Pract 2

```
import java.util.*;

// Represents a node in the graph
class Node {
    String id; // ID of the node
    int heuristicValue; // Heuristic value for A* algorithm
    List<Edge> edges = new ArrayList<>(); // List of edges connected to this node

    // Constructor
    Node(String id, int heuristicValue) {
        this.id = id;
        this.heuristicValue = heuristicValue;
    }

    // Method to add an edge from this node to another node
    void addEdge(Node node, int weight) {
        edges.add(new Edge(node, weight));
    }

    // Override toString method to print node ID
    @Override
    public String toString() {
        return id;
    }
}

// Represents an edge between two nodes
class Edge {
    Node node; // Destination node
    int weight; // Weight of the edge

    // Constructor
    Edge(Node node, int weight) {
        this.node = node;
        this.weight = weight;
    }
}

// Class implementing the A* algorithm
class AStar {
    // Comparator for comparing nodes based on combined cost (gScore + heuristicValue)
    static class NodeComparator implements Comparator<Node> {
        Map<Node, Integer> gScore; // Map to store gScores of nodes

        // Constructor
        NodeComparator(Map<Node, Integer> gScore) {
            this.gScore = gScore;
        }

        // Compare method
        @Override
        public int compare(Node node1, Node node2) {
            return (gScore.get(node1) + node1.heuristicValue) - (gScore.get(node2) +
node2.heuristicValue);
        }
    }
```

```
// A* algorithm implementation
static List<Node> aStarAlgo(Node startNode, Node stopNode) {
   Set<Node> openSet = new HashSet<>(); // Set of open nodes
   Set<Node> closedSet = new HashSet<>(); // Set of closed nodes
   Map<Node, Integer> gScore = new HashMap<>(); // Map to store gScores of nodes
   Map<Node, Node> parents = new HashMap<>(); // Map to store parent nodes

   // Initialize gScore and parents maps
   gScore.put(startNode, 0);
   parents.put(startNode, startNode);

   // Add start node to open set
   openSet.add(startNode);

   // Loop until open set is empty
   while (!openSet.isEmpty()) {
      // Get node with minimum combined cost from open set
      Node n = Collections.min(openSet, new NodeComparator(gScore));

      // Check if stop node is reached
      if (n == stopNode) {
         // Reconstruct and return the path
         List<Node> path = new ArrayList<>();
         while (parents.get(n) != n) {
            path.add(n);
            n = parents.get(n);
         }
         path.add(startNode);
         Collections.reverse(path);
         System.out.println("Path found: " + path);
         return path;
      }

      // Remove current node from open set and add to closed set
      openSet.remove(n);
      closedSet.add(n);

      // Explore neighbors of current node
      for (Edge edge : n.edges) {
         Node m = edge.node;
         int weight = edge.weight;

         // Skip if neighbor is in closed set
         if (closedSet.contains(m)) {
            continue;
         }

         // Calculate tentative gScore for neighbor
         int tentativeGScore = gScore.get(n) + weight;

         // If neighbor is not in open set, add it
         if (!openSet.contains(m)) {
            openSet.add(m);
         } else if (tentativeGScore >= gScore.getOrDefault(m, Integer.MAX_VALUE)) {
            // If tentative gScore is not better than current gScore, skip
            continue;
```

```
            }

            // Update parent and gScore for neighbor
            parents.put(m, n);
            gScore.put(m, tentativeGScore);
          }
        }

        // If open set becomes empty, no path exists
        System.out.println("Path does not exist!");
        return null;
    }
}

public class Main {
    public static void main(String[] args) {
        // Create nodes representing vertices of the graph
        Node A = new Node("A", 11);
        Node B = new Node("B", 6);
        Node C = new Node("C", 5);
        Node D = new Node("D", 7);
        Node E = new Node("E", 3);

        // Add edges between nodes
        A.addEdge(B, 6);
        A.addEdge(D, 3);
        B.addEdge(A, 6);
        B.addEdge(C, 3);
        B.addEdge(D, 2);
        C.addEdge(B, 3);
        C.addEdge(D, 1);
        C.addEdge(E, 5);
        D.addEdge(B, 2);
        D.addEdge(C, 1);
        D.addEdge(E, 8);
        E.addEdge(C, 5);
        E.addEdge(D, 8);

        // Call A* algorithm with start node A and stop node E
        AStar.aStarAlgo(A, E);
    }
}
```