ANC Assessed Exercise

Robert Allison 1102085a

Source Code Listings

DVRouting.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;


public class DVRouting {

    public static void main (String[] args){

        if (args.length == 0 ){
            System.out.println("Program usage: 'java DVRouting 
C:/input/file/path'");
            System.exit(0);
        }

        // Various argument trackers and flags (We need to track everything the 
user could enter!)
        String arg;
        int i = 1;
        int exchanges = 0;
        boolean stable = false;
        boolean splitHorizon = false;
        boolean trace = false;
        boolean fail = false;
        boolean change = false;
        String viewNode = "";
        String traceRoute = "";
        String failLink = "";
        String changeLink = "";
        String nodeA = null;
        String nodeB = null;
        String failNodeA = null;
        String failNodeB = null;
        String changeNodeA = null;
        String changeNodeB = null;
        int noticeExchange = 0;
        int failExchange = 0;
        int changeExchange = 0;
        int changeValue = 0;
        while (i < args.length && args[i].startsWith("-")) {
            arg = args[i++];

            // check for arguments and arguments for the arguments
            if (arg.equals("-help")){
                System.out.println("Command use: java -jar DVRouting [-s/-e #] 
[-v 'Node'] [-t 'N1,N2,EX'] [-f 'N1,N2,EX'] [-c 'N1,N2,C,EX'] [-split]");
                System.exit(0);
            }
            else if (arg.equals("-e")) {
                if (i < args.length)
                    exchanges = Integer.parseInt(args[i++]);
                else {
                    System.err.println("-e requires a number '-e 10'");
                    System.exit(0);
                }
            }
            else if (arg.equals("-v")) {
                if (i < args.length)
                    viewNode = args[i++];
                else {
                    System.err.println("-v requires a node to view 'N1'");
                    System.exit(0);
                }
            }
            else if (arg.equals("-t")) {
                if (i < args.length) {
                    traceRoute = args[i++];
                    String[] items = traceRoute.split(",");
                    nodeA = items[0];
                    nodeB = items[1];
                    noticeExchange = Integer.parseInt(items[2]);
                    trace = true;
                } else {
                    System.err.println("-t requires a two nodes and an exchange
number to trace on");
                    System.exit(0);
                }
            }
            else if (arg.equals("-f")) {
                if (i < args.length) {
                    failLink = args[i++];
                    String[] items = failLink.split(",");
                    failNodeA = items[0];
                    failNodeB = items[1];
                    failExchange = Integer.parseInt(items[2]);
                    fail = true;
                } else {
                    System.err.println("-f requires a two nodes and an exchange
number to fail on");
                    System.exit(0);
                }
            }
            else if (arg.equals("-c")) {
                if (i < args.length) {
                    changeLink = args[i++];
                    String[] items = changeLink.split(",");
                    changeNodeA = items[0];
                    changeNodeB = items[1];
                    changeValue = Integer.parseInt(items[2]);
                    changeExchange = Integer.parseInt(items[3]);
                    change = true;
                } else {
                    System.err.println("-c requires a two nodes, a cost and an
exchange number to change on");
                    System.exit(0);
                }
            }
            else if (arg.equals("-s")){
                stable = true;
            }
            else if (arg.equals("-split")){
                splitHorizon = true;
            }
        }

        System.out.println("Getting file");
        // The name of the file to open.
        String fileName = args[0];

        // Holds one line at a time
        String line = null;

        try {
            // Read the file
            FileReader fr =
                    new FileReader(fileName);

            // Wrap FileReader in BufferedReader.
            BufferedReader br =
                    new BufferedReader(fr);

            // Grab first line ( which should be list of nodes)
            line = br.readLine();
            //System.out.println(line);
            // Strip brackets { }, split into list
            String strippedLine = line.replaceAll("[\\[\\](){}]","");
            String[] nodes = strippedLine.split(",");

            //Create the nodes, add to list
            ArrayList<DVNode> nodeList = new ArrayList<DVNode>();
            for (String label: nodes){
                nodeList.add(new DVNode(label));
            }

