**Part A.** (20 points) Paste a copy of the completed source code for the Forwarder class below. Highlight your changes by making them **bold** (you may omit sections of the original program that contain no added code). Remember to also place a complete copy in the repository before you make your final commit. *Your committed version should have no extraneous print statements.*

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

/** Forwarder for an overlay IP router.
* This class implements a basic packet forwarder for a simplified
* overlay IP router. It runs as a separate thread.
* An application layer thread provides new packet payloads to be
* sent using the provided send() method, and retrieves newly arrived
* payloads with the receive() method. Each application layer payload
* is sent as a separate packet, where each packet includes a protocol
* field, a ttl, a source address and a destination address.
*/
public class Forwarder implements Runnable {
    private int myIp; // this node's ip address in overlay
    private int debug; // controls amount of debugging output
    private Substrate sub; // Substrate object for packet IO
    private double now; // current time in ns
    private final double sec = 1000000000; // # of ns in a second

    // forwarding table maps contains (prefix, link#) pairs
    private ArrayList<Pair<Prefix, Integer>> fwdTbl;

    // queues for communicating with SrcSnk
    private ArrayBlockingQueue<Packet> fromSrc;
    private ArrayBlockingQueue<Packet> toSnk;

    // queues for communicating with Router
    private ArrayBlockingQueue<Pair<Packet, Integer>> fromRtr;
    private ArrayBlockingQueue<Pair<Packet, Integer>> toRtr;

    private Thread myThread;
    private boolean quit;

    /** Initialize a new Forwarder object.
     * @param myIp is this node's IP address in the overlay network,
     * expressed as a raw integer.
     * @param sub is a reference to the Substrate object that this object
     */
```
* uses to handle the socket IO
* @param debug controls the amount of debugging output
*/
Forwarder(int myIp, Substrate sub, int debug) {
    this.myIp = myIp; this.sub = sub; this.debug = debug;
    // initialize forwarding table with a default route to link 0
    fwdTbl = new ArrayList<Pair<Prefix, Integer>>();
    fwdTbl.add(new Pair<Prefix, Integer>(new Prefix(0, 0), 0));
    // create queues for SrcSnk and Router
    fromSrc = new ArrayBlockingQueue<Packet>(1000, true);
    toSnk = new ArrayBlockingQueue<Packet>(1000, true);
    fromRtr = new ArrayBlockingQueue<Pair<Packet, Integer>>(1000, true);
    toRtr = new ArrayBlockingQueue<Pair<Packet, Integer>>(1000, true);
    quit = false;
}
/** Start the Forwarder running. */
public void start() throws Exception {
    myThread = new Thread(this); myThread.start();
}
/** Terminate the Forwarder. */
public void stop() throws Exception {
    quit = true;
    myThread.join();
}
/** This is the main thread for the Forwarder object.
* It inserts payloads received from the application layer into
* packets, which it sends to the substrate, after determining the
* appropriate outgoing link.
* It also sends packets from the Router to a specified link.
* In addition, it receives packets from the Substrate and
* forwards them either to the application layer, to the
* Router object or back to the substrate on a different
* outgoing link.
*/
public void run() {
    now = 0; double t0 = System.nanoTime() / sec;
    Packet p;
    Pair<Packet, Integer> pp;
    int lnk = 0;
    Pair<String, String> sp;

    while (!quit) {
        double rightnow = System.nanoTime() / sec - t0;
        synchronized(this) { now = rightnow; }
        if (sub.incoming()) {
            // substrate has an incoming packet
            pp = sub.receive();
            p = pp.left; lnk = pp.right;
            p.ttl -= 1;
            if (p.destAdr == myIp) {
                // if addressed to me, send to sink
            }
        }
    }
}
// or router; discard if buffer is full
if (p.protocol == 1) {
    toSnk.offer(p);
} else if (p.protocol == 2) {
    toRtr.offer(pp);
}
} else if (p.ttl > 0) {
    // otherwise forward if ttl not expired,
    // we have a route and substrate can
    // take it; otherwise, discard it
    int lnk = lookup(p.destAdr);
    if (lnk >= 0 && sub.ready(lnk))
        sub.send(p,lnk);
}
} else if (fromRtr.size() > 0) {
    pp = fromRtr.peek();
    if (sub.ready(pp.right)) {
        // router has a packet to be sent
        try {
            fromRtr.take();
        } catch(Exception e) {
            System.err.println("Forwarder: "
            + "can't read router "+"queue");
            System.exit(1);
        }
        sub.send(pp.left,pp.right);
    }
}
} else if (fromSrc.size() > 0) {
    p = fromSrc.peek();
    lnk = lookup(p.destAdr);
    if (sub.ready(lnk)) {
        // source has a packet to be sent
        try {
            fromSrc.take();
        } catch(Exception e) {
            System.err.println("Forwarder: 
            + "can't read source "+"queue");
            System.exit(1);
        }
        sub.send(p,lnk);
    }
} else {
    try {
        Thread.sleep(1);
    } catch(Exception e) {
        System.err.println("Forwarder:run: "
            + "can't sleep "+e);
    }
}

/** Add a route to the forwarding table.
   * @param nuPrefix is a prefix to be added
   * @param nuLnk is the number of the link on which to forward

- 3 -
* packets matching the prefix
* If the table already contains a route with the specified
* prefix, the route is updated to use nuLnk. Otherwise,
* a route is added.
* If debug>0, print the forwarding table when done
*/
public synchronized void addRoute(Prefix nuPrefix, int nuLnk) {
    for (Pair<Prefix,Integer> rte : fwdTbl) {
        if (rte.left.equals(nuPrefix)) {
            rte.right = nuLnk;
            if (debug >= 1) printTable();
            return;
        }
    }
    fwdTbl.add(new Pair<Prefix,Integer>(nuPrefix,nuLnk));
    if (debug >= 1) printTable();
}
/** Print the contents of the forwarding table. */
public synchronized void printTable() {
    String s = String.format("Forwarding table (%.3f)\n",now);
    for (Pair<Prefix,Integer> rte : fwdTbl)
        s += String.format("%s %s\n", rte.left, rte.right);
    System.out.println(s);
}
/** Lookup route in forwarding table. 
* @param ip is an integer representing an IP address to lookup 
* @return nextHop link number or -1, if no matching entry. 
*/
private synchronized int lookup(int ip) {
    int bestLink, matchLeng;
    bestLink = matchLeng = -1;
    for (Pair<Prefix,Integer> rte : fwdTbl) {
        if (rte.left.matches(ip) && rte.left.leng > matchLeng) {
            bestLink = rte.right; matchLeng = rte.left.leng;
        }
    }
    return bestLink;
}
Part B. (30 points) Paste a copy of the completed source code for the Router class below. Highlight your changes by making them bold (you may omit sections of the original program that contain no added code). Remember to also place a complete copy in the repository before you make your final commit. Your committed version should have no extraneous print statements.

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

/** Router module for an overlay router. *
 * Objects in this class implements a distributed *
 * routing algorithm in order to maintain a routing table. *
 * The routing protocol is a path-vector style protocol, but *
 * is used as an intra-domain protocol. Each router periodically *
 * sends a path vector on each of its incident links. In general, *
 * a path vector consists of an advertised prefix, a timestamp *
 * for the advertisement, a path cost and a list of IP addresses, *
 * defining the path to the router that issued the advertisement. *
 * The initial vector sent by a router lists just the advertising *
 * router's IP address and has a path cost of 0. *
 * When a router receives a path-vector from another router, *
 * it first checks to see if it is already in the path, *
 * in which case it discards the vector. If the vector is *
 * not discarded, it is extended (by adding the current router's *
 * address to the beginning of the path) and forwarded on all incident *
 * links, except the one on which it arrived. The path cost of the *
 * vector is also increased by the current cost of the link on which *
 * it arrived. *
 * The received vector is also used to update this *
 * router's routing state. If the new vector yields a path to *
 * the advertised prefix that has a lower cost than the current route *
 * for that prefix, then the current route is updated to *
 * reflect the lower cost (and possibly different next-hop). *
 * The new path vector may also cause an update to the current *
 * route if the current route is more than 30 seconds old. *
 * The router also sends periodic hello packets to each of its *
 * neighbors. These are echoed back to indicate that a peer router *
 * is still alive. These packets are also used to monitor the *
 * round-trip delay to the neighbor. These delays are used to *
 * determine link costs. *
 * The payloads of routing packets are formatted as ASCII text. *
 * Here are some examples of the format. A hello packet is simply *
 * RPv0 *
 * type: hello *
 * timestamp: 123.456 *
 * The reply to a hello packet is *
 * RPv0
```
type: hello2u
timestamp: 123.456

