Group 22
Anushka Gupta (220101014)
Aradhya Jain (220101015)
Arush Jain (220101016)
Aryan Arya (220101018)

Assignment 5

Part 1: Implementing the Distance Vector Routing Algorithm

1) Created Member variables of the class DistanceVectorRouting.

```
class DistanceVectorRouting {
private:
    int numNodes, numIter;
    vector<vector<int>> dist;
    vector<vector<int>> nextHop;
    vector<vector<pair<int, int>>> adjList; // adjacency list representation
};
```

`dist`: A 2D vector to store the shortest path cost between nodes, initialized to infinity (`INF`) .
`nextHop`: Stores the next-hop node in the shortest path between each source and destination.
`adjList`: An adjacency list to store direct neighbors and edge costs for each node.

2) Initialization of adjList, dist and nextHop.

```
// Distance to self is 0
for (int i = 1; i <= numNodes; i++) {
    dist[i][i] = 0;
    nextHop[i][i] = i;
}

// Set initial distances based on the provided edges and build adjacency list
for (const auto& edge : edges) {
    int src = edge[0], dest = edge[1], cost = edge[2];
    adjList[src].push_back({dest, cost});
    adjList[dest].push_back({src, cost});
    dist[src][dest] = cost;
    dist[dest][src] = cost;
    nextHop[src][dest] = dest;
    nextHop[dest][src] = src;
}
```

3) Implementation of the DVR algorithm

```
void runDVRAlgorithm() {
    bool updated;
    vector<vector<int>> prevDist;
    numIter = 0;
    do {
        updated = false;
        prevDist = dist; // Copy current distances to use as the reference for this iteration
        numIter++;
        for (int src = 1; src <= numNodes; src++) {
            for (int dest = 1; dest <= numNodes; dest++) {
                if (src == dest) continue;

                int minCost = INF; // Start with the direct distance

                // Calculate the minimum distance from neighbors
                for (const auto& neighbor : adjList[src]) {
                    if (prevDist[neighbor.first][dest] != INF) {
                        int newCost = neighbor.second + prevDist[neighbor.first][dest];
                        if (newCost < minCost) {
                            minCost = newCost;
                            nextHop[src][dest] = neighbor.first;
                        }
                    }
                }

                // Update if we found a shorter path
                if (dist[src][dest] != minCost) {
                    dist[src][dest] = minCost;
                    updated = true;
                }
            }
        }
    } while (updated);
}
```

- **Neighbor Cost Calculation**: For each source-destination pair, it calculates the minimum path cost through each neighbor, updating the cost and next-hop if a shorter path is found.
- **Repeat Until Convergence**: This process repeats until no further updates are needed, indicating that each node's routing table has stabilized with the shortest paths to all other nodes.

Output:

```
PS C:\Users\aradh\Downloads> ./a
Number of Iterations : 3
Routing Table for Node 1:
  To Node 1: Cost = 0, Next Hop = 1
  To Node 2: Cost = 3, Next Hop = 2
  To Node 3: Cost = 5, Next Hop = 2
  To Node 4: Cost = 6, Next Hop = 2
  To Node 5: Cost = 12, Next Hop = 2

Routing Table for Node 2:
  To Node 1: Cost = 3, Next Hop = 1
  To Node 2: Cost = 0, Next Hop = 2
  To Node 3: Cost = 2, Next Hop = 3
  To Node 4: Cost = 3, Next Hop = 3
  To Node 5: Cost = 9, Next Hop = 3

Routing Table for Node 3:
  To Node 1: Cost = 5, Next Hop = 1
  To Node 2: Cost = 2, Next Hop = 2
  To Node 3: Cost = 0, Next Hop = 3
  To Node 4: Cost = 1, Next Hop = 4
  To Node 5: Cost = 7, Next Hop = 4

Routing Table for Node 4:
  To Node 1: Cost = 6, Next Hop = 3
  To Node 2: Cost = 3, Next Hop = 3
  To Node 3: Cost = 1, Next Hop = 3
  To Node 4: Cost = 0, Next Hop = 4
  To Node 5: Cost = 6, Next Hop = 5

Routing Table for Node 5:
  To Node 1: Cost = 12, Next Hop = 4
  To Node 2: Cost = 9, Next Hop = 4
  To Node 3: Cost = 7, Next Hop = 4
  To Node 4: Cost = 6, Next Hop = 4
  To Node 5: Cost = 0, Next Hop = 5
```

Observations:

The output shows the correct tables for all the nodes and the Next Hop to reach that node.

