Distance Vector Routing Algorithm with Edge Break and its Solution

Assignment 5 CS344 Fall 2024

Group 37:

Adarsh Gupta (220101003)
Tanvi Doshi (220101102)
Vasudha (220101108)
Yash Jain (220101115)

Table of Contents:

Code Link: https://github.com/CoolSunflower/DVR-Simulation

Part 1: Distance Vector Routing & Link Failure

Theory

The distance vector algorithm is a distributed, asynchronous, iterative, and self-stopping routing algorithm. Each router sends its distance vector to only its neighboring routers, the receiving router performs updation using the Bellman-Ford equation and if the distance vector of the router changes this information is then propagated to all neighbouring routers and so on.

Implementation

The general flow of the simulation is as follows:

- We start by instantiating the distance vectors of each router and its neighbors with the information provided as input.
- Then we run the distance vector computation step in each router and update its distance vector one by one until convergence.
- Once a set of optimal distance vectors for each node is reached, this routing information is printed.
- To simulate link failure:
    - The edge is removed from the list of available edges and also the nodes are removed from the neighbor lists for each node
    - We then rerun the DV algorithm on the updated topology and wait for either convergence or routing loops and count-to-infinity problem

The key features of the code include initialisization of a network of nodes, updation of distance vectors based on neighboring nodes' information, handle link failures, and check for routing issues like the "count-to-infinity" problem.
The functions, code, and description are as follows :

Initialisation:

```
void initializeNodes(vector<Node>& nodes, vector<Edge>& edges, int N) {
    nodes.resize(N + 1); // Assuming nodes are numbered from 1 to N
    for (int i = 1; i <= N; ++i) {
        nodes[i].id = i;
    }
    for (auto& edge : edges) {
        nodes[edge.src].neighbors.push_back(edge.dest);
        nodes[edge.dest].neighbors.push_back(edge.src);
    }
}
```

The **initializeNodes** function sets up the network by initialising each node and establishing direct neighbours based on the provided edges. It resizes the nodes vector to N + 1 for 1-based indexing and assigns the `id` of each node from 1 to N. Then, it populates the neighbours for each node by iterating through the edges and registering the source and destination nodes as mutual neighbours, ensuring that each node's neighbours vector accurately reflects its direct connections.

```
void initializeDistanceVectors(vector<Node>& nodes, vector<Edge>& edges, int N) {
    for (int i = 1; i <= N; ++i) {
        nodes[i].distanceVector.clear();
        nodes[i].nextHop.clear();
        for (int j = 1; j <= N; ++j) {
            if (i == j) {
                nodes[i].distanceVector[j] = 0;
                nodes[i].nextHop[j] = j;
            } else {
                nodes[i].distanceVector[j] = INFINITY;
                nodes[i].nextHop[j] = -1;
            }
        }
    }
```

The **initializeDistanceVectors** function establishes the initial distance vectors and next-hop tables for each node in the network. For each node i (from 1 to N), it sets the distance to itself as 0 and the next hop to itself. Distances to all other nodes are initialised to INFINITY, with the next hop set to -1, indicating no known path. The function then assigns costs for directly connected neighbours using the edges vector; if a node has a direct connection to another node, the actual edge cost is assigned in the distance vector, and the next-hop table for the destination is set to the neighbour itself. This provides each node with an initial understanding of its direct connections while considering all other nodes as unreachable.

