Lecture 13: IP Routing algorithms: RIP and OSPF

1 Review of Last Time

Last time we started discussing the Network layer. We discussed the concept of an IP address and the *network part* of the address, a.k.a. the *network prefix.* We talked about how a single datalink network is a collection of hosts and bridges where the bridges all "speak" the datalink protocol and MAC addresses are used for forwarding. We also talked about how routers speak IP and use IP addresses. We discussed the notion of an "IP-hop" between two routers, which moves us to a new network.

Review Question: How many networks do you see in Figure 1?

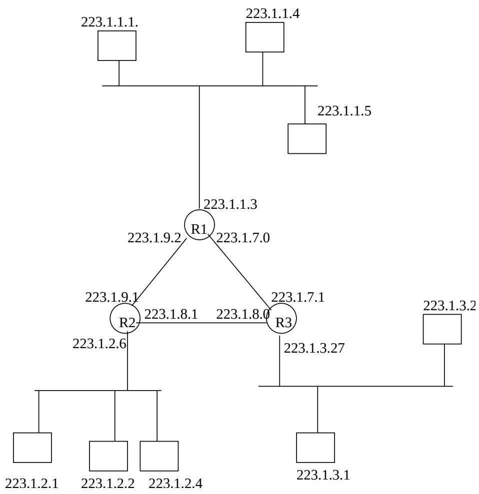

Figure 1: *3 Ethernet networks and 3 point-to-point networks. Network prefixes shown are: 223.1.1, 223.1.7, 223.1.9, 223.1.8, 223.1.2, 223.1.3*

Answer: 6.

2 Routing Algorithms – setup

Up until now, we have assumed that the forwarding table at each router has been handed to the router, as if by magic. That is, every router knows, for each network address, what is the best next router (next IP hop).

We will now discuss some algorithms for coming up with the forwarding table at a router. That is, we will describe routing algorithms which tell us the best next hop to get to a given destination network.

We have been careful to distinguish between **forwarding tables** and **routing tables**. Both of these are generally stored in the static RAM in a router (we will get to router internals later), but in *separate* data structures. The forwarding table at a router contains a mapping from a network number to an outgoing interface and the MAC address of the next hop. The routing table on the other hand is the table built up by the routing algorithm as a *precursor* to building the forwarding table. It generally contains mappings from network numbers to next hops, but may also contain a lot of additional information such as costs of getting from one network to all other networks, or full path information — anything necessary to help figure out what are the best routes. The goal of the forwarding table is to be very fast. Therefore the forwarding table needs to contain the bare minimum necessary info, in the easiest-to-use form. The forwarding table may even be implemented in hardware.

When learning about routing algorithms, you should always be asking yourself two questions. The first is: How well does this routing algorithm **scale**? For the next two algorithms that I will show you, the answer is: not very well at all. The algorithms that I will be showing you in this lecture are only intended to work for the case of 100 or fewer routers (e.g. the network for a single university campus). However, we need to study these algorithms because they will serve as building blocks for the hierarchical routing infrastructure used in the Internet today.

The other question you should be asking is: Is this routing algorithm **distributed** (can be run independently on each node) or **centralized** (rely on one central router to store all the info)? Centralized solutions are hard to scale, so we always favor distributed algorithms. All the algorithms that we discuss will be distributed.

You can envision the following abstract model of a network shown in Figure 2:

In Figure 2, the **nodes** in the graph represent **routers**. The edges represent physical links between the routers. (Note that one link might consist of an entire datalink network — bunch of Ethernets connected by bridges — but we've purposely abstracted-away this detail). On each link you see a cost $c(i, j)$, which is the cost of going from node i to node j. In the real world this might represent the latency between the routers, queueing delay, bandwidth, or something else. We will go more into what costs represent later. For

