ADA BONUS PROBLEM

Ayushi Jain, 2019031
Kesar Shrivastava, 2019051

Algorithm using Priority Queue

The shortest path in the given graph includes every other vertex in the graph.

The algorithm for finding the shortest path when each of the edge fails is as follows:
We will first apply Dijkstra on the graph and store the shortest distance of every vertex from s in an array. This is the usual Dijkstra in which we push the nodes in the priority queue using the distance from the source vertex as the label.

Now, we have the shortest distance of every vertex. From this we can get the path from s to t. This will be the shortest path between s and t.

There are two cases for the edge that fails

- The edge is not in the shortest s-t path. In this case the removal of that edge does not create any difference and the shortest path remains the same.
- The edge is in the shortest s-t path. In this case the length of the path will increase.

For the second case while pushing the nodes in the priority queue we need to make sure that the edge that has failed does not get added to the priority queue. **This can be done by taking into consideration the nodes that we are adding to the priority queue that are connected to the existing set of vertices S, by any edge other than the one we are excluding.** However, this time we changed the condition of the priority queue, as opposed to normal dijkstra. This time the condition is the total path length

(minimum). Suppose we are considering edge e = (u,v), the label would be the ds(u) + dt(v) + length(u - v). (ds(u) and dt(v) is the distance between s and u, and v and t respectively)
In this way we run dijkstra two times whose time complexity is O(E log V), hence the complexity of the algorithm is O(E log V).

The algorithm works because the given graph has the following properties:

- Shortest s-t path has all vertices in the graph
- All edge weights are positive

So, as all edge weights are positive, this means that we can apply dijkstra on the graph, to store all the minimum weight distances from s for all the vertices.

So, say the shortest path between S and T is of the form:

S -- 1 -- 2 -- 3 -- 4 -- 5 -- T

So, we will do the following steps to achieve the objective of the question:

- ➔ Take a hashmap named answer that stores the shortest distance between S and T corresponding to the situation where the edge is removed.
- ➔ Now we iterate through the list. So now we have two sets A and V - A. Initially A is empty. First we include S in A. So, we will check for the node that is reachable from all nodes in A, without the edge [S - 1] and has minimum total path value, say x: (dist[S - x] + dist[x - T] + edge[S - x]), where dist[x - T] = dist[S - T] - dist[S - x]. Fetching the node x will take O(log (V)) time as we are using a priority queue. As we do this step for all edges in the shortest path, which can be <= E, the total time complexity is O(E log V). Then the total distance between S and T without the edge [S - 1] is t = d + dist[T] - dist[x]. Please note that doing this is possible only because of condition 1, and the Dijkstra algorithm. So we put the pair {[S - 1]: t} in the hashmap.

➔ Similarly, in the next step, we will check for [1 - 2], then [2 - 3] and so on.