The timestamp in the hello packet is the time in seconds from an arbitrary starting point at the sending router. Timestamps are echoed back in the replies to hello packets.

A path vector advertisement takes the form

RPv0
type: advert
pathvec: 1.5.0.0/16 345.678 0.346 1.2.0.1 1.2.3.4 1.5.4.3

Each path vector starts with an advertised prefix, followed by a timestamp for the vector and the cost of the path. The remainder of the path vector is a list of the IP addresses of the routers along the path, ending with the router that originated the advertisement (1.5.4.3 in the example).

The timestamp for a path vector represents the time at which the advertising router sent this path vector. It is forwarded without change by other routers and used by routers when deciding how to update their routing table entries.

The cost of a path is the sum of its link costs and is expressed in seconds.

```java
public class Router implements Runnable {
    private Thread myThread; // thread that executes run() method
    private int myIp; // ip address in the overlay
    private String myIpString; // String representation
    private ArrayList<Prefix> pfxList; // list of prefixes to advertise
    private ArrayList<NborInfo> nborList; // list of info about neighbors

    private class LinkInfo { // class used to record link information
        public int peerIp; // IP address of peer in overlay net
        public double cost; // in seconds
        public boolean gotReply; // flag to detect hello replies
        public int helloState; // set to 3 when hello reply received
        // decremented whenever hello reply // is not received; when 0, link is down

        // link cost statistics
        public int count;
        public double totalCost;
        public double minCost;
        public double maxCost;

        LinkInfo() {
            cost = 0; gotReply = true; helloState = 3;
            count = 0; totalCost = 0; minCost = 10; maxCost = 0;
        }
    }
    private ArrayList<LinkInfo> lnkVec; // indexed by link number

    private class Route { // routing table entry
```
public Prefix pfx; // destination prefix for route
public double timestamp; // time this route was generated
public double cost; // cost of route in ns
public LinkedList<Integer> path; // list of router IPs;
// destination at end of list
public int outLink; // outgoing link for this route
}
private ArrayList<Route> rteTbl; // routing table
private Forwarder fwdr; // reference to Forwarder object
private double now; // current time in ns
private static final double sec = 1000000000; // ns per sec
private int debug; // controls debugging output
private boolean quit; // stop thread when true

