**RIPHAH INTERNATIONAL UNIVERSITY, ISLAMABAD**

****

**Lab 7**

**Bachelors of Computer science – 6th semester**

**Subject:** Artificial Intelligence Lab

**Submitted to:** Ma’am Ayesha

**Submitted by:** Munaza Malik

**Sap Id:** 44956

**Date:** 25th March, 2025

**Question 01:**

Write a program to traverse a graph using the shortest BFS algorithm.



**Code:**

from collections import deque

def bfs\_shortest\_path(graph, start, goal):

# Queue to store paths

queue = deque([[start]])

# Set to track visited nodes

visited = set()

while queue:

path = queue.popleft() # Get first path from the queue

node = path[-1] # Get last node from the path

if node == goal:

return path # Return the shortest path

if node not in visited:

visited.add(node)

for neighbor in graph.get(node, []):

new\_path = list(path) # Copy current path

new\_path.append(neighbor) # Append neighbor to path

queue.append(new\_path) # Add new path to queue

return None # If no path found

# Graph from the image

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"]

}

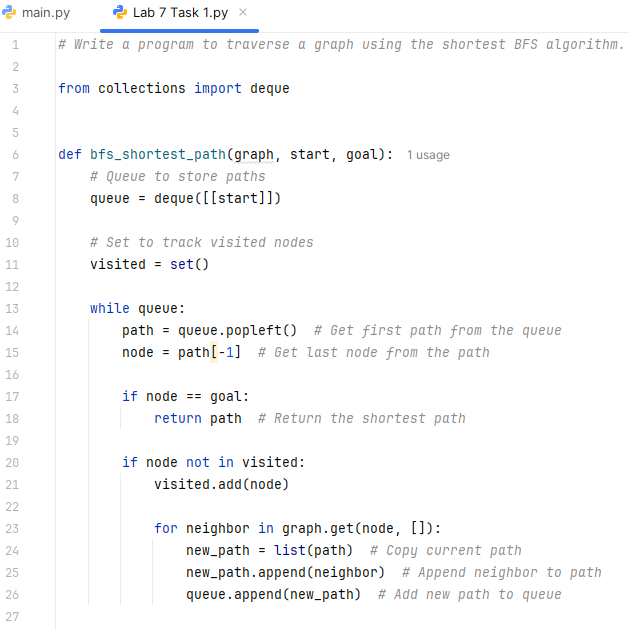
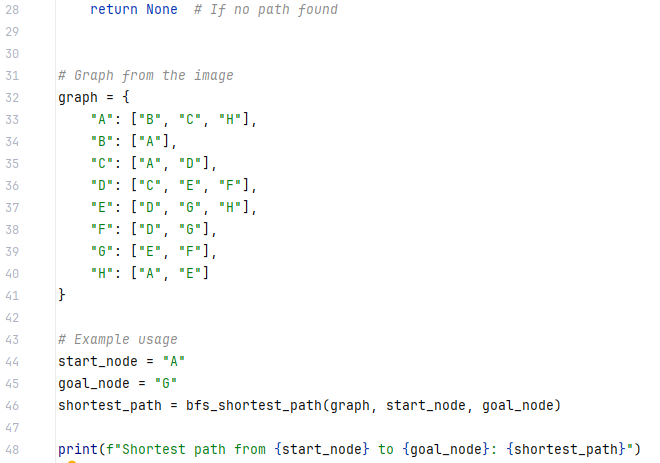
# Example usage

start\_node = "A"

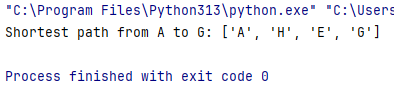
goal\_node = "G"

shortest\_path = bfs\_shortest\_path(graph, start\_node, goal\_node)

print(f"Shortest path from {start\_node} to {goal\_node}: {shortest\_path}")

**Output:**



**Question 02:**

Write a program for Depth First Search on the graph below



**Code:**

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

if visited is None:

visited = set()

visited.add(start)

print(start, end=" ") # Print visited node

for neighbor in graph.get(start, []):

if neighbor not in visited:

dfs(graph, neighbor, visited)