            ArrayList<DVEdge> edgeList = new ArrayList<DVEdge>();
            // Get the edges
            while((line = br.readLine()) != null) {
                // Strip out the brackets, split
                strippedLine = line.replaceAll("[\\[\\](){}]","");
                String[] edge = strippedLine.split(",");
                // Get nodes on the edge
                DVNode a = null;
                DVNode b = null;
                for (DVNode n: nodeList){
                    if (n.getLabel().equals(edge[0])){
                        a = n;
                    }
                    else if(n.getLabel().equals(edge[1])){
                        b = n;
                    }
                }
                if (a == null || b == null) // Could not find nodes in nodelist
                    System.out.println("Could not resolve edge between: " +
edge[0] + " - " + edge[1]);
                else {
                    // Add edge to list of edges
                    int cost = Integer.parseInt(edge[2]);
                    DVEdge newEdge = new DVEdge(a, b, cost);
                    edgeList.add(newEdge);
                    //Add node as neighbour to each node
                    a.addNeighbour(new DVNeighbour(b, cost));
                    b.addNeighbour(new DVNeighbour(a, cost));
                }
            }
            // Create the graph using the node/edge lists
            if (!nodeList.isEmpty() && !edgeList.isEmpty()) {
                DVGraph routingGraph = new DVGraph(nodeList, edgeList,
splitHorizon);

                //Operations on the graph
                System.out.println("Graph constructed successfully.");
                if (splitHorizon){
                    System.out.println("Split Horizon: ON");
                }
                System.out.println(routingGraph);
                int counter = 1;
                // If going until stability, loop
                if (stable) {
                    while (!routingGraph.isStable()){
                        System.out.println("Exchange " + (counter));
                        // Check our flags to see if it's time to use them
                        if (trace && counter == noticeExchange){
                            System.out.println(routingGraph.findPath(nodeA,
nodeB));
                        }
                        if (fail && counter == failExchange){
                            routingGraph.failLink(failNodeA, failNodeB);
                            System.out.println("Failing Link: " + failNodeA + "
- " + failNodeB);
                        }
                        if (change && counter == changeExchange){
                            routingGraph.changeLink(changeNodeA, changeNodeB,
changeValue);
                            System.out.println("Changing Link: " + changeNodeA +
" - " + changeNodeB + ": " + changeValue);
                        }
                        if (!viewNode .equals("")){
                            routingGraph.printRoutingTable(viewNode);
                        }
                        routingGraph.doExchange();
                        routingGraph.stableNode(viewNode);
                        routingGraph.checkStability();
                        counter++;
                    }
                    System.out.println("Stability achieved");
                }
                // Else exchange selected number of times
                else {
                    for (int j = 0; j < exchanges; j++){
                        System.out.println("Exchange " + (counter));
                        // check flags
                        if (trace && counter == noticeExchange){
                            System.out.println(routingGraph.findPath(nodeA,
nodeB));
                        }
                        if (fail && counter == failExchange){
                            System.out.println("Failing Link: " + failNodeA + "
- " + failNodeB);
                            routingGraph.failLink(failNodeA, failNodeB);
                        }
                        if (change && counter == changeExchange){
                            routingGraph.changeLink(changeNodeA, changeNodeB,
changeValue);
                            System.out.println("Changing Link: " + changeNodeA +
" - " + changeNodeB + ": " + changeValue);
                        }
                        if (!viewNode .equals("")){
                            routingGraph.printRoutingTable(viewNode);
                        }
                        routingGraph.doExchange();
                        counter++;
                    }
                }
                br.close();
            }
            // We couldn't make the graph
            else {
                System.out.println("Not enough information to construct 
graph.");
            }
        }
        // Catch exceptions
        catch(FileNotFoundException ex) {
            System.out.println("Unable to open file '" + fileName + "'");
        }
        catch(IOException ex) {
            System.out.println(
                    "Error reading file '"
                            + fileName + "'");
        }
    }