/** Initialize a new Router object. *
 * @param myIp is an integer representing the overlay IP address of
 * this node in the overlay network
 * @param fwdr is a reference to the Forwarder object through which
 * the Router sends and receives packets
 * @param pfxList is a list of prefixes advertised by this router
 * @param nborList is a list of neighbors of this node
 * @param debug is an integer that controls the amount of debugging
 * information that is to be printed */
Router(int myIp, Forwarder fwdr, ArrayList<Prefix> pfxList,
 ArrayList<NborInfo> nborList, int debug) {
    this.myIp = myIp; this.myIpString = Util.ip2string(myIp);
    this.fwdr = fwdr; this.pfxList = pfxList;
    this.nborList = nborList; this.debug = debug;
    lnkVec = new ArrayList<LinkInfo>();
    for (NborInfo nbor : nborList) {
        LinkInfo lnk = new LinkInfo();
        lnk.peerIp = nbor.ip;
        lnk.cost = nbor.delay;
        lnkVec.add(lnk);
    }
    rteTbl = new ArrayList<Route>();
    quit = false;
}

/** Instantiate and start a thread to execute run(). */
public void start() {
    myThread = new Thread(this); myThread.start();
}

/** Terminate the thread. */
public void stop() throws Exception { quit = true; myThread.join(); }

/** This is the main thread for the Router object. */
* Every second, we send a hello packet to our neighbors
* and every 10 seconds we advertise our prefixes to our
* neighbors.
* * When we get a hello, we echo it back and when we get
* a reply to our own hello, we use it to update our link costs.
* * When we get an advertisement from a neighbor, we first check
* if we're in the path and discard it if so. If not, we update
* our own routing state and forward, as appropriate.
* */

public void run() {
    double t0 = System.nanoTime()/sec;
    now = 0;
    double helloTime, pvSendTime;
    helloTime = pvSendTime = now;
    while (!quit) {
        now = System.nanoTime()/sec - t0;
        if (now > helloTime + 1) {
            sendHellos(); helloTime = now;
        } else if (now > pvSendTime + 10) {
            sendAdverts(); pvSendTime = now;
        } else if (fwdr.incomingPkt()) {
            handleIncoming();
        } else {
            try { Thread.sleep(1);
            } catch(Exception e) {
                System.err.println("Router: run " +
                "can't sleep " + e);
                System.exit(1);
            }
        }
    }

    String s = String.format("Router link cost statistics
    peerIp count avgCost minCost maxCost
    " +
    "%s %d %.3f %.3f %.3f\n",
    Util.ip2string(lnk.peerIp), lnk.count,
    lnk.totalCost/lnk.count,
    lnk.minCost, lnk.maxCost);
    System.out.println(s);
}

/** Lookup route in routing table.
 * @param pfx is IP address prefix to be looked up.
 * @return a reference to the Route that matches the prefix or null
 */
private Route lookupRoute(Prefix pfx) {
    for (Route rte : rteTbl) {
        if (pfx.equals(rte.pfx)) return rte;
    }
    return null;
}

/** Add a route to the routing table.*/
/** @param rte is a route to be added to the table; no check is done to make sure this route does not conflict with an existing route */
private void addRoute(Route rte) {
    rteTbl.add(rte);
}

/** Update a route in the routing table. 
  * @param rte is a reference to a route in the routing table. 
  * @param nuRte is a reference to a new route that has the same prefix as rte 
  * @return true if rte is modified, else false 
  * This method replaces certain fields in rte with fields in nuRte. Specifically, 
  * if nuRte has a link field that refers to a disabled link, ignore it and return false 
  * else, if both routes have the same path and link, then the timestamp and cost fields of rte are updated 
  * else, if nuRte has a cost that is less than .9 times the cost of rte, then all fields in rte except the prefix fields are replaced with the corresponding fields in nuRte 
  * else, if nuRte is at least 20 seconds newer than rte (as indicated by their timestamps), then all fields of rte except the prefix fields are replaced 
  * else, if the link field for rte refers to a link that is currently disabled, replace all fields in rte but the prefix fields 
  */
private boolean updateRoute(Route rte, Route nuRte) {
    int lnk = rte.outLink; int nuLnk = nuRte.outLink;
    if (lnkVec.get(nuRte.outLink).helloState == 0) return false;
    if (rte.path.equals(nuRte.path) && lnk == nuLnk) {
        rte.timestamp = nuRte.timestamp;
        rte.cost = nuRte.cost;
        return true;
    }
    if (nuRte.cost < .9*rte.cost || nuRte.timestamp > rte.timestamp + 20 || lnkVec.get(rte.outLink).helloState == 0) {
        rte.cost = nuRte.cost;
        rte.timestamp = nuRte.timestamp;
        rte.path = nuRte.path;
        rte.outLink = nuRte.outLink;
        return true;
    }
    return false;
}

/** Send hello packet to all neighbors. 
  * First check for replies. If no reply received on some link,
* update the link status by subtracting 1. If that makes it 0,
* the link is considered down, so we increase the cost of all
* routes using that link.
*/
public void sendHellos() {
    int lnk = 0;
    for (LinkInfo lnkInfo : lnkVec) {
        // if no reply to the last hello, subtract 1 from
        // link status if it's not already 0
        if (!lnkInfo.gotReply && lnkInfo.helloState > 0)
            lnkInfo.helloState--;

        // send new hello, after setting gotReply to false
        lnkInfo.gotReply = false;
        Packet p = new Packet();
        p.protocol = 2; p.ttl = 100;
        p.srcAdr = myIp; p.destAdr = lnkInfo.peerIp;
        p.payload = String.format("RPv0\nntype: hello\n" + "timestamp: %.4f\n", now);
        fwdr.sendPkt(p,lnk);
    }
}

