LAB-05
Computer Networks
Group-18

200101086 Ravipati Swarna
220123041 Nelapati Meghana
220101049 Kasani Keerthi Sarika
220123029 Kavuri Veda Varsha

Task-1

The code prompts the user for the number of nodes and edges in a weighted, undirected graph, accepting edges in the source, destination, and cost format. It initializes a routing table with self-loops set to zero and other distances set to infinity.

Algorithm Implementation

The core functionality is implemented through a **Bellman-Ford-like** approach. The algorithm iteratively updates the routing table for each node based on the costs of directly connected nodes. This process runs N−1 times, where N is the number of nodes, ensuring that all shortest paths are computed.

```
// Bellman-Ford-like algorithm to update the routing table
void bellman_ford(int N, vector<vector<int>>& routing_table, vector<tuple<int, int, int>>& edges) {
    for (int k = 0; k < N-1; ++k) {  // Run N-1 times
        for (auto& edge : edges) {
            int u, v, cost;
            tie(u, v, cost) = edge;
            u--; v--;  // Convert to 0-indexed
            for (int node = 0; node < N; ++node) {
                if (routing_table[node][u] != INF) {
                    routing_table[node][v] = min(routing_table[node][v], routing_table[node][u] + cost);
                }
                if (routing_table[node][v] != INF) {
                    routing_table[node][u] = min(routing_table[node][u], routing_table[node][v] + cost);
                }
            }
        }
    }
}
```

Link Failure Simulation

The program enables users to simulate link failures by removing specified edges. After each removal, the routing table is recalculated to reflect updated costs, mimicking real-world scenarios where network links fail and routers must adjust their routes.

```
Enter the number of nodes and edges: 5 6
Enter the edges in the format: source destination cost (e.g., 1 2 3):
1 2 3
1 3 5
2 3 2
2 4 4
3 4 1
4 5 6
Routing Table before Link Failure:
Routing Table:
     1   2   3   4   5
1 | 0   3   5   6   12
2 | 3   0   2   3   9
3 | 5   2   0   1   7
4 | 6   3   1   0   6
5 | 12   9   7   6   0

Enter the nodes between which the link has failed (or enter 0 0 to exit): 3 4
Routing Table after Link Failure (3-4):
Routing Table:
     1   2   3   4   5
1 | 0   3   5   7   13
2 | 3   0   2   4   10
3 | 5   2   0   6   12
4 | 7   4   6   0   6
5 | 13   10   12   6   0

No count-to-infinity problem.
```

Count-to-Infinity Problem Detection

To access network stability, the program checks for the count-to-infinity problem. If any distance in the routing table exceeds a threshold (set to 100), it indicates potential issues in the routing protocol, highlighting the importance of monitoring and managing dynamic networks.

```
Enter the nodes between which the link has failed (or enter 0 0 to exit): 4 5
Routing Table after Link Failure (4-5):
Routing Table:
     1   2   3   4   5
1 | 0   3   5   6   INF
2 | 3   0   2   3   INF
3 | 5   2   0   1   INF
4 | 6   3   1   0   INF
5 | INF  INF  INF  INF  0

Count-to-infinity problem detected.
Enter the nodes between which the link has failed (or enter 0 0 to exit): 0 0
```