}
```

DVGraph.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;

/**
 * Undirected graph representing a simple network
 */
public class DVGraph {

    private HashMap<String, DVNode> nodes;
    private ArrayList<DVEdge> edges;
    private boolean stable;
    private boolean split;

    public DVGraph(ArrayList<DVNode> nodeList, ArrayList<DVEdge> edgeList, 
boolean splitH) {
        edges = edgeList;
        // Put nodes in list into hashmap with label key
        nodes = new HashMap<String, DVNode>();
        for (DVNode n : nodeList){
            nodes.put(n.getLabel(), n);
        }
        stable = false;
        split = splitH;
    }

    public boolean isStable() {
        return stable;
    }

    public void setStable(boolean stable) {
        this.stable = stable;
    }

    public boolean isSplit() {
        return stable;
    }

    public void setSplit(boolean split) {
        this.split = split;
```

```
}

// Check if all nodes in the graph are stable
public void checkStability(){
    for (String k: nodes.keySet()){
        if (nodes.get(k).isStable()){
            setStable(true);
        }
        else {
            setStable(false);
        }
    }
}

public void doExchange(){
    // Get all nodes to send vectors to direct neighbours
    for (String key : nodes.keySet()){
        nodes.get(key).sendVectors(split);
    }
    // Get nodes to update costs
    for (String key : nodes.keySet()){
        nodes.get(key).calculateVectors();
    }
}

public void printNodes(){
    for (String key : nodes.keySet()){
        System.out.println(key);
    }
}

public void printNeighbours(){
    for (String key : nodes.keySet()){
        nodes.get(key).printNeighbours();
    }
}

// Print a node's routing table
public void printRoutingTable(String s){
    DVNode n = nodes.get(s);
    System.out.println(n.getRt().toString());
}

public void failLink(String a, String b){
     // Get the nodes
    DVNode nodeA = nodes.get(a);
    DVNode nodeB = nodes.get(b);
    DVNeighbour failedNode = null;
    // find the neighbour in each list and delete it
    for (DVNeighbour n : nodeA.getNeighbours()){
        if (n.getNeighbour().equals(nodeB)){
            failedNode = n;
        }
    }
    nodeA.removeNeighbour(failedNode);
    // Also remove it from the routing table;
    nodeA.getRt().changeRow(failedNode.getNeighbour());
    // Do the same with the other side of the link
    for (DVNeighbour n : nodeB.getNeighbours()){
        if (n.getNeighbour().equals(nodeA)){
            failedNode = n;
        }
    }
    nodeB.removeNeighbour(failedNode);
```

```
        nodeB.getRt().changeRow(failedNode.getNeighbour());
    }

    public void changeLink(String a, String b, int c){
        DVNode nodeA = nodes.get(a);
        DVNode nodeB = nodes.get(b);
        DVNeighbour changedNode = null;
        // find the neighbour in each list and change the cost
        for (DVNeighbour n : nodeA.getNeighbours()){
            if (n.getNeighbour().equals(nodeB)){
                n.setCost(c);
            }
        }
        // Do the same with the other side of the link
        for (DVNeighbour n : nodeB.getNeighbours()){
            if (n.getNeighbour().equals(nodeA)){
                n.setCost(c);
            }
        }
    }

    public String findPath(String a, String b){
        // get the nodes we need from the node list
        DVNode nodeA = nodes.get(a);
        DVNode nodeB = nodes.get(b);
        boolean found = false;
        DVNode currentNode = null;
        // List to store the path we create
        LinkedList<DVNode> nodePath = new LinkedList<DVNode>();
        nodePath.add(nodeA);
        currentNode = nodeA.getNextLink(nodeB);
        // Check if a link to the node exists in the table
        if (currentNode == null){
            return "No link between " + nodeA.getLabel() + " and " +
nodeB.getLabel();
        }
        else {
            // There is a link, loop until we reach it and push the path into
the list
            while (!found){
                nodePath.add(currentNode);
                if (currentNode.equals(nodeB)){
                    found = true;
                }
                currentNode = currentNode.getNextLink(nodeB);
            }
        }
        // Print out the path to a buffer, and return it
        StringBuffer sb = new StringBuffer();
        sb.append("Path from " + nodeA.getLabel() + " to " + nodeB.getLabel() +
"\n");
        for (DVNode n : nodePath){
            if (n.equals(nodeB)){
                sb.append(n.getLabel());
            }
            else {
                sb.append(n.getLabel() + " => ");
            }
        }
        return sb.toString();
    }

    // Print all nodes and edges in graph
    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append("Distance Vector Routing Graph\n");
        sb.append("Nodes:\n");
        for (String key : nodes.keySet()){
            sb.append(nodes.get(key).toString() + "\n");
        }
        sb.append("Edges:\n");
        for (DVEdge e : edges){
            sb.append(e.toString() + "\n");
        }
        sb.append("END\n");
        return sb.toString();
    }
}
```