/** Send initial path vector to each of our neighbors. */
public void sendAdverts() {
    for (Prefix pfx : pfxList) {
        for (int lnk = 0; lnk < nborList.size(); lnk++) {
            if (lnkVec.get(lnk).helloState == 0) continue;
            Packet p = new Packet();
            p.protocol = 2; p.ttl = 100;
            p.srcAdr = myIp;
            p.destAdr = lnkVec.get(lnk).peerIp;
            p.payload = String.format("RPv0\nntype: advert\n" + "pathvec: %s %.3f 0 %s\n",pfx.toString(), now, myIpString);
            fwdr.sendPkt(p,lnk);
        }
    }
}

/** Retrieve and process packet received from Forwarder.
* For hello packets, we simply echo them back.
* For replies to our own hello packets, we update costs.
* For advertisements, we update routing state and propagate
* as appropriate.
*/
public void handleIncoming() {
    // parse the packet payload
    Pair<Packet,Integer> pp = fwdr.receivePkt();
    Packet p = pp.left; int lnk = pp.right;
    String[] lines = p.payload.split("\n");
    if (!lines[0].equals("RPv0")) return;
    String[] chunks = lines[1].split(":");
    if (!chunks[0].equals("type")) return;
    String type = chunks[1].trim();
// handle advertisements separately
if (type.equals("advert")) { handleAdvert(lines, lnk); return; }

// handle hello packets and replies
chunks = lines[2].split(":");
if (!chunks[0].equals("timestamp")) return;
if (type.equals("hello")) {
    // send reply
    Packet reply = new Packet();
    reply.protocol = 2; reply.ttl = 100;
    reply.srcAdr = myIp; reply.destAdr = p.srcAdr;
    reply.payload = "RPv0\ntype: hello2u\n" +
    lines[2] + "\n";
    if (fwdr.ready4pkt()) fwdr.sendPkt(reply, lnk);
    return;
}
if (!type.equals("hello2u")) return;

// handle reply by updating lnkCost
LinkInfo lnkInfo = lnkVec.get(lnk);
lnkInfo.gotReply = true; lnkInfo.helloState = 3;
double timestamp = Double.parseDouble(chunks[1]);
double cost = (now - timestamp)/2;
lnkInfo.cost = .9*lnkInfo.cost + .1*cost;
lnkInfo.count++; lnkInfo.totalCost += cost;
lnkInfo.minCost = Math.min(cost, lnkInfo.minCost);
lnkInfo.maxCost = Math.max(cost, lnkInfo.maxCost);
}

/** Handle an advertisement received from another router. *
 * @param lines is a list of lines that defines the packet;
 * the first two lines have already been processed at this point *
 * @param lnk is the number of link on which the packet was received *
 */
private void handleAdvert(String[] lines, int lnk) {
    // example path vector line
    // pathvec: 1.2.0.0/16 345.678 52 1.2.0.1 1.2.3.4
    String[] chunks = lines[2].split(":");
    if (!chunks[0].trim().equals("pathvec")) return;
    chunks = chunks[1].trim().split(" ");
    if (chunks.length < 4) return;
    Route nuRte = new Route();
    nuRte.path = new LinkedList<Integer>();
    nuRte.pfx = new Prefix(chunks[0]);
    nuRte.timestamp = Double.parseDouble(chunks[1]);
    nuRte.cost = Double.parseDouble(chunks[2])
    + lnkVec.get(lnk).cost;

    // copy path to nuRte; abort early if loop found
    for (int i = 3; i < chunks.length; i++) {
        if (chunks[i].equals(myIpString)) return;
        nuRte.path.add(Util.string2ip(chunks[i]));
    }
    nuRte.outLink = lnk;
// look for a matching route in the routing table
// and update as appropriate; whenever an update
// changes the path, print the table if debug>0;
// whenever an update changes the output link,
// update the forwarding table as well

Route rte = lookupRoute(nuRte.pfx);
if (rte == null) {
    addRoute(nuRte);
    if (debug >= 1) printTable();
    fwdr.addRoute(nuRte.pfx, nuRte.outLink);
} else {
    boolean pathChange = (!nuRte.path.equals(rte.path));
    boolean linkChange = (nuRte.outLink != rte.outLink);
    if (!updateRoute(rte, nuRte)) return;
    // print the table if debug>0 and path changed
    if (debug >= 1 && pathChange) printTable();
    // update forwarding table when link changes
    if (linkChange) {
        fwdr.addRoute(nuRte.pfx, nuRte.outLink);
    }
}

// extend path and forward to neighbors
String payload = String.format("RPv0\ntype: advert\n" + "pathvec: %s %.3f %.4f %s", nuRte.pfx.toString(),
    nuRte.timestamp, nuRte.cost, myIpString);
for (int i = 3; i < chunks.length; i++)
    payload += " " + chunks[i];
payload += "\n";

// and send it to neighbors
int outLnk = 0;
for (NborInfo nbor : nborList) {
    if (outLnk != lnk) {
        Packet p = new Packet();
        p.protocol = 2; p.ttl = 100;
        p.srcAdr = myIp; p.destAdr = nbor.ip;
        p.payload = payload;
        fwdr.sendPkt(p, outLnk);
    }
    outLnk++;
}

/** Print the contents of the routing table. */
public void printTable() {
    String s = String.format("Routing table (%.3f)%n" + "prefix timestamp cost path link%n", now);
    for (Route rte : rteTbl) {
        s += String.format("%s %.3f %.3f",
            rte.pfx.toString(), rte.timestamp, rte.cost);
        for (int r : rte.path)
            s += String.format("%s", Util.ip2string(r));
        s += String.format("%d", rte.outLink);
        if (lnkVec.get(rte.outLink).helloState == 0)
            s += String.format("** disabled link");
        s += "\n";
    }
}
System.out.println(s);
**Part C.** Put your files for this lab in the directory `~/473/lab5`. In this part, you will be running some tests using the configuration and script you will find in the `net1` sub-directory.

