# Project 1 – Distance Vector Routing

Due: Oct 7, 2022 11:59 PM

## 1 Problem Statement

The goal of this project is for you to learn to implement distributed algorithms for intradomain routing, where all routers run an algorithm that allows them to transport packets to their destination, but no central authority determines the forwarding paths. You will implement code to run at a router, and we will provide a routing simulator that builds a graph connecting your routers to each other and to simulated hosts on the network. By the end of this project, you will have implemented a version of a distance vector protocol that computes efficient paths across the network.

## 2 Getting a bit more Concrete

In much of the class material, we discussed routing abstractly, i.e., the algorithms discussed were used on graphs and computed distances between every pair of nodes. In the real world, we have both switches/routers and hosts, and you're primarily concerned with whether **hosts** can reach other **hosts**. Which is to say, for the purposes of this assignment, you need not compute routes to other routers—only to other hosts.

Similarly, we often speak about abstract **links**. Links in the real world are often a combination of a **port** (or **interface**) on one device, a **cable**, and a port on another device. A device is often not aware so much of a link as a whole as it is aware of its own side of the link, i.e., its own port. Ports are typically numbered. When a device sends data out of one of its own ports, the data travels through the cable and is received by the port on the other side. The API functions in the simulator reflect this: they deal in ports, not in links.[1]

## 3 Simulation Environment

You will be developing your router in Python 3.8 under a simulation environment provided by us. The simulation environment, as well as your router implementation, lives under the `simulator` directory; you should `cd` into this directory before entering any terminal commands provided in this document.

In the simulation environment, every type of network device (e.g., a host or your router) is modeled by a subclass of the `Router` class. Each `Router` has a number of ports, each of which may be connected to a neighboring entity (e.g., a host or another router). Each link connecting two entities has a **latency**—think of it as the link's propagation delay. Your `Router` sends and receives `Packet`'s to and from its neighbors.

The `Router`, `Packet` and `Ports` classes are defined in the `cs168.dv` module. Relevant methods of these classes are displayed in Figures 1 and 2 3; now might be a good time to skim through them. You can learn more about the simulation environment from the Simulator Guide.

Before we begin, let's make sure that your Python version is supported. Type in your terminal:

```
$ python --version
```

You should be good to go if the printed version has the form `Python 3.8.*`.

[1] Don't get these confused with the logical "ports" that are part of transport layer protocols like TCP and UDP. The ports we're talking about here are actual holes that you plug cables into!

```
class Router (api.Entity)
    add_static_route (self, host, port)
        Adds a static route to a host directly connected to this router.
        Called by the framework when a host is connected.
        host - the host that got connected
        port - the port that the host got connected to
        You should override this method.

    handle_data_packet (self, packet, in_port)
        Called by the framework when a data packet arrives at this router.
        packet - a Packet (or subclass)
        in_port - port number it arrived on
        You should override this method.

    send_routes (self, force=False, single_port=None)
        Send route advertisements for all routes in the table.
        force - if True, advertises ALL routes in the table; otherwise, advertises
        only those routes that have changed since the last advertisement.
        single_port - if not None, sends updates only to that port; to be used in
        conjunction with handle_link_up.
        You should override this method.

    handle_route_advertisement (self, route_dst, port, route_latency)
        Called when the router receives a route advertisement from a neighbor.
        route_dst - the destination of the advertised route.
        route_latency - latency from the neighbor to the destination
        port - the port that the advertisement arrived on.
        You should override this method.

    expire_routes (self)
        Clears out expired routes from table.
        You should override this method.

    handle_link_up (self, port, latency)
        Called by the framework when a link is attached to this router.
        port - local port number associated with the link
        latency - the latency of the attached link
        You should override this method.

    handle_link_down (self, port)
        Called by the framework when a link is unattached from this router.
        port - local port number associated with the link
        You should override this method.

    send (self, packet, port)
        Sends the given packet out on the specified port.
        packet - the packet to forward.
        port - the port for the packet to be sent on.
        Do not override this method.

    send_route (self, port, dst, latency)
        Creates a routing packet with the specified destination and latency
        and sends the packet out on the port.
        Note that the destination is not the destination of the packet
        but the advertised route destination.
        port - the port for the route to be sent on.
        dst - the destination being advertised by the route.
        latency - the latency for the advertised route.
        Do not override this method.
    log (self, format, *args)
        Produces a log message in Python terminal
        format - The log message as a Python format string
        args - Arguments for the format string
        Do not override this method.

    s_log (self, format, *args)
        Logs the only these messages, if this router is selected in the simulator.
        format - The log message as a Python format string
        args - Arguments for the format string
        Do not override this method.
```