DVEdge.java

```
/**
 * Distance Vector Routing edge connecting two DV Nodes with a weight
 */
public class DVEdge {

    private DVNode a;
    private DVNode b;
    private int weight;

    public DVEdge(DVNode a, DVNode b, int weight){
        this.a = a;
        this.b = b;
        this.weight = weight;
    }

    public DVNode getA() {
        return a;
    }

    public void setA(DVNode a) {
        this.a = a;
    }

    public DVNode getB() {
        return b;
    }

    public void setB(DVNode b) {
        this.b = b;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Edge{" +
                "a=" + a +
                ", b=" + b +
                ", weight=" + weight +
                '}';
    }
}
```

DVNode.java

```
import java.util.*;

/**
 * Simple Distance Vector Routing Node object
 */
public class DVNode {

    private String label;
    private RoutingTable rt;
    private ArrayList<DVNeighbour> neighbours;
    private ArrayList<CVector> recievedVectors;
    private boolean stable;

    public DVNode(String label){
        this.label = label;
        this.rt = new RoutingTable(this);
        neighbours = new ArrayList<DVNeighbour>();
        recievedVectors = new ArrayList<CVector>();
    }

    public String getLabel() { return label; }

    public void setLabel(String label) {
        this.label = label;
    }

    public RoutingTable getRt() { return rt; }

    public boolean isStable() {
        return stable;
    }

    public void setStable(boolean stable) {
        this.stable = stable;
    }

    public ArrayList<DVNeighbour> getNeighbours(){
            return neighbours;
    }

    public void addNeighbour(DVNeighbour n){
        // Check if neighbour already exists
        if (!neighbours.contains(n)){
            neighbours.add(n);
        }
    }

    public void removeNeighbour(DVNeighbour n){
        // Check if neighbour exists
        if (neighbours.contains(n)){
            neighbours.remove(n);
        }
    }

    public void printNeighbours(){
        System.out.println(this.getLabel());
        for (DVNeighbour n : neighbours){
            System.out.println(n);
        }
    }

    // Get a vector and add it to list of recieved vectors
    public void recieveVector(CVector vector){
        recievedVectors.add(vector);
    }

    public void sendVectors(boolean split){
        // Send routing table vector to all neighbours
        // NOTE: Add split horizon here
        for (DVNeighbour n : neighbours){
            n.recieveVector(rt.getVector(n, split));
        }
    }

    public void calculateVectors(){
        // Give routing table neighbour vectors and calculate new costs
        rt.calculateNewCosts(recievedVectors, neighbours);
        recievedVectors.clear();
    }

    public DVNode getNextLink(DVNode n){
        // Get next link from routing table;
        return rt.getNextLink(n);
    }

    @Override
    public String toString() {
        return "Node {'" +
                label + '\'' +
                '}';
    }
}
```