# Graph from the image

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"]

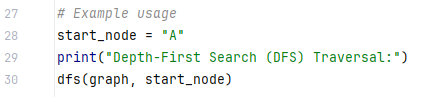
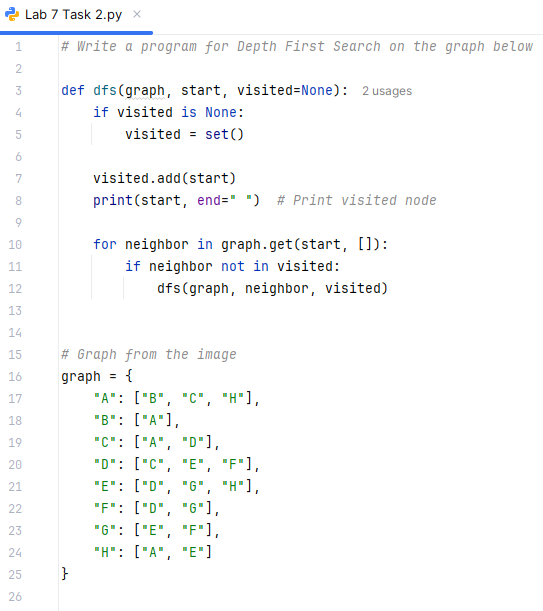
}

# Example usage

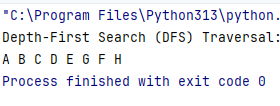
start\_node = "A"

print("Depth-First Search (DFS) Traversal:")

dfs(graph, start\_node)



**Output:**



**Question 03:**

**8-puzzle problem:**

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space

* Solve this problem using the BFS algorithm in python.
* Take an example matrix of 3x3 and a goal matrix of 3x3.
* Must give a dry run of your example

**Code:**

from collections import deque

class Puzzle:

def \_\_init\_\_(self, initial, goal):

self.initial = initial # Initial puzzle state

self.goal = goal # Goal state

self.rows, self.cols = 3, 3 # Puzzle size

def find\_blank(self, state):

"""Find the position of the blank (0) in the 3x3 puzzle."""

for i in range(self.rows):

for j in range(self.cols):

if state[i][j] == 0:

return i, j # Return row, col of blank space

def move(self, state, x, y, new\_x, new\_y):

"""Move the blank space in the given direction."""

new\_state = [row[:] for row in state] # Deep copy

new\_state[x][y], new\_state[new\_x][new\_y] = new\_state[new\_x][new\_y], new\_state[x][y]

return new\_state

def get\_neighbors(self, state):

"""Generate all possible moves from the current state."""

x, y = self.find\_blank(state)

moves = []

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right

for dx, dy in directions:

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < self.rows and 0 <= new\_y < self.cols: # Check boundaries

moves.append(self.move(state, x, y, new\_x, new\_y))

return moves

def bfs(self):

"""Solve the puzzle using BFS."""

queue = deque([(self.initial, [])]) # (state, path)

visited = set()

while queue:

current\_state, path = queue.popleft()

if current\_state == self.goal:

return path + [current\_state] # Solution found

state\_tuple = tuple(map(tuple, current\_state))

if state\_tuple in visited:

continue # Skip already visited states

visited.add(state\_tuple)

for neighbor in self.get\_neighbors(current\_state):

queue.append((neighbor, path + [current\_state]))

return None # No solution found

def print\_solution(self, solution):

"""Print the steps to reach the goal state."""

for step, state in enumerate(solution):

print(f"Step {step}:")

for row in state:

print(row)

print()

# Example Initial and Goal State

initial\_state = [

[1, 2, 3],

[0, 4, 6],

[7, 5, 8]

]

goal\_state = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 0]

]

# Solve the puzzle

puzzle = Puzzle(initial\_state, goal\_state)

solution = puzzle.bfs()

