```
import java.io.*;
import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.*;
import java.util.Scanner;

public class DijkstrasAlgorithm {
    public static void main(String[] args) throws IOException {
        // Read input from file and base parameters like # of vertices and edges,
and current source
        Scanner scanner = new Scanner(new File("cop3503-dijkstra-input.txt"));
        int vertices = scanner.nextInt();
        int source = scanner.nextInt() - 1; // Convert to zero-indexed
        int edges = scanner.nextInt();

        // setting up our data structure to store graph--a double list such that
each node represented in a list,
        // and each node has another list for neighbors, and each neighbor has a
int array storing parent and cost
        List<List<int[]>> graph = new ArrayList<>();
        for (int i = 0; i < vertices; i++) {
            graph.add(new ArrayList<>());
        }

        // initializing graph by reading input and also storing neighbors and costs
        for (int i = 0; i < edges; i++) {
            int from = scanner.nextInt() - 1;
            int to = scanner.nextInt() - 1;
            int weight = scanner.nextInt();
            graph.get(from).add(new int[]{to, weight});
            graph.get(to).add(new int[]{from, weight}); // Doubly-connected
        }

        // create and initialize int arrays to store the dijkstra values like
distance and parent
        int[] distance = new int[vertices];
        int[] parent = new int[vertices];
        Arrays.fill(distance, Integer.MAX_VALUE);
        Arrays.fill(parent, -1);
        distance[source] = 0;

        // create priority queue that compares the weight in the int array to
implement iteration of djikstra
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a ->
a[1]));
        pq.add(new int[]{source, 0});

        // iterating until all nodes included
        while (!pq.isEmpty()) {
            // get the node with the lowest cost
            int[] current = pq.poll();
            int node = current[0];
            int dist = current[1];

            // if distance is greater than the current node, continue
            if (dist > distance[node]) continue;

            // for each neighbor of this current node . . .
            for (int[] neighbor : graph.get(node)) {
```

```
                  // get distance through current node
                int neighborNode = neighbor[0];
                int weight = neighbor[1];
                int newDist = distance[node] + weight;

                // if the distance is less than the current distance
                if (newDist < distance[neighborNode] ||
                    (newDist == distance[neighborNode] && node <
parent[neighborNode])) {
                  // replace current distance and replace neighbor parent and add
to iteration
                    distance[neighborNode] = newDist;
                    parent[neighborNode] = node;
                    pq.add(new int[]{neighborNode, newDist});
                }
            }
        }

        // Open output file
        PrintWriter writer = new PrintWriter(new File("cop3503-dijkstra-output-
bathina-kashyap.txt"));
        writer.println(vertices);

        // Print each node in the format: node distance parent
        for (int i = 0; i < vertices; i++) {
            int nodeId = i + 1;
            int nodeDistance = (distance[i] == Integer.MAX_VALUE) ? -1 :
distance[i];
            int nodeParent = (parent[i] == -1) ? -1 : (parent[i] + 1);

            // Check if the node is the source; it should have -1 for both parent
and distance
            if (i == source) {
                writer.println(nodeId + " -1 -1");
            } else { // else just print the node, its distance, and parent
                writer.println(nodeId + " " + nodeDistance + " " + nodeParent);
            }
        }

        scanner.close();
        writer.close();
    }
}
```