1. **(5 points)** Draw a diagram showing the logical links joining the three routers in the overlay network defined by the configuration files `r1`, `r2` and `r3`. Label the inter-router links with their assigned link costs.

   ![Diagram](image)

2. **(5 points)** Run `script1` in the `net1` sub-directory by typing
   ```bash
   ./script1 .333 20 static
   ```
   Paste a copy of the output below.

   ```plaintext
delta= .333  runlength= 20 static
*********** log  1  ***********
Final Report
Routing table (28.244)
prefix timestamp cost path link
1.3.0.0/16 20.001 0.013 1.3.0.1 1
1.2.0.0/16 20.001 0.022 1.2.0.1 0
Forwarding table (28.245)
0.0.0.0/0 0
1.3.0.0/16 1
1.2.0.0/16 0
Router link cost statistics
peerIp count avgCost minCost maxCost
1.2.0.1 25 0.023 0.021 0.036
1.3.0.1 25 0.013 0.011 0.027
SrcSnk statistics
destIp count avgDelay minDelay maxDelay
1.3.0.1 30 0.023 0.011 0.044
1.2.0.1 31 0.029 0.021 0.047
*********** log  2  ***********
Final Report
Routing table (28.233)
prefix timestamp cost path link
1.1.0.0/16 20.001 0.022 1.1.0.1 1
1.3.0.0/16 20.001 0.035 1.1.0.1 1.3.0.1 1
Forwarding table (28.232)
0.0.0.0/0 0
1.1.0.0/16 1
1.3.0.0/16 1
```
Router link cost statistics
peerIp  count  avgCost  minCost  maxCost
1.3.0.1  25  0.052  0.050  0.065
1.1.0.1  25  0.023  0.021  0.036

SrcSnk statistics
destIp  count  avgDelay  minDelay  maxDelay
1.3.0.1  34  0.036  0.032  0.043
1.1.0.1  27  0.031  0.021  0.044

*********** log 3 ***********

Final Report

Routing table (28.239)
prefix  timestamp  cost  path  link
1.1.0.0/16  20.001  0.012  1.1.0.1  0
1.2.0.0/16  20.001  0.034  1.1.0.1  1.2.0.1  0

Forwarding table (28.239)
0.0.0.0/0  0
1.1.0.0/16  0
1.2.0.0/16  0

Router link cost statistics
peerIp  count  avgCost  minCost  maxCost
1.1.0.1  26  0.013  0.011  0.026
1.2.0.1  25  0.052  0.051  0.065

SrcSnk statistics
destIp  count  avgDelay  minDelay  maxDelay
1.1.0.1  31  0.024  0.011  0.045
1.2.0.1  30  0.036  0.032  0.043

3. (5 points) For each pair of routers $r_i$ and $r_j$, write down the shortest path from $r_i$ to $r_j$ and the total cost of that path. Verify that the final routing tables and forwarding tables printed by script1 are consistent with these shortest paths.

\( r_1 \) to \( r_2 \): path \( r_1, r_2 \) cost .02
\( r_1 \) to \( r_3 \): path \( r_1, r_3 \) cost .01
\( r_2 \) to \( r_3 \): path \( r_2, r_1, r_3 \) cost .03

The routing table for \( r_1 \) shows that its routes to the other two routers both use the direct links. The routes from \( r_2 \) both pass through \( r_1 \), as do the routes from \( r_3 \). This is consistent with the high cost link connecting \( r_2 \) and \( r_3 \). The costs in the routing table are a little bit larger than the costs in the configuration files, which makes sense because the routing table shows actual measured delays, which include software and network delays.

The forwarding tables are consistent with the routing tables.
4. (5 points) Paste a screenshot of the monitoring window from your script1 run below.

Note how the bandwidth on some links changes part way through the run. Explain why this happens. How are packets routed during the first few seconds of the run? Why does this happen?

The routing algorithm does not distribute any advertisements until 10 seconds into the run. So, up until this point, packets are forwarded using a default forwarding table entry that sends all packets to link 0. This causes r1 to forward all of its packets to r2, r2 forwards all of its packets to r3 and r3 forwards all of its packets to r1. This default forwarding table entry is placed in the routing table by the constructor for the Forwarder.

5. (5 points) How many packets per second should be sent on the link from onl router port 1.1 to onl router port 2.0 during the first part of the run? Your answer should include all packets sent by the routing algorithm and all packets sent by the SrcSnk that would travel over this link. Explain your answer. Does your answer match the observed packet rates?

R1 sends a hello packet every second to r2 over this link. It also replies to the hello packets that r2 sends to it. So that’s 2 packets per second. Now, the SrcSnk at r1 sends 3 packets per second to r2 and 3 packets per second to r3. These all travel over this link during the first part of the run. So that adds 6 packets per second, giving us 8 packets per second. Finally, the SrcSnk at r3 sends 3 packets per second to r2 that pass over this link. That brings the total to 11 packets per second, which is what we observe.