4) Implementing the removal of an edge(given its source and destination)

```
void removeEdge(int src, int dest) {
    // Remove edge from adjacency list
    adjList[src].erase(remove_if(adjList[src].begin(), adjList[src].end(),
                                 [dest](const pair<int, int>& p) { return p.first == dest; }),
                       adjList[src].end());
    adjList[dest].erase(remove_if(adjList[dest].begin(), adjList[dest].end(),
                                  [src](const pair<int, int>& p) { return p.first == src; }),
                        adjList[dest].end());
    dist[src][dest] = INF;
    dist[dest][src] = INF;
    nextHop[src][dest] = -1;
    nextHop[dest][src] = -1;
}
```

This function removes an edge between `src` and `dest` in the adjacency list.
It sets the distance (`dist[src][dest]` and `dist[dest][src]`) to infinity (`INF`) to indicate that these nodes are unreachable.
Additionally, `nextHop` for the affected nodes is set to `-1`, marking that there is no valid next-hop for these nodes in either direction.

5) Updated DVA algorithm to check if the problem of count to infinity exists.

```
bool runDVRAlgorithm() {
    bool updated;
    bool countToInf = false;
    vector<vector<int>> prevDist;
    numIter = 0;
    do {
        updated = false;
        prevDist = dist; // Copy current distances to use as the reference for this iteration
        numIter++;
        for (int src = 1; src <= numNodes; src++) {
            for (int dest = 1; dest <= numNodes; dest++) {
                if (src == dest) continue;

                int minCost = INF; // Start with the direct distance

                // Calculate the minimum distance from neighbors
                for (const auto& neighbor : adjList[src]) {
                    if (prevDist[neighbor.first][dest] != INF) {
                        int newCost = neighbor.second + prevDist[neighbor.first][dest];
                        if (newCost < minCost) {
                            minCost = newCost;
                            nextHop[src][dest] = neighbor.first;
                        }
                    }
                }

                // Update if we found a shorter path
                if (dist[src][dest] != minCost) {
                    dist[src][dest] = minCost;
                    updated = true;
                }

                if(dist[src][dest]>100 && dist[src][dest]<INF) return true;
            }
        }
    } while (updated);
    return countToInf;
}
```

Distance Calculation:

- As before, this part of the function iterates over all pairs of nodes, calculating the minimum path cost for each source-destination pair (`src` to `dest`) through neighboring nodes.
- If a shorter path is found, `dist` and `nextHop` tables are updated accordingly, and `updated` is set to `true` to indicate that changes were made.

Count-to-Infinity Check:

- After updating a distance, the algorithm now checks if `dist[src][dest]` has exceeded a threshold (100 in this case) but is still finite (not `INF`).
- If the distance exceeds 100, this indicates a possible count-to-infinity problem. In this case, the function immediately returns `true`.

Loop Termination:

- If no updates were made in an iteration (`updated` remains `false`), the `do-while` loop exits, indicating that the routing tables have stabilized.
- If the function detects a count-to-infinity scenario (a distance exceeds 100), it returns `true` to signal this condition; otherwise, it returns `false`, confirming that the algorithm has converged successfully.

6) Running the algorithm again after removing an edge.

```
DistanceVectorRouting dvr(numNodes, edges);

// Initial run to generate routing tables
dvr.runDVRAlgorithm();
cout << "Initial Routing Tables:\n";
dvr.printRoutingTables();

// Simulate the failure of the link between nodes 4 and 5
cout << "\nSimulating link failure between nodes 4 and 5...\n";
dvr.removeEdge(4,5);

// Re-run the DVR algorithm after link failure
bool countToInfinity = dvr.runDVRAlgorithm();

// Check if count-to-infinity problem was detected
if (countToInfinity) {
    cout << "Count-to-Infinity problem detected: Some distances exceed 100.\n";
    // Print routing tables after link failure
    cout << "Routing Tables after link failure:\n";
    dvr.printRoutingTables();
} else {
    cout << "No count-to-infinity problem detected.\n";
}
```

