**DAY-2 LAB PROGRAMS**

S.Kangana Sri

192210006

CSA1760-Artificial Intelligence

**7.BFS PROBLEM**

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

visited.add(node)

print(node)

for neighbor in graph[node]:

if neighbor not in visited:

queue.append(neighbor)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D'],

'C': ['A','E','F'],

'D': ['B'],

'E': ['C' ],

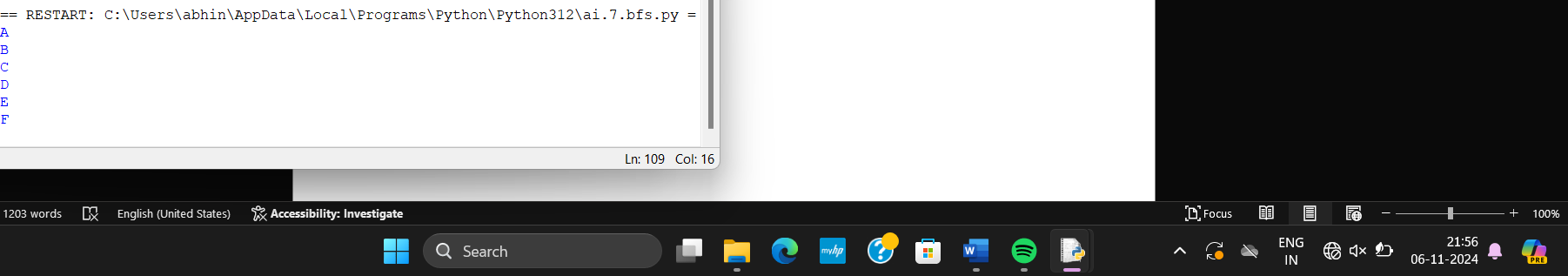
'F': [ 'C']

}

start\_node = 'A'

bfs(graph, start\_node)

**OUTPUT:**



**8.DFS PROBLEM**

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start)

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

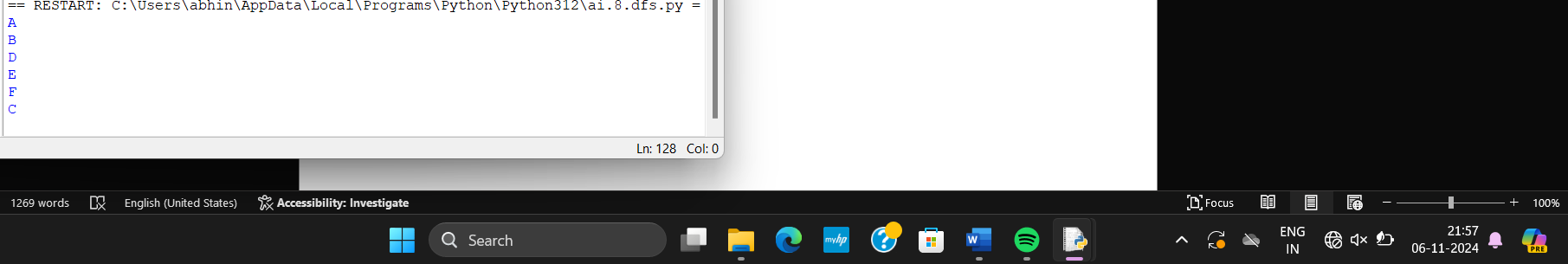
'F': ['C', 'E']

}

start\_node = 'A'

dfs(graph, start\_node)

