```
import sun.nio.ch.Net;
import sun.security.x509.GeneralName;

import java.util.*;
import java.io.*;
/**
 * Created by Arthur.
 * This code can be used to calculate the characteristics
 */
public class Characteristics {
    public static int p;
    public static int t;
    public static Vector<Vector<Network>> edges;
    public static Vector<Network> vertices;
    public static Vector<Vector<Node>> disjointPaths;
    public static Network N;
    public static Network spanningTree;
    public static BipartiteGraph Gn;
    public static BipartiteGraph Zn;
    public static Vector<Network> visited;
    public static Boolean enewick;

    public static void main(String[] args) {
        String networkFile = args[0];
        File file = new File(networkFile);
        BufferedReader reader = null;
        String newick = null;
        Vector<String> newicks = new Vector<>();
        String edge = null;
        Vector<String> edges = new Vector<>();
        enewick = false;
        try {
            reader = new BufferedReader(new FileReader(file));
            if ((newick = reader.readLine()) != null) {
                if (newick.substring(0, 1).equals("(")) {
                    enewick = true;
                }
            }
            reader.close();
            reader = new BufferedReader(new FileReader(file));
```

```
    if (enewick) {
        while ((newick = reader.readLine()) != null) {
            newicks.add(newick);
        }
    } else {
        while ((edge = reader.readLine()) != null) {
            edges.add(edge);
        }
    }
    reader.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
    return;
} catch (IOException e) {
    e.printStackTrace();
    return;
}
if (enewick){
    for (String n : newicks) {
        N = Network.newick2netwerk(n);
    }
}
else {
    visited = new Vector<>();
    for (String n: edges) {
        String[] split = n.split(" ");
        Network node1 = contains(split[0]);
        if (node1 == null){
            node1 = new Network();
            node1.label = split[0];
        }
        Network node2 = contains(split[1]);
        if (node2 == null){
            node2 = new Network();
            node2.label = split[1];
        }
        node1.children.add(node2);
        node2.parents.add(node1);
        visited.add(node1);
        visited.add(node2);
```

```
        }

        N = visited.firstElement();
        while(N.parents.size()>0){
            N = N.parents.firstElement();
        }
        N.isRoot = true;
        N.setup(0);
        N.setUnseen();
    }
    // N is the root node

    //Create a bipartite graph as defined by GN
    p = vertexDisjointPaths();
    System.out.println("p(N) = " + p);
    System.out.println("l(N) = " + p);
    N.toDot("DotNetwork");

    rootedSpanningTree();
    //SpanningTree is the Network.
    N.toDot("DotNetwork", "DotSpanningTree", spanningTree);

    /*
    N.setUnseen();
    int numRets = N.countRets(0);
    System.out.println("rets = " + numRets);
    N.setUnseen();
    */


    treeBasedNetwork();
    System.out.println("t(N) = " + t);
    /*System.out.println(N.toString());
    System.out.println(spanningTree.toString());
    */
    N.toDot("DotTreeBasedVersion");
}


public static void setLabels(Network net){
```

```
    if (net.label == null) {
        net.label = Integer.toString(net.number);
    }
    for (Network child: net.children){
        setLabels(child);
    }
}

public static Network contains(String n){
    if (visited == null){
        return null;
    }
    for (Network node: visited) {
        if (node.label.equals(n)){
            visited.remove(node);
            return node;
        }
    }
    return null;
}

/**
 * vertexDisjointPaths is used to calculate the characteristic p(N).
 * @return p, the minimum number of vertex disjoint paths that partition the v
 */
public static int vertexDisjointPaths(){
    // this also numbers the vertices
    N.cleanNetwork();
    int[] num = new int[1];
    num[0] = Network.TAXON_LABELS.size() + 1;
    vertices = N.getVertices(num);
    if (enewick){
        setLabels(N);
    }

    // construct vector with all edges
    edges = N.getEdges();

    //Create the bipartite graph Gn.
    Gn = network2BipartiteGN(num);
```

```
    //Finds the maximum matching in Gn
    Gn = findMaxMatching(Gn);

    //Find the unmatched nodes in V2
    Vector<Node> unmatchedV2 = findUnmatchedV2(Gn);

