#MUHAMMAD HUZAIFA KHILJI

#f24-002

#LAB\_7

import heapq  # Import the heapq module for priority queue implementation

class Node:

    """

    Represents a node in the graph for A\* search.

    """

    def \_\_init\_\_(self, data, h\_cost):

        self.data = data

        self.neighbors = []  # List of neighboring nodes (tuples: (neighbor\_node, edge\_cost))

        self.h\_cost = h\_cost  # Heuristic cost (estimated cost to goal)

        self.g\_cost = 0     # Cost from the start node to this node

        self.f\_cost = 0     # Total cost (g\_cost + h\_cost)

    def \_\_lt\_\_(self, other):

        """

        Comparison method for priority queue (heapq).  Compares f\_costs.

        """

        return self.f\_cost < other.f\_cost

    def \_\_repr\_\_(self):

        return f"Node({self.data}, h={self.h\_cost}, g={self.g\_cost}, f={self.f\_cost})"

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

    """

    Performs A\* search on a graph.

    Args:

        start\_node: The starting node.

        goal\_node: The goal node.

    """

    open\_set = [start\_node]  # Priority queue (heapq) of nodes to explore

    closed\_set = set()      # Set of nodes already evaluated

    path = []

    start\_node.g\_cost = 0

    start\_node.f\_cost = start\_node.h\_cost  # f = g + h

    print("A\* Search:")

    while open\_set:

        # Get the node with the lowest f\_cost from the open set

        current\_node = heapq.heappop(open\_set)

        if current\_node == goal\_node:

            print("Goal Node Found!")

            # Reconstruct the path from start to goal

            while current\_node:

                path.append(current\_node.data)

                # Backtrack.  In a full implementation, you'd store the 'came\_from'

                # attribute in each node during the search.  For this simplified

                # example, we assume the graph structure allows us to reach

                # the start by going through the parents.

                if current\_node == start\_node:

                    break

                # A\* doesn't store parent directly.  This is a simplified way to reconstruct.

                for node in closed\_set: # VERY INEFFICIENT WAY TO BACKTRACK.

                    for neighbor, edge\_cost in node.neighbors:

                        if neighbor == current\_node:

                            current\_node = node

                            break

                    else:

                        continue

                    break

            print("Path:", path[::-1])  # Reverse the path to get start -> goal

            return

        closed\_set.add(current\_node)  # Move current node to the closed set

        for neighbor, edge\_cost in current\_node.neighbors:

            tentative\_g\_cost = current\_node.g\_cost + edge\_cost

            if neighbor in closed\_set and tentative\_g\_cost >= neighbor.g\_cost:

                continue  # Skip if neighbor is in closed set with a lower cost

            if neighbor not in closed\_set or tentative\_g\_cost < neighbor.g\_cost:

                neighbor.g\_cost = tentative\_g\_cost

                neighbor.f\_cost = tentative\_g\_cost + neighbor.h\_cost

                # In a proper A\* implementation, you would also set

                # neighbor.came\_from = current\_node  # To track the path

                if neighbor not in open\_set:

                    heapq.heappush(open\_set, neighbor)  # Add neighbor to the open set

                    #print(f"Added to open set: {neighbor}")

# --- Example Graph for A\* ---

# Heuristic costs (h\_cost) are estimates of the distance to the goal (G)

node\_a = Node("A", h\_cost=10)

node\_b = Node("B", h\_cost=8)

node\_c = Node("C", h\_cost=5)

node\_d = Node("D", h\_cost=7)

node\_e = Node("E", h\_cost=3)

node\_f = Node("F", h\_cost=6)

node\_g = Node("G", h\_cost=0)  # Goal node

# Connect nodes with edge costs (tuples: (neighbor\_node, edge\_cost))

node\_a.neighbors = [(node\_b, 2), (node\_c, 9), (node\_d, 15)]

node\_b.neighbors = [(node\_a, 2), (node\_e, 10)]

node\_c.neighbors = [(node\_a, 9), (node\_f, 1)]

node\_d.neighbors = [(node\_a, 15), (node\_g, 15)]

node\_e.neighbors = [(node\_b, 10), (node\_f, 5), (node\_g, 8)]

node\_f.neighbors = [(node\_c, 1), (node\_e, 5), (node\_g, 5)]

node\_g.neighbors = []  # Goal node has no neighbors

# Perform A\* search from A to G

a\_star\_search(node\_a, node\_g)  # Output: A B E G