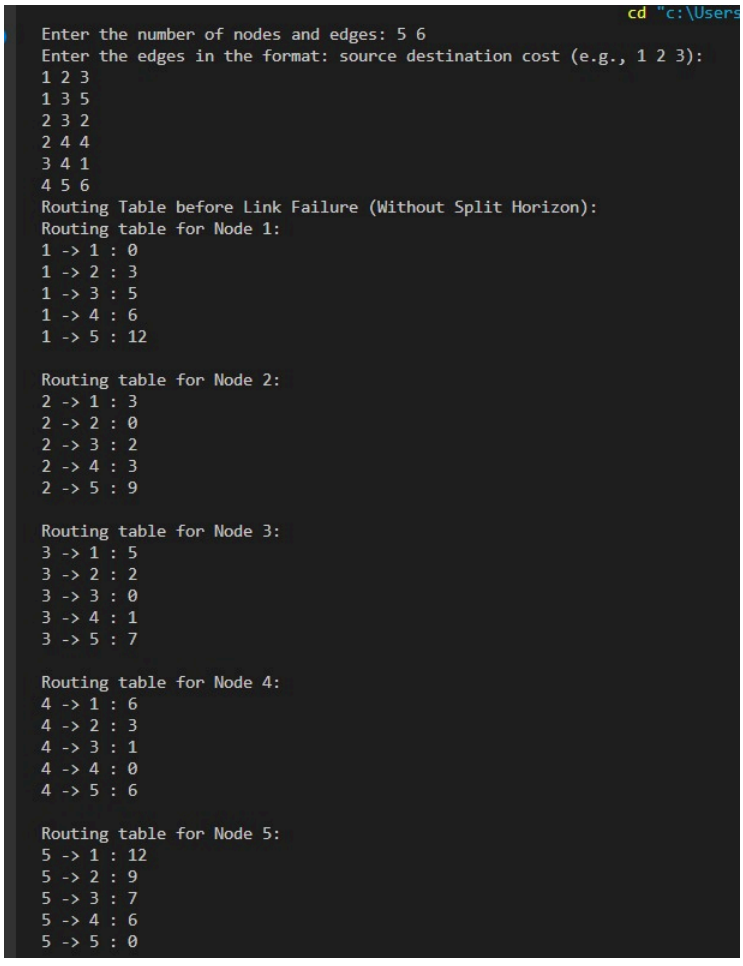
You can check the detailed implementation in the methods and outputs mentioned below..

SPLIT HORIZON

Split horizon is a method used by distance vector protocols to prevent network routing loops. The basic principle is simple: Never send routing information back in the direction from which it was received.

But here to implement this, I stored the next hop for every router while using split horizon. Checking for every router, when I have the reaching router as the one where I have failed the link,
I check the next hop, if it is the other node of the failed link, then I assume it to be infinity, calculating again, we get the final distance between the given nodes.

Using the example given in the question:



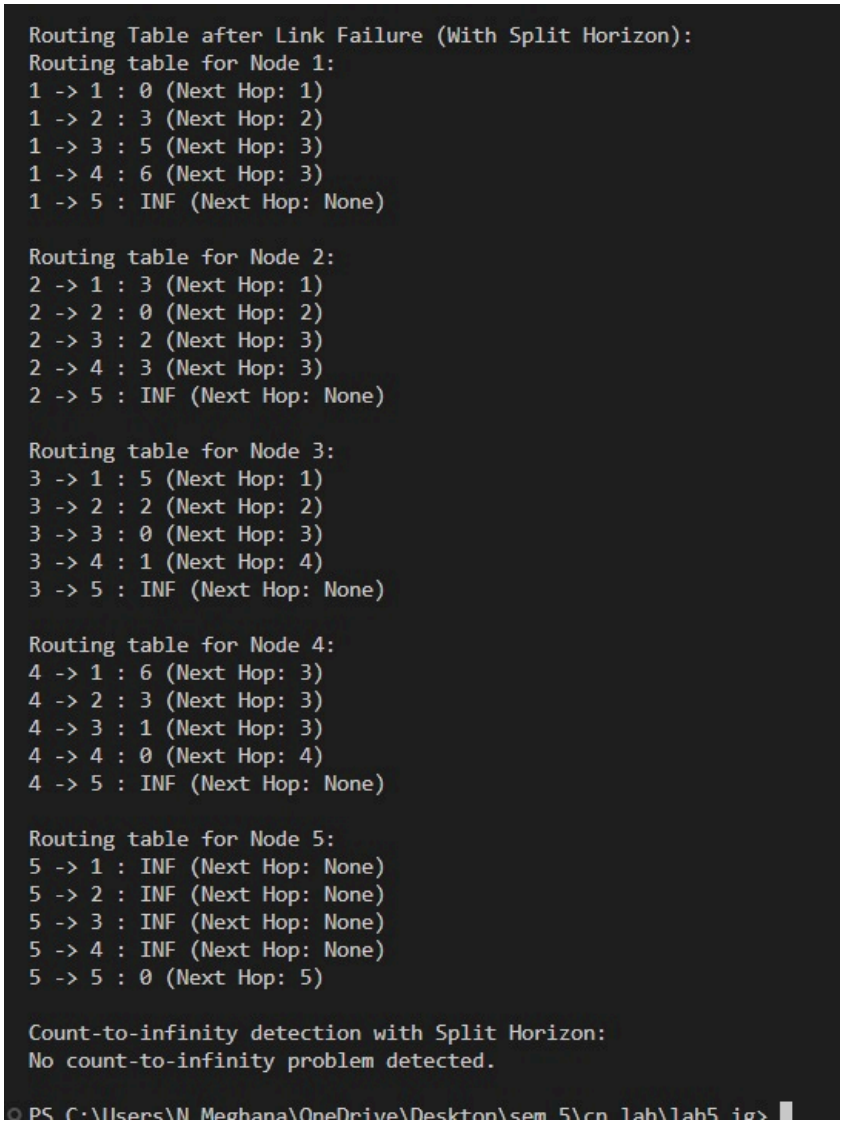
This is the initial routing table, without any link failure

```
Enter the nodes between which the link has failed (e.g., 4 5): 4 5
Routing Table after Link Failure (Without Split Horizon):
Routing table for Node 1:
1 -> 1 : 0
1 -> 2 : 3
1 -> 3 : 5
1 -> 4 : 6
1 -> 5 : INF

Routing table for Node 2:
2 -> 1 : 3
2 -> 2 : 0
2 -> 3 : 2
2 -> 4 : 3
2 -> 5 : INF

Routing table for Node 3:
3 -> 1 : 5
3 -> 2 : 2
3 -> 3 : 0
3 -> 4 : 1
3 -> 5 : INF

Routing table for Node 4:
4 -> 1 : 6
4 -> 2 : 3
4 -> 3 : 1
4 -> 4 : 0
4 -> 5 : INF

Routing table for Node 5:
5 -> 1 : INF
5 -> 2 : INF
5 -> 3 : INF
5 -> 4 : INF
5 -> 5 : 0
```