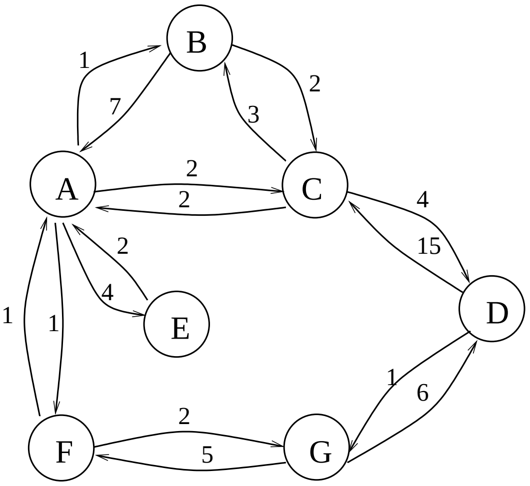

Figure 2: *Cost graph. Nodes represent routers. Link costs represent cost of traversing link. Goal is to find least cost path between all pairs of routers.*

simplicity for illustrating the algorithm we will often assume that $c(i,j) = c(j,i)$, although this is not necessary in the algorithm. We will also often assume that $c(i,j) = 1$, $\forall i,j$, although this too is not necessary and is just done to make the presentation easier to follow. The goal is to find the lowest-cost path between any two nodes.

Question: What is the cost metric if all links have cost 1?

Answer: Cost of getting from router X to router Y is just the number of IP hops between X and Y.

For both the algorithms that we describe below, it will be assumed that each router starts out knowing the cost of the link to each of its immediate neighbors, but nothing else.

3 Distance Vector Routing Algorithm

The **Distance Vector** algorithm can be guessed by its name:

Each node maintains a **vector** containing its current belief about the distances (costs) to all other nodes. Each node then repeatedly distributes that vector to its immediate neighbors.

To illustrate this algorithm, consider the cost graph shown in Figure 3.

Each node keeps track of the shortest distance to all other nodes, and of the next hop on this shortest path to each of the other nodes.

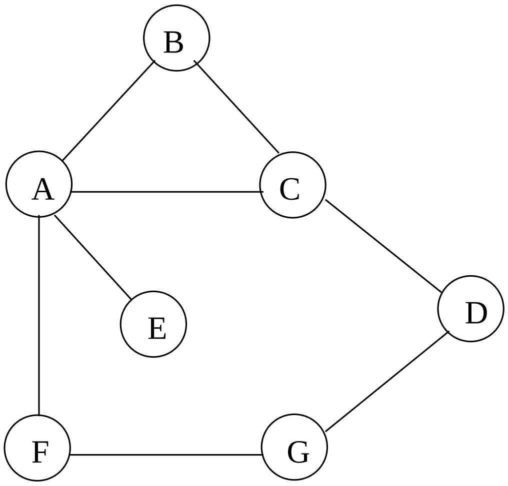

Figure 3: *Simpler cost graph. All costs are 1. Cost represents hop count.*

3.1 The static case: Costs don't change

We will illustrate the state at each node via the table below. **Important Note: Each node only knows one row of this table!!**

Initially, the table describing what each node knows looks as follows:

Information stored at node	*Distance to Reach Node*						
	A	B	C	D	E	F	G
A	0^A	1^B	1^C	∞	1^E	1^F	∞
B	1^A	0^B	1^C	∞	∞	∞	∞
C	1^A	1^B	0^C	1^D	∞	∞	∞
D	∞	∞	1^C	0^D	∞	∞	1^G
E	1^A	∞	∞	∞	0^E	∞	∞
F	1^A	∞	∞	∞	∞	0^F	1^G
G	∞	∞	∞	1^D	∞	1^F	0^G

The numerical entry corresponds to the known distance in number of hops. The exponent corresponds to the first hop. For example, A's router currently thinks the distance to D is ∞, whereas A's router knows the distance to C is 1 and the next hop is C.

Now every node to sends its own table, (it's row, or vector) to each of its **neighbors**. Let's suppose that all the nodes do this **instantaneously**.

After this step, the table of what each node knows will look as follows:

Information stored at node	*Distance to Reach Node*						
	A	B	C	D	E	F	G
A	0^A	1^B	1^C	2^C	1^E	1^F	2^F
B	1^A	0^B	1^C	2^C	2^A	2^A	∞
C	1^A	1^B	0^C	1^D	2^A	2^A	2^D
D	2^C	2^C	1^C	0^D	∞	2^G	1^G
E	1^A	2^A	2^A	∞	0^E	2^A	∞
F	1^A	2^A	2^A	2^G	2^A	0^F	1^G
G	2^F	∞	2^D	1^D	2^A	1^F	0^G