After the nodes and distance vectors for each node are initialised, we simulate the distance vector routing algorithm in the following loop:

```
    // Run the DVR algorithm until convergence
    bool updated;
    int iteration = 0; // Iteration counter
    do {
        updated = updateDistanceVectors(nodes, N, method);

        // Increment iteration counter
        iteration++;

        // Generate filename
        string filename = "distance_vectors_iteration_" + to_string(iteration) + ".txt";

        // Print the current distance vectors to file
        printDistanceVectorsToFile(nodes, N, filename);

    } while (updated);
```

After each iteration, the intermediate routing table gets saved, if needed for later analysis.

updateDistanceVector function:

The purpose of this code is that it updates the distance vectors based on the information exchanged between neighbouring nodes.

```
bool updateDistanceVectors(vector<Node>& nodes, int N, int method) {
    vector<map<int, int>> oldDVs;
    for(int i = 1; i <= N; i++){
        oldDVs.push_back(nodes[i].distanceVector);
    }

    for (int i = 1; i <= N; ++i) {
        for (int neighbor : nodes[i].neighbors) {
            // For each neighbor, create a copy of the distance vector to send
            map<int, int> dvToSend = nodes[i].distanceVector;

            // Need to store the old distance vector of the neigbor for calculations
            map<int, int> oldDV = nodes[neighbor].distanceVector;

            // Now, neighbor updates its distance vector based on the received dvToSend
            for(int j = 1; j <= N; j++){
                // Neighbor needs to update its distance vector for all enteries
                if(neighbor == j) continue; // Skip its own entry

                int minCost = INFINITY;
                for(int neighborsNeighbor : nodes[neighbor].neighbors){
                    // for all neighbors of this neighbor we need to find minimum for all destination nodes
                    // except for i, for them we need to use the one supplied by dvToSend
                    if(neighborsNeighbor == i){
                        if((oldDV[neighborsNeighbor] + dvToSend[j]) < minCost){
                            minCost = oldDV[neighborsNeighbor] + nodes[neighborsNeighbor].distanceVector[j];
                            nodes[neighbor].distanceVector[j] = minCost;
                            nodes[neighbor].nextHop[j] = neighborsNeighbor;
                        }
                    }else{
                        if((oldDV[neighborsNeighbor] + nodes[neighborsNeighbor].distanceVector[j]) < minCost){
                            minCost = oldDV[neighborsNeighbor] + nodes[neighborsNeighbor].distanceVector[j];
                            nodes[neighbor].distanceVector[j] = minCost;
                            nodes[neighbor].nextHop[j] = neighborsNeighbor;
                        }
                    }
                }
            }
        }
    }

    // To decide whether updated or not
    // we need to compare old distance vectors to new distance vectors
    for(int i = 1; i <= N; i++){
        for(int j = 1; j <= N; j++){
            if(i == j) continue;
            if(oldDVs[i - 1][j] != nodes[i].distanceVector[j]){
                return true;
            }
        }
    }
    return false;
}
```

Explanation:

- The function creates a copy of the old distance vectors for comparison later.
- For each node, it iterates through its neighbours and prepares to send its current distance vector.
- Each neighbour updates its own distance vector by considering:

  1.The distance through the sending node and the cost to reach the destination from the sending node.

  2.For each destination, it determines if a shorter path exists through the sending neighbour and updates the distance vector and next hop if a shorter path is found.

- It finally checks if any distance vectors were updated and returns true if any changes occurred; otherwise, it returns false

After the do-while loop gets over, that symbolizes that each router has found its optimum routing table under the current scenario. This then gets printed on the terminal along with the next hop routers for each destination node.

Next, we will simulate link failure, by the following method:

- Remove edge from list of edges of both neighbors
- Set the distance vector components to infinity and next hop routers to -1

```
// Remove the edge from edges list
edges.erase(remove_if(edges.begin(), edges.end(), [failSrc, failDest](Edge& e) {
    return (e.src == failSrc && e.dest == failDest) || (e.src == failDest && e.dest == failSrc);
}), edges.end());

// Update neighbors
nodes[failSrc].neighbors.erase(remove(nodes[failSrc].neighbors.begin(), nodes[failSrc].neighbors.end(), failDest),
                               nodes[failSrc].neighbors.end());
nodes[failDest].neighbors.erase(remove(nodes[failDest].neighbors.begin(), nodes[failDest].neighbors.end(), failSrc),
                                nodes[failDest].neighbors.end());

nodes[failSrc].distanceVector[failDest] = INFINITY;
nodes[failDest].distanceVector[failSrc] = INFINITY;
nodes[failSrc].nextHop[failDest] = -1;
nodes[failDest].nextHop[failSrc] = -1;
```