Output:(After removal of an edge)

```
Simulating link failure between nodes 4 and 5...
Count-to-Infinity problem detected: Some distances exceed 100.
Routing Tables after link failure:
Number of Iterations : 91
Routing Table for Node 1:
  To Node 1: Cost = 0, Next Hop = 1
  To Node 2: Cost = 3, Next Hop = 2
  To Node 3: Cost = 5, Next Hop = 2
  To Node 4: Cost = 6, Next Hop = 2
  To Node 5: Cost = 102, Next Hop = 3

Routing Table for Node 2:
  To Node 1: Cost = 3, Next Hop = 1
  To Node 2: Cost = 0, Next Hop = 2
  To Node 3: Cost = 2, Next Hop = 3
  To Node 4: Cost = 3, Next Hop = 3
  To Node 5: Cost = 100, Next Hop = 4

Routing Table for Node 3:
  To Node 1: Cost = 5, Next Hop = 1
  To Node 2: Cost = 2, Next Hop = 2
  To Node 3: Cost = 0, Next Hop = 3
  To Node 4: Cost = 1, Next Hop = 4
  To Node 5: Cost = 97, Next Hop = 4

Routing Table for Node 4:
  To Node 1: Cost = 6, Next Hop = 3
  To Node 2: Cost = 3, Next Hop = 3
  To Node 3: Cost = 1, Next Hop = 3
  To Node 4: Cost = 0, Next Hop = 4
  To Node 5: Cost = 100, Next Hop = 3

Routing Table for Node 5:
  To Node 1: Unreachable
  To Node 2: Unreachable
  To Node 3: Unreachable
  To Node 4: Unreachable
  To Node 5: Cost = 0, Next Hop = 5
```

Observations:

After simulating a link failure between nodes 4 and 5, the routing tables enter the "count-to-infinity" problem, where certain distances (such as the cost to reach Node 5) continuously increase. This is evident from the very high values (e.g., 100 or more) seen in the routing tables, which indicate that nodes are unsuccessfully trying to update their paths to Node 5 through other nodes.

The DVR algorithm performs 91 iterations, eventually detecting that some nodes exceed the threshold cost of 100, signaling the count-to-infinity problem. As a result, the unreachable routes to Node 5 stabilize at "Unreachable" for most nodes, while Node 5's own table reflects only itself as reachable.

Part 2: Implementing the Poisoned Reverse Mechanism

1) Implementing the Poisoned Reverse technique.

```
bool runDVRAlgorithmWithPoisonedReverse() {
    bool updated;
    bool countToInf = false;
    vector<vector<int>> prevDist;
    numIter = 0;
    do {
        updated = false;
        prevDist = dist; // Copy current distances to use as the reference for this iteration
        numIter++;
        for (int src = 1; src <= numNodes; src++) {
            for (int dest = 1; dest <= numNodes; dest++) {
                if (src == dest) continue;

                int minCost = INF; // Start with the infinity cost

                // Calculate the minimum distance from neighbors with Poisoned Reverse applied
                for (const auto& neighbor : adjList[src]) {
                    if (prevDist[neighbor.first][dest] != INF) {
                        int newCost = neighbor.second + prevDist[neighbor.first][dest];

                        // Apply Poisoned Reverse: if the next hop to 'dest' via 'neighbor' is 'src' itself, set the distance to infinity
                        if (nextHop[neighbor.first][dest] == src) {
                            newCost = INF;
                        }
```

```
                    if (newCost < minCost) {
                        minCost = newCost;
                        nextHop[src][dest] = neighbor.first;
                    }
                }
            }

            // Update if we found a shorter path
            if (dist[src][dest] != minCost) {
                dist[src][dest] = minCost;
                updated = true;
            }

            // Detect count-to-infinity issue if distance exceeds 100 but is not infinity
            if (dist[src][dest] > 100 && dist[src][dest] < INF) return true;
        }
    }
} while (updated);
return countToInf;
}
```

The function calculates the minimum distance to each destination using neighbors' distances, but with the **Poisoned Reverse** technique:

- If the best next hop for reaching `dest` from `neighbor` points back to `src` (indicating a potential routing loop), the distance (`newCost`) is artificially set to infinity (`INF`).
- This discourages `src` from routing traffic to `dest` via `neighbor`, effectively blocking routing loops from forming.
- The algorithm updates `minCost` and `nextHop[src][dest]` only if a shorter path is found that doesn't route through a loop.
- **Updating Distances:**
    - The algorithm updates `dist[src][dest]` if a shorter, valid path is discovered, setting `updated` to `true` to continue iterating.
    - If the distance for any node exceeds 100 without reaching infinity (`INF`), the function returns `true`, signaling a count-to-infinity situation, thus stopping the algorithm early.
- **Loop Termination:**
    - The loop continues until no further updates occur (`updated == false`), indicating that the routing tables have stabilized.
    - The function returns `countToInf` as `false` if convergence is reached without encountering the count-to-infinity issue.

2) Re-run of the simulation

```
Simulating link failure between nodes 4 and 5...
Count-to-Infinity problem detected: Some distances exceed 100.
Routing Tables after link failure with Poisoned Reverse:
Number of Iterations : 37
Routing Table for Node 1:
  To Node 1: Cost = 0, Next Hop = 1
  To Node 2: Cost = 3, Next Hop = 2
  To Node 3: Cost = 5, Next Hop = 2
  To Node 4: Cost = 6, Next Hop = 2
  To Node 5: Cost = 96, Next Hop = 3

Routing Table for Node 2:
  To Node 1: Cost = 3, Next Hop = 1
  To Node 2: Cost = 0, Next Hop = 2
  To Node 3: Cost = 2, Next Hop = 3
  To Node 4: Cost = 3, Next Hop = 3
  To Node 5: Cost = 93, Next Hop = 4

Routing Table for Node 3:
  To Node 1: Cost = 5, Next Hop = 1
  To Node 2: Cost = 2, Next Hop = 2
  To Node 3: Cost = 0, Next Hop = 3
  To Node 4: Cost = 1, Next Hop = 4
  To Node 5: Cost = 101, Next Hop = 2

Routing Table for Node 4:
  To Node 1: Cost = 6, Next Hop = 3
  To Node 2: Cost = 3, Next Hop = 3
  To Node 3: Cost = 1, Next Hop = 3
  To Node 4: Cost = 0, Next Hop = 4
  To Node 5: Cost = 89, Next Hop = 3

Routing Table for Node 5:
  To Node 1: Unreachable
  To Node 2: Unreachable
  To Node 3: Unreachable
  To Node 4: Unreachable
  To Node 5: Cost = 0, Next Hop = 5
```

3) Observations

Since Poisoned Reverse prevents a routing loop between two nodes by assigning an infinite weight to that edge, it cannot stop routing loops involving more than two nodes. However, the limit of 100 distance is reached more quickly because it utilizes a larger loop. As a result, we are able to detect the "count-to-infinity" problem in fewer iterations.

Part 3: Implementing the Split Horizon Mechanism

1) Implementation of the Split Horizon technique in DVR algorithm.

```
bool runDVRAlgorithmWithSplitHorizon() {
    bool updated;
    bool countToInf = false;
    vector<vector<int>> prevDist;
    numIter = 0;
    do {
        updated = false;
        prevDist = dist; // Copy current distances to use as the reference for this iteration
        numIter++;
        for (int src = 1; src <= numNodes; src++) {
            for (int dest = 1; dest <= numNodes; dest++) {
                if (src == dest) continue;

                int minCost = INF; // Start with the infinity cost

                // Calculate the minimum distance from neighbors with Split Horizon applied
                for (const auto& neighbor : adjList[src]) {
                    // Apply Split Horizon: Ignore route if the next hop from 'neighbor' to 'dest' is 'src' itself
                    if (nextHop[neighbor.first][dest] == src) {
                        continue;
                    }
```

```
                    if (prevDist[neighbor.first][dest] != INF) {
                        int newCost = neighbor.second + prevDist[neighbor.first][dest];
                        if (newCost < minCost) {
                            minCost = newCost;
                            nextHop[src][dest] = neighbor.first;
                        }
                    }
                }

                // Update if we found a shorter path
                if (dist[src][dest] != minCost) {
                    dist[src][dest] = minCost;
                    updated = true;
                }

                // Detect count-to-infinity issue if distance exceeds 100 but is not infinity
                if (dist[src][dest] > 100 && dist[src][dest] < INF) return true;
            }
        }
    } while (updated);
    return countToInf;
```