Finally, after one more step of distance-vector, we will have:

Information stored at node	*Distance to Reach Node*						
	A	B	C	D	E	F	G
A	0^A	1^B	1^C	2^C	1^E	1^F	2^F
B	1^A	0^B	1^C	2^C	2^A	2^A	3^C
C	1^A	1^B	0^C	1^D	2^A	2^A	2^D
D	2^C	2^C	1^C	0^D	3^C	2^G	1^G
E	1^A	2^A	2^A	3^A	0^E	2^A	3^A
F	1^A	2^A	2^A	2^G	2^A	0^F	1^G
G	2^F	3^D	2^D	1^D	2^F	1^F	0^G

At this point, you will observe that the distance vector algorithm **converges**, i.e., the routing tables will now cease to change.

Question: Assuming that the network itself doesn't change (i.e. distances don't change), how many iterations will it take until the distance vector algorithm converges?

Answer: Consider the number of hops in the shortest path between each pair of routers. Now take the longest of these shortest paths. That is how many iterations we will need.

Question: Prove that the above answer is correct.

Answer: Consider any arbitrary two nodes, X and Y. Let P be the shortest path between X and Y and let l be the number of hops along this path. That is, $P = \{X, R_1, R_2, R_3, \ldots, R_{l-1}, Y\}$. I claim that if the shortest path between two nodes has length l, then that path, or an equally short one, will be discovered within l steps. We prove this by induction. Observe that every subpath of P is also a shortest path (otherwise P could be made shorter, which is a contradiction). At this point we can just invoke induction to say that after $l-1$ steps the $l-1$ hop path between R_1 and Y will be established. Therefore, after l steps, the l hop path between X and Y will be established.

If you don't like proofs by induction, try this argument: After the first iteration, the 1-hop path between R_{l-1} and Y will be established. Therefore, after 2 iterations, the 2-hop path between R_{l-2} and Y will be established. Therefore after 3 iterations, the 3-hop path between R_{l-3} and Y will be established, etc..

Question: Why isn't the convergence time $l/2$ iterations?

Answer: A router using distance-vector only talks to its immediate neighbors. It looks at the paths advertised by its neighbors, and appends itself onto that path. A router trying to deduce the length of the path to x does not add its path to k and k's path to x, unless k is an immediate neighbor.

3.2 Dynamic case: Costs change

Distance vector works by having the nodes send **periodic updates** to each other. A node might also send a **triggered update** if it detects a change on one of its neighboring links, causing a change in its routing table.

Question: What do you think happens when the link between F and G fails.

Answer: Node F detects the failure perhaps by testing, or perhaps by simply missing a periodic update). Node F updates its routing table to say that its distance to G is infinity and sends a triggered update to its neighbors. When A receives the update from F, it knows that it's route to G is through F, so it sets its distance to G also to infinity. At around the same time, however A hears from C and learns that it can get to G in 3 hops via C, so it updates its table and sends this info to F which updates its table and the system stabilizes again.

Unfortunately, there are slightly different situations where updates are **not synchronized** which prevent the network from ever stabilizing. Imagine the following simple network shown in Figure 4.

Before the increase in the link cost, the routing table looks like this:

Information	*Distance to Reach Node*		
stored at node	X	Y	Z
X	0^X	4^Y	5^Y
Y	4^X	0^Y	1^Z
Z	5^Y	1^Y	0^Z

Now the link cost increases from 4 to 60. Let's consider only a small portion of the above table which includes the costs of going from Y to X and the cost of going from Z to X.