6. (5 points) How many packets per second should be sent on the link from the onl router port 1.1 to onl router port 2.0 during the second part of the run? Explain your answer. Does your answer match the observed packet rates

During this part, the packets between r1 and r3 no longer pass over the link from 1.1 to 2.0. This causes the rate to drop to 8 packets per second, which is consistent with what we observe on the chart.
7. (5 points) Run `script1` again by typing

```
./script1 333 20 static debug
```

and paste a copy of the resulting log file below. Add comments to the output in bold to explain how the advertisements trigger changes in the routing table. Also, explain why some received advertisements do not trigger changes to the routing table.

```
/192.168.7.1:31313 sending to /192.168.4.2:31313 at 10.024
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.1.0.1
RPv0
type: advert
pathvec: 1.2.0.0/16 10.000 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.4.2:31313 at 10.018
protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.1.0.0/16 10.001 0 1.1.0.1

This is the first ad received by r2. It triggers the addition of a route to the previously empty routing table at r2. This also triggers the addition of a route to the forwarding table.

Routing table (10.032)
prefix timestamp cost path link
1.1.0.0/16 10.001 0.022 1.1.0.1 1

Forwarding table (10.041)
0.0.0.0/0 0
1.1.0.0/16 1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.055
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1
RPv0
type: advert
pathvec: 1.2.0.0/16 10.000 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.4.2:31313 at 10.038
protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.3.0.0/16 10.001 0.0292 1.1.0.1 1.3.0.1

Here, we are getting an ad from r3, that comes indirectly through r1. This triggers the addition of a new route, as can be seen below.

Routing table (10.067)
prefix timestamp cost path link
1.1.0.0/16 10.001 0.022 1.1.0.1 1

This ad also comes from r3, but because the corresponding route has a higher cost than the indirect route, it does not affect the routing table.

Routing table (10.067)
prefix timestamp cost path link
1.1.0.0/16 10.001 0.022 1.1.0.1 1
1.3.0.0/16 10.001 0.051 1.1.0.1 1.3.0.1 1

Forwarding table (10.072)

0.0.0.0/0 0
1.1.0.0/16 1
1.3.0.0/16 1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.079
protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.1.0.0/16 10.001 0.0127 1.3.0.1 1.1.0.1

This ad originated at r1, but was forwarded through r3. Because has a higher cost than the direct route in the table, it had no effect

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.097
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1
RPv0
type: advert
pathvec: 1.1.0.0/16 10.001 0.0221 1.2.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.128
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1
RPv0
type: advert
pathvec: 1.3.0.0/16 10.001 0.0513 1.2.0.1 1.1.0.1 1.3.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.121
protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.2.0.0/16 10.000 0.0355 1.3.0.1 1.1.0.1 1.2.0.1

This ad originated at r2, so it is simply discarded by r2.

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 20.024
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.1.0.1
RPv0
type: advert
pathvec: 1.2.0.0/16 20.000 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.4.2:31313 at 20.005
protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.1.0.0/16 20.002 0 1.1.0.1

This is a repeat of the earlier ad. This and all remaining ads have no effect on the routing table, beyond updating the timestamp and cost.

/192.168.7.1:31313 received from /192.168.4.2:31313 at 20.034
protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.3.0.0/16 20.002 0.0189 1.1.0.1 1.3.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.054
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1
RPv0
type: advert
pathvec: 1.2.0.0/16 20.000 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.055
protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.3.0.0/16 20.002 0 1.3.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.074
protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.1.0.0/16 20.002 0.0125 1.3.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.083
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1
RPv0
type: advert
pathvec: 1.1.0.0/16 20.002 0.0227 1.2.0.1 1.1.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.078
protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1
RPv0
type: advert
pathvec: 1.2.0.0/16 20.000 0.0352 1.3.0.1 1.1.0.1 1.2.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.101
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1
RPv0
type: advert
pathvec: 1.3.0.0/16 20.002 0.0416 1.2.0.1 1.1.0.1 1.3.0.1

Final Report

Routing table (28.247)
prefix timestamp cost path link
1.1.0.0/16 20.002 0.023 1.1.0.1 1
1.3.0.0/16 20.002 0.042 1.1.0.1 1.3.0.1 1

Forwarding table (28.248)
0.0.0.0/0 0
1.1.0.0/16 1
1.3.0.0/16 1

Router link cost statistics
peerIp count avgCost minCost maxCost
1.3.0.1 25 0.053 0.051 0.073
1.1.0.1 25 0.023 0.020 0.037

SrcSnk statistics
destIp count avgDelay minDelay maxDelay
1.3.0.1 28 0.037 0.032 0.043
1.1.0.1 33 0.029 0.021 0.043
8. (15 points) In this part, you will disable and re-enable one of the links while script1 is running. This is done using a filter that is installed on onl router port 1.1. Click on this port in the RLI and select “Filter Table” from the menu. This will show you a “delete filter” which causes all packets received on this link to be discarded. At the right end of the filter table entry you will see a check box. Click on the check box and select “Commit” from the file menu in order to turn on the filter (effectively disabling the link). To turn off the filter (and re-enable the link), uncheck the box and select “Commit” again.

Now, run script1 again (with the filter turned off) by typing

```
script1 .333 100 static debug
```

after the script has run for about 30 seconds, turn on the filter. Then wait another 30 seconds and turn off the filter. Paste a copy of the log2 file from your run below and add comments in bold, explaining all changes to the routing table at r2.

```
Routing table (10.035)
prefix timestamp cost path link
1.1.0.0/16 10.001 0.022 1.1.0.1 1
```

```
Forwarding table (10.044)
0.0.0.0/0 0
1.1.0.0/16 1
```

```
Routing table (10.051)
prefix timestamp cost path link
1.1.0.0/16 10.001 0.022 1.1.0.1 1
1.3.0.0/16 10.000 0.034 1.1.0.1 1.3.0.1 1
```

The routes shown above are from the initial exchange of advertisements.