I have generated a link failure between 4 and 5

```
Count-to-infinity detection without Split Horizon:
Node 1 has a count-to-infinity problem to Node 5
Node 2 has a count-to-infinity problem to Node 5
Node 3 has a count-to-infinity problem to Node 5
Node 4 has a count-to-infinity problem to Node 5
Node 5 has a count-to-infinity problem to Node 1
Node 5 has a count-to-infinity problem to Node 2
Node 5 has a count-to-infinity problem to Node 3
Node 5 has a count-to-infinity problem to Node 4
```

These are the count to infinity problems generated without the split horizon method.



Here, using split horizon, it has calculated the next hop at every step, when the next is one of the node and the destination node is also one of the nodes when the link has failed, then the **distance** will be **infinity** here and the **next hop** is made **none.** Since 5 has only one link connected to the rest of the graph, and that is 4, and we broke that link, the distance to 5 from any node is INF.

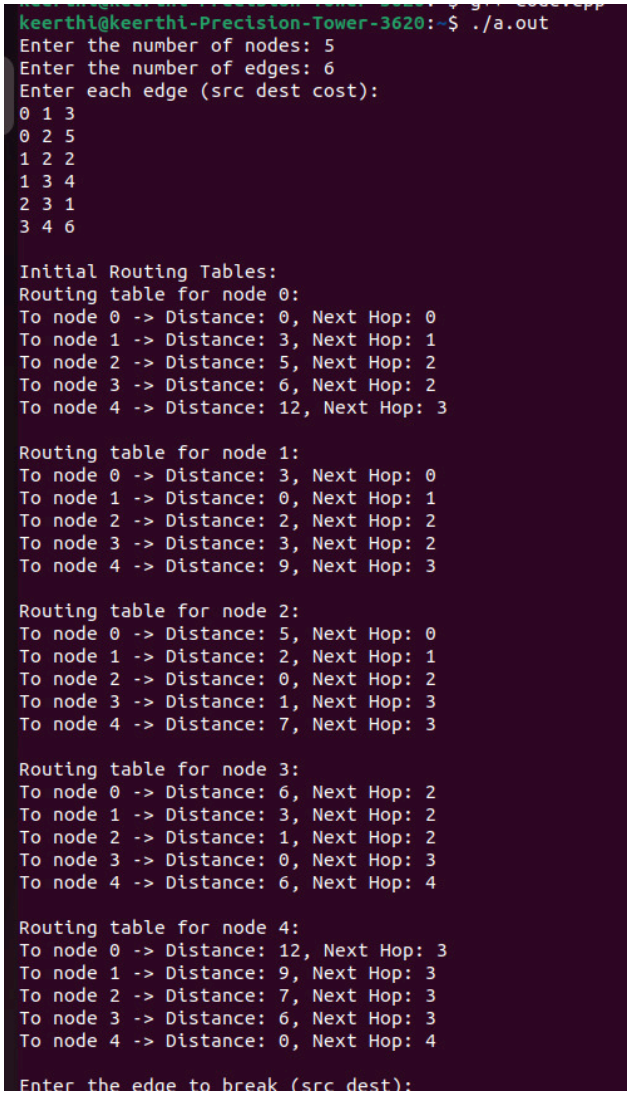
So finally, **no count to infinity problem is detected** using the Split Horizon Method.