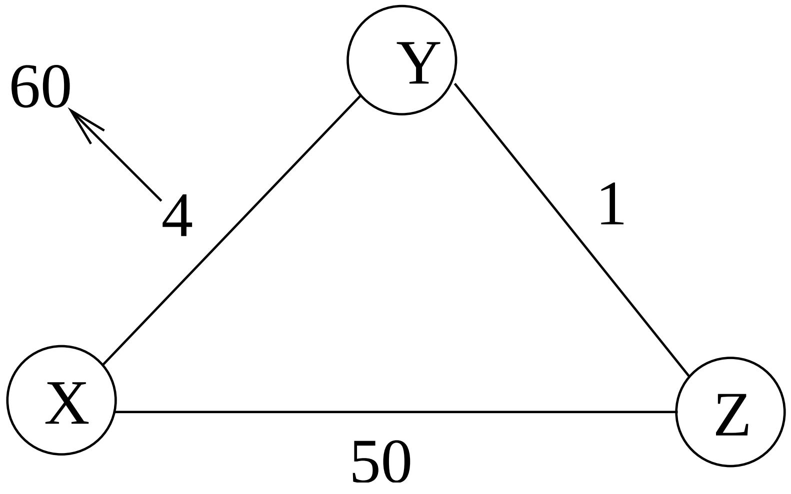

Figure 4: *Network with increase in link cost.*

Information stored at node	*Distance to Reach Node* X
Y	4^X
Z	5^Y

Now Y hears the bad news, so it has to rethink how to get to node X. Before it was going to get to node X via node X, which would cost 4. Now that path costs 60. However, Y has an old update from Z telling it that Z can get to node X in 5 steps, so Y decides that it can get to node X in 6 steps if it goes through Z. Y's table entry then becomes:

Information stored at node	*Distance to Reach Node* X
Y	6^Z
Z	5^Y

Y then sends Z a triggered update. Z has to rethink how to get to X. It could go directly, which would cost 50, or it could go through Y which would cost $6 + 1$. Z decides to go through Y and updates its table entry to look like this:

Information stored at node	*Distance to Reach Node* X
Y	6^Z
Z	7^Y

Z then sends Y a triggered update. Y will end up changing its entry as follows:

Information stored at node	*Distance to Reach Node*
	X
Y	8^Z
Z	7^Y

This process continues on and on for about 50 steps. If there is a link failure (sending cost to ∞), this process could go on for a very long time. This is called the **count to infinity problem**.

Question: There are many partial solutions to this problem. Can you think of any?

Answer:

One idea is to use a relatively small number, like 16, as an approximation to infinity, to make sure that this looping doesn't go on forever. (This of course assumes that the diameter of our network is < 16).

Another solution is to make sure that a node never sends a route that it learned from a neighbor back to the same neighbor. Specifically if Z's route to X was via Y, then Z should advertise to Y that Z's cost to X is ∞. This is a white lie, but is necessary to keep Y from wanting to use Z. If Z later finds a route to X which isn't via Y, then it can advertise to Y its true cost of getting to X. This is called **poisoned reverse**. The problem with this is that it eliminates routing loops of length 2, not more.

Another solution is to have nodes delay advertising, but this also delays convergence.

In practice the first of these ideas seems to be the one used.

Note: There is a lot more research in this area that I don't have time to cover. If you are interested in such research, please come see me and I'll give you more material.

Question: We discussed a link failing ... what about if a node fails?

Answer: Observe that it's impossible to tell the difference between a failed node and failed link, so behavior is the same.

Question: Suppose a link's cost improves, rather than getting worse. Do we still have a count to infinity like problem?

Answer: No. Good news travels fast. Bad news travels slowly.

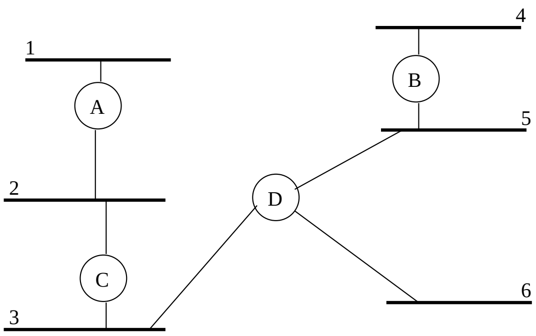

Figure 5: *Network on which we might run RIP.*

4 RIP

So you might be wondering at this point why I ever bothered showing you this bogus Distance Vector algorithm, with it's many convergence problems. The answer is: because this bogus algorithm is what's actually used in the Internet.

RIP (Routing Information Protocol) is the name of the Distance Vector algorithm which is actually applied in the Internet. It is widely popular, despite the above convergence issues.

There are almost no differences between RIP and the Distance Vector algorithm that we described above.