**Figure 1:** Relevant methods of the Router superclass.

```
class Packet (object)
    self.src
    Packets have a source address.
    You generally don't need to set it yourself.  The "address" is actually a
    reference to the sending Entity, though you shouldn't access its attributes!

    self.dst
    Packets have a destination address.
    In some cases, packets aren't routeable -- they aren't supposed to be
    forwarded by one router to another.  These don't need destination addresses
    and have the address set to None.  Otherwise, this is a reference to a
    destination Entity.

    self.trace
    A list of every Entity that has handled the packet previously. This is
    here to help you debug. Don't use this information in your router logic.
```

**Figure 2:** Relevant methods of the Packet class.

```
class Ports (object)
    get_latency (self, port)
        Returns the latency of the link associated with that port.
        port - the port whose latency will be returned.

    get_all_ports (self)
        Returns a list of all of the active ports connected to this router.
```

**Figure 3:** Relevant methods of the Ports class.

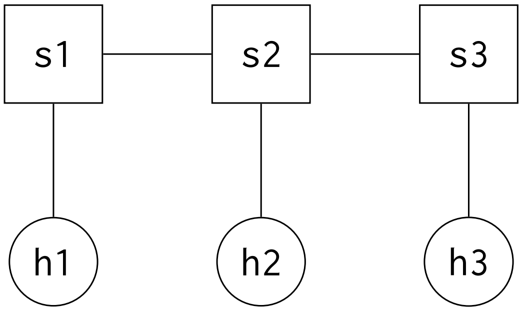

**Figure 4:** Linear topology with three hosts. Circles denote hosts, squares denote routers, and lines denote links. A link has a latency of 1 unless otherwise specified.

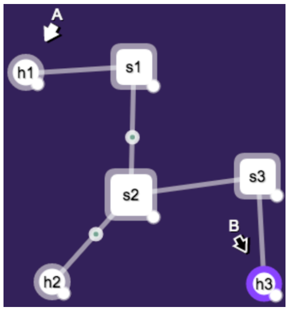

**Figure 5:** Sending a "ping" from the visualizer in the hub example. (Packets shown are "pong" packets sent by h3 back to h1 being flooded by hub s2.)

## 4 Warm-up Example: Hub

To get you started, we have provided an implementation of a **hub**—a network device that floods any packet it receives to all of its ports (other than the port that the packet came from). *The hub is already implemented and you don't need to submit anything for this section.*

Take a look at the hub implementation in examples/hub.py. Having no need to record any routes, the hub only implements the handle_data_packet method to flood data packets.

Let's try out the hub on a **linear** topology with three hosts (Figure 4):

```
$ python3 simulator.py --start --default-switch-type=examples.hub topos.linear --n=3
```

You can now access the **visualizer** at http://127.0.0.1:4444 using your browser; you should see the hosts and routers displayed against a purple background. Let's now make host h1 send a ping packet to host h3. You can either type into the Python terminal:

```
>>> h1.ping(h3)
```

or you can send the ping from the visualizer by: (1) selecting h1 and pressing A on the keyboard; (2) selecting h3 and pressing B; and (3) pressing P to send a ping from host A to host B (Figure 5).

You should see the "ping" and "pong" packets being delivered between h1 and h3. You should also see both packets delivered to h2 despite it not being the recipient. This behavior is expected since the hub simply floods packets everywhere. You may also observe what's going on from the log messages printed to the Python terminal:[2]

```
WARNING:user:h2:NOT FOR ME: <Ping h1->h3 ttl:17> s1,s2,h2
DEBUG:user:h3:rx: <Ping h1->h3 ttl:16> s1,s2,s3,h3
WARNING:user:h2:NOT FOR ME: <Pong <Ping h1->h3 ttl:16>> s3,s2,h2
DEBUG:user:h1:rx: <Pong <Ping h1->h3 ttl:16>> s3,s2,s1,h1
```

[2] You can see a ttl field printed for each packet. The simulator automatically assigns each packet a "time to live" (TTL)—any packet will only be forwarded up to some maximum number of times. The TTL is managed entirely by the simulator; you should **not** read or write the ttl field.