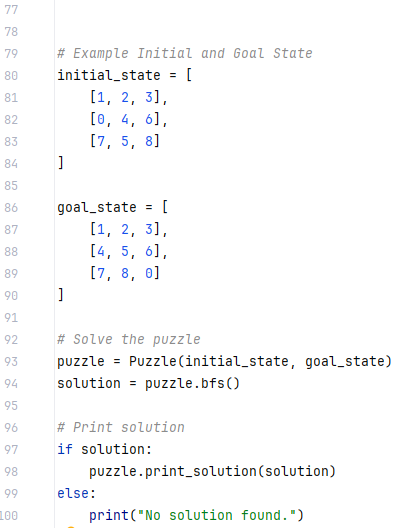
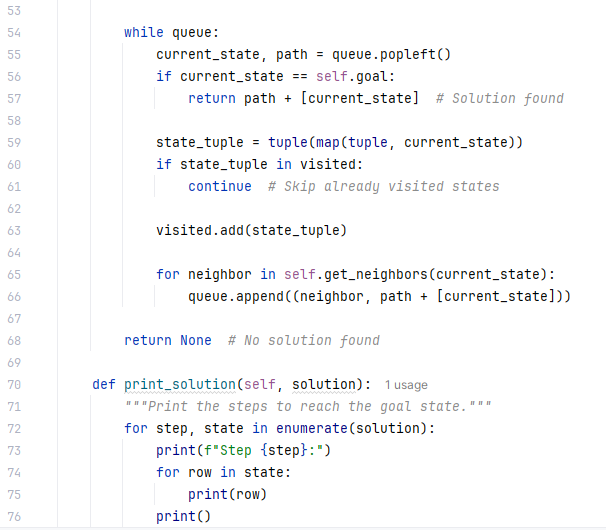
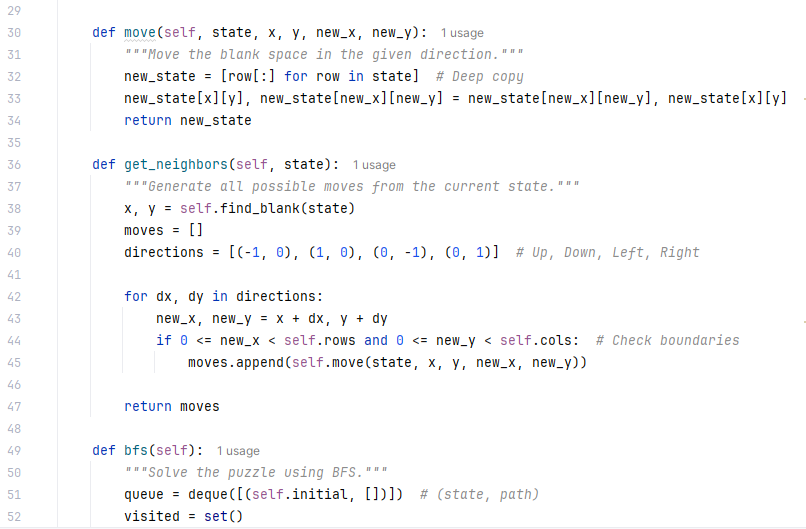
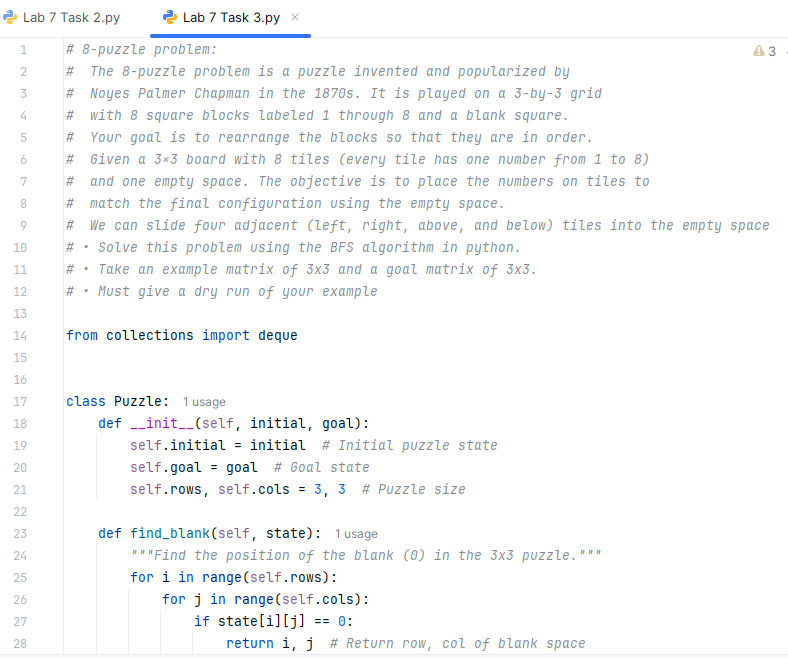
# Print solution

if solution:

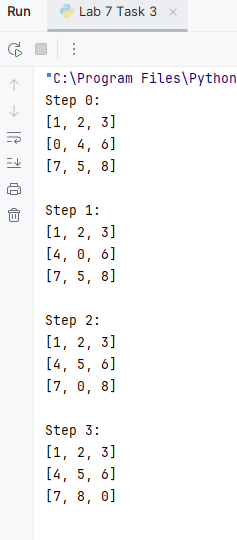
puzzle.print\_solution(solution)

else:

print("No solution found.")

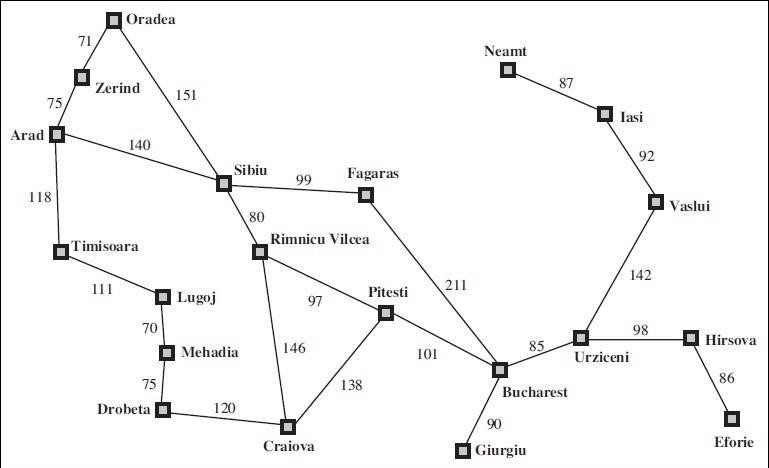


**Output:**



**Question 04:**

Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a depth first search to find the corresponding path.



**Code:**

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

if path is None:

path = []

if visited is None:

visited = set()

path.append(start)

visited.add(start)

if start == goal:

return path # If we reach the goal, return the path

for neighbor in graph.get(start, []):

if neighbor[0] not in visited:

new\_path = dfs(graph, neighbor[0], goal, path.copy(), visited.copy())

if new\_path: # If a valid path is found, return it

return new\_path

return None # If no path is found

# Romania Map as an Adjacency List with distances

romania\_map = {

"Arad": [("Zerind", 75), ("Timisoara", 118), ("Sibiu", 140)],

"Zerind": [("Arad", 75), ("Oradea", 71)],

"Oradea": [("Zerind", 71), ("Sibiu", 151)],

"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)],

"Sibiu": [("Arad", 140), ("Oradea", 151), ("Fagaras", 99), ("Rimnicu Vilcea", 80)],

"Fagaras": [("Sibiu", 99), ("Bucharest", 211)],

"Rimnicu Vilcea": [("Sibiu", 80), ("Craiova", 146), ("Pitesti", 97)],

"Pitesti": [("Rimnicu Vilcea", 97), ("Craiova", 138), ("Bucharest", 101)],

"Bucharest": [("Fagaras", 211), ("Pitesti", 101), ("Urziceni", 85), ("Giurgiu", 90)],

"Giurgiu": [("Bucharest", 90)],

"Urziceni": [("Bucharest", 85), ("Vaslui", 142), ("Hirsova", 98)],

"Hirsova": [("Urziceni", 98), ("Eforie", 86)],

"Eforie": [("Hirsova", 86)],

"Vaslui": [("Urziceni", 142), ("Iasi", 92)],

"Iasi": [("Vaslui", 92), ("Neamt", 87)],

"Neamt": [("Iasi", 87)]