After this we will restart the DVR algorithm from this stage and check for count to infinity problem anytime we update the distance vector by comparing it against the value 100 (given in assignment).

```
// Re-run the DVR algorithm until convergence or until any distance exceeds 100
bool countToInfinity = false;
iteration = 0; // Reset iteration counter
do {
    updated = updateDistanceVectors(nodes, N, method);
    countToInfinity = checkCountToInfinity(nodes, N);
    if (countToInfinity) {
        cout << "Count-to-infinity problem detected.\n";
        break;
    }

    // Increment iteration counter
    iteration++;

    // Generate filename
    string filename = "distance_vectors_after_failure_iteration_" + to_string(iteration) + ".txt";

    // Print the current distance vectors to file
    printDistanceVectorsToFile(nodes, N, filename);

} while (updated);
```

The checkCountToInfinity function works as follows:

-The function examines each node's distance vector and checks if any distance exceeds a threshold (100) but is still less than INFINITY.
-If such a distance is found, it prints a message indicating the node and destination, and sets a flag (countToInfinity) to true.
-Returns true if any distances exceeding the threshold are detected; otherwise, returns false.

```
bool checkCountToInfinity(const vector<Node>& nodes, int N) {
    bool countToInfinity = false;
    for (int i = 1; i <= N; ++i) {
        for (auto& entry : nodes[i].distanceVector) {
            if (entry.second > 100 && entry.second < INFINITY) {
                cout << "Node " << i << " has distance >100 to Node " << entry.first << ".\n";
                countToInfinity = true;
            }
        }
    }
    return countToInfinity;
}
```

Output on Sample test case:

Original Calculate Routing Table:

```
Routing tables after running DVR algorithm:
Routing table for Node 1:
Destination     Cost    Next Hop
1               0       1
2               3       2
3               5       2
4               6       2
5               12      2

Routing table for Node 2:
Destination     Cost    Next Hop
1               3       1
2               0       2
3               2       3
4               3       3
5               9       3

Routing table for Node 3:
Destination     Cost    Next Hop
1               5       1
2               2       2
3               0       3
4               1       4
5               7       4

Routing table for Node 4:
Destination     Cost    Next Hop
1               6       2
2               3       2
3               1       3
4               0       4
5               6       5

Routing table for Node 5:
Destination     Cost    Next Hop
1               12      4
2               9       4
3               7       4
4               6       4
5               0       5
```

After breaking link between 4 and 5:

```
Simulate Link Failure between
Node A: 4
Node B: 5
Node 1 has distance >100 to Node 5.
Count-to-infinity problem detected.

Routing tables after link failure:
Routing table for Node 1:
Destination     Cost    Next Hop
1               0       1
2               3       2
3               5       2
4               6       2
5               102     2

Routing table for Node 2:
Destination     Cost    Next Hop
1               3       1
2               0       2
3               2       3
4               3       3
5               99      3

Routing table for Node 3:
Destination     Cost    Next Hop
1               5       1
2               2       2
3               0       3
4               1       4
5               99      4

Routing table for Node 4:
Destination     Cost    Next Hop
1               6       2
2               3       2
3               1       3
4               0       4
5               98      3

Routing table for Node 5:
Destination     Cost    Next Hop
1               12      4
2               9       4
3               7       4
4               INF     -
5               0       5
```

Part 2: Adding Poisoned Reverse

Theory:

In the distance vector algorithm, we often face a challenge when link costs get updated; a given node might end up sending routing information to a route from where the information actually came from leading to a routing loop and the famous count-to-infinity problem. In order to prevent such routing loops, the poisoned reverse technique is used where a node advertises to its parent(i.e. The route from which information came) that its distances to other nodes that exist in the route are infinite so that the parent knows that it is the parent!