The biggest difference is that RIP has to get routers to learn how to forward packets to various *networks*, rather than the idealized graph model above where routers only learn about other routers. However this is an easy change. Rather than advertising the cost of reaching other routers, the routers advertise the cost of reaching networks. For example, in Figure 5 below, router C would advertise to its neighboring routers the fact that it can reach networks 2 and 3 in zero hops, networks 5 and 6 in 1 hop, and network 4 in 2 hops.

Thus the RIP packet format is a list of

(network-address, distance, next-hop router)

3-tuples.

RIP routers send periodic advertisements every 30 seconds, as well as the triggered advertisements. If a router hasn't received an advertisement from a neighbor in 180 seconds, it presumes the neighbor is dead, and updates its cost to infinity. RIP assumes all link costs are 1, just as we have above. RIP uses the number 16 to represent infinity, to prevent the count-to-infinity problem. Thus the maximum diameter of networks using RIP is 15 hops.

RIP is actually implemented as an application-layer protocol, running on top of UDP, although it is implementing network layer functionality (confusing, huh?). Here's how it works: Suppose you have a UNIX workstation serving as a router. A process called **routed** executes the RIP protocol, i.e., maintains the routing table, and exchanges messages with **routed** processes running in neighboring routers via port 520. Note that the routers are still considered layer 3 devices, since their main function, forwarding, is done at layer 3. However the routers have the higher layers available for administrative work like routing table maintenance.

Question: How can you see the routing table?

Answer: I thought that `netstat -rn` would do it, but it seems that `netstat -rn` doesn't show the costs of the paths. The *TCP/IP Illustrated Vol 1* book by Stevens recommends using `ripquery -n` to see the full routing table (with costs).

5 Link-State Routing Algorithm

The Link-State Routing algorithm, like the Distance-Vector Routing algorithm is only meant to work on a relatively small network (no more than about 100 routers). We will discuss next lecture how Link-State and Distance Vector are used in Internet routing, given the much larger scope of the Internet. The Link-State algorithm has advantages over the Distance-Vector algorithm in that it has much better stability and does not generate as much traffic. On the minus side, it requires storing more info at a node and doing much more computation at the node.

We again start with the assumption that each node knows only the distance to its direct neighbors.

The idea behind this algorithm is as follows: Each node knows the distance to its neighbors. This is called **link-state info**. If each node could send just that link-state info to every other node, then every node would have enough information to reconstruct the **whole graph**. This would certainly give every node sufficient information to now determine the shortest path from itself to all other nodes.

What happens in link-state routing is that each node sends its link-state info out on its immediate links. Every node which receives this link-state info **forwards** it to all its neighbors (except the neighbor from which it received the info). This process is called **reliable flooding**. Reliability is important here because each node needs to be sure that everyone in the whole graph has received its link state info.

Thus, whereas in Distance-Vector, every node sends out everything it thinks about every other node (it's full gossip vector), in Link-State, a node only send out what it **knows for sure**.

Here's how it works:

Each node creates a **Link-State Packet (LSP)**. The LSP created by node X contains:

1. ID of X
2. List of neighbors of X and cost to them
3. Sequence number
4. TTL (time-to-live) field.

The first two entities are the key data.

Question: What do you think the sequence number above is for?

Answer: Each time a node generates a new LSP, it increments the sequence number by 1. The sequence number makes sure that the most recent copy of LSP is is the one to get propagated. Old copies and duplicate copies get discarded.

Question: What do you think the TTL field is for?

Answer: We want to minimize the number of LSP packets, so we need to be able to kill them off after a while.

To ensure reliability, transmission of LSPs between adjacent routers uses a **reliable link layer protocol**, known as the Exchange Protocol, with ACKs and retransmits.

Just as in the Distance vector algorithm, there are periodic updates and also updates triggered by link failures. In practice the time between periodic updates is very long (maybe even hours).

Once all the nodes have the LSP for all other nodes, each node makes a **full map of the topology of the network** and decides the best route from itself to every other destination.

Question: Given a full map of the topology of the network, to determine the best router from a node to all other nodes, what algorithm can we use?

Answer: From your algorithms class: **Dijkstra's algorithm**.

Consider the graph shown in Figure 6. The nodes represent routers. The costs on the edges represent costs of getting from one router to another.

Question: Consider a node s. The goal of s is to find, for each node d, ... FILL IN BLANK ...?