}

# Find the path using DFS

start\_city = "Arad"

goal\_city = "Bucharest"

dfs\_path = dfs(romania\_map, start\_city, goal\_city)

# Print the DFS path

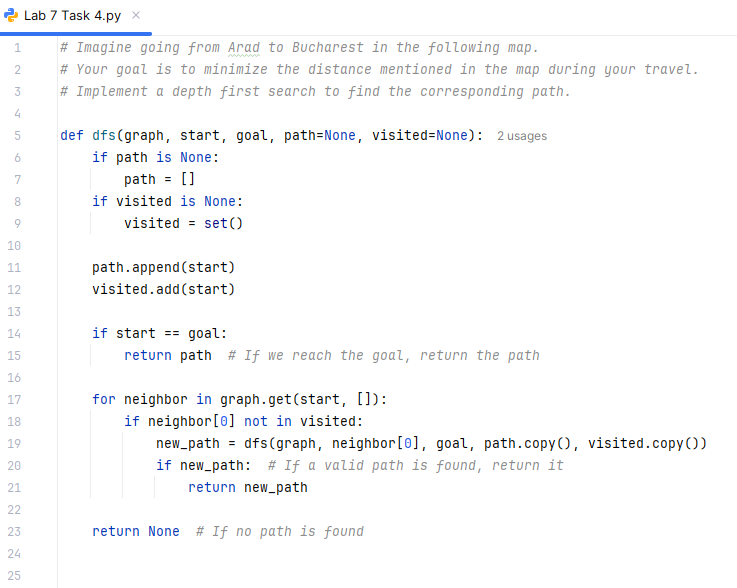
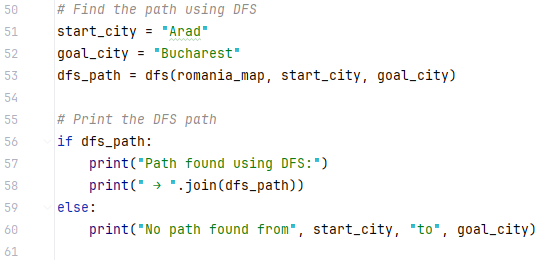
if dfs\_path:

print("Path found using DFS:")

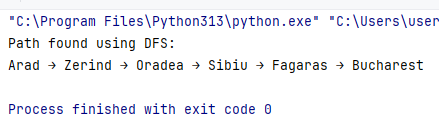
print(" → ".join(dfs\_path))

else:

print("No path found from", start\_city, "to", goal\_city)

**Output:**



**Question 05:**

Create a graph with weighted edges.

Implement A\* to find the shortest path between two nodes.

**Question 06:**

Implement a Basic Minimax for Tic-Tac-Toe

 Create a **3x3 Tic-Tac-Toe board**.

 Use **Minimax** to find the best move for a player.

 Assume 'X' is the maximizer and 'O' is the minimizer.

 Use a recursive function that assigns **+1 (win), -1 (loss), or 0 (draw)**.

 Implement **a function to check winning conditions**.

**Code:**

import heapq

def a\_star(graph, start, goal, heuristics):

priority\_queue = []

heapq.heappush(priority\_queue, (0, start)) # (cost, node)

came\_from = {start: None}

cost\_so\_far = {start: 0}

while priority\_queue:

current\_cost, current\_node = heapq.heappop(priority\_queue)

if current\_node == goal:

break

for neighbor, weight in graph[current\_node]:

new\_cost = cost\_so\_far[current\_node] + weight

if neighbor not in cost\_so\_far or new\_cost < cost\_so\_far[neighbor]:

cost\_so\_far[neighbor] = new\_cost

priority = new\_cost + heuristics[neighbor] # A\* Formula: g(n) + h(n)

heapq.heappush(priority\_queue, (priority, neighbor))

came\_from[neighbor] = current\_node

# Reconstruct path

path = []

node = goal

while node:

path.append(node)

node = came\_from[node]

path.reverse()

return path, cost\_so\_far[goal]