POISONED REVERSE MECHANISM

Poison reverse can be thought of as an alternative to split horizon. With poison reverse, route advertisements that would be suppressed by split horizon are instead advertised with a distance of infinity.

Here to implement this, whenever we generate a link failure, we update the routing tables of the nodes in the link failure then go on to the neighbouring(next hop) nodes and update the routing tables accordingly and so on..

We used zero based indexing, and the output for the input given in the question:



After finding the shortest path using bellman ford, the initial routing paths are printed.

```
Enter the edge to break (src dest): 3 4

Routing Tables after applying Poisoned Reverse:
Routing table for node 0:
To node 0 -> Distance: 0, Next Hop: 0
To node 1 -> Distance: 3, Next Hop: 1
To node 2 -> Distance: 5, Next Hop: 2
To node 3 -> Distance: 6, Next Hop: 2
To node 4 -> Distance: INFINITY, Next Hop: -1

Routing table for node 1:
To node 0 -> Distance: 3, Next Hop: 0
To node 1 -> Distance: 0, Next Hop: 1
To node 2 -> Distance: 2, Next Hop: 2
To node 3 -> Distance: 3, Next Hop: 2
To node 4 -> Distance: INFINITY, Next Hop: -1

Routing table for node 2:
To node 0 -> Distance: 5, Next Hop: 0
To node 1 -> Distance: 2, Next Hop: 1
To node 2 -> Distance: 0, Next Hop: 2
To node 3 -> Distance: 1, Next Hop: 3
To node 4 -> Distance: INFINITY, Next Hop: -1

Routing table for node 3:
To node 0 -> Distance: 6, Next Hop: 2
To node 1 -> Distance: 3, Next Hop: 2
To node 2 -> Distance: 1, Next Hop: 2
To node 3 -> Distance: 0, Next Hop: 3
To node 4 -> Distance: INFINITY, Next Hop: -1

Routing table for node 4:
To node 0 -> Distance: INFINITY, Next Hop: -1
To node 1 -> Distance: INFINITY, Next Hop: -1
To node 2 -> Distance: INFINITY, Next Hop: -1
To node 3 -> Distance: INFINITY, Next Hop: -1
To node 4 -> Distance: 0, Next Hop: 4

keerthi@keerthi-Precision-Tower-3620:~/Downloads$
```

Now we generated a link failure between nodes 3 and 4, here the only link for 4 to the graph is through 3, we remove this, making the distance from all the nodes to 4 as infinity, we implement this using poison reverse mechanism as follows:
we check in the routing table for node 3 and 4,
Checking for 3: we see its neighbours as 1 and 2, we check for the path from 1 to 4 and 2 to 4, if the next hop node is 3, here it is, so we make the distance from 1 to 4 and 2 to 4 as INF. these changes are made in routing table for node 4
Since the changes were made, then we move to the updated neighbours' tables(1,2) and change accordingly
We also move to node 0, the neighbour of 1 and 2(as they were updated) and update it
Checking for 4: we compute the routing table for node 4 similarly, it is simple here since 4 doesn't have any neighbours after the link is removed making all the distances INF from the node.

The final tables for every node are printed as above.