Modification to code:

The updateDistanceVector function is modified to add poisoned reverse. Before updating the distance vectors of the neighbours of a node during propagation the following lines are added:

```
            // Poisoned Reverse
            for (auto& entry : dvToSend) {
                int dest = entry.first;
                if (nodes[i].nextHop[dest] == neighbor && dest != neighbor) {
                    dvToSend[dest] = INFINITY;
                }
            }
```

These lines of code check that the node which is sending its distance vector to its neighbours(i.e. Node i) checks that its path to a given destination does not pass through/have the next hop as the neighbour itself. If this is the case, then this neighbour advertises to the neighbour that its distance to the destination is infinity. This ensures that the neighbour does not update so as to route through node i itself and thus lead to a routing loop.

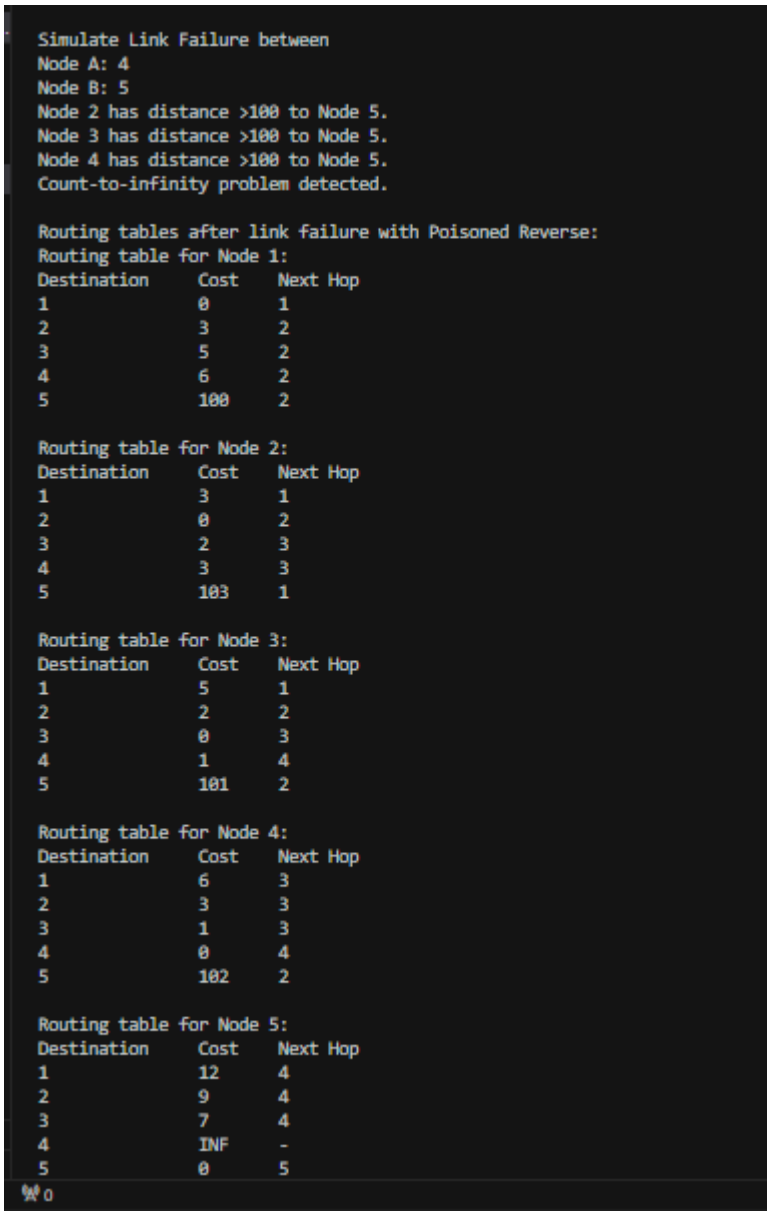
Output on Sample test case:

For the given sample graph in the question the output is as follows:

Routing tables Original:

```
Routing tables after running DVR algorithm with Poisoned Reverse:
Routing table for Node 1:
Destination     Cost    Next Hop
1               0       1
2               3       2
3               5       2
4               6       2
5               12      2

Routing table for Node 2:
Destination     Cost    Next Hop
1               3       1
2               0       2
3               2       3
4               3       3
5               9       3

Routing table for Node 3:
Destination     Cost    Next Hop
1               5       1
2               2       2
3               0       3
4               1       4
5               7       4

Routing table for Node 4:
Destination     Cost    Next Hop
1               6       3
2               3       3
3               1       3
4               0       4
5               6       5

Routing table for Node 5:
Destination     Cost    Next Hop
1               12      4
2               9       4
3               7       4
4               6       4
5               0       5
0
```

Routing tables after link failure between nodes 4 and 5:



```
Simulate Link Failure between
Node A: 4
Node B: 5
Node 2 has distance >100 to Node 5.
Node 3 has distance >100 to Node 5.
Node 4 has distance >100 to Node 5.
Count-to-infinity problem detected.

Routing tables after link failure with Poisoned Reverse:
Routing table for Node 1:
Destination     Cost    Next Hop
1               0       1
2               3       2
3               5       2
4               6       2
5               100     2

Routing table for Node 2:
Destination     Cost    Next Hop
1               3       1
2               0       2
3               2       3
4               3       3
5               103     1

Routing table for Node 3:
Destination     Cost    Next Hop
1               5       1
2               2       2
3               0       3
4               1       4
5               101     2

Routing table for Node 4:
Destination     Cost    Next Hop
1               6       3
2               3       3
3               1       3
4               0       4
5               102     2

Routing table for Node 5:
Destination     Cost    Next Hop
1               12      4
2               9       4
3               7       4
4               INF     -
5               0       5
```

Part 3: Adding Split Horizon

Theory

Split horizon is another technique to tackle routing loops and count-to-infinity problems, but instead of advertising infinite distance to the parent, the node does not send any information about that particular route to the parent and hence avoids the parent using the node itself as a path to route to the destination.

Modification to code

The updateVectorDistance function is modified to ensure that the node sending its updated distance vector to its neighbours does not send information about a route to that neighbour which is the next-hop for the given node to the destination of the route. Here is the code added before updating neighbour's distance vector:

```
            // Split Horizon
            for (auto it = dvToSend.begin(); it != dvToSend.end();) {
                int dest = it->first;
                if (nodes[i].nextHop[dest] == neighbor && dest != neighbor) {
                    // Remove the route from the advertisement
                    it = dvToSend.erase(it);
                } else {
                    ++it;
                }
            }
```

These lines iterate through the updated distance vector of a node(say node i) and if the neighbour is the next-hop for a given destination, then that entry of the distance vector is erased, else we continue iterating.