DVNeighbour.java

```
/**
 * Data class to store direct neighbour links and their cost
 */
public class DVNeighbour {

    private DVNode neighbour;
    private int cost;

    public DVNeighbour(DVNode n, int c){
        neighbour = n;
        cost = c;
    }

    public DVNode getNeighbour() {
        return neighbour;
    }

    public void setNeighbour(DVNode neighbour) {
        this.neighbour = neighbour;
    }

    public int getCost() {
        return cost;
    }
    public void setCost(int cost) {
        this.cost = cost;
    }

    public void recieveVector(CVector c){
        neighbour.recieveVector(c);
    }

    @Override
    public String toString() {
        return "Neighbour{" +
                "neighbour=" + neighbour.getLabel() +
                ", cost=" + cost +
                '}';
    }
}
```

CVector.java

```
import java.util.ArrayList;
import java.util.HashMap;

/**
 * Vector class to store costs data and reference node
 */
public class CVector {

    private DVNode node;
    private HashMap<DVNode, Integer> vector;

    public CVector(DVNode n){
        node = n;
        vector = new HashMap<DVNode, Integer>();
    }

    public DVNode getNode() {
        return node;
    }

    public void setNode(DVNode node) {
        this.node = node;
    }

    public HashMap<DVNode, Integer> getVector() {
        return vector;
    }

    public void clear(){
        vector.clear();
    }

    public void addCost(DVNode node, int cost){
        vector.put(node, cost);
    }

    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append("Vector " + node.getLabel() + "\n");
        for (DVNode key : vector.keySet()) {
            sb.append("Node: " + key.getLabel() + "; Cost: " + vector.get(key) +
"\n");
        }
        return sb.toString();
    }
}
```

