**Name :Sahil Kishor Shirodkar TY-CS-D BATCH 2 ROLL NO :06**

**Assignment 3:INFORMED SEARCH (ASTAR ALGORITHM).**

**CODE:-**

import java.util.\*;

class Node {

int id;

int g;

int h;

Node parent;

Node(int id, int g, int h) {

this.id = id;

this.g = g;

this.h = h;

this.parent = null;

}

}

class Graph {

private Map<Integer, List<int[]>> graph = new HashMap<>();

void addEdge(int from, int to, int weight) {

if (!graph.containsKey(from)) {

graph.put(from, new ArrayList<>());

}

graph.get(from).add(new int[]{to, weight});

}

List<int[]> getNeighbors(int node) {

return graph.getOrDefault(node, Collections.emptyList());

}

}

public class Assignment3 {

public static void main(String[] args) {

Graph graph = new Graph();

graph.addEdge(1,2,4);

graph.addEdge(1,3,2);

graph.addEdge(2,3,5);

graph.addEdge(2,4,10);

graph.addEdge(3,5,3);

graph.addEdge(5,4,4);

graph.addEdge(4,6,11);

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the starting node: ");

int startNode = scanner.nextInt();

System.out.print("Enter the destination node: ");

int goalNode = scanner.nextInt();

List<Integer> path = findShortestPath(graph, startNode, goalNode);

if (path != null) {

System.out.println("Shortest Path from " + startNode + " to " + goalNode + ": " + path);

} else {

System.out.println("No path found from " + startNode + " to " + goalNode);

}

scanner.close();

}

static List<Integer> findShortestPath(Graph graph, int startNode, int goalNode) {

PriorityQueue<Node> openList = new PriorityQueue<>(Comparator.comparingInt(node -> node.g + node.h));

Map<Integer, Integer> gValues = new HashMap<>();

openList.add(new Node(startNode, 0, heuristic(startNode, goalNode)));

gValues.put(startNode, 0);

while (!openList.isEmpty()) {

Node currentNode = openList.poll();

if (currentNode.id == goalNode) {

return reconstructPath(currentNode);

}

for (int[] neighborEdge : graph.getNeighbors(currentNode.id)) {

int neighborNode = neighborEdge[0];

int edgeWeight = neighborEdge[1];

int tentativeG = currentNode.g + edgeWeight;

int heuristicvalue= heuristic(neighborNode,goalNode);

if (!gValues.containsKey(neighborNode) || tentativeG < gValues.get(neighborNode)) {

gValues.put(neighborNode, tentativeG);

Node neighbor = new Node(neighborNode, tentativeG, heuristic(neighborNode, goalNode));

neighbor.parent = currentNode;

openList.add(neighbor);

}

}

}

return null;

}

static List<Integer> reconstructPath(Node goalNode) {

List<Integer> path = new ArrayList<>();

Node current = goalNode;

while (current != null) {

path.add(current.id);

current = current.parent;

}

Collections.reverse(path);

return path;

}

static int heuristic(int node, int goalNode) {

return Math.abs(node - goalNode);

}

}