    //Generate P with unique maximum sequences. Used for p(N) and t(N).
    disjointPaths = findUniqueMaxSequences(Gn, unmatchedV2);

    //Calculate characteristic p
    int p = disjointPaths.size()- numOfLeaves(vertices);
    return p;
}

/**
 * Creates a Bipartite network Gn from a set of edges.
 * @param total highest numbered label.
 * @return the bipartite graph Gn with set V1 and V2 with a matching.
 */
public static BipartiteGraph network2BipartiteGN(int[] total){
    BipartiteGraph GN = new BipartiteGraph();
    GN.Edges = edges;
    GN.maxNumLabel = total[0];
    N.setUnseen();
    for (Vector<Network> e: edges) {
        Network start = e.get(0);
        Network end = e.get(1);

        Node startNode;
        Node endNode;

        if(!start.seen) {
            start.seen = true;
            startNode = new Node(start.number,true, start.label);
            GN.V1.add(startNode);
            GN.V2.add(new Node(start.number + GN.maxNumLabel, start.label));
        }
        else{
            startNode =  GN.getNodeV1(start.number);
```

```
        }

        if(!end.seen) {
            end.seen = true;
            endNode = new Node(end.number + GN.maxNumLabel, end.label);
            GN.V1.add(new Node(end.number,true, end.label));
            GN.V2.add(endNode);
        }
        else{
            endNode =  GN.getNodeV2(end.number + GN.maxNumLabel);
        }

        startNode.neighbours.add(endNode);
        endNode.neighbours.add(startNode);

        if (startNode.matched == null && endNode.matched == null){
            startNode.matched = endNode;
            endNode.matched = startNode;
        }

    }
    return GN;
}

/**
 * This function finds a maximum matching and changes the graph accordingly.
 * @param Gn with a matching
 * @return Gn with maximum matching
 */
public static BipartiteGraph findMaxMatching(BipartiteGraph Gn){
    BipartiteMatching maxGn = new BipartiteMatching(Gn);
    return maxGn.Gn;
}

/**
 * Function to return all unmatched nodes in V2.
 * @param Gn
 * @return
 */
public static Vector<Node> findUnmatchedV2(BipartiteGraph Gn){
```

```
    Vector<Node> unmatched = new Vector<>();
    for (Node n : Gn.V2){
        if (n.matched == null){
            unmatched.add(n);
        }
    }
    return unmatched;
}

/**
 * Function to find all unique maximum sequences.
 * @param Gn
 * @param U2
 * @return all unique maximum sequences.
 */
public static Vector<Vector<Node>> findUniqueMaxSequences(BipartiteGraph Gn, V
    Vector<Vector<Node>> paths = new Vector<>();
    for (Node n: U2){
        Vector<Node> p = new Vector<>();
        p.add(n);
        Node next = Gn.getNodeV1(n.number% Gn.maxNumLabel);
        while (next.matched != null){
            p.add(next.matched);
            next = Gn.getNodeV1(next.matched.number% Gn.maxNumLabel);
        }
        paths.add(p);
    }
    for (Vector<Node> path: paths){
        for (Node n: path){
            n.number = n.number%Gn.maxNumLabel;
        }
    }
    return paths;
}

/**
 * function to count all leaves in the network.
 * @param vertices are all the vertices of the network.
 * @return the number of leaves.
 */
```

```
public static int numOfLeaves(Vector<Network> vertices){
    int i = 0;
    for (Network n: vertices){
        if (n.children.size()==0){
            i++;
        }
    }
    return i;
}

public static void rootedSpanningTree(){
    //Find the path in P traversing the root of N, denoted by pi_rho;
    Vector<Node> rootPath = findRootPath();
    extendP();
    createSpanningTree(rootPath);
}

/**
 * Finds the path in P which traverses the root of N
 * @return the path which traverses the root of N
 */
public static Vector<Node> findRootPath(){
    for (Vector<Node> path: disjointPaths) {
        for(Node n: path){
            if(n.number == N.number){
                disjointPaths.remove(path);
                return path;
            }
        }
    }
    return null;
}

public static void extendP(){
    for (Vector<Node> path: disjointPaths){
        Node extendThis = path.get(0);
        Node extendTo = extendThis.neighbours.get(0);
        path.add(extendTo);
    }
}
```

```
public static Network createSpanningTree(Vector<Node> rootPath){
    boolean[] isMade = new boolean [Gn.maxNumLabel];
    Network[] allST = new Network[Gn.maxNumLabel];

    for(int i=0; i<rootPath.size()-1;i++){
        Node currentNode = rootPath.get(i);
        Network current;
