RIPHAH
INTERNATIONAL
UNIVERSITY

Artificial Intelligence

Lab 07 Tasks

Name: Dua Amir

Sap ID: 47849

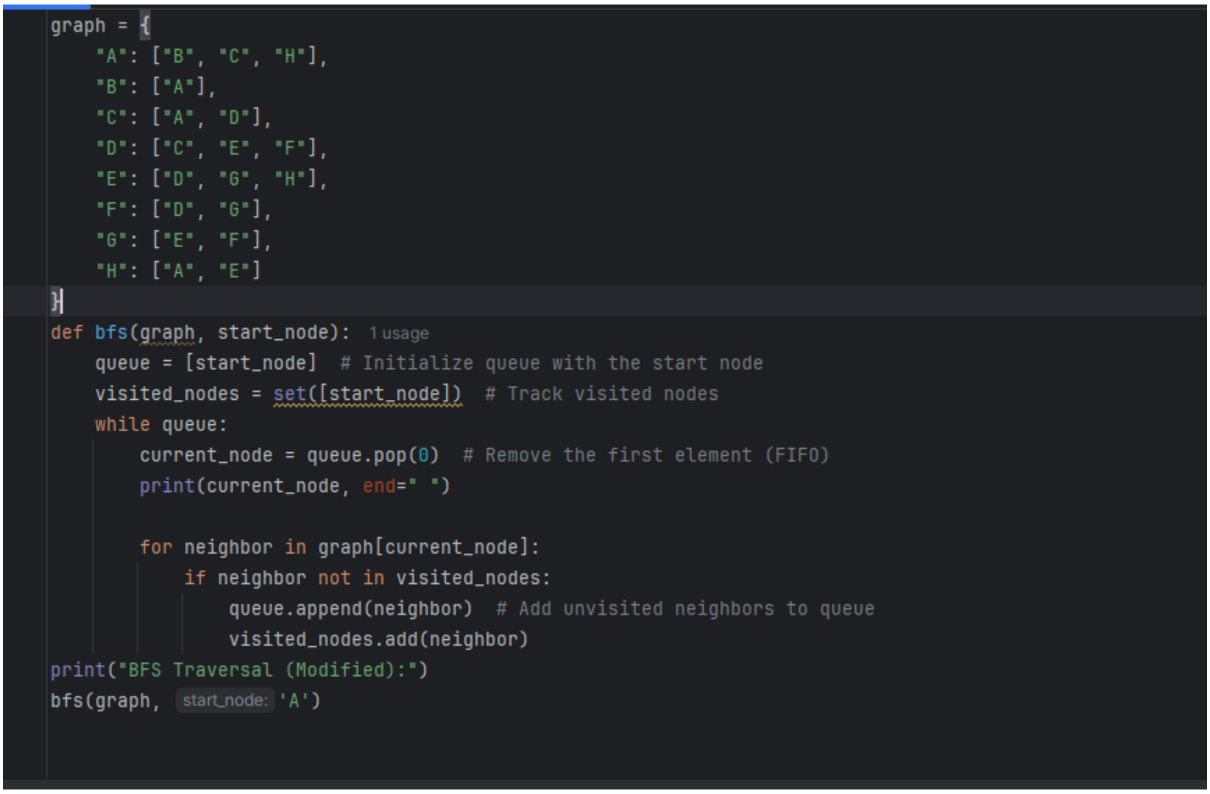
Batch: BSCS-6th semester

Lab Instructor:

Mam Ayesha Akram

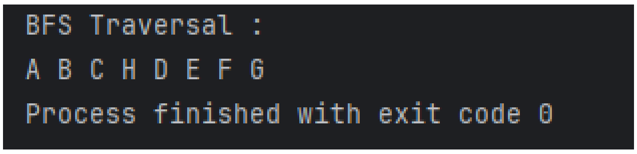
Task#1

Solution:



```
graph = {
    "A": ["B", "C", "H"],
    "B": ["A"],
    "C": ["A", "D"],
    "D": ["C", "E", "F"],
    "E": ["D", "G", "H"],
    "F": ["D", "G"],
    "G": ["E", "F"],
    "H": ["A", "E"]
}
def bfs(graph, start_node):  1 usage
    queue = [start_node]  # Initialize queue with the start node
    visited_nodes = set([start_node])  # Track visited nodes
    while queue:
        current_node = queue.pop(0)  # Remove the first element (FIFO)
        print(current_node, end=" ")

        for neighbor in graph[current_node]:
            if neighbor not in visited_nodes:
                queue.append(neighbor)  # Add unvisited neighbors to queue
                visited_nodes.add(neighbor)
print("BFS Traversal (Modified):")
bfs(graph,  start_node: 'A')
```