Recall from class that flooding is problematic when the network has loops. Let's see this in action by launching the simulator with the topos.candy topology, which has a loop (Figure 6):

```
$ python3 simulator.py --start --default-switch-type=examples.hub topos.candy
```

Now, send a ping from host h1a to host h2b. You should be seeing a lot more log messages in the terminal, and the visualizer should be showing routers forwarding superfluous packets for quite a while. Oops! This is why our next step will be to implement a more capable distance vector router.

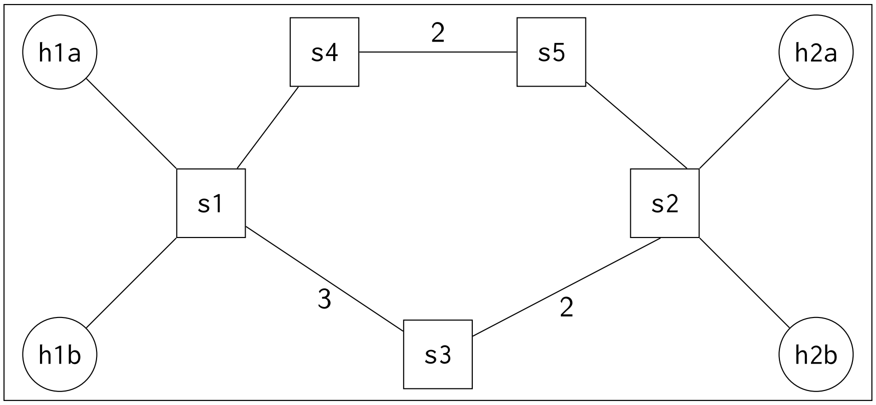

**Figure 6:** The topos.candy topology, which has a loop. A link has latency 1 unless otherwise specified. We'll be using this topology for demonstrative purposes throughout the project.

## 5 Distance Vector Router: Working Section

We've provided a skeleton dv_router.py file with the beginnings of a DVRouter class for you to flesh out. The DVRouter class inherits from the DVRouterBase class.

To guide your implementation of DVRouter, we have split the implementation process into **three sections**, each of which contains a subset of the **ten total stages** and focuses on one aspect of the router. You should follow along stage by stage, and by the end, you will have implemented a functional distance vector router!

## Testing

To help you check your work, we provide you with unit and comprehensive tests. These tests will be the **only** tests that we'll use to grade your submission, i.e., there will be no hidden tests (see Section 6 for details on grading).

The **unit tests** test each aspect of the router separately and correspond to the **10** stages of implementation.[3] After you finish each stage, you should run the unit tests for that stage (and all previous stages) and make sure you pass them. For example, after you finish Stage 5, you can run unit tests for the first five stages by invoking:

[3] There are also "Stage 0" tests that simply make sure certain parts of the skeleton code are intact. You will not be graded on the "Stage 0" tests.

```
$ python3 dv_unit_tests.py 5
```

If your tests fail, you should read the test's documentation and source code in `dv_unit_tests.py` to figure out what the test does. You should not change any tests except for debugging purposes.

**Hint:** When you debug a test failure, you should read not only the documentation for a single test function (e.g., `test_handle_link_up`), but also the docstring for the enclosing class (e.g., `TestAdvertise`), which may include useful information about the common setup of all tests in this stage.

**Note:** The unit tests are **not** guaranteed to be comprehensive—it is possible for your implementation to have a defect in one stage that manifests itself by failing unit tests for a later stage, or for your implementation to pass all the unit tests but fail the comprehensive test.

The **comprehensive test** checks to make sure that your **completed** router has good routing behavior on pseudo-randomly generated topologies. The test proceeds in rounds; in each round, it changes the topology by adding and/or removing some random links, waits a bit for the routes to converge, sends pings from every host to every other host, and then checks to ensure that the "ping" packets arrive in time. You **will not be graded based on the results of the comprehensive test.** This is for your understanding only. After you finish your implementation, you may run the comprehensive test like this (provided you are in the simulator directory):

```
$ bash run_comprehensive.sh
>>> start()
```

This command will launch the simulator on a pseudorandomly generated topology with 5 switches and 10 links. The comprehensive test will start running after you enter `start()` into the Python terminal. As usual, you may observe the test progress in the visualizer.