        Network next;
        if(!isMade[currentNode.number]){
            isMade[currentNode.number] = true;
            current = new Network(currentNode);
            spanningTree = current;
            spanningTree.isRoot = true;
            allST[current.number] = current;
        }
        else{
            current = allST[currentNode.number];
        }

        Node nextNode = rootPath.get(i+1);
        if(!isMade[nextNode.number]){
            isMade[nextNode.number] = true;
            next = new Network(nextNode);
            allST[next.number] = next;
        }
        else{
            next = allST[nextNode.number];
        }
        current.children.add(next);
        next.parents.add(current);
    }

    for(Vector<Node> path: disjointPaths){
        Node currentNode = path.remove(path.size()-1);
        Network current;
        Network next;
        if(!isMade[currentNode.number]){
            isMade[currentNode.number] = true;
            current =  new Network(currentNode);
```

```
            allST[current.number] = current;
        }
        else{
            current =  allST[currentNode.number];
        }

        Node nextNode = path.get(0);
        if(!isMade[nextNode.number]){
            isMade[nextNode.number] = true;
            next =  new Network(nextNode);
            allST[next.number] = next;
        }
        else{
            next =  allST[nextNode.number];
        }
        current.children.add(next);
        next.parents.add(current);

        for(int i=1; i<path.size();i++){
            current = next;
            nextNode = path.get(i);
            if(!isMade[nextNode.number]){
                isMade[nextNode.number] = true;
                next =  new Network(nextNode);
                allST[next.number] = next;
            }
            else{
                next =  allST[nextNode.number];
            }
            current.children.add(next);
            next.parents.add(current);
        }
    }
    setLeaves(spanningTree);
    return spanningTree;
}

public static void setLeaves(Network N){
    for (Network child: N.children){
        if (child.children.size() == 0){
```

```
            child.isLeaf = true;
        }
        setLeaves(child);
    }
}


public static void treeBasedNetwork(){
    int numOfLabels = Gn.maxNumLabel;
    N.setUnseen();

    Vector<Network> reticulations = new Vector<>();
    Vector<Network> rets = findReticulations(N, reticulations);
    Zn = network2BipartiteZN(numOfLabels, rets);

    Zn = findMaxMatching(Zn);
    Vector<Node> unmatchedRets = findUnmatchedRets(Zn);
    t = attachLeaves(unmatchedRets, rets, Zn.maxNumLabel);
}

public static Vector<Network> findReticulations(Network N, Vector<Network> ret
    for (Network child: N.children) {
        if (!child.seen) {
            child.seen = true;
            if (child.parents.size() > 1) {
                rets.add(child);
            }
        }
        rets = findReticulations(child, rets);
    }
    return rets;
}

public static BipartiteGraph network2BipartiteZN(int total, Vector<Network> re
    BipartiteGraph Zn = new BipartiteGraph();
    N.setUnseen();

    for (Network ret: rets) {
        for (Network par : ret.parents) {
            Vector<Network> edge = new Vector<>();
```

```
                edge.add(ret);
                edge.add(par);
                Zn.Edges.add(edge);
            }

            Node r;
            if (!ret.retSeen) {
                ret.retSeen = true;
                r = new Node(ret.number, true, ret.label);
                Zn.V1.add(r);
            }
            else {
                r = Zn.getNodeV1(ret.number);
            }

            for (Network par : ret.parents){
                Node p;
                if (par.parents.size() < 2) {
                    if (!par.parSeen) {
                        par.parSeen = true;
                        p = new Node(par.number, par.label);
                        if (par.parents.size() > 1) {
                            p.isRet = true;
                        }
                        Zn.V2.add(p);
                    } else {
                        p = Zn.getNodeV2(par.number);
                    }
                    r.addNeighbour(p);

                    if (r.matched == null && p.matched == null){
                        r.matched = p;
                        p.matched = r;
                    }
                }
            }
        }