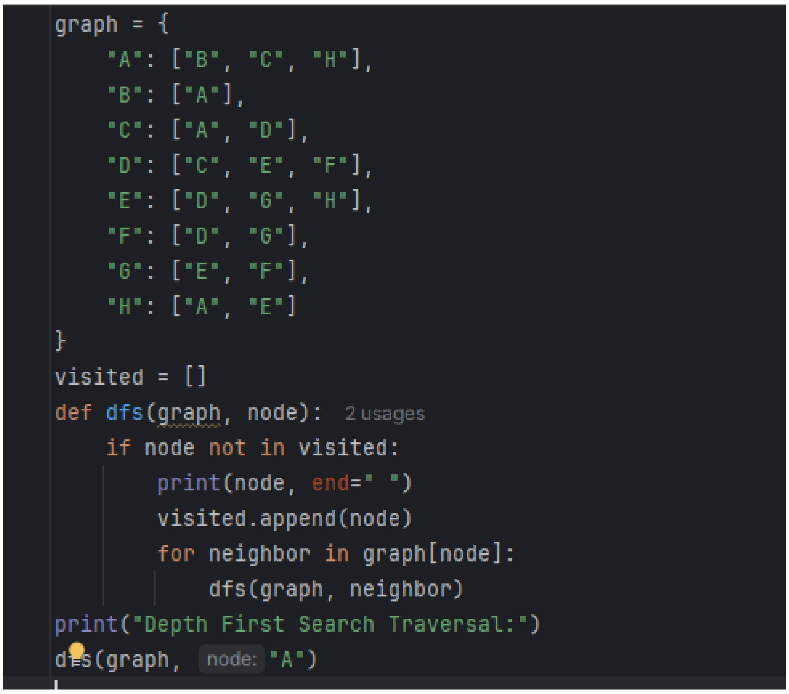
Output:



```
BFS Traversal :
A B C H D E F G
Process finished with exit code 0
```

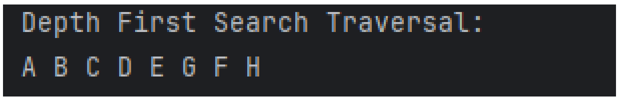
Task#2

Solution:



```python
graph = {
    "A": ["B", "C", "H"],
    "B": ["A"],
    "C": ["A", "D"],
    "D": ["C", "E", "F"],
    "E": ["D", "G", "H"],
    "F": ["D", "G"],
    "G": ["E", "F"],
    "H": ["A", "E"]
}
visited = []
def dfs(graph, node):
    if node not in visited:
        print(node, end=" ")
        visited.append(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor)
print("Depth First Search Traversal:")
dfs(graph, "A")
```

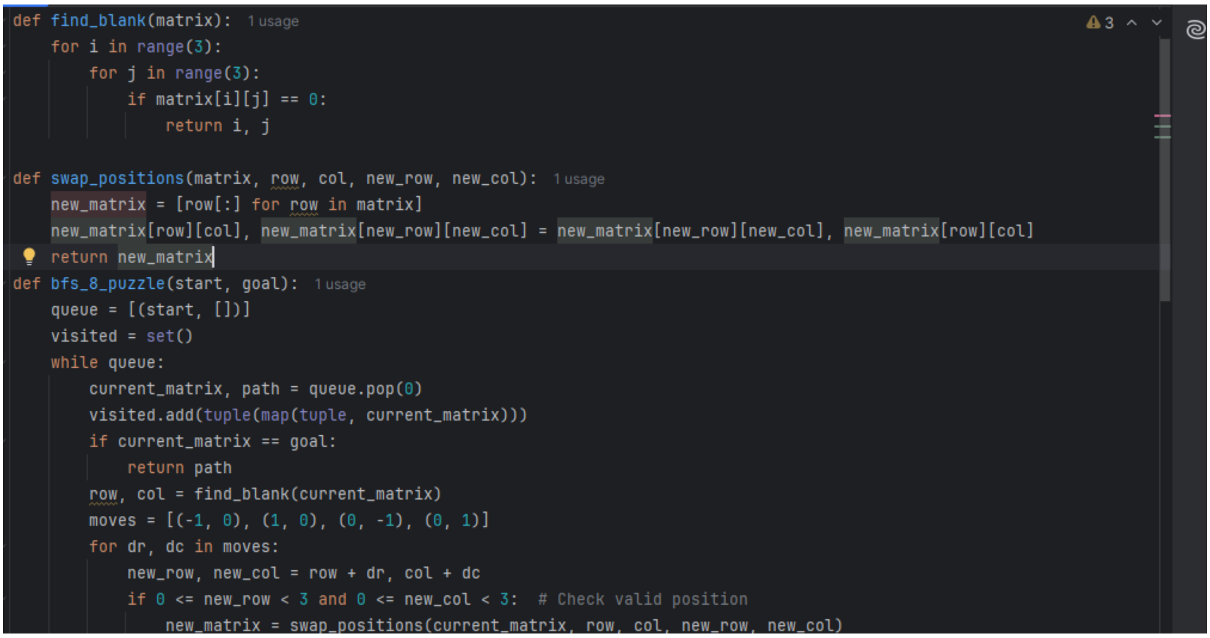
Output:



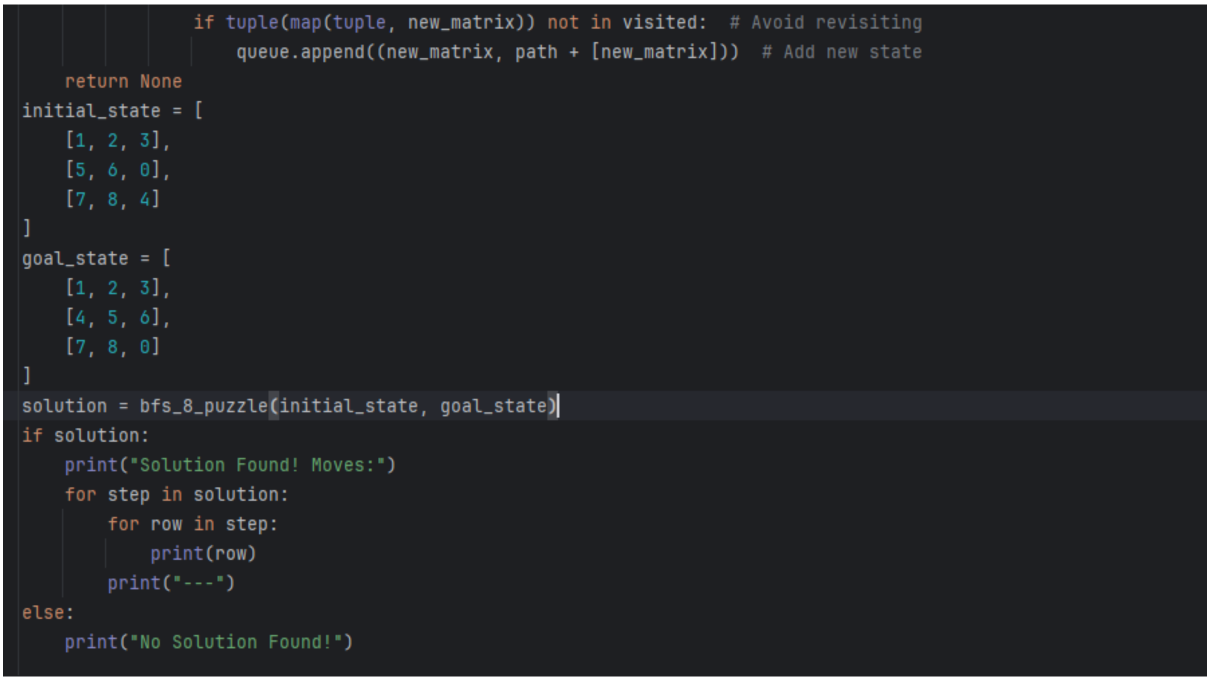
```
Depth First Search Traversal:
A B C D E G F H
```

Task#3

Solution:



```python
def find_blank(matrix):
    for i in range(3):
        for j in range(3):
            if matrix[i][j] == 0:
                return i, j

def swap_positions(matrix, row, col, new_row, new_col):
    new_matrix = [row[:] for row in matrix]
    new_matrix[row][col], new_matrix[new_row][new_col] = new_matrix[new_row][new_col], new_matrix[row][col]
    return new_matrix
def bfs_8_puzzle(start, goal):
    queue = [(start, [])]
    visited = set()
    while queue:
        current_matrix, path = queue.pop(0)
        visited.add(tuple(map(tuple, current_matrix)))
        if current_matrix == goal:
            return path
        row, col = find_blank(current_matrix)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dr, dc in moves:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:  # Check valid position
                new_matrix = swap_positions(current_matrix, row, col, new_row, new_col)
```



```
                if tuple(map(tuple, new_matrix)) not in visited:  # Avoid revisiting
                    queue.append((new_matrix, path + [new_matrix]))  # Add new state
    return None
initial_state = [
    [1, 2, 3],
    [5, 6, 0],
    [7, 8, 4]
]
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
solution = bfs_8_puzzle(initial_state, goal_state)
if solution:
    print("Solution Found! Moves:")
    for step in solution:
        for row in step:
            print(row)
        print("---")
else:
    print("No Solution Found!")
```

Dry Run:

* Initial State

1 2 3
5 6 0 < Blank tile 0
7 8 4

* Goal State:

1 2 3
4 5 6
7 8 0 < Blank tile 0

1) Move 4 to blank position

1 2 3
4 5 6
7 8 0

2) Move 6 to blank positions

1 2 3
5 0 4
7 8 6

3) Move 5 to blank position

1 2 3
0 5 4
7 8 6

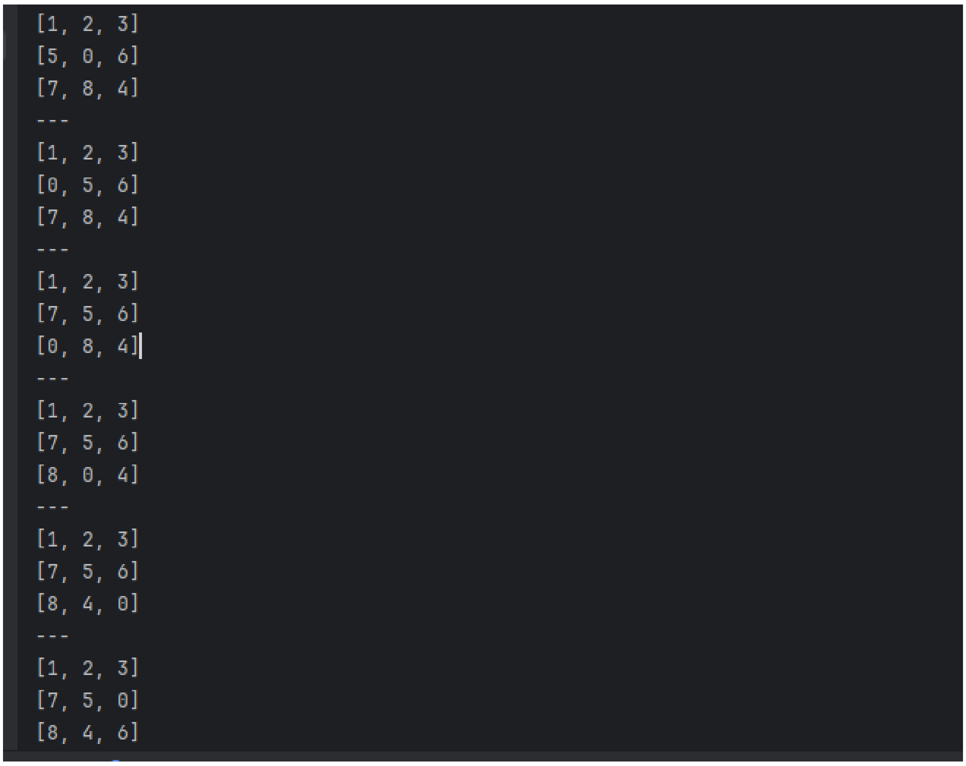
4) Move 9 to blank position

1 2 3
4 5 0
7 8 6

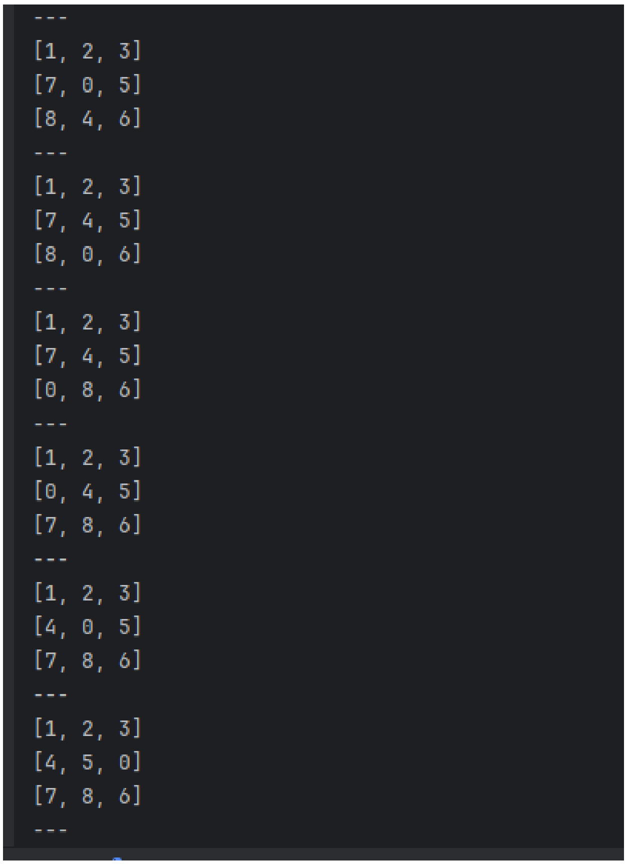
5) Move 6 to blank position (Goal reached)

1 2 3
4 5 6
7 8 0

Output:



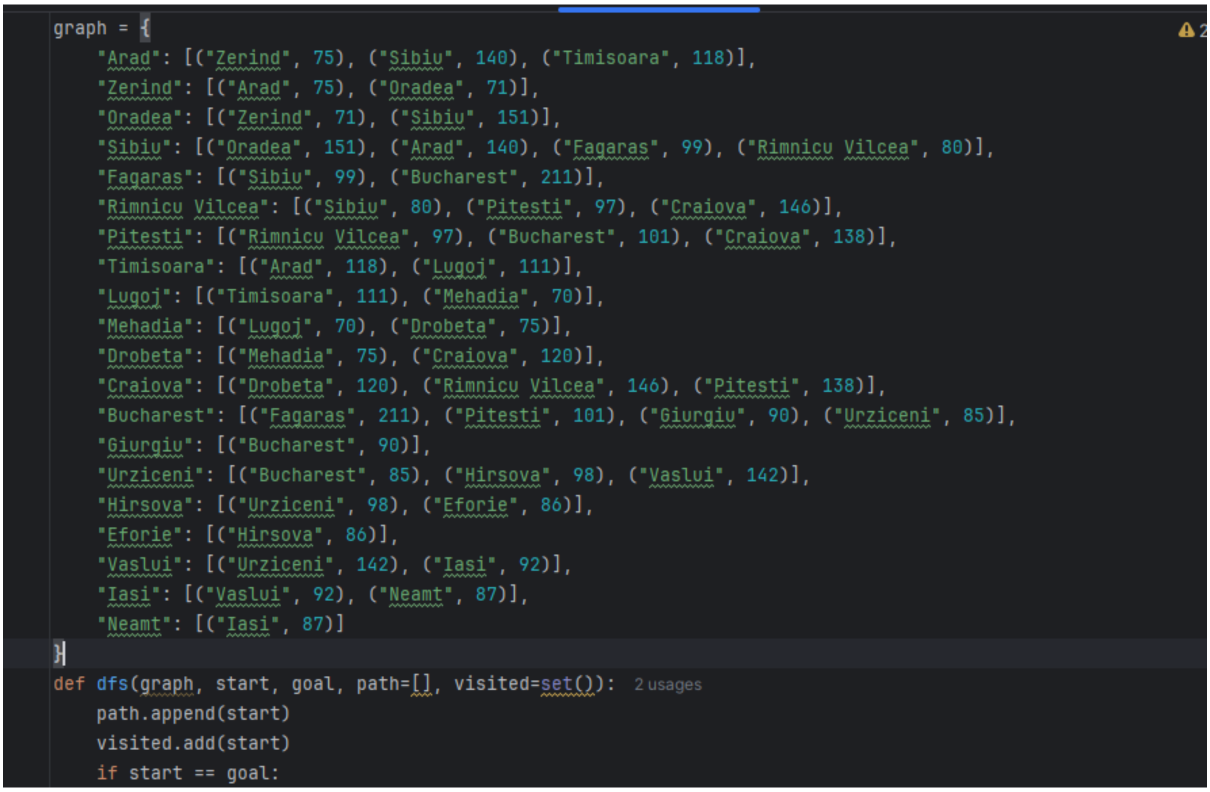
```
[1, 2, 3]
[5, 0, 6]
[7, 8, 4]
---
[1, 2, 3]
[0, 5, 6]
[7, 8, 4]
---
[1, 2, 3]
[7, 5, 6]
[0, 8, 4]
---
[1, 2, 3]
[7, 5, 6]
[8, 0, 4]
---
[1, 2, 3]
[7, 5, 6]
[8, 4, 0]
---
[1, 2, 3]
[7, 5, 0]
[8, 4, 6]
```



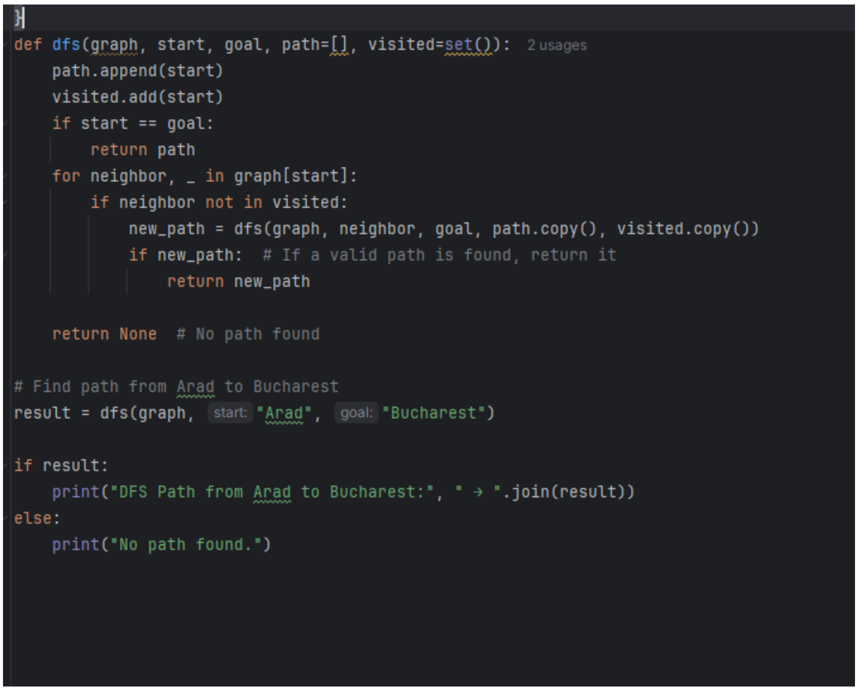
```
---
[1, 2, 3]
[7, 0, 5]
[8, 4, 6]
---
[1, 2, 3]
[7, 4, 5]
[8, 0, 6]
---
[1, 2, 3]
[7, 4, 5]
[0, 8, 6]
---
[1, 2, 3]
[0, 4, 5]
[7, 8, 6]
---
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]
---
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
---
```

Task#4

Solution:

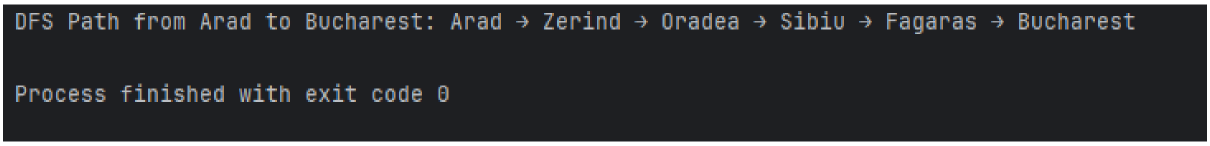


```
graph = {
    "Arad": [("Zerind", 75), ("Sibiu", 140), ("Timisoara", 118)],
    "Zerind": [("Arad", 75), ("Oradea", 71)],
    "Oradea": [("Zerind", 71), ("Sibiu", 151)],
    "Sibiu": [("Oradea", 151), ("Arad", 140), ("Fagaras", 99), ("Rimnicu Vilcea", 80)],
    "Fagaras": [("Sibiu", 99), ("Bucharest", 211)],
    "Rimnicu Vilcea": [("Sibiu", 80), ("Pitesti", 97), ("Craiova", 146)],
    "Pitesti": [("Rimnicu Vilcea", 97), ("Bucharest", 101), ("Craiova", 138)],
    "Timisoara": [("Arad", 118), ("Lugoj", 111)],
    "Lugoj": [("Timisoara", 111), ("Mehadia", 70)],
    "Mehadia": [("Lugoj", 70), ("Drobeta", 75)],
    "Drobeta": [("Mehadia", 75), ("Craiova", 120)],
    "Craiova": [("Drobeta", 120), ("Rimnicu Vilcea", 146), ("Pitesti", 138)],
    "Bucharest": [("Fagaras", 211), ("Pitesti", 101), ("Giurgiu", 90), ("Urziceni", 85)],
    "Giurgiu": [("Bucharest", 90)],
    "Urziceni": [("Bucharest", 85), ("Hirsova", 98), ("Vaslui", 142)],
    "Hirsova": [("Urziceni", 98), ("Eforie", 86)],
    "Eforie": [("Hirsova", 86)],
    "Vaslui": [("Urziceni", 142), ("Iasi", 92)],
    "Iasi": [("Vaslui", 92), ("Neamt", 87)],
    "Neamt": [("Iasi", 87)]
}
def dfs(graph, start, goal, path=[], visited=set()):
    path.append(start)
    visited.add(start)
    if start == goal:
```



```
}
def dfs(graph, start, goal, path=[], visited=set()):
    path.append(start)
    visited.add(start)
    if start == goal:
        return path
    for neighbor, _ in graph[start]:
        if neighbor not in visited:
            new_path = dfs(graph, neighbor, goal, path.copy(), visited.copy())
            if new_path:  # If a valid path is found, return it
                return new_path

    return None  # No path found

# Find path from Arad to Bucharest
result = dfs(graph, "Arad", "Bucharest")

if result:
    print("DFS Path from Arad to Bucharest:", " → ".join(result))
else:
    print("No path found.")
```

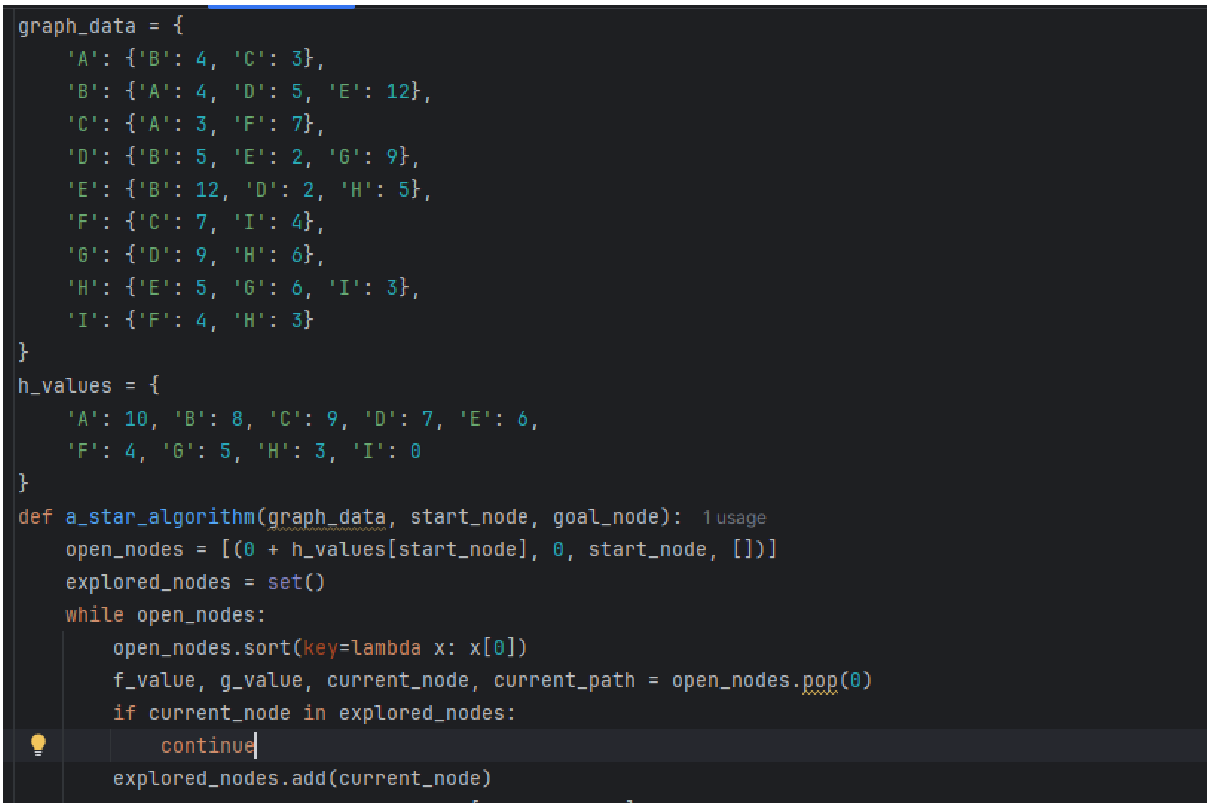
Output:



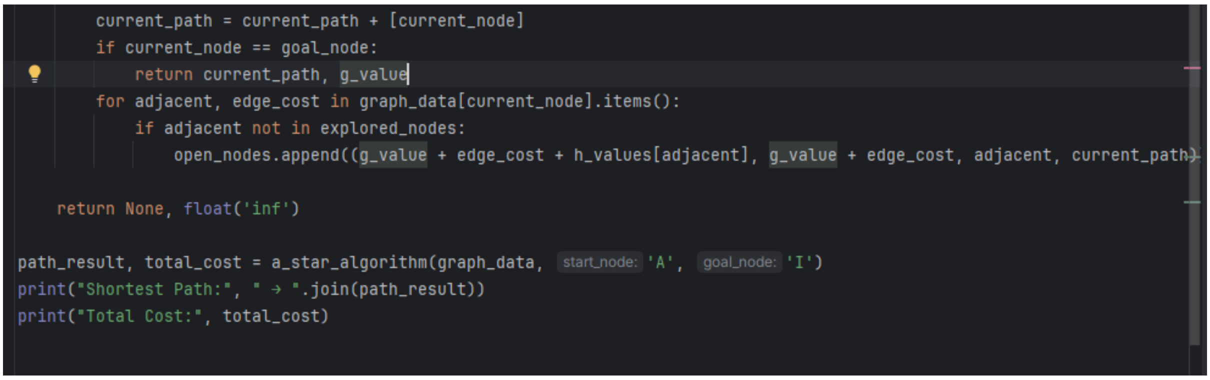
```
DFS Path from Arad to Bucharest: Arad → Zerind → Oradea → Sibiu → Fagaras → Bucharest

Process finished with exit code 0
```

Task#5

Solution:

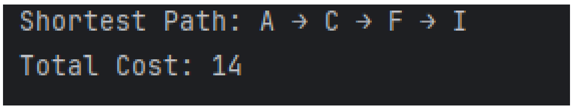


```
graph_data = {
    'A': {'B': 4, 'C': 3},
    'B': {'A': 4, 'D': 5, 'E': 12},
    'C': {'A': 3, 'F': 7},
    'D': {'B': 5, 'E': 2, 'G': 9},
    'E': {'B': 12, 'D': 2, 'H': 5},
    'F': {'C': 7, 'I': 4},
    'G': {'D': 9, 'H': 6},
    'H': {'E': 5, 'G': 6, 'I': 3},
    'I': {'F': 4, 'H': 3}
}
h_values = {
    'A': 10, 'B': 8, 'C': 9, 'D': 7, 'E': 6,
    'F': 4, 'G': 5, 'H': 3, 'I': 0
}
def a_star_algorithm(graph_data, start_node, goal_node):  1 usage
    open_nodes = [(0 + h_values[start_node], 0, start_node, [])]
    explored_nodes = set()
    while open_nodes:
        open_nodes.sort(key=lambda x: x[0])
        f_value, g_value, current_node, current_path = open_nodes.pop(0)
        if current_node in explored_nodes:
            continue
        explored_nodes.add(current_node)
```



```
        current_path = current_path + [current_node]
        if current_node == goal_node:
            return current_path, g_value
        for adjacent, edge_cost in graph_data[current_node].items():
            if adjacent not in explored_nodes:
                open_nodes.append((g_value + edge_cost + h_values[adjacent], g_value + edge_cost, adjacent, current_path)

    return None, float('inf')

path_result, total_cost = a_star_algorithm(graph_data, start_node: 'A', goal_node: 'I')
print("Shortest Path:", " → ".join(path_result))
print("Total Cost:", total_cost)
```

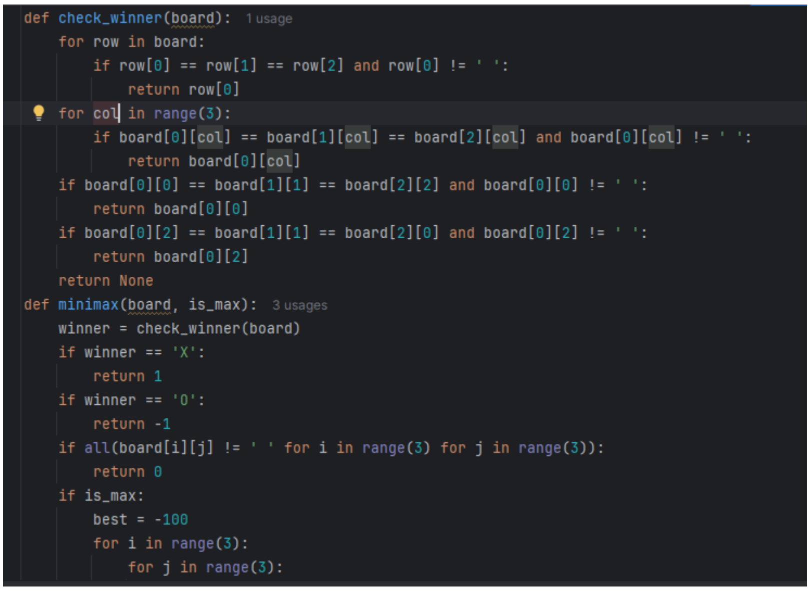
Output:



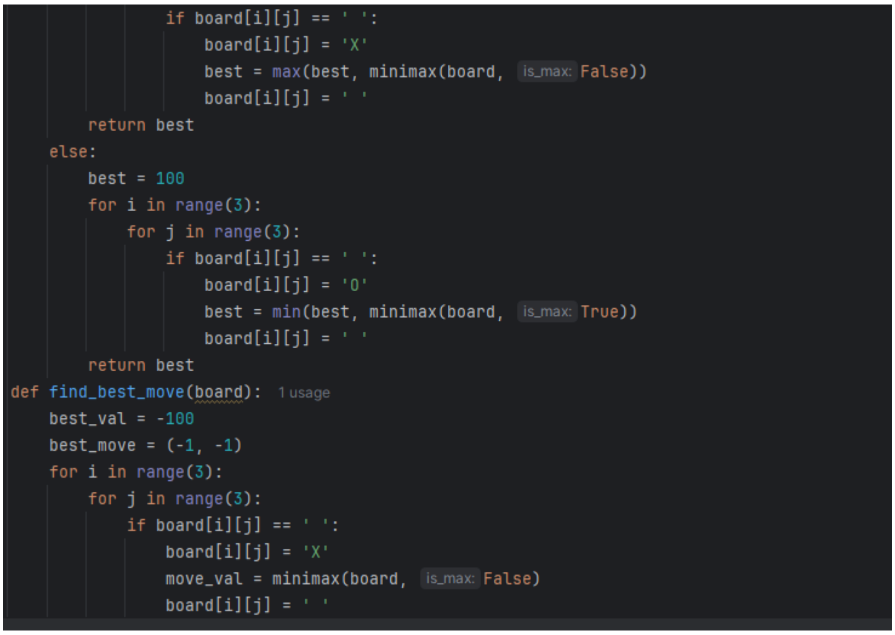
```
Shortest Path: A → C → F → I
Total Cost: 14
```

Task#6

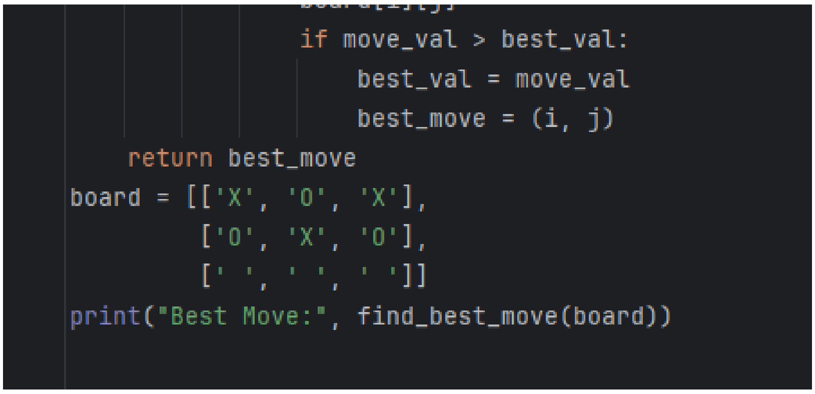
Solution:



```
def check_winner(board):  1 usage
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != ' ':
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != ' ':
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' ':
        return board[0][2]
    return None
def minimax(board, is_max):  3 usages
    winner = check_winner(board)
    if winner == 'X':
        return 1
    if winner == 'O':
        return -1
    if all(board[i][j] != ' ' for i in range(3) for j in range(3)):
        return 0
    if is_max:
        best = -100
        for i in range(3):
            for j in range(3):
```

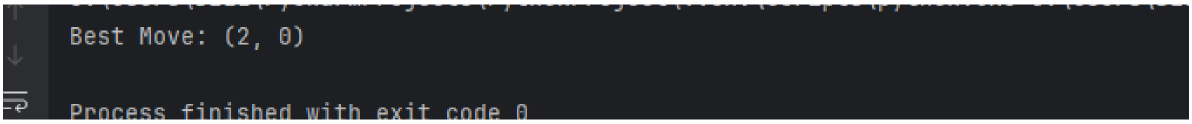


```
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    best = max(best, minimax(board,  is_max: False))
                    board[i][j] = ' '
        return best
    else:
        best = 100
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    best = min(best, minimax(board,  is_max: True))
                    board[i][j] = ' '
        return best
def find_best_move(board):  1 usage
    best_val = -100
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                move_val = minimax(board,  is_max: False)
                board[i][j] = ' '
```



```
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)
    return best_move
board = [['X', 'O', 'X'],
         ['O', 'X', 'O'],
         [' ', ' ', ' ']]
print("Best Move:", find_best_move(board))
```

Output:



```
Best Move: (2, 0)

Process finished with exit code 0
```