The comprehensive test will run indefinitely until a test failure occurs. If that happens, you can type commands into the Python terminal and use the visualizer to debug.

The command that starts the comprehensive test comes with two random seeds (specified by the −seed flag). The first seed controls the random topology generation; the second controls how the test changes the topology in each round. Feel free to provide different seeds to test your router under different scenarios.

For maximum efficiency, the comprehensive test always runs your router with "poisoning and triggered/incremental" turned on (you'll understand what this means once you get to stage 10).

**Note:** The unit and comprehensive tests are subject to change. Any changes we make before the deadline will be publicized on Ed.

## Requirements

Before we get started with the implementation, let's lay down some ground rules:

- Your `DVRouter` implementation must live entirely in `dv_router.py`; **do not** add other files.
- You should **not** touch the simulator code itself or the unit tests. Nor should you write code that dynamically modifies the simulator or the tests. Additionally, don't override any of the methods

which aren't clearly intended to be overridden, and don't alter any "constants." *You will receive zero credit for turning in a solution that modifies the simulator itself or otherwise subverts the assignment. If you're not sure about something: ask.*

- Your `DVRouter` instances should communicate with other `DVRouter` instances **only** via the sending of packets. Global variables, class variables, calling methods on other instances, etc., are not allowed—each `DVRouter` instance should be entirely standalone!
- The constructor (`__init__`) we provide for `DVRouter` defines several instance variables for the class (e.g., `self.table`). **Do not** modify or remove these definitions; you'll be using them later on. Similarly, **do not** remove any existing method definitions; you'll be filling in their implementations.
- However, **feel free to add your own instance variables and/or helper methods**, as long as they don't break the unit and comprehensive tests.
- Your `DVRouter` implementation must work with the unmodified `cs168/dv.py` file (which contains some helper classes).
- You should not need any additional `import` statements. It would be fine for you to use, say, Python's `collections` module. However, you should **not** use (or need to use!) the `time`, `threading`, or `socket` modules. If you have questions, ask!
- You must solve this project **individually**. You may not share code with anyone, including any custom test code that you may write. You may discuss the assignment requirements or your solutions—*away from a computer and without sharing code*—but you should not discuss the detailed nature of your solution. Also, don't put your code in a public repository. We expect you all to uphold high academic integrity and pride in doing **your own work**. Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct[4].

Let's get started on implementing `DVRouter`.

## Code Structure

Utilize this section throughout the project whenever you are unsure about the underlying architecture of the Distance Vector router.

### Data Structures

In this project we employ the use of several data structures that correspond to the state kept by routers including the forwarding table (`Table`) and the ports (`Ports`).

**Table:** For each host your router maintains a table entry corresponding to the current "best" route to that host. Each table entry (`TableEntry`) is denoted by the following characteristics from that best route:

- `dst`: the destination for this route
- `port`: the port that this route takes (i.e., a packet to `dst` should be sent out of this port)
- `latency`: the latency of the route **from this router** to the destination.
- `expire_time`: the timestamp (in seconds) at which this route expires.

[4] `http://students.berkeley.edu/uga/conduct.pdf`

Consider the following code which constructs a table object and two table entries with destinations h1 and h2, using ports p1 and p2, latencies of 10s and 20s, and expiry times of 20s from now:

```
t = Table()
t[h1] = TableEntry(dst=h1, port=p1, latency=10, expire_time=api.current_time()+20)
t[h2] = TableEntry(dst=h2, port=p2, latency=20, expire_time=api.current_time()+20)

for host, entry in t.items():  # <-- This is how you iterate through a dict.
    print ”Route to {} has latency {}”.format(host, entry.latency)
```

Note: You should call `api.current_time()` to acquire the current timestamp.
Note: If you are not using keyword parameters (e.g. `dst=h1`), make sure you place the parameters in the correct order (`dst, port, latency, expire_time`) when creating a `TableEntry`.

A `TableEntry` object is immutable. If you wish to update an attribute, you should create a new `TableEntry` object with updated attributes.

Your router keeps its table in the instance variable `self.table`, which is a Python dictionary mapping each host to the corresponding `TableEntry` object.