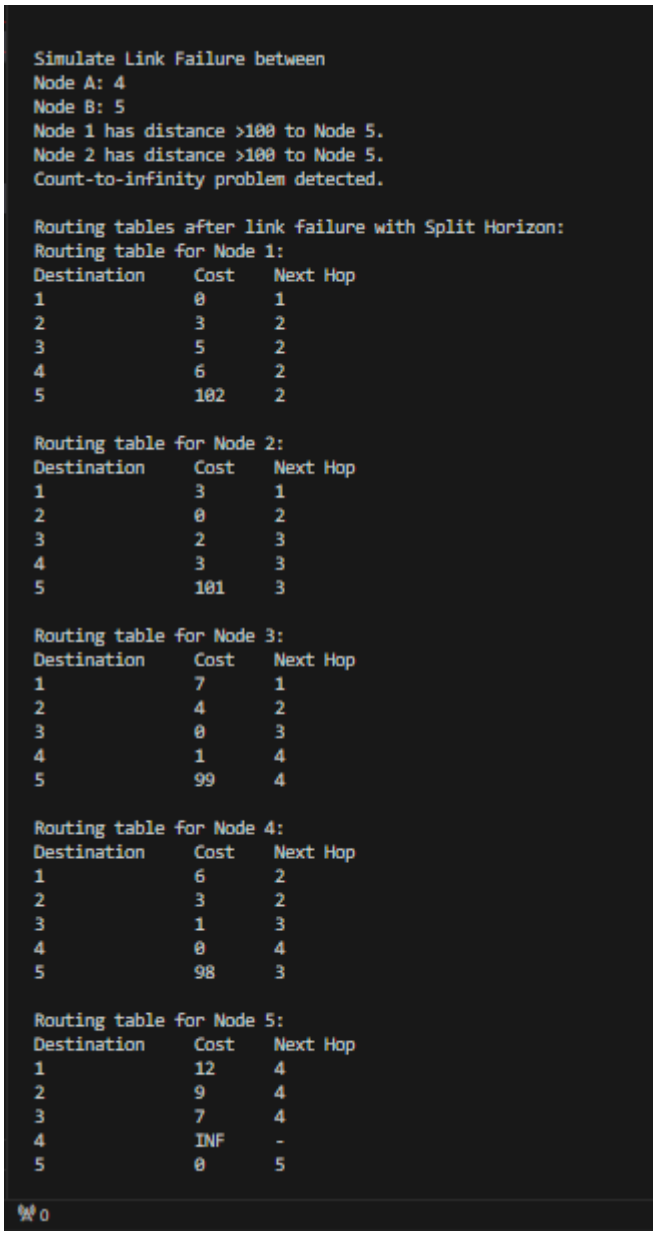
Output:

For the given sample graph in the question the output is as follows:

Routing tables originally:

```
Routing tables after running DVR algorithm with Split Horizon:
Routing table for Node 1:
Destination     Cost    Next Hop
1               0       1
2               3       2
3               5       2
4               6       2
5               12      2

Routing table for Node 2:
Destination     Cost    Next Hop
1               3       1
2               0       2
3               2       3
4               3       3
5               9       3

Routing table for Node 3:
Destination     Cost    Next Hop
1               7       1
2               4       2
3               0       3
4               1       4
5               7       4

Routing table for Node 4:
Destination     Cost    Next Hop
1               6       2
2               3       2
3               1       3
4               0       4
5               6       5

Routing table for Node 5:
Destination     Cost    Next Hop
1               12      4
2               9       4
3               7       4
4               6       4
5               0       5
```

Routing tables after link failure between nodes 4 and 5:



```
Simulate Link Failure between
Node A: 4
Node B: 5
Node 1 has distance >100 to Node 5.
Node 2 has distance >100 to Node 5.
Count-to-infinity problem detected.

Routing tables after link failure with Split Horizon:
Routing table for Node 1:
Destination     Cost    Next Hop
1               0       1
2               3       2
3               5       2
4               6       2
5               102     2

Routing table for Node 2:
Destination     Cost    Next Hop
1               3       1
2               0       2
3               2       3
4               3       3
5               101     3

Routing table for Node 3:
Destination     Cost    Next Hop
1               7       1
2               4       2
3               0       3
4               1       4
5               99      4

Routing table for Node 4:
Destination     Cost    Next Hop
1               6       2
2               3       2
3               1       3
4               0       4
5               98      3

Routing table for Node 5:
Destination     Cost    Next Hop
1               12      4
2               9       4
3               7       4
4               INF     -
5               0       5
```

Conclusion

None of the given algorithms fix the general count-to-infinity problem because of routing loops. In any case, these modifications to the original algorithm do fix the count-to-infinity problem in some cases, particularly, in the absence of routing loops or routing loops of size equal to 3.