**OUTPUT:**



**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'

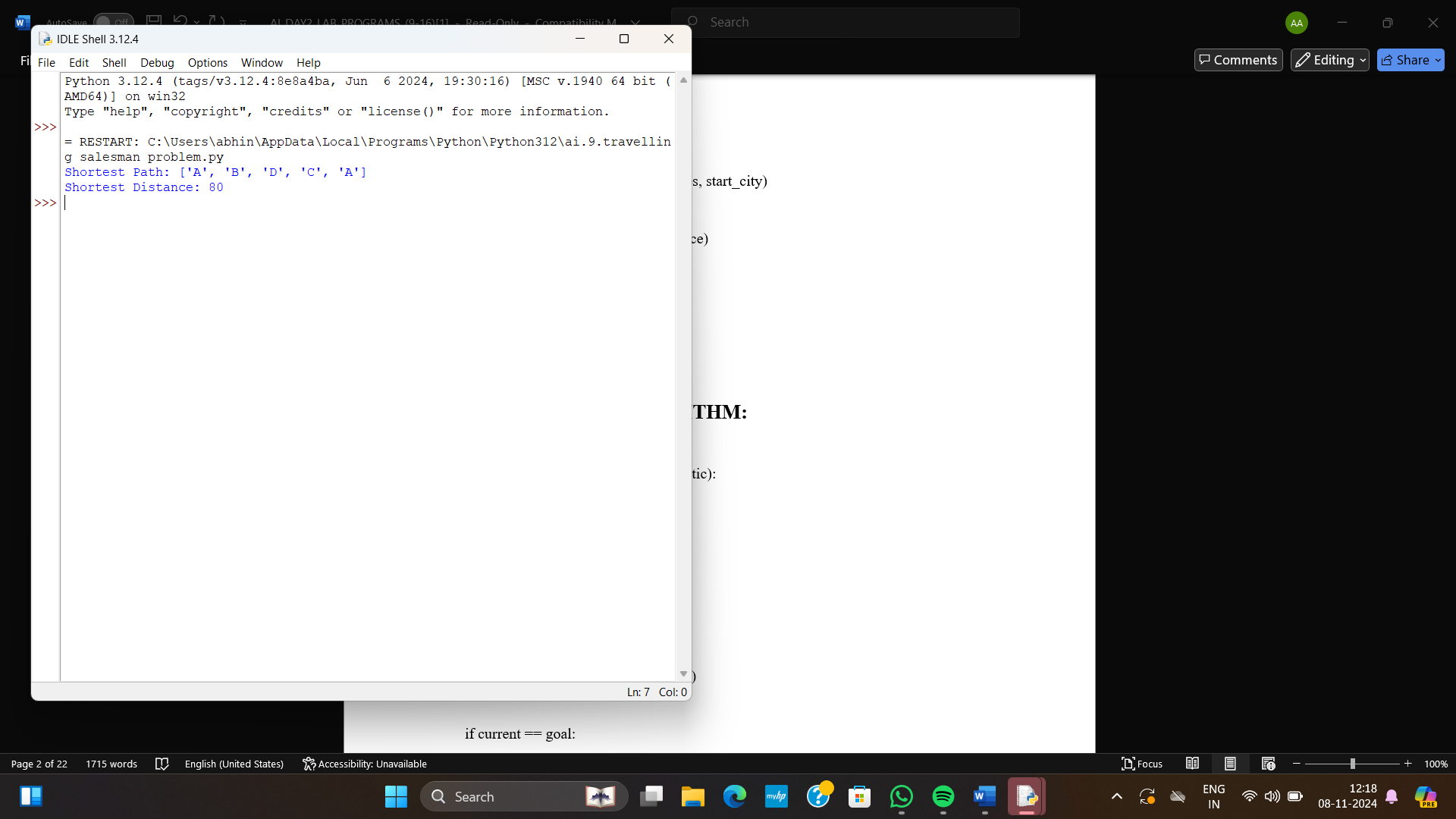
cities = ['A', 'B', 'C', 'D']