**Ports:** Maintains the current set of ports and the latencies of their associated links. See 3 for the api.

**Flags:** The following flags will denote the different modes the router is in. We will tell you explicitly when to include them in your code:

- `self.SPLIT_HORIZON`
- `self.POISON_REVERSE`
- `self.POISON_ON_LINK_DOWN`
- `self.POISON_EXPIRED`
- `self.SEND_ON_LINK_UP`

One final note, we have tried to manage the complexity of this project in a way that's beneficial for you. In general if we haven't told you to employ the use of a certain method or flag then you likely don't need it yet. Our goal is NOT to trick you!

## Forwarding

Recall from class that when a router receives a packet on the data plane it will consult its table to determine that packet's outgoing link to be forwarded. This section will encompass all functionality related to the forwarding that routers perform in Distance Vector.

### Stage 1/10: Static Routes

For each host that **directly** connects to your router, your router should record a **static route** to that host. The code you write here will be called magically by the framework. For the purpose of this assignment these routes should be assigned the latency recorded in the link latency table and should never expire (expiry time = `FOREVER`).

**Implementation:** ▶ Implement the `add_static_route` method to add a static route (by adding a `TableEntry` object) to the appropriate host slot in your routing table; this method is called by the framework for every host that directly connects to your router.

You can check the routing table for any switch by typing, e.g., for s1:

```
>>> print(s1.table)
```

Check Your Understanding: Now that we have implemented static routes what do you expect to happen if we try to send a packet along a path?

Demo: Using NetVis you can check exactly what will happen in the above described case. Start the simulator using the `topos.simple` topology with the following command (remember you can see the demo by logging into your browser and going to 127.0.0.1:4444):

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.simple
```

Try to send a ping from host h1 to host h2 (i.e., `h1.ping(h2)`). What happened? Did it align with your expectation?

## Stage 2/10: Forwarding

It's now clear that we need to implement some forwarding logic in the router! Using its routing table, your router can forward packets on the data plane.

**Implementation:**
▶ Implement the `handle_data_packet` method to handle data packets appropriately. The `handle_data_packet` method is called whenever a data packet arrives at your router. Use `send` found in Figure 1.

**Note:** If no route exists for a packet's destination, your router should drop the packet (do nothing). And if the latency is greater than or equal to `INFINITY` you should also drop the packet (we will further adapt this case later).

Demo: Let's again use the `topos.simple` topology. Start the simulator as the previous stage and try to send a ping from host h1 to host h2. Hopefully it arrived!

We should now be confident in our forwarding capabilities. Run the unit tests associated with this section and see how you did.

# Routing Fundamentals

With forwarding down surely we have mastered the art of network communication!

## Stage 3/10: Sending Routing Tables Advertisements

Check Your Understanding: Given our current implementation what should happen when pings are sent from one host to another in the following scenarios:

- Scenario 1: Two hosts are connected to the same router
- Scenario 2: Two hosts are connected to their own unique router with those routers connected

A. Pings fail in both scenarios B. Pings succeed in Scenario 1 but not 2 C. Pings succeed in Scenario 2 but not 1 D. Pings succeed in both scenarios

Demo: Use the `topos.simple` topology again to run through Scenario 1 and Scenario 2. As you may notice, what we have tried out in previous stages (i.e., `h1.ping(h2)`) is a case of Scenario 1; but how would you reproduce Scenario 2? What do you observe and why?

As you can see in a path with routers connected to one another the ping is dropped which signals that the router does not know where to forward the packet. Individually each of the routers do not have a grasp on the entire topology, but together they know how to reach both destinations. We will now implement the sending portion of those control messages that routers use to communicate views of the network.

For routers to learn one another's routes, each router must advertise the routes in its table to its neighbors. There are many scenarios where a router should send route advertisements, let's work on one of those scenarios.

**Implementation**:
**Timer:** Your router should advertise routes periodically (every time the timer fires, the timer firing is already implemented in the base class) in order to refresh the routes and keep them from expiring. The framework automatically advertises all routes in the table by calling `self.send_routes(force=True)`. The `force` argument taken by `send_routes` dictates whether to advertise **all** routes (if `force=True`) or to advertise only those routes that have changed since the last advertisement (if `force=False`). ▶ Implement the `send_routes` method for the `force=True` case. For now, don't worry about the `force=False` case or `single_port!=None` case — they won't be needed until a later stage on incremental updates.

To advertise a route, you should use the `send_route` from Figure 1.

Check Your Understanding: Given our current implementation what should happen to a ping in Scenario 2? A. Pings fail B. Pings succeed

Demo: On the simulator, repeat what you did for Scenario 1 and Scenario 2 on `topos.simple`. Now you can observe routing advertisement packets (shown in purple dots) periodically sent from each switch. However, as you may observe, we are still not able to successfully ping under Scenario 2.

## Stage 4/10: Handle Route Advertisements

Your router sends route advertisements to its neighbors; but these advertisements need to be handled! Ultimately we want to select the best route possible to reach a destination. How do we know when a route is better if they perform the same?

Check Your Understanding: When the current route and a new route have the same performance how do you know which one is better. Like most decisions this one navigates a trade-off. What is this trade-off and how should you navigate it?

Answer: The trade-off here is between correctness and stability.

**Implementation:**
▶ Implement the `handle_route_advertisement` method, which is called by the framework when your router receives a route advertisement from a neighbor. This method should update the router's table with the better of the current route and the new route, or, if the two routes have equal performance, break

ties by **choosing the new route**. Each time you receive a route advertisement, you should set the route‘s expiry time route to `self.ROUTE_TTL` seconds in the future ($15\,\mathrm{s}$ by default).

**Note:** To get the current time in seconds, call `api.current_time()`. **Do not** use Python's `time` module.

Demo: Start the simulator with `topos.square`:

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.square
```

