Distance Vector Routing

```
class Router:
    def __init__(self, name, neighbors):
        self.name = name
        self.neighbors = neighbors  # List of (neighbor_name, cost)
        self.routing_table = {name: (0, name)}  # Distance to self is 0
        for neighbor, cost in neighbors:
            self.routing_table[neighbor] = (cost, neighbor)  # Initial distance to neighbors

    def update_routing_table(self, neighbor_router):
        updated = False
        for destination, (neighbor_distance, _) in neighbor_router.routing_table.items():
            if destination not in self.routing_table:
                self.routing_table[destination] = (neighbor_distance +
self.routing_table[neighbor_router.name][0], neighbor_router.name)
                updated = True
            else:
                current_distance, next_hop = self.routing_table[destination]
                new_distance = neighbor_distance +
self.routing_table[neighbor_router.name][0]
                if new_distance < current_distance:
                    self.routing_table[destination] = (new_distance, neighbor_router.name)
                    updated = True
        return updated

    def display_routing_table(self):
        print(f"Routing table for router {self.name}:")
        for destination, (distance, next_hop) in self.routing_table.items():
            print(f"Destination: {destination}, Distance: {distance}, Next hop: {next_hop}")
        print("\n")

def distance_vector_routing(routers):
    changes = True
    iteration = 0
    while changes:
        changes = False
        print(f"--- Iteration {iteration} ---")
        for router in routers:
            for neighbor_name, _ in router.neighbors:
                neighbor_router = next(r for r in routers if r.name == neighbor_name)
                if router.update_routing_table(neighbor_router):
                    changes = True
        iteration += 1
```

```
        for router in routers:
            router.display_routing_table()

    r1 = Router('A', [('B', 1), ('C', 4)])
    r2 = Router('B', [('A', 1), ('C', 2), ('D', 7)])
    r3 = Router('C', [('A', 4), ('B', 2), ('D', 3)])
    r4 = Router('D', [('B', 7), ('C', 3)])

    routers = [r1, r2, r3, r4]
    distance_vector_routing(routers)
```

Output

```
D:\GITHUB\LAB>python -u "d:\GITHUB\LAB\5TH SEMESTER\CNS\Assign 6\d.py"
--- Iteration 0 ---
Routing table for router A:
Destination: A, Distance: 0, Next hop: A
Destination: B, Distance: 1, Next hop: B
Destination: C, Distance: 3, Next hop: B
Destination: D, Distance: 6, Next hop: C
Routing table for router B:
Destination: B, Distance: 0, Next hop: B
Destination: A, Distance: 1, Next hop: A
Destination: C, Distance: 2, Next hop: C
Destination: D, Distance: 5, Next hop: C
Routing table for router C:
Destination: C, Distance: 0, Next hop: C
Destination: A, Distance: 3, Next hop: B
Destination: B, Distance: 2, Next hop: B
Destination: D, Distance: 3, Next hop: D
Routing table for router D:
Destination: D, Distance: 0, Next hop: D
Destination: B, Distance: 5, Next hop: C
Destination: C, Distance: 3, Next hop: C
Destination: A, Distance: 6, Next hop: C

--- Iteration 1 ---
Routing table for router A:
Destination: A, Distance: 0, Next hop: A
Destination: B, Distance: 1, Next hop: B
Destination: C, Distance: 3, Next hop: B
Destination: D, Distance: 6, Next hop: C
Routing table for router B:
Destination: B, Distance: 0, Next hop: B
Destination: A, Distance: 1, Next hop: A
```

```
Destination: C, Distance: 2, Next hop: C
Destination: D, Distance: 5, Next hop: C
Routing table for router C:
Destination: C, Distance: 0, Next hop: C
Destination: A, Distance: 3, Next hop: B
Destination: B, Distance: 2, Next hop: B
Destination: D, Distance: 3, Next hop: D
Routing table for router D:
Destination: D, Distance: 0, Next hop: D
Destination: B, Distance: 5, Next hop: C
Destination: C, Distance: 3, Next hop: C
Destination: A, Distance: 6, Next hop: C
```

Link State Routing Protocol

```
import java.util.*;
class LinkStateRouting {
   private Map<String, Map<String, Integer>> graph;
   public LinkStateRouting() {
      graph = new HashMap<>();
   }

   public void addEdge(String source, String destination, int cost) {
      graph.putIfAbsent(source, new HashMap<>());
      graph.putIfAbsent(destination, new HashMap<>());
      graph.get(source).put(destination, cost);
      graph.get(destination).put(source, cost); // Assuming undirected graph
   }

   public void computeShortestPaths(String start) {
      Map<String, Integer> distances = new HashMap<>();
      Set<String> visited = new HashSet<>();
      PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));

      for (String node : graph.keySet()) {
         distances.put(node, Integer.MAX_VALUE);
      }
      distances.put(start, 0);
      pq.offer(new Node(start, 0));
      while (!pq.isEmpty()) {
         Node current = pq.poll();
         if (visited.contains(current.name)) continue;
         visited.add(current.name);
         for (Map.Entry<String, Integer> neighbor : graph.get(current.name).entrySet()) {
            String neighborNode = neighbor.getKey();
            int newDist = distances.get(current.name) + neighbor.getValue();
```

```
            if (newDist < distances.get(neighborNode)) {
                distances.put(neighborNode, newDist);
                pq.offer(new Node(neighborNode, newDist));
            }
        }
    }

    printPaths(start, distances);
}

private void printPaths(String start, Map<String, Integer> distances) {
    System.out.println("Shortest paths from " + start + ":");
    for (String node : distances.keySet()) {
        if (node.equals(start)) continue;
        System.out.println("To " + node + " (cost: " + distances.get(node) + ")");
    }
}

private static class Node {
    String name;
    int cost;
    Node(String name, int cost) {
        this.name = name;
        this.cost = cost;
    }
}

public static void main(String[] args) {
    LinkStateRouting lsRouting = new LinkStateRouting();
    lsRouting.addEdge("A", "B", 1);
    lsRouting.addEdge("A", "C", 4);
    lsRouting.addEdge("B", "C", 2);
    lsRouting.addEdge("B", "D", 5);
    lsRouting.addEdge("C", "D", 1);
    lsRouting.computeShortestPaths("A");
}
}
```

Output:

```
Shortest paths from A:
To B (cost: 1)
To C (cost: 3)
To D (cost: 4)
```