RoutingTable.java

```
import java.util.*;

/**
 * Abstracted Data Structure for nodes to hold routing information
 */
public class RoutingTable {

    private LinkedList<RTRow> table;
    private CVector vector;
    private DVNode tableNode;

    public RoutingTable(DVNode n) {
        table = new LinkedList<RTRow>();
        // Add self to table with cost 0
        table.add(new RTRow(n, 0, null));
        vector = new CVector(n);
        tableNode = n;
    }

    // Helper function to find the minimum item in a list
    public static int minIndex (ArrayList<Integer> list) {
        return list.indexOf (Collections.min(list)); }

    public void addRow(RTRow row) {
        table.add(row);
    }

    public void deleteRow(DVNode n) {
        for (RTRow r : table){
            if (r.getNode().equals(n)){
                table.remove(table.indexOf(r));
            }
        }
    }

    public void changeRow(DVNode n) {
        for (RTRow r : table){
            if (r.getNode().equals(n)){
                r.setCost(Integer.MAX_VALUE);
            }
        }
    }


    public CVector getVector(DVNeighbour n, boolean split){
         //System.out.println(tableNode.getLabel() + " creating vector for " +
n.getNeighbour().getLabel());
        // Clear previous vector information
        vector = new CVector(tableNode);
        // Gather data from cost column
        if (!table.isEmpty()) {
            for (RTRow r : table) {
                // Check for split horizon to avoid loops
                if (split && n.getNeighbour().equals(r.getNextLink())){
                       //System.out.println("Omitting: "+ r);
                    continue;
                }
```

```
                   //System.out.println("Adding: "+ r);
                vector.addCost(r.getNode(), r.getCost());
            }
        }
        return vector;
    }

    public DVNode getNextLink(DVNode n){
        for (RTRow r : table){
            if (r.getNode() == n){
                return r.getNextLink();
            }
        }
        return null;
    }

    // Convert entire table to string for comparison;
    public String convertToString(){
        StringBuffer sb = new StringBuffer();
        for (RTRow r : table) {
            sb.append(r);
        }
        return sb.toString();
    }

    // Get the routing table in a readable manner
    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append("-----------------------------\n");
        sb.append("Table for " + tableNode.getLabel() + "\n");
        for (RTRow r : table){
            sb.append(r.toString() + "\n");
        }
        sb.append("Stable: " + tableNode.isStable() + "\n");
        sb.append("-----------------------------\n");
        return sb.toString();
    }

    public void calculateNewCosts(ArrayList<CVector> vectors,
ArrayList<DVNeighbour> neighbours) {
        // Update routing table based on cost data from neighbours
        // Clear current costs
        String tableString = this.convertToString();
        table = new LinkedList<RTRow>();
        // Add node back into its own table;
        table.add(new RTRow(tableNode, 0, null));
        HashMap<DVNode, ArrayList<vectorElement>> knownRoutes = new
HashMap<DVNode, ArrayList<vectorElement>>();
        for (CVector cv : vectors) {
            int totalcost = 0;
            for (DVNode n: cv.getVector().keySet()){
                if (n.getLabel().equals(tableNode.getLabel())){
                    continue;
                }
                // If we don't have an entry for the node we're asking about,
add it
                if (!knownRoutes.containsKey(n)){
                    knownRoutes.put(n, new ArrayList<vectorElement>());
                }
                // Add the node being checked (for next link), and its total
cost, factoring in neighbour links
                // Check if the neighbour exists
                boolean found = false;
                for (DVNeighbour nb : neighbours){
                    if
(nb.getNeighbour().getLabel().equals((cv.getNode().getLabel()))){
                        // check for broken links
                        if (nb.getCost() == Integer.MAX_VALUE ||
cv.getVector().get(n) == Integer.MAX_VALUE) {
                            totalcost = Integer.MAX_VALUE;
                        }
                                else {
                            totalcost = nb.getCost() + cv.getVector().get(n);
                        }
                        found = true;
                    }
                }
                   // Add this item as a new vectorElement under node key (Node:
{<NextLink, Cost>,...})
                if (found){
                    knownRoutes.get(n).add(new vectorElement(cv.getNode(),
totalcost));
                }
                else {
                    System.out.println("Neighbour not found: " + cv.getNode());
                }
            }
        }
        // We should now have a list of all reachable nodes, with a list of path
cost and next link nodes
        ArrayList<vectorElement> routes = new ArrayList<vectorElement>();
        int minValue = Integer.MAX_VALUE;
        DVNode thisNode = null;
        DVNode minNode = null;
        // First, find the minimum value and the next link node
        for (DVNode key : knownRoutes.keySet()){
            routes = knownRoutes.get(key);
              // System.out.println("Key: " + key);
            for (vectorElement ve : routes){
                  // System.out.println("Node: " + ve.getNode().getLabel() + ";
Cost: " + ve.getCost());
                if (ve.getCost() < minValue){
                    minValue = ve.getCost();
                    minNode = ve.getNode();
                }
            }
            // Found lowest cost for this key, add it to table
            thisNode = key;
            RTRow row = new RTRow(thisNode, minValue, minNode);
            table.add(row);
            minValue = Integer.MAX_VALUE;
        }
         // Check if current table matched the previous table
        String newTableString = convertToString();
        if (tableString.equals(newTableString)){
            tableNode.setStable(true);
        }
         else {
               tableNode.setStable(false);
            }
    }
}

// Temporary class for storing vector elements
class vectorElement {
    private DVNode node;
    private int cost;
    public vectorElement(DVNode n, int c){
        node = n ;
        cost = c;
    }

    public DVNode getNode(){
        return node;
    }

    public int getCost(){
        return cost;
    }
}
```

RtRow.java

```
/**
 * Container object that represents a routing table row, storing three different
elements
 */
public class RTRow {

    private DVNode node;
    private int cost;
    private DVNode nextLink;

    public RTRow(DVNode node, int cost, DVNode nextLink) {
        this.node = node;
        this.cost = cost;
        this.nextLink = nextLink;
    }

    public DVNode getNode() { return node; }

    public void setNode(DVNode node) { this.node = node; }

    public int getCost() { return cost; }

    public void setCost(int cost) { this.cost = cost; }

    public DVNode getNextLink() { return nextLink; }

    public void setNextlink(DVNode nextlink) {
        this.nextLink = nextLink;
    }

    @Override
    public String toString() {
        if (nextLink == null){
            return  "node: '" + node.getLabel() + '\'' + ", cost: " + cost + ",
nextlink: None";
        }
        else {
            return  "node: '" + node.getLabel() + '\'' + ", cost: " + cost + ",
nextlink: " + nextLink.getLabel();
        }
    }
}
```