Send a sequence of pings between two corners of the square. What do you observe?

Check Your Understanding: We can see in this example that packets from the same host toward the same destination can take different paths - is generally considered sub-optimal. On the one hand, a new route represents state that is more recent and is therefore more indicative of the current state of the network. On the other hand, always choosing a new route (when breaking ties) can lead to more unstable paths. Next, we will change the way router break ties to see if we can make paths more stable.

Furthermore, if the candidate route for replacement comes from the same port shown in the current route for the same destination, then we should always update that route. Why? Firstly, you are already using their route so that neighbor serves as the ground truth for the latency of that route. Secondly, in the case that the latency is the same you want to reflect the new expiry time of the route.

**Implementation:**

- Update the `handle_route_advertisement` method to break ties by **choosing the current route**
- Furthermore, you should update the `handle_route_advertisement` method to increase the cost of a route if you receive an advertisement from the same port.

Demo: Re-launch that same square topology (`topos.square`) and send the same sorts of pings, we should now see packets traverse the same path. It‘s important to recognize that we should now be able to send packets between hosts with an arbitrary number of routers between them. Go ahead try a random topology! Good job![5]

Problem Demo: Before moving on to the next stage, let‘s explore what happens when a link goes down. Launch (or reload) the simulator with `topos.candy` topology:

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.candy
```

Wait for all routes to propagate. Then remove the link between routers s4 and s5 by either selecting the two routers and pressing E in NetVis or typing:

```
>>> s4.unlinkTo(s5)
```

and try sending a ping from h1a to h2b. Even though there still "exists" a route between the two hosts through router s3, you will see the ping packet get forwarded to router s4 and get dropped - s4 wants to forward the packet to s5, to which it is no longer connected! In the next stage, you will be adding functionality to handle route removals correctly and as a result the ping would take the alternate route.

[5] To run simulation with a random topology, use `topos.rand` with different parameters (e.g., run `python simulator.py --default-switch-type=dv_router topos.rand --switches=5 --links=10 --seed=1`).

## Stage 5/10: Handling Routing Tables Timeouts

When a route expires we should remove it from the table. We will implement this by periodically scanning our table entries to see if any have expired.

**Implementation:**
Previously, you assigned expiry times to your peer table entries. ▶ Implement the `expire_routes` method, which should clear out any expired routes. We encourage you to log the fact that a route has expired using `self.s_log` or `self.log` but this is not required. The `expire_routes` method is automatically called when the Router's timer goes off.

Demo: Start the simulator with `topos.candy` topology. Now let's bring down link the between routers s4 and s5 again. Roughly $15\,\mathrm{s}$ after the link down event - this is when the old route has expired - you should see the ping packet being forwarded correctly along the alternate route.

Your distance vector router now has all its basic functionality! Don't forget to run your tests for all of those stages!

# Routing Enhancements

Your distance vector router has all of the basic functionality! It can route packets to all destinations and find alternate paths if the existing ones fail. But since there's always room for improvement, let's make it more efficient!

## Stage 6/10: Split Horizon (Let's Get Loopy)

Problem Demo: Start the simulator with the `topos.linear` topology with three hosts:

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.linear --n=3
```

Wait for all routes to propagate. Send a ping from host h3 to host h1 (this should work). Now bring down the link between s1 and s2 (`s1.unlinkTo(s2)`) and try sending pings from h3 to h1. You should see the packets get dropped at s2. Eventually this route will timeout, and s2 will get a new advertisement from s3. Now when you send a ping from h3 to h1, what happens and why?