Answer: For each node s and each node d we want to know the first hop on the least-cost path from s to d.

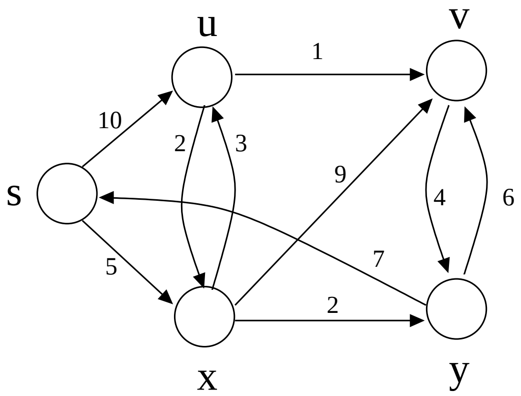

Figure 6: *Graph on which we'll apply Dijkstra's algorithm.*

Dijkistra's algorithm provides s with this information.

In the algorithm below, $D(v)$ represents the current estimated cost of going from s to v (stands for "distance to v"). $P(v)$ represents the node just before v on the currently-shortest path from s to v (stands for "parent of v"). Finally, $c(x, y)$ represents the cost on the link from x to y.

Dijkstra's Algorithm:

```
// Initialization part

S = {s}
For all nodes v{
     If v adjacent to s
          D(v) = c(s,v)
          P(v) = s
     Else
          D(v) = infinity
     }

// Main part

LOOP until all nodes in S{
     Find w not in S such that D(w) is a minimum
     Add w to S
     Update D(v) for all v adjacent to w as follows:
        If D(v) > D(w) + c(w,v){
```

```
            D(v) = D(w) + c(w,v);
            P(v) =  w;
        }
    }
```

Figure 7 shows Dijkstra's algorithm in action. Each successive graph is another iteration through the LOOP.

Question: Remember that what node s really wants is the "next hop" to each possible node. How does s get this info?

Answer: For each node, we can trace back the parent list. For example, v's parent is u. u's parent is x and x's parent is s. So s can enter in its forwarding table that the next hop to v is x.

Question: Does s need to "draw out" the whole graph to implement Dijkstra's algorithm?

Answer: Absolutely not. The link-state info and a few data structures suffice (need to store current distances, current parents, etc.).

Question: How many iterations does it take until the algorithm terminates?

Answer: The number of nodes in the graph.

Question: Why does the Dijkstra Algorithm work?

Answer: Dijkstra's Algorithm maintains the following invariant:

Whenever a node v is added to S, it must hold that $D(v)$ represents the shortest distance to v from the source s.

To see how this invariant is maintained, we argue by induction.

Let v be the ith node added to S. Assume by induction that the 1st, 2nd, ..., $i-1$th node added to S all have the property that at the time they were added to S, their shortest-possible path from s had already been found.

Now consider node v. Suppose by contradiction that at the time v is added to S the current path from s to v is suboptimal. Let P_{curr} represent the current path. Let P_{opt} represent the optimal (shortest) path from s to v. P_{curr} must differ from P_{opt}. Let's look for the last point *starting from the right* where the two paths agree.

Example:

$$P_{opt} : s, \dots\dots\dots\dots, m, b, \dots\dots v$$

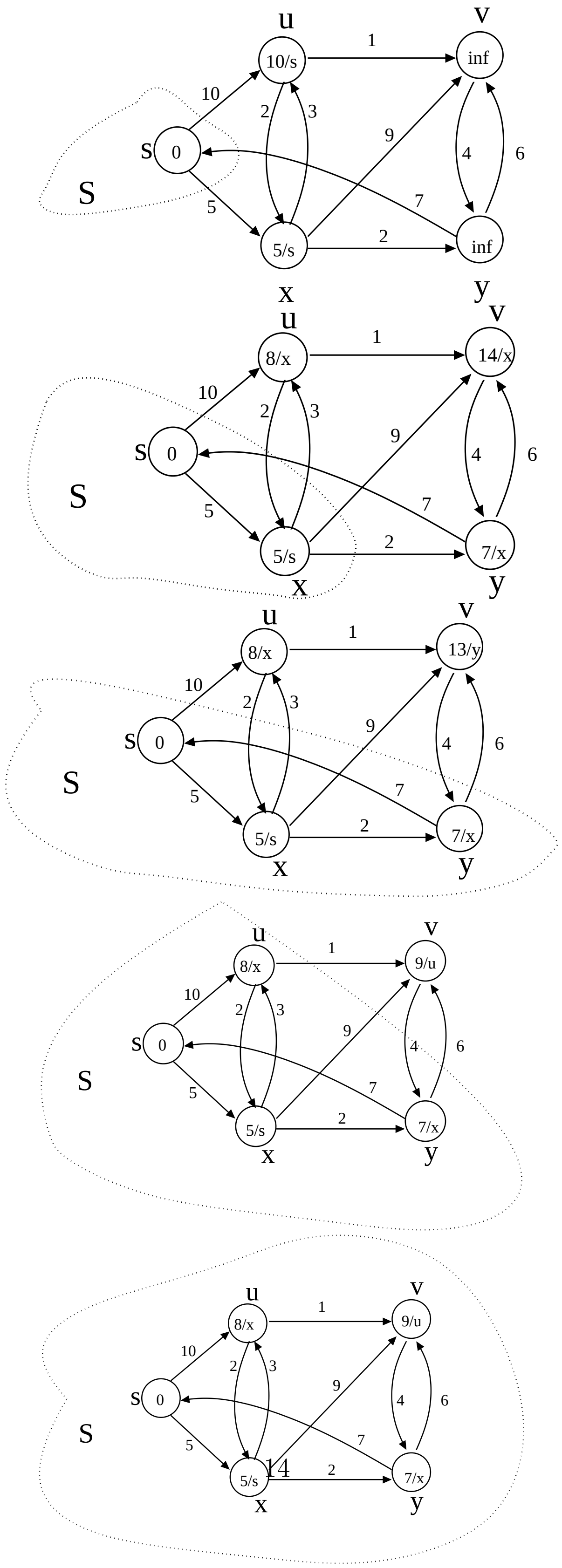

Figure 7: *Dijkstra's algorithm in action. Within each node v is shown $D(v)/P(v)$.*

$$P_{curr} : s, ..., z, b, ...v$$

In the example above, this is node b. By definition, all subpaths of P_{opt} must be optimal. Thus the path from s to b shown in P_{opt} must be the shortest path from s to b. But then at the time when b was added to the set S, the path from s to b was what is shown under P_{curr} which is suboptimal. But this contradicts the induction hypothesis.

Comparison of Link-State and Distance-Vector

Question: How much traffic does the Link-State algorithm generate as compared to the Distance-Vector algorithm?

Answer: Link-State creates much less traffic. Remember that in Link-State, only the LSPs are propagated. A node of degree d sends off an LSP of size d. In Distance-Vector, a whole vector is propagated. The size of this vector may be the number of nodes in the whole graph $>> d$.

Question: How much information must be stored at a node under the Link-State algorithm as compared with the Distance-Vector algorithm?

Answer: The Link-State has to store a lot more. Link-State stores one LSP for every node in the network, as opposed to Distance-Vector which only stores a couple fields for each node in the network.

Question: In Distance-Vector when a link or node fails, we can run into the count-to-infinity problem. Does this happen under the Link-State algorithm?

Answer: No. Say a link fails between nodes u and x. Nodes u and x send LSPs. Note, no one else has to send their own LSP (light on traffic!). Upon receiving the new LSPs, each node reruns Dijkstra's algorithm. Every node now comes up with new shortest path which doesn't involve the broken link.

Question: Interesting algorithms question: Is there a modification we can make to Dijkstra's algorithm, so that a change on a single link doesn't mean you have to rerun the algorithm from scratch?

6 OSPF

If RIP is the most widely-used IP routing algorithm, then OSPF – Open Shortest Path First Protocol – is a close second. The OSPF algorithm is a Link-State algorithm. It is very similar to the Link-State algorithm described above, except that a router's LSP doesn't carry information about neighboring routers, but rather about the neighboring **networks**, since the goal, of course, is to build a table that specifies the next hop for each *network*.

OSPF was developed after RIP and therefore has a few advanced features that have only recently being added to RIP, or that RIP still doesn't have. These include:

- Authentication of routing messages, so a malicious router/host can't suddenly send a message saying that it can reach every network with cost 0.
- Load balancing. A network might be configured to purposely show equal cost for two routes to the same place. OSPF will allow the router to split the traffic between these routes.