PSEUDO CODE

```
public class Main
{
   public static void dijkstra_1(int[][] grid, Node[] nodes,
int[] dist, int src, int destination){
        Arrays.fill(dist, Integer.MAX_VALUE);
        PriorityQueue<Node> pq = new PriorityQueue<>(new
Comparator<Node>(){
            public int compare (Node a1, Node a2){
                return Integer.compare(a1.path, a2.path);
            }
        });
        pq.add(nodes[src]);
        HashSet<Integer> set = new HashSet<>();
        nodes[src].dist = 0;
        while (!pq.isEmpty()){
            Node k = pq.poll();
            set.add(k.value);
            for (Integer lk : k.list){
                if (!set.contains(lk)){
                    nodes[lk].dist = Math.min(k.dist +
grid[lk][k.value], nodes[lk].dist);
                    dist[lk] = nodes[lk].dist;
                    pq.add(nodes[lk]);

                }
            }
        }
   }
```

```
   public static void dijkstra_2( int[][] grid, HashMap<String,
Integer> hash, ArrayList<int[]> edges, int[] dist, Node[] nodes,
int source, int destination){
   Node[] arr1 = new Node[dist.length];
   int n = nodes.length;
   for (int i = 0; i < nodes.length; i ++){ arr1[i] = nodes[i];}
   Arrays.sort(arr1, new Comparator<Node>(){
       public int compare(Node a1, Node a2){
           return Integer.compare(a1.dist, a2.dist);}
   });
   PriorityQueue<Node> pq =new PriorityQueue<>(new
Comparator<Node>() {
        @Override
        public int compare(Node o1, Node o2) {
            return o1.path - o2.path;
        }
   });
   HashSet<Integer> set = new HashSet<>();
   for (int i = 0; i < n - 1; i++){
       int k1 = nodes[i].value;
       int k2 = nodes[i + 1].value;

       for (int r: nodes[i].list){
           if (!(nodes[i].value == k1 && nodes[i + 1].value ==
k2)){
//to avoid considering the edge from i to i + 1 th node in
sequence.
if (!set.contains(r)){
```

```
 nodes[r].path = Math.min(nodes[r].path, nodes[i].dist +
nodes[destination].dist - nodes[i + 1].dist +
grid[r][nodes[i].value]);
              pq.add(nodes[r]); set.add(r);}
            }
            if (!pq.isEmpty()){
                Node ans = pq.poll();
                hash.put(i + "_" + (i + 1), ans.path);
            }
            else {
                hash.put(i + "_" + (i + 1), Integer.MAX_VALUE);
            }
        }
    }
}
   public static void main(String[] args) throws IOException {
        Reader.init(System.in);
        int n = Reader.nextInt();
        ArrayList<int[]> edges = new ArrayList<>();
        Node[] nodes = new Node[n + 1];
        for (int i= 0; i < n; i++){nodes[i] = new
Node(Reader.nextInt());}
        int q = Reader.nextInt();
        System.out.println("Enter source and destination node
values");
        int source = Reader.nextInt(); int destination =
Reader.nextInt();
        System.out.println("Queries");
        int[][] grid = new int[n][n];
        for (int i = 0; i < q; i++){
            int a = Reader.nextInt(); int b = Reader.nextInt();
            nodes[a].list.add(b);
```

```
            edges.add(new int[]{a, b});

        }
        int[] dist = new int[n + 1];
        dijkstra_1(grid, nodes, dist, source, destination);
        for (int i = 0; i < dist.length; i ++){
            nodes[i].dist = dist[i];
        }
        HashMap<String, Integer> hash = new HashMap<>();
        dijkstra_2(grid, hash, edges, dist, nodes, source,
destination);
        hash.forEach((key, value) -> System.out.println(key + " "
+ value));
    }

}
    class Node{
        int dist = Integer.MAX_VALUE; int value = -1;
        int path = Integer.MAX_VALUE;
        ArrayList<Integer> list = new ArrayList<>();
        public Node(int value){
        this.value = value;
        }
        }
```

Algorithm using Segment Trees

There are two types of edges in the graph: those which are in the shortest path from s to t and the ones which are not.

The edges which are not in the path if removed does not create any difference to the shortest distance of t from s, however, if the edges from

the shortest path are removed then the shortest distance must increase. Let these edges be called the optimal edges. Now, these edges can be easily identified by running Dijkstra on the graph once and finding the shortest path from s to t. All the edges in this path are optimal edges since we have assumed that there is only one unique shortest path from s to t.

Let the number of optimal edges be k. Now, we define a set of all vertices v as A_i such that there exists a shortest path from the source to v using at most i optimal edges.

The main idea:
Consider an optimal edge e_i, connecting two sets of vertices A_i and A_{i+1}. Let E_i be the set of all edges connecting these two sets then the shortest path between s and t without the edge e_i will be:
$(u,v) \in E_i$ -> Min { distance of u from s + length(u,v) + distance of t from v }

Let the final array be Ans[1...E] where the j-th entry contains the shortest s-t path in the graph G/e. Thus, for all the sets A_i and A_j such that $i<j$, we need to consider the minimum path length to connect these sets by considering the non-optimal edges. That is for each non-optimal edge k from i to j:
Ans[k] = min{ Ans[k], distance of u from s + length(u,v) + distance of t from v}.
Note: This algorithm works if all edge lengths are distinct.
In other words, we are updating the distance at each iteration. However, this cannot be done in the given time bound so to solve this problem we can use segment trees. A Segment Tree is a data structure that allows answering range queries over an array effectively, with updations. The update of Ans[k] can be done in O(log k) time by segment trees.
Hence we initialise the segment tree and as stated above we iterate over all the non-optimal edges when the optimal edge connecting two sets of the vertices in which optimal edge was connected. If e=(u,v) is an edge

crossing from Ai to Aj such that j>i, then we update Ans[i], Ans[i+1], ..., Ans[j−1].

Since we have already identified the path between s and t we know if an edge is an optimal edge or not, hence in the final array if we encounter an edge that is in the optimal edge then we fill it the value calculated above and if not then we do have the shortest distance calculated initially.

Now we run this for E number of edges and every query using the segment tree uses a time of O(log k). Since the maximum value k can take is E, the time complexity becomes O(E log E) which is equivalent to O(E log V). Also, we run Dijkstra once over the graph whose complexity is also O(E log V). Hence the overall time complexity of the algorithm is O(E log V).