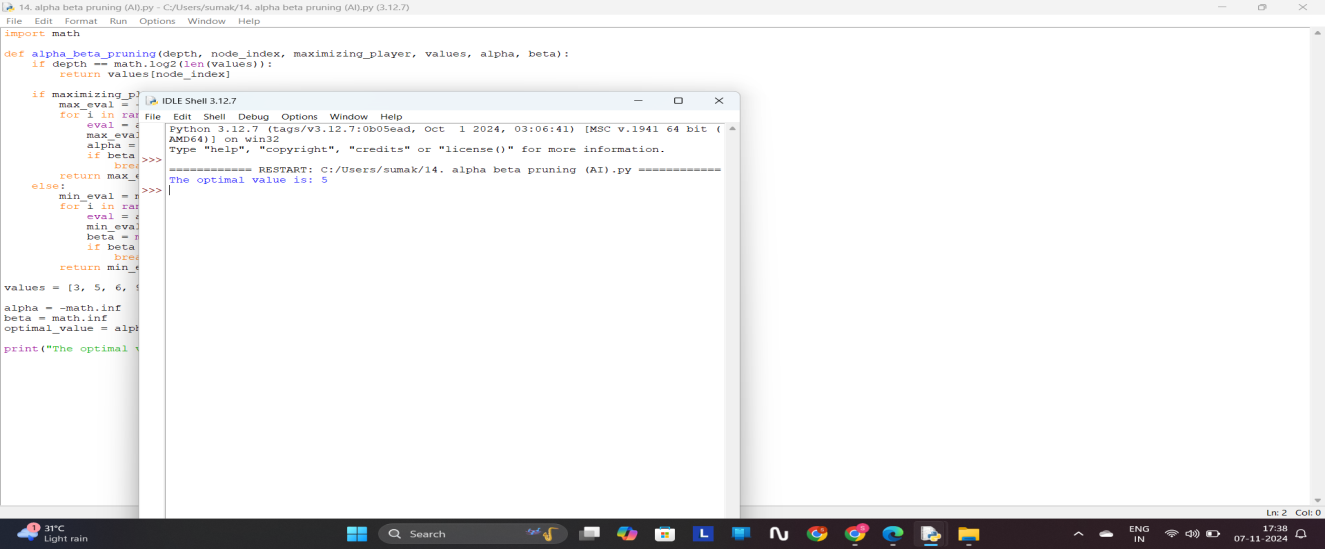
alpha = -math.inf

beta = math.inf

optimal\_value = alpha\_beta\_pruning(0, 0, True, values, alpha, beta)

print("The optimal value is:", optimal\_value)

**OUTPUT:**



**15. DECISION TREE:**

import math

from collections import Counter

def entropy(labels):

total\_count = len(labels)

label\_counts = Counter(labels)

entropy\_value = 0.0

for count in label\_counts.values():

probability = count / total\_count

entropy\_value -= probability \* math.log2(probability)

return entropy\_value

def information\_gain(data, labels, feature\_index):

original\_entropy = entropy(labels)

total\_count = len(labels)

feature\_values = [row[feature\_index] for row in data]

value\_counts = Counter(feature\_values)

weighted\_entropy = 0.0

for value, count in value\_counts.items():

subset\_labels = [labels[i] for i in range(total\_count) if data[i][feature\_index] == value]

weighted\_entropy += (count / total\_count) \* entropy(subset\_labels)

return original\_entropy - weighted\_entropy

def build\_tree(data, labels, features, depth=0):

if len(set(labels)) == 1:

return labels[0]

if len(features) == 0:

return Counter(labels).most\_common(1)[0][0]

gains = [information\_gain(data, labels, i) for i in range(len(features))]

best\_feature\_index = gains.index(max(gains))

tree = {features[best\_feature\_index]: {}}

unique\_values = set(row[best\_feature\_index] for row in data)

for value in unique\_values:

subset\_data = [row[:best\_feature\_index] + row[best\_feature\_index + 1:] for row in data if

row[best\_feature\_index] == value]

subset\_labels = [labels[i] for i in range(len(data)) if data[i][best\_feature\_index] == value]

new\_features = features[:best\_feature\_index] + features[best\_feature\_index + 1:]

subtree = build\_tree(subset\_data, subset\_labels, new\_features, depth + 1)

tree[features[best\_feature\_index]][value] = subtree

return tree

def predict(tree, sample):

if not isinstance(tree, dict):

return tree

feature = next(iter(tree))

feature\_value = sample.get(feature)

if feature\_value in tree[feature]:

return predict(tree[feature][feature\_value], sample)

else:

return None

data = [

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

]

labels = [0, 1, 0, 1, 0]

features = ['Feature 1', 'Feature 2']

decision\_tree = build\_tree(data, labels, features)