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

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

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

**OUTPUT:**



**10. A 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)

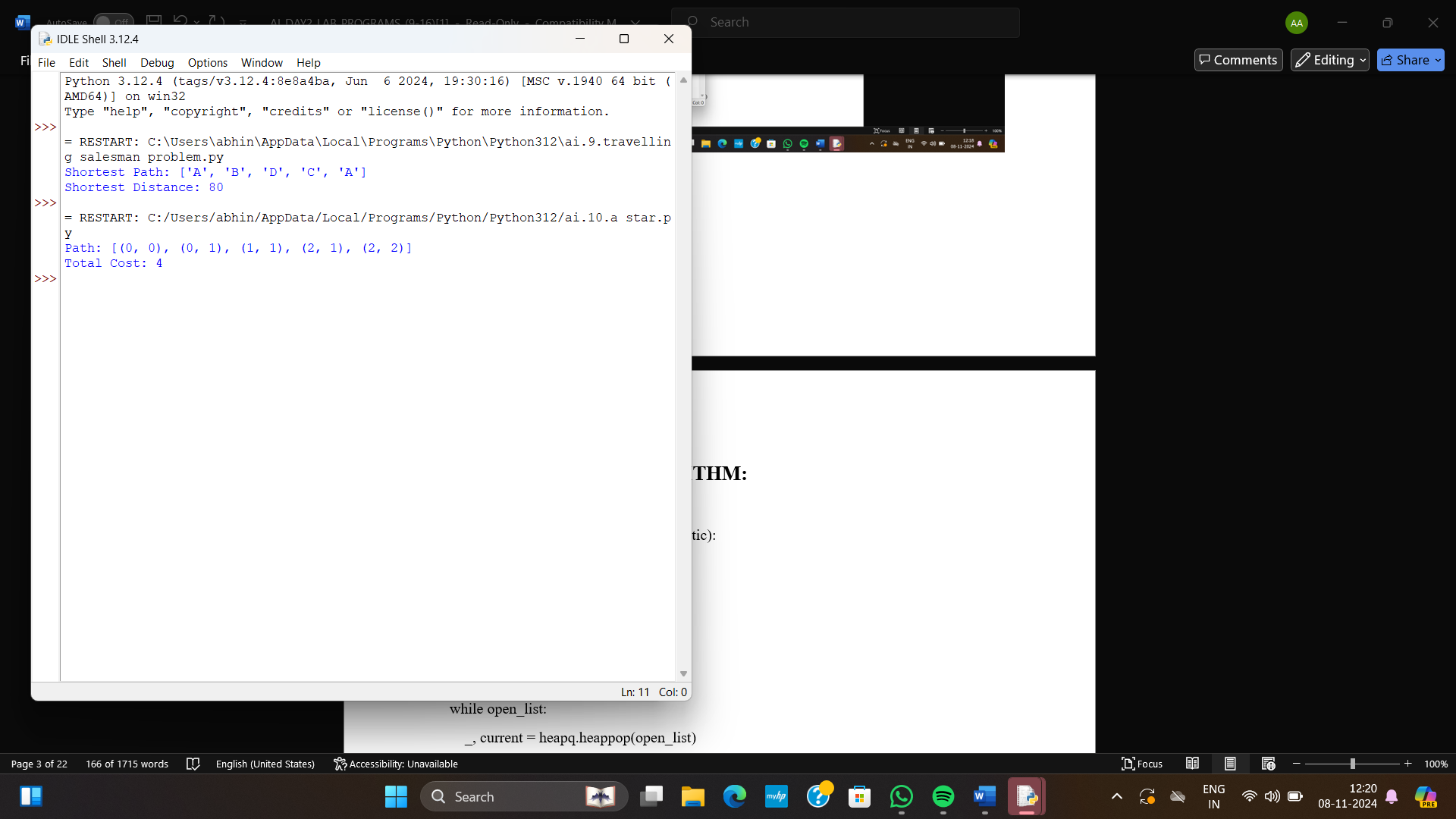
goal = (2, 2)

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:")