        Zn.maxNumLabel = total;
        return Zn;
    }
```

```
public static Vector<Node> findUnmatchedRets(BipartiteGraph Zn){
    Vector<Node> unmatched = new Vector<>();
    for (Node n : Zn.V1){
        if (n.matched == null){
            unmatched.add(n);
        }
    }
    for (Node n: Zn.V2){
        if (n.matched == null && n.isRet){
            unmatched.add(n);
        }
    }
    return unmatched;
}

public static int attachLeaves(Vector<Node> unmatchedRets, Vector<Network> ret
    int addedleaves = 0;
    for (Node unRet: unmatchedRets){
        Network ret = new Network();
        for (Network r: rets) {
            if (unRet.number == r.number) {
                ret = r;
                break;
            }
        }
        Network parent = ret.parents.get(0);
        parent.children.remove(ret);
        ret.parents.remove(parent);

        Network extra1 = new Network();
        labelNum++;
        extra1.number = labelNum;
        extra1.label = "Extra " + Integer.toString(labelNum);

        Network extra2 = new Network();
        labelNum++;
        extra2.number = labelNum;
        extra2.label = "Extra " + Integer.toString(labelNum);
```

```
            parent.children.add(extra1);

            extra1.parents.add(parent);
            extra1.children.add(extra2);
            extra1.children.add(ret);

            extra2.parents.add(extra1);
            extra2.isLeaf = true;
            addedleaves++;
        }
        return addedleaves;
    }
}
```

```
import java.util.*;

/**
 * Nodes used in the bipartite graph representations
 */

class Node {
    Vector<Node> neighbours;
    Node matched;
    int number;
    boolean isRet;
    boolean isV1 = false;
    String label;
```

```
    public Node(int n, String l){
        number = n;
        neighbours = new Vector<>();
        matched = null;
        isRet = false;
        label = l;
    }

    public Node(int n, boolean b, String l){
        number = n;
        neighbours = new Vector<>();
        matched = null;
        isV1 = b;
        label = l;
    }

    /**
     * Adds Node neighbour as a neighbour.
     * @param node to be added as a neighbour
     */
    public void addNeighbour(Node node) {
        this.neighbours.add(node);
        node.getNeighbours().add(this);
    }

    /**
     * @return neighbours of the Node.
     */
    public Vector<Node> getNeighbours() {
        return neighbours;
    }
}
```

```
import java.util.*;
import java.io.*;
```

```
class Network {

    Vector<Network> children;
    Vector<Network> parents;
    Vector<Double> retEdgeSupport;
    boolean isLeaf;
    int number;
    String label;
    Vector TreeVertices;
    boolean isRoot;
    boolean processed;
    int retNum;
    boolean seen;
    boolean retSeen;
    boolean parSeen;
    static Vector<String> TAXON_LABELS = new Vector();
    int MAX_RET;

    public Network() {
        isLeaf = false;
        label = null;
        parents = new Vector<>();
        children = new Vector<>();
        TreeVertices = new Vector<>();
        retNum = -1;
        seen = false;
        processed = false;
        isRoot = false;
        number = 0;
        retEdgeSupport = new Vector<>();
    }

    public Network(Node n){
        isLeaf = false;
        label = n.label;
        parents = new Vector<>();
        children = new Vector<>();
        TreeVertices = new Vector<>();
        retNum = -1;
```

```
    seen = false;
    processed = false;
    isRoot = false;
    number = n.number;
    retEdgeSupport = new Vector<>();
}

public void setup(int ret){
    for (Network n: this.children){
        if (!n.seen){
            n.seen = true;
            if (n.parents.size()>1){
                n.retNum=ret;
                ret += 1;
                TAXON_LABELS.add(n.label);
            }
            if (n.children.size()==0){
                n.isLeaf = true;
                TAXON_LABELS.add(n.label);
            }
            n.setup(ret);
        }
    }
}

public static Network newick2netwerk(String newick) {
    if (newick.endsWith(";")) {
        int lastclosepar = newick.lastIndexOf(")");
        newick = newick.substring(0, lastclosepar + 1);
    } else {
        return null;
    }
    Network N = newick2netwerk(newick, new Vector<>());
    N.isRoot = true;
    N.cleanNetwork();

    // suppress indegree-1 outdegree-1
    N.suppress();

    return N;
```

```
}