Neighbor Evaluation with Split Horizon:

```
for (const auto& neighbor : adjList[src]) {
    // Apply Split Horizon: Ignore route if the next hop from 'neighbor' to 'dest' is 'src' itself
    if (nextHop[neighbor.first][dest] == src) {
        continue;
    }
```

If the next hop from a neighbor to the destination is the source itself, that route is ignored. This prevents routing information from being sent back to the source node that might lead to routing loops or incorrect information propagation.

Cost Calculation and Update:

```
if (prevDist[neighbor.first][dest] != INF) {
    int newCost = neighbor.second + prevDist[neighbor.first][dest];
    if (newCost < minCost) {
        minCost = newCost;
        nextHop[src][dest] = neighbor.first;
    }
}
```

Purpose: If the neighbor's distance to the destination is not infinity, it calculates a new potential cost for reaching the destination via that neighbor.
If this new cost is less than the current minimum cost (`minCost`), it updates `minCost` and sets the next hop for the source to be that neighbor.

Rerun of the simulation:

```
Simulating link failure between nodes 4 and 5...
Count-to-Infinity problem detected: Some distances exceed 100.
Routing Tables after link failure with Split Horizon:
Number of Iterations : 37
Routing Table for Node 1:
  To Node 1: Cost = 0, Next Hop = 1
  To Node 2: Cost = 3, Next Hop = 2
  To Node 3: Cost = 5, Next Hop = 2
  To Node 4: Cost = 6, Next Hop = 2
  To Node 5: Cost = 96, Next Hop = 3

Routing Table for Node 2:
  To Node 1: Cost = 3, Next Hop = 1
  To Node 2: Cost = 0, Next Hop = 2
  To Node 3: Cost = 2, Next Hop = 3
  To Node 4: Cost = 3, Next Hop = 3
  To Node 5: Cost = 93, Next Hop = 4

Routing Table for Node 3:
  To Node 1: Cost = 5, Next Hop = 1
  To Node 2: Cost = 2, Next Hop = 2
  To Node 3: Cost = 0, Next Hop = 3
  To Node 4: Cost = 1, Next Hop = 4
  To Node 5: Cost = 101, Next Hop = 2

Routing Table for Node 4:
  To Node 1: Cost = 6, Next Hop = 3
  To Node 2: Cost = 3, Next Hop = 3
  To Node 3: Cost = 1, Next Hop = 3
  To Node 4: Cost = 0, Next Hop = 4
  To Node 5: Cost = 89, Next Hop = 3

Routing Table for Node 5:
  To Node 1: Unreachable
  To Node 2: Unreachable
  To Node 3: Unreachable
  To Node 4: Unreachable
  To Node 5: Cost = 0, Next Hop = 5
```

Observations

Slow Convergence with Count-to-Infinity Problem:

- Despite using split horizon, the count-to-infinity problem still persists, as evident from the high costs between nodes (e.g., Node 1's cost to Node 5 is 96, and Node 3's cost to Node 5 is 101). This indicates that split horizon alone is not sufficient to prevent count-to-infinity entirely, particularly when the network has cycles and alternative paths.

Effectiveness of Split Horizon in Some Routes:

- The costs to Node 5 for Nodes 1, 2, 3, and 4 have become extremely high (close to 100 or beyond). This shows that split horizon attempts to prevent nodes from updating their routes with information learned from their neighbors, thus slowing down the incorrect propagation of routes. However, the ultimate convergence to a stable state where distances remain high reflects the inefficiency of split horizon alone in larger or more complex network topologies.

Increased Iteration Count:

- It took 37 iterations for the routing tables to converge, which is significantly longer than the initial convergence (3 iterations). This increase is due to the slow propagation of route updates caused by split horizon, which attempts to avoid routing loops but does not completely prevent them in the presence of link failures.