# Graph with weighted edges

graph = {

"A": [("B", 4), ("C", 2)],

"B": [("A", 4), ("D", 5)],

"C": [("A", 2), ("D", 8), ("E", 10)],

"D": [("B", 5), ("C", 8), ("E", 2), ("F", 6)],

"E": [("C", 10), ("D", 2), ("F", 3)],

"F": [("D", 6), ("E", 3)]

}

# Example heuristic values (straight-line distance estimate to 'F')

heuristics = {

"A": 7, "B": 6, "C": 5, "D": 3, "E": 1, "F": 0

}

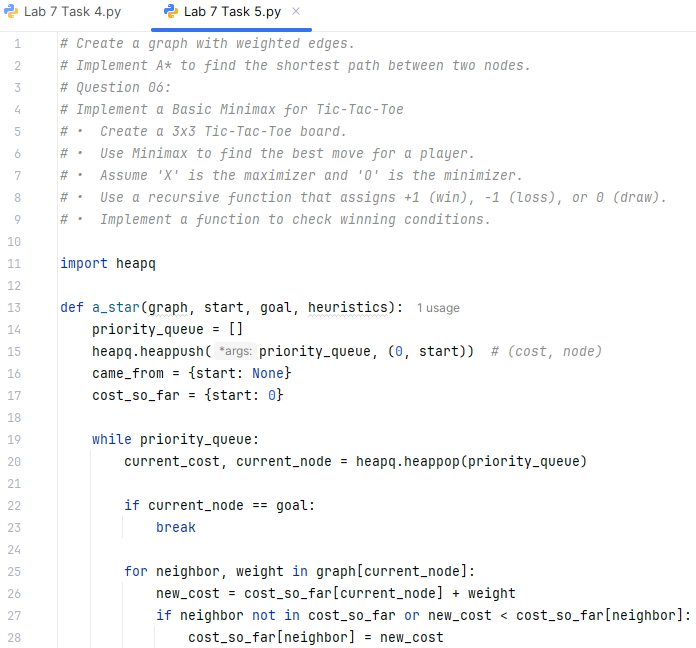
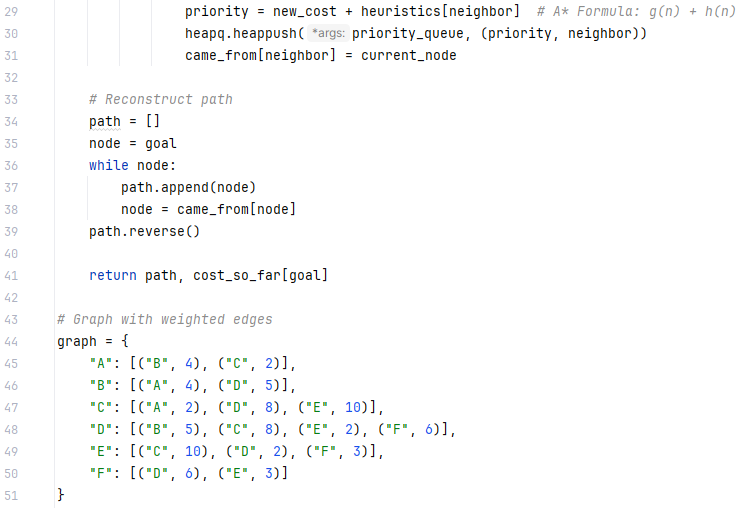
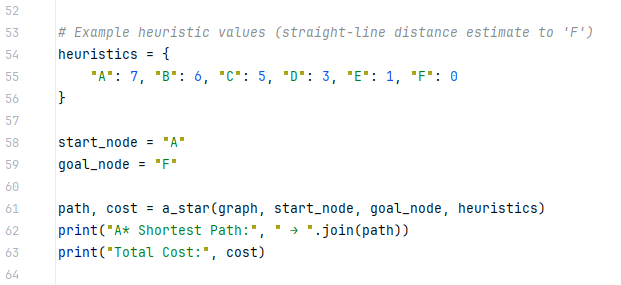
start\_node = "A"

goal\_node = "F"

path, cost = a\_star(graph, start\_node, goal\_node, heuristics)

print("A\* Shortest Path:", " → ".join(path))

print("Total Cost:", cost)

**Output:**