Check Your Understanding: How long will the routing loop above continue? What is the larger problem here and how can we tackle this issue?

Answer: Without fixing this problem each router will believe that the other has a path to a destination when in reality **neither do**. This will continue until either one of them gets a real path to the destination (which will have bounded cost) or the heat death of the universe.

There is a principled solution here, maybe you've already guessed it. If you are using a route to destination D through neighbor N, dont advertise destination D to neighbor N – it would be sure to describe a loop! This technique is called split horizon.

**Implementation:**
▶ Edit the `send_routes` method to support split horizon, but only if the `self.SPLIT_HORIZON` flag is turned on.

Demo: Launch the same topology and proceed in the same way as before, but set self.SPLIT_HORIZON to true. Now we shouldn't see any counting!

## Stage 7/10: Poison Reverse (Still loopy)

Problem Demo: Let's start the simulator with a slightly more complex topology topos.double_triangle:

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.double_triangle
```

For the purpose of this demo start with split horizon off (set the self.SPLIT_HORIZON flag to False). Disconnect s2 by removing all links that connect it to other routers:

```
>>> s2.unlinkTo(s1)
>>> s2.unlinkTo(s3)
>>> s2.unlinkTo(s4)
```

Router s3 and s4 now form a loop, each one believes that the other has a path to the destination, so any packet sent between them for that destination will bounce back and forth.

Will the loop ever end? It seems so - we eventually came to the right result, but it is slow (why?). Let's see if split horizon will save us! Turn on split horizon and rerun the topology, you should see that now we converge in only a single interval; an improvement!

Check Your Understanding: Can we do better than this and if so, by how much?

With split horizon, however, we still had to wait for the invalid route to expire. Let's get more aggressive with loop prevention by actively advertising the non-existence of a route to the port that the route takes. We can advertise non-existence by sending an "infinite" value which is much larger than any real latency. This technique is called **poison reverse**.

**Implementation:**

▶ Augment your send_routes method to implement poisoned reverse - to each neighbor, advertise as unreachable any destination for which this router's route goes through that neighbor. (These would be route advertisements with latency=INFINITY.) You should send poisoned reverse advertisements **only if** self.POISON_REVERSE is True. When poison reverse mode is off, you should still implement split horizon if the self.SPLIT_HORIZON mode is on! (We'll run tests with at most one of the two flags enabled.)

Check Your Understanding: How long will it take to get correct state now?

Demo: Relaunch the same topology performing the same process (with the self.SPLIT_HORIZON flag set to True). Now it forms the s3/s4 loop with one advertisement, but the next advertisement is a poison which breaks the loop. The next time s3/s4 get an advertisement from s1, it gets correct state.

## Stage 8/10: Counting to Infinity

The demo in the previous stage was a special case. We can only detect a loop when its between two adjacent nodes.

Problem Demo: Launch the topos.loopy topology:

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.loopy
```

Wait for all routes to propagate. Disconnect s1 from s2:

```
>>> s2.unlinkTo(s1)
```

You can print the tables of the routers: How long will the routers count? It seems to be forever - despite all of our hard work!

Check Your Understanding: Will we ever stabilize? Can split horizon or poison reverse save the day?

To fix this problem we need some threshold (infinity) to recognize when counting will probably never end. Then we can simply not accept any routes with a latency higher than our threshold.

**Implementation:** ▶ Utilize `INFINITY` to represent a threshold for unrealistically high latency. Upgrade your `handle_route_advertisements` methods to support the following functionality:

- If a poisoned advertisement (latency `INFINITY`) matches the destination and port of a current route replace it with the poisoned entry for poison propagation
- Do not recharge the timer of a poisoned route in your table when a new advertisement comes in
- Any incoming routes with latency `INFINITY` that don't match destination and port with a current route should be ignored

Lastly, make sure that incoming packets are never sent on a poisoned route (you may want to check your handle_data_packet() implementation) just because its in the table.