public static Network newick2netwerk(String newick, Vector<Network> reticulati
    int lastclosepar = newick.lastIndexOf(")");
    int lasthash = newick.lastIndexOf("#");
    int lastcolon = newick.lastIndexOf(":");

    // get rid of weights
    if (lastcolon > lastclosepar & lastcolon > lasthash) {
        return newick2netwerk(newick.substring(0, lastcolon), reticulations);
    }

    Network N = new Network();

    if (newick.startsWith("(")) {
        if (lastclosepar < newick.length() - 1 && newick.charAt(lastclosepar +
            // a new reticulation
            reticulations.add(N);
            N.retNum = new Integer(newick.substring(lastclosepar + 3, newick.l
            Network child = newick2netwerk(newick.substring(0, lastclosepar +
            N.children.add(child);
            child.parents.add(N);
            return N;
        } else {
            // split vertex
            int openpar = 0;
            int closepar = 0;
            int start = 1;
            Vector<String> childrenNewick = new Vector<>();
            for (int i = 0; i < newick.length(); i++) {
                if (newick.charAt(i) == '(') {
                    openpar++;
                }
                if (newick.charAt(i) == ')') {
                    closepar++;
                }
                if ((openpar == closepar + 1) && (newick.charAt(i) == ',')) {
                    childrenNewick.add(newick.substring(start, i));
                    start = i + 1;
                }
```

```
                if (i == newick.length() - 1) {
                    childrenNewick.add(newick.substring(start, i));
                }
            }

            for (String childNewick : childrenNewick) {
                Network child = newick2netwerk(childNewick, reticulations);
                N.children.add(child);
                child.parents.add(N);
            }
            return N;
        }

    } else {
        if (newick.startsWith("#H")) {
            // a reticulation
            N.retNum = Integer.parseInt(newick.substring(2, newick.length()));
            for (Network reticulation : reticulations) {
                if (reticulation.retNum == N.retNum) {
                    // an existing reticulation
                    N.children.add(reticulation);
                    reticulation.parents.add(N);
                    return N;
                }
            }
        } else {
            // a leaf
            if (newick.contains("#")) {
                // a reticulation leaf
                int hash = newick.indexOf("#");
                N.retNum = Integer.parseInt(newick.substring(hash + 2, newick.

                // check if we've already seen this reticulation
                for (Network reticulation : reticulations) {
                    if (reticulation.retNum == N.retNum) {
                        // an existing reticulation
                        N.children.add(reticulation);
                        reticulation.parents.add(N);
                        return N;
                    }
```

```
                }

                // apparently this is a new reticulation
                reticulations.add(N);
                Network child = new Network();
                child.isLeaf = true;
                String lab = newick.substring(0, hash);
                child.label = lab;
                N.children.add(child);
                child.parents.add(N);
                TAXON_LABELS.add(lab);
            } else {
                // a normal leaf
                N.isLeaf = true;
                N.label = newick;
                TAXON_LABELS.add(newick);
            }
        }
    }
    return N;
}

public Vector<Network> getVertices(int num[]) {
    Vector out = new Vector();
    if (number != 0) {
        return out; // already visited
    }
    if (isLeaf) {
        // this is a leaf
        number = TAXON_LABELS.indexOf(label) + 1;
        out.add(this);
    } else {
        number = num[0];
        out.add(this);
        for (Network child : children) {
            num[0]++;
            out.addAll(child.getVertices(num));
        }
    }
    return out;
```

```
}

public Vector<Vector<Network>> getEdges() {
    Vector out = new Vector();
    for (Network child : children) {
        Vector edge = new Vector();
        edge.add(this);
        edge.add(child);
        out.add(edge);
    }
    seen = true;
    for (Network child : children) {
        if (!child.seen) {
            out.addAll(child.getEdges());