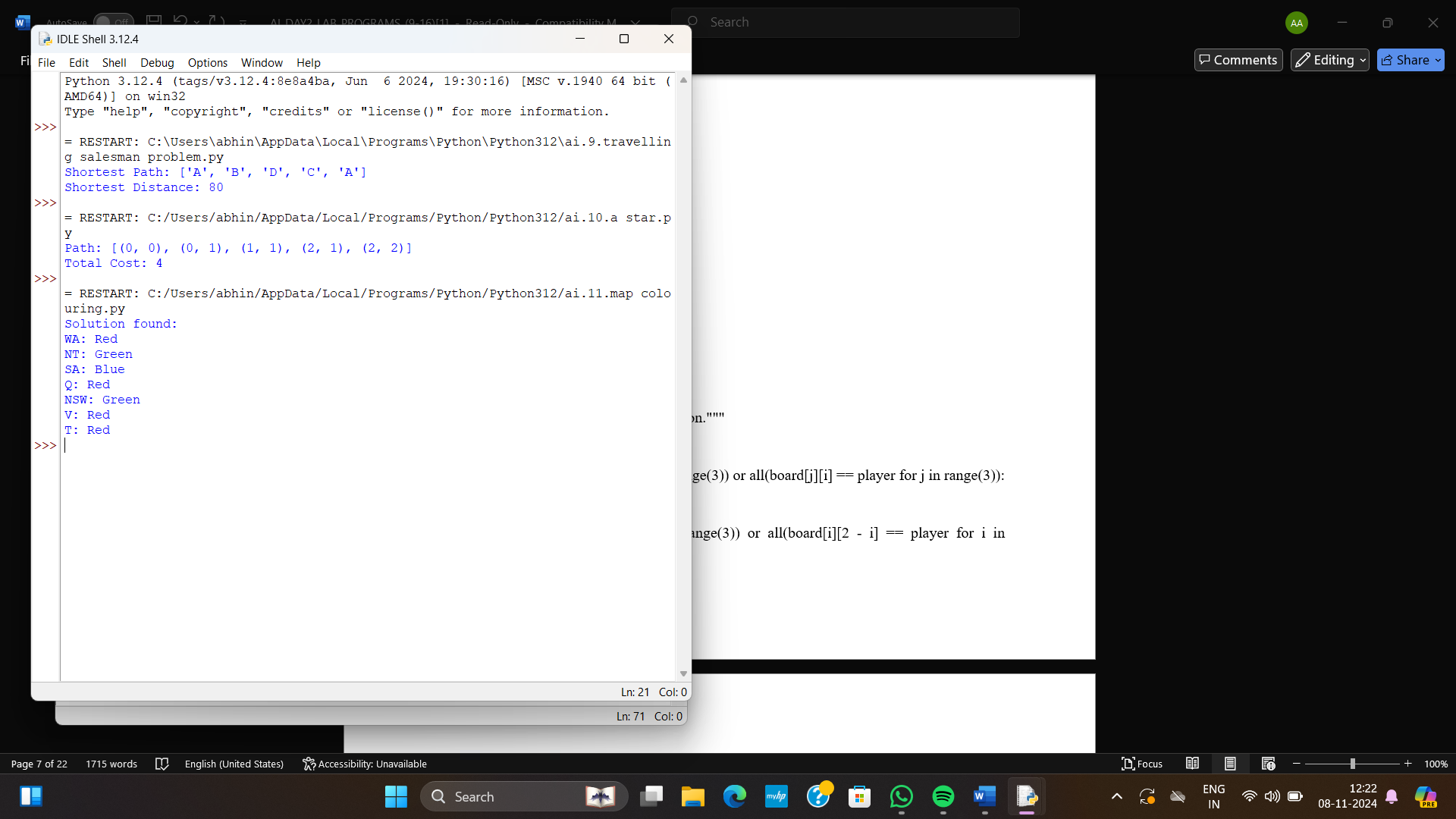
for region, color in solution.items():

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:**