Demo: Let's repeat the previous simulation (launch the `topos.loopy` topology and disconnect s1 and s2. How long will the routers count? Now the routers reach infinity and stops counting!

## Stage 9/10: Poisoning Expired Routes

Problem Demo: Launch the `topos.linear` topology with 7 hosts:

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.linear --n=7
```

Wait for all routes to converge (e.g., on s7 you should be able to see a route to h1; this may take a while!). Then, disconnect s1 from s2. How long will it take for the route to s1 to be updated the other routers?

Can we leverage techniques from the previous stage to fix this? Employing the intuition from the previous Stage, instead of waiting for timeouts, we should perform some sort of poisoning. That is, when an entry expires we will not only remove it from our routing table but will also advertise the corresponding route's latency as infinite.

**Implementation:** ▶ Augment your code to implement expired route poisoning when `self.POISON_EXPIRED` is set to `True`. Since poison advertisement packets can be dropped, your router should make sure to advertise poisoned routes **periodically** for at least `ROUTE_TTL` seconds ($15\,\mathrm{s}$ by default), e.g., by keeping the poison routes in the routing table so that the periodic route advertisements will do the poisoning.

Ideally, these periodic advertisements should halt after a while (e.g., removing the poison routes from the routing table after X seconds), because it is of little use to keep advertising the **nonexistence** of a route. Note that we will not test you on this!

Demo: Relaunch the `topos.loopy` topology and disconnect `s1` from `s2`. Now our routes propagate much faster!

Check Your Understanding: What do route removals now propagate relative to?

## Stage 10/10: Becoming Eventful

You're almost done!

Timers provide good guarantees of eventual convergence. But, even more optimal behavior results from performing actions in response to network events.

Check Your Understanding: What sorts of network events warrant a response and what should you do in relation to those events?

In this stage, you'll be implementing incremental and triggered updates - your router will advertise routes every time its table is updated (**triggered**), and will advertise only those routes that have changed (**incremental**).

You will implement incremental updates by augmenting the `send_routes` method to handle the `force=False` case (ignore the `single_port` parameter for now), where only route advertisements that differ from before should be sent. Here is our recommended implementation strategy:

- Maintain a "history" data structure that records the latest route advertisement **sent** out of each port for each destination host. You might want to use a tuple of the relevant properties of a route.
- Implement `send_routes(force=False)` so that it sends a route advertisement only if (1) it is missing from the "history" data structure, or (2) it differs from the corresponding entry in the history.
- Implement `send_routes(force=True)` to simply send all route advertisements, ignoring the history.
- In either case, `send_routes` should update the "history" data structure as appropriate.

**Implementation:** For incremental updates: ▶ Implement the `force=False` case for the `send_routes` method. Then, to enable triggered updates: ▶ Call `send_routes(force=False)` wherever necessary.

Next, when a link comes up on a port, your router should get its new neighbor up to speed by immediately advertising all of its routes to that port only, using the `single_port!=None` case, if the `SEND_ON_LINK_UP` flag is on. ▶ Add this feature to the `handle_link_up` method.

Lastly, when a link goes down, all routes using that link should be invalidated. ▶ Implement the `handle_link_down` method, which is called by the framework when a link goes down, to remove any routes that go through that link - and poison and immediately send any routes that need to be updated only when the `POISON_ON_LINK_DOWN` flag is on.

Demo: Start the simulator with any topology you like. You should now see a much lower time to responds to network events compared to earlier versions of the router. Revisit those demos and appreciate the result of a solid design decision.

Check Your Understanding: If we now have events what is the purpose of a timer?

**Congratulations you have implemented a router that performs a variant of Distance Vector! Fun fact, this implementation was closest to Routing Information Protocol (RIP) which became part of UNIX when included in the BSD! Be sure to run the tests for each stage.**

## 6 Submitting your work and Grading

To submit your implementation, upload your entire repo on Gradescope. We recommend using the GitHub integration on Gradescope to do this. Be sure to familiarize yourself with the **late policy** outlined in the syllabus on the course website.

The grade for your `DVRouter` implementation is determined in the following way:

- **100%** of your grade will come from the unit tests that we have provided to you. This portion of your grade is split equally among the ten stages (i.e., 10% for each stage). Within each stage, all tests are weighted equally.

Any further details on grading will be posted on Ed.