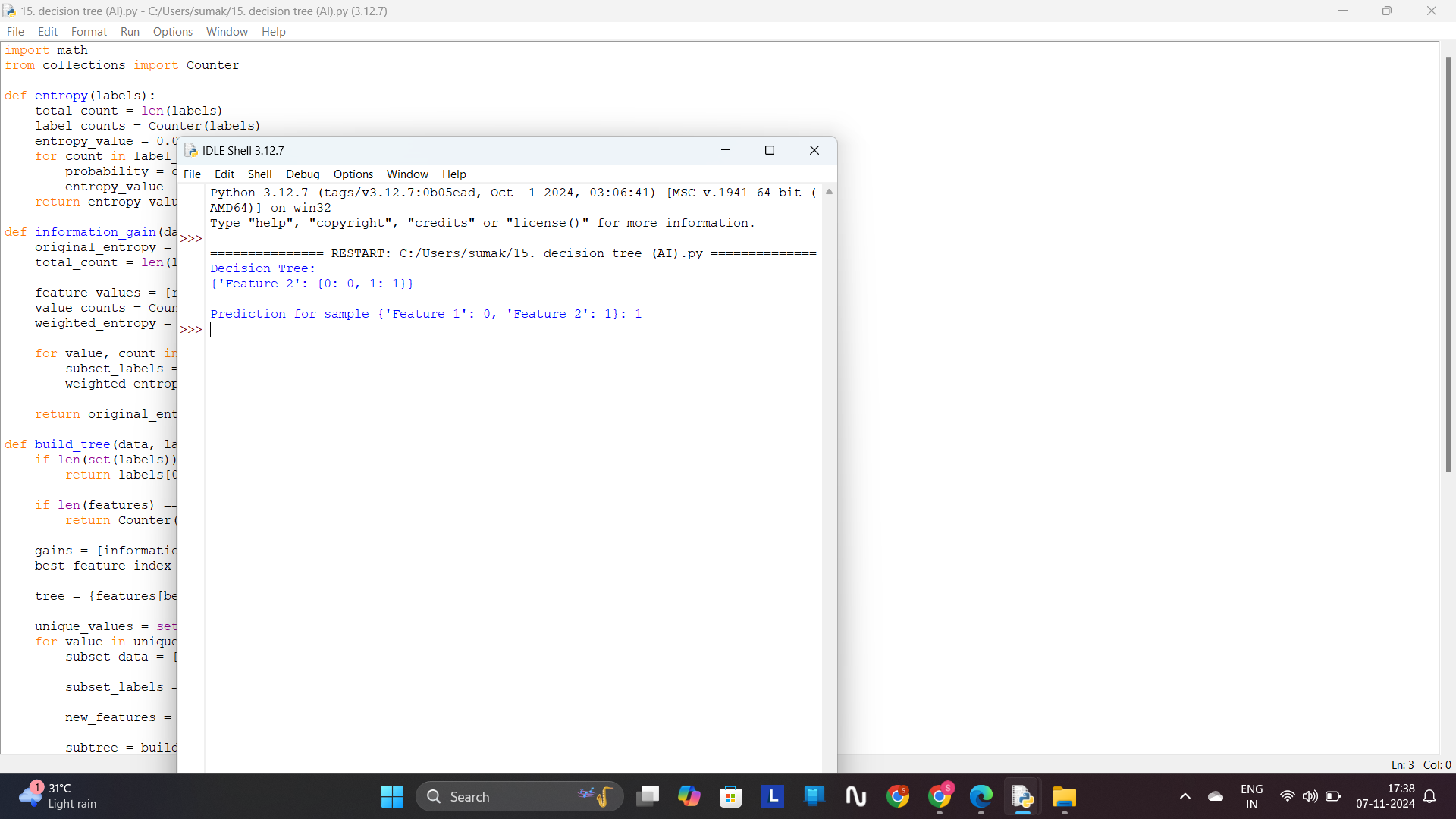
print("Decision Tree:")

print(decision\_tree)

sample = {'Feature 1': 0, 'Feature 2': 1}

print("\nPrediction for sample {}: {}".format(sample, predict(decision\_tree, sample)))

**OUTPUT:**



**16. FORWARD NEURAL NETWORK:**

import math

import random

def sigmoid(x):

return 1 / (1 + math.exp(-x))

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.weights\_input\_hidden = []

for \_ in range(self.input\_size):

self.weights\_input\_hidden.append([random.uniform(-1, 1) for \_ in range(self.hidden\_size)])

self.bias\_hidden = [random.uniform(-1, 1) for \_ in range(self.hidden\_size)]

self.weights\_hidden\_output = []

for \_ in range(self.hidden\_size):

self.weights\_hidden\_output.append([random.uniform(-1, 1) for \_ in range(self.output\_size)])

self.bias\_output = [random.uniform(-1, 1) for \_ in range(self.output\_size)]

def feedforward(self, inputs):

hidden\_sum = []

for j in range(self.hidden\_size):

neuron\_sum = 0

for i in range(self.input\_size):

neuron\_sum += inputs[i] \* self.weights\_input\_hidden[i][j]

neuron\_sum += self.bias\_hidden[j]

hidden\_sum.append(neuron\_sum)

hidden\_output = [sigmoid(x) for x in hidden\_sum]

output\_sum = []

for k in range(self.output\_size):

neuron\_sum = 0

for j in range(self.hidden\_size):

neuron\_sum += hidden\_output[j] \* self.weights\_hidden\_output[j][k]

neuron\_sum += self.bias\_output[k]

output\_sum.append(neuron\_sum)

final\_output = [sigmoid(x) for x in output\_sum]

return final\_output

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

nn = NeuralNetwork(input\_size=2, hidden\_size=3, output\_size=1)

input\_data = [0.7, 0.3]

output = nn.feedforward(input\_data)

print("Input:", input\_data)

print("Output:", output)

**OUTPUT:**