        }
    }
    return out;
}

public void suppress() {
    for (Network child : children) {
        child.suppress();
        if (child.children.size() == 1 && child.parents.size() == 1) {
            // indegree-1 outdegree-1
            // suppress
            Network grandchild = child.children.elementAt(0);
            children.setElementAt(grandchild, children.indexOf(child));
            grandchild.parents.setElementAt(this, grandchild.parents.indexOf(c
        }
    }
}

public void cleanNetwork() {
    MAX_RET = 0;
    seen = false;
    retSeen = false;
    parSeen = false;
    processed = false;
    retNum = -1;
    number = 0;
```

```
    if (!isLeaf) {
        for (Network child : children) {
            child.cleanNetwork();
        }
    }
}

public void setUnseen(){
    seen = false;
    for (Network child: children) {
        child.setUnseen();
    }
}
/*public String toString() {
    String output;
    // returns eNewick string of the network
    if (isLeaf) {
        return label + ":1.0";
    }

    String childString1 = ((Network) children.elementAt(0)).toString();

    if (parents.size() > 1) {
        // reticulation
        if (seen) {
            output = "#H" + retNum;
        } else {
            MAX_RET++;
            retNum = MAX_RET;
            seen = true;
            if (childString1.startsWith("(")) {
                output = childString1 + "#H" + retNum;
            } else {
                output = "(" + childString1 + ")" + "#H" + retNum;
            }
        }
        return output;
    }

    output = "(" + childString1;
```

```
    for (int i = 1; i < children.size(); i++) {
        output += "," + ((Network) children.elementAt(i)).toString();
    }
    output += "):1.0";
    if (parents.isEmpty()) {
        output += ";";
    }
    return output;
}*/

public void toDot(String s){
    this.setUnseen();
    try {
        PrintWriter writer = new PrintWriter(s + ".txt", "UTF-8");
        writer.println("graph {");
        this.toDot(writer);
        writer.println("}");
        writer.close();
    }
    catch (FileNotFoundException e){
        System.out.println("FileNotFoundException: " + e.getMessage());
    }
    catch (UnsupportedEncodingException e){
        System.out.println("UnsupportedEncodingException: " + e.getMessage());
    }
}

public void toDot(String networkname, String stname, Network spanningTree){
    this.setUnseen();
    spanningTree.setUnseen();
    try {
        PrintWriter writer = new PrintWriter(networkname + ".txt", "UTF-8");
        writer.println("graph {");
        this.toDot(writer);
        writer.println("}");
        writer.close();

        this.setUnseen();
```

```
            writer = new PrintWriter(stname + ".txt", "UTF-8");
            writer.println("strict graph {");
            this.toDot(writer);
            spanningTree.toDotSt(writer);
            writer.println("}");
            writer.close();
        }
        catch (FileNotFoundException e){
            System.out.println("FileNotFoundException: " + e.getMessage());
        }
        catch (UnsupportedEncodingException e){
            System.out.println("UnsupportedEncodingException: " + e.getMessage());
        }
    }

    public void toDot(PrintWriter writer){
        for (Network child: this.children){
            if (!child.seen) {
                child.seen = true;
                writer.println("\t\"" + this.label + "\" -- \"" + child.label + "\
                child.toDot(writer);
            }
            else{
                writer.println("\t\"" + this.label + "\" -- \"" + child.label + "\
            }
        }
    }

    public void toDotSt(PrintWriter writer){
        for (Network child: this.children){
            if (!child.seen) {
                child.seen = true;
                writer.println("\t\"" + this.label + "\" -- \"" + child.label + "\
                child.toDotSt(writer);
            }
            else{
                writer.println("\t\"" + this.label + "\" -- \"" + child.label + "\
            }
        }
    }
```

```
    public int countRets(int rets){
        for (Network child: this.children){
            if(!child.seen) {
                child.seen = true;
                if (this.parents.size() > 1) {
                    rets++;
                }
            }
            rets = child.countRets(rets);
        }
        return rets;
    }

}
```