```
Forwarding table (10.055)
0.0.0.0/0 0
1.1.0.0/16 1
1.3.0.0/16 1
```

```
Routing table (40.057)
prefix timestamp cost path link
1.1.0.0/16 30.002 0.022 1.1.0.1 1 ** disabled link
1.3.0.0/16 40.001 0.052 1.3.0.1 0
```

Here we see that the route to r3 has switched over to use the direct link. However, the route to r1 still uses the disabled link because we have not yet received an advertisement for the alternate path.

```
Forwarding table (40.066)
0.0.0.0/0 0
1.1.0.0/16 1
1.3.0.0/16 0
```

```
Routing table (40.082)
prefix timestamp cost path link
1.1.0.0/16 40.003 0.064 1.3.0.1 1.1.0.1 0
1.3.0.0/16 40.001 0.052 1.3.0.1 0
```

The above routing table entry was presumably triggered by the arrival of an advertisement for r1 on the indirect path. So now both r1 and r3 are reached through link 0, which goes to r3.
Forwarding table (40.086)
0.0.0.0/0 0
1.1.0.0/16 0
1.3.0.0/16 0

Routing table (80.044)
prefix timestamp cost path link
1.1.0.0/16 70.005 0.064 1.3.0.1 1.1.0.1 0
1.3.0.0/16 80.002 0.034 1.1.0.1 1.3.0.1 1

Here we see that the routing table has reverted back to the direct path for r1, as a result of the link re-enabled.

Routing table (80.048)
prefix timestamp cost path link
1.1.0.0/16 80.009 0.022 1.1.0.1 1
1.3.0.0/16 80.002 0.034 1.1.0.1 1.3.0.1 1

And finally, we see the route to r3 revert back to the indirect path, which has a lower cost.

Routing table (80.050)
prefix timestamp cost path link
1.1.0.0/16 100.011 0.022 1.1.0.1 1
1.3.0.0/16 100.003 0.034 1.1.0.1 1.3.0.1 1

Final Report

Routing table (108.167)
prefix timestamp cost path link
1.1.0.0/16 100.011 0.022 1.1.0.1 1
1.3.0.0/16 100.003 0.034 1.1.0.1 1.3.0.1 1

Router link cost statistics
peerIp count avgCost minCost maxCost
1.3.0.1 105 0.052 0.051 0.065
1.1.0.1 68 0.022 0.021 0.037

SrcSnk statistics
destIp count avgDelay minDelay maxDelay
1.3.0.1 132 0.042 0.031 0.054
1.1.0.1 153 0.040 0.021 0.065
Explain the packet rates leaving ONL router port 1.1 and 2.0 during the period when the link is disabled. Justify the specific numerical values observed.

*Before the link is disabled, we see an average of 8 packets per second leaving ports 1.1 and 2.0. This corresponds to 2 packets per second for hello packets and replies, plus six packets per second for the application packets sent between r1 and r2 and between r3 and r2. After the link is disabled, this drops to 1 packet per second from 1.1 and 2 packets per second from 2.0. It seems surprising that it should be non-zero, until we realize that the filter only stops the packets entering port 1.1. So while the application packets are no longer using the link joining the two ONL routers, the hello packets from both routers are still being sent on this link, and the replies from r2 are being sent to r1.*
**Part D.** In this part, you will be using the configuration in the net2 subdirectory.

1. (10 points) Using the information in the provided configuration files draw a network graph that represents this network. Each node in the graph should be labeled with the router number (e.g. \( r_1, r_2, \ldots \)) and the last two components of the IP address of its ONL host (so, for example, \( r_1 \) runs on the host whose address is 192.168.6.1, so label its node in the graph as with “\( r_1/6.1 \)”). Each link should be labeled with its cost and for each router, the endpoints of the links incident to it should be labeled 0, 1, 2,... where these local link numbers are determined by the order in which the neighbors are listed in the configuration file. For example, here is the relevant section from the configuration file for \( r_1 \).

neighbor: 1.3.0.1 192.168.2.5 .03
neighbor: 1.6.0.1 192.168.5.2 .05
neighbor: 1.9.0.1 192.168.1.1 .04

The link connecting to \( r_3 \) (which has IP address 1.3.0.1 in the overlay) would have an index of 0 at \( r_1 \). The link connecting to \( r_6 \) (which has IP address 1.6.0.1 in the overlay) would have an index of 1 at \( r_1 \). The link connecting to \( r_9 \) (which has IP address 1..0.1 in the overlay) would have an index of 2 at \( r_1 \).

Find a shortest path tree in your network graph, rooted at router 2. Show the edges in the shortest path tree using heavy weight lines.
2. (10 points) Run the provided `script2` by typing

```bash
script2 .01 20 static
```

Paste a screenshot of the monitoring window from your run here.

![Monitoring Window](image)

Paste the portion of the output from `log2` showing the final routing table at `r2`.

```
Routing table (28.321)
prefix  timestamp  cost  path   link
1.5.0.0/16  20.001  0.012  1.5.0.1  3
1.4.0.0/16  20.001  0.032  1.4.0.1  0
1.3.0.0/16  20.001  0.045  1.9.0.1  1.3.0.1  4
1.8.0.0/16  20.001  0.044  1.5.0.1  1.8.0.1  3
1.9.0.0/16  20.009  0.023  1.9.0.1  4
1.6.0.0/16  20.002  0.050  1.9.0.1  1.6.0.1  4
1.1.0.0/16  20.000  0.062  1.9.0.1  1.6.0.1  1.1.0.1  4
1.7.0.0/16  20.001  0.062  1.7.0.1
```

Do the routes in your routing table match the shortest path tree in your network graph? If not, explain why not.

*It is consistent. Note that we use link 4 to reach nodes `r1`, `r3`, `r6` and `r9` and the path to all of these matches the tree path above. The others paths and costs also match up with the diagram. The costs are somewhat higher than the shortest-path costs in the diagram, but that is consistent with the fact that the delays measured by the routers are generally larger than the values specified in the configuration files.*
3. (10 points) Run script2 by typing

    script2 .01 20 static debugg

Show all advertisements for prefix 1.7.0.0/* that are received by r5 during the first round of advertisements (the ones that occur at around 10 seconds).