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;

public class BipartiteMatching {

    public BipartiteGraph Gn;
    private boolean[] visited;            // visited[v] = true iff v is reachable
    private int[] edgeTo;                // edgeTo[v] = w if v-w is last edge on p

    /**
     * Determines a maximum matching (and a minimum vertex cover)
     * in a bipartite graph.
     *
     * @param  bGraph the bipartite graph
     */
    public BipartiteMatching(BipartiteGraph bGraph) {
        Gn = bGraph;
        // alternating path algorithm
        while (hasAugmentingPath(Gn)) {
```

```
        // find one endpoint t in alternating path
        Node t = null;
        Vector<Node> V = new Vector<>();
        V.addAll(Gn.V1);
        V.addAll(Gn.V2);
        for(Node n : V){
            if(n.matched == null && edgeTo[n.number] != -1){
                t = n;
                break;
            }
        }

        // update the matching according to alternating path in edgeTo[] array
        for (int v = t.number; v != -1; v = edgeTo[edgeTo[v]]) {
            Node v1;
            Node w1;
            if (t.isV1) {
                v1 = Gn.getNodeV1(v);
                w1 = Gn.getNodeV2(edgeTo[v]);
            } else {
                v1 = Gn.getNodeV2(v);
                w1 = Gn.getNodeV1(edgeTo[v]);
            }
            v1.matched = w1;
            w1.matched = v1;
        }
    }
}


/**
 * an alternating path is a path whose edges belong alternately to the matchin
 * to the matching
 *
 * an augmenting path is an alternating path that starts and ends at unmatched
 */
private boolean hasAugmentingPath(BipartiteGraph Gn) {
    visited = new boolean[Gn.maxNumLabel*2];
    edgeTo = new int[Gn.maxNumLabel*2];
```

```
        for (int i = 0; i < Gn.maxNumLabel*2; i++) {
            edgeTo[i] = -1;
        }

        // breadth-first search (starting from all unmatched vertices on one side
        Queue<Node> queue = new LinkedList<>();
        for (Node n : Gn.V1) {
            if (n.matched == null) {
                queue.add(n);
                visited[n.number] = true;
            }
        }

        // run BFS, stopping as soon as an alternating path is found
        while (!queue.isEmpty()) {
            Node s = queue.poll();
            for (Node t : s.neighbours) {
                // either (1) forward edge not in matching or (2) backward edge in
                if (isResidualGraphEdge(s, t) && !visited[t.number]) {
                    edgeTo[t.number] = s.number;
                    visited[t.number] = true;
                    if (t.matched == null){
                        return true;
                    }
                    queue.add(t);
                }
            }
        }


        return false;
    }

    // is the edge s-t a forward edge not in the matching or a reverse edge in the
    private boolean isResidualGraphEdge(Node s, Node t) {
        if ((s.matched != t) &&  s.isV1) return true;
        if ((s.matched == t) && !s.isV1) return true;
        return false;
    }
}
```

```
import java.util.Vector;

public class BipartiteGraph {
    Vector<Node> V1;
    Vector<Node> V2;
    Vector<Vector<Network>> Edges;
    int maxNumLabel;

    public BipartiteGraph(){
        V1 = new Vector<>();
        V2 = new Vector<>();
        Edges = new Vector<>();
    }

    public Node getNodeV1(int index){
        for (Node n: this.V1) {
            if (n.number == index) {
                return n;
            }
        }
        return null;
    }

    public Node getNodeV2(int index){
        for (Node n: this.V2) {
            if (n.number == index) {
                return n;
            }
        }
        return null;
    }
}
```