```
/192.168.2.3:31313 received from /192.168.7.1:31313 at 10.138
protocol=2 ttl=100 srcAdr=1.8.0.1 destAdr=1.5.0.1
RPv0
type: advert
pathvec: 1.7.0.0/16 10.005 0.0222 1.8.0.1 1.7.0.1
--
/192.168.2.3:31313 received from /192.168.2.4:31313 at 10.180
protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.5.0.1
RPv0
type: advert
pathvec: 1.7.0.0/16 10.005 0.0618 1.2.0.1 1.7.0.1
--
/192.168.2.3:31313 received from /192.168.1.1:31313 at 10.332
protocol=2 ttl=100 srcAdr=1.9.0.1 destAdr=1.5.0.1
RPv0
type: advert
pathvec: 1.7.0.0/16 10.005 0.0878 1.9.0.1 1.2.0.1 1.7.0.1
```

Which of r5’s neighbors send it advertisements for this prefix? Why do these neighbors send ads and the others do not? Do your best to explain your observations based on the delays that ads will experience as they pass through the network.

Routers 2, 8 and 9 send ads to r5, but r6 does not. It makes sense that r6 does not, because r5 receives its first ad from r7 about 50 ms after r7 first issues its ads. It then forwards an ad for r7 to r6, which receives it about 90 ms after r7 issued its ads. The next ad to reach r6 from r7 would arrive 10 ms later, since it would take the path r7, r4, r2, r9, r6 which has a cost of 100 ms. The case of r9 is seems curious, since the first ad from r7 to reach r9 should also be the one that comes through r5. So, I would not expect r9 to send an ad to r5. It appears that for this particular run, the ad that passes through r5 is not the first to arrive at r9. I confirmed this by examining the log9 file. Although this is not entirely consistent with the delays shown in the graph, it probably just reflects random delays at the hosts in the onl testbed (due to OS activity, etc). Note that a delay difference of just 10 ms would account for the observed anomaly.
4. (10 points) Run script2 by typing

```
script2 .01 120 debug
```

Paste a screenshot of the monitoring window below. To get the entire run on the display, you will need to zoom out, by clicking repeatedly on the arrow head at the right end of the horizontal axis.

![Monitoring Window](image.png)

Type the command

```
grep "1.1.0.0.16....." log*
```

and paste the results below. Remove all lines that are identical to the one above them. Highlight all the places after time 10, where the paths defined by the route change, by making them bold.

```
log2:1.1.0.0/16 10.000 0.273 1.9.0.1 1.3.0.1 1.1.0.1 4
log2:1.1.0.0/16 10.000 0.235 1.5.0.1 1.6.0.1 1.1.0.1 3
log2:1.1.0.0/16 30.001 0.288 1.9.0.1 1.1.0.1 4
log2:1.1.0.0/16 40.001 0.383 1.9.0.1 1.1.0.1 4
log2:1.1.0.0/16 50.002 0.556 1.9.0.1 1.1.0.1 4
log2:1.1.0.0/16 70.003 0.518 1.9.0.1 1.3.0.1 1.1.0.1 4
log2:1.1.0.0/16 80.004 0.498 1.9.0.1 1.1.0.1 4
log2:1.1.0.0/16 90.004 0.451 1.9.0.1 1.1.0.1 4
log2:1.1.0.0/16 120.006 0.625 1.9.0.1 1.1.0.1 4
log3:1.1.0.0/16 10.000 0.067 1.1.0.1 1
log3:1.1.0.0/16 20.000 0.140 1.1.0.1 1
log3:1.1.0.0/16 30.001 0.224 1.1.0.1 1
log3:1.1.0.0/16 40.001 0.246 1.1.0.1 1
log3:1.1.0.0/16 60.002 0.150 1.1.0.1 1
log3:1.1.0.0/16 70.003 0.129 1.1.0.1 1
log3:1.1.0.0/16 100.004 0.262 1.1.0.1 1
log3:1.1.0.0/16 110.005 0.311 1.1.0.1 1
log3:1.1.0.0/16 120.006 0.281 1.1.0.1 1
log4:1.1.0.0/16 10.000 0.340 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1 1
log4:1.1.0.0/16 30.001 0.510 1.2.0.1 1.9.0.1 1.1.0.1 1
log4:1.1.0.0/16 30.001 0.407 1.3.0.1 1.1.0.1 2
log4:1.1.0.0/16 40.001 0.559 1.3.0.1 1.1.0.1 2
log4:1.1.0.0/16 70.003 0.401 1.3.0.1 1.1.0.1 2
log4:1.1.0.0/16 80.004 0.455 1.3.0.1 1.1.0.1 2
log4:1.1.0.0/16 90.004 0.454 1.3.0.1 1.1.0.1 2
```
Paste a copy of your network graph below and highlight the shortest path tree defined by the routes going to \( r_1 \) at time 15, by making the links heavy weight.
Find a time when the shortest path tree to $r_1$ differs from the one at time 15. Paste another copy of your network graph below and highlight the links in the shortest path tree at that time. During what time period is this shortest path tree used?

The shortest path tree shown below is used from time 40 to time 50.