Question: How can you imagine making use of this load balancing feature? How should the router split the incoming traffic?

Question: Perhaps a router should send the first packet it receives along Path 1, the second along Path 2, the third along Path 1, the fourth along Path 2, and so forth ... Is this a good idea?

Answer: No, This scheme will potentially cause a lot of out-of-order packets and therefore more loss. We don't want to take one flow and split it.

Question: So what might the router do?

Answer: The router might pick a path based on the destination port, or based on the source IP address (e.g. traffic from even numbered IP addresses goes on Path 1, traffic from odd numbered IP addresses goes on Path 2.).

OSPF also has several important hierarchical features. It allows the network to be broken into sub-areas. Your textbook explains this well, and we don't have time to cover this in class. Please read about this in your textbook.

OSPF allows for all sorts of other advanced features. For example, it allows the use of **different cost metrics** (not just hop count) and also allows more than one cost metric to be assigned to a link, so that different cost metrics could be used for different applications. It is important to note that so far these fancy features have *not* been widely deployed. There is still a lot of work left to be done on how to choose good metrics in the first place. I will say a few words here about what's been tried:

The difficulty in choosing a cost metric

Queue Length metric The original ARPANET used queue length as a routing metric (high queue length = high distance). The problem with this is that queue length is a dynamic metric which changes frequently, and has the problem that the "best route" for one node is not independent of the "best route" for another node.

Question: Why would using queue length as a cost metric possibly cause instability problems?

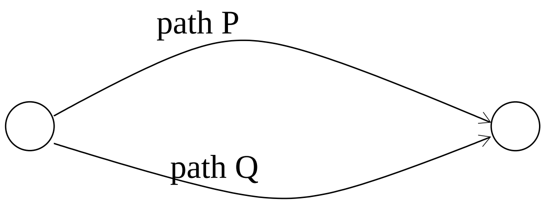

Figure 8: *Two paths P and Q*

Answer: Suppose that there are two major non-intersecting paths, P and Q. Suppose P has a moment of low load, and this is advertised to all nodes. All nodes simultaneously change their routes to P. Now P has very heavy load. All nodes change their mind and go back to Q. But now Q is heavily-loaded and so on and so forth. This oscillation may seem like a big problem. In truth this almost never happens because dynamic cost metrics are rarely used and certainly not used alone.

Aside from causing the above-described instability, the queue length metric also doesn't achieve the real goal of sending packets to short, high bandwidth links so as to lessen overall load on the network.

Sum of delays metric The second attempt for a routing metric was to let

$$delay = QueueingTime + TransmissionTime + PropagationTime$$

This had two problems: First, transmission time above was only computed for a single packet. In the case where the file being transmitted is large, we'd like transmission time to count more, since we want to make sure the file is sent over a high-bandwidth link. Second, it turned out that when propagation time was small, it was dwarfed by queueing time. The problem with this is that the algorithm sometimes ended up preferring a route with many many hops and short queueing delays at each hop to a single-hop route with large queueing delays at the router. This may work well for a single message, but is a poor use of overall network resources.

Revised ARPANET routing metric The third attempt tried to limit the range of each delay type so that none could dominate. Queueing time was changed to "utilization" and was only allowed to count if utilization was high. It was allowed to count more in the case of low bandwidth links and less in the case of high-bandwidth links. Also, different scales were used for satellite links as opposed to terrestrial links. The Peterson/Davie textbook, p. 303, Fig 4.21, shows a graph of how much utilization counts under different link types. Your book makes the important point that all these "slopes, offsets, and breakpoints for the curves in Fig 4.21 were arrived by a great deal of **trial and error**." The point you should take away is that coming up with a single universal metric of delay is really hard. In practice it is done by the system administrator of the autonomous system. Remember the OSPF is only run on networks of size 100 or fewer routers and all within one autonomous system (one company).

7 AS Hierarchy

So far we have seen two routing algorithms. Both however are only practical in the case of networks with fewer than 100 routers.

But the Internet has **many more routers** and **over 50,000 networks**!

The size of the Internet creates several big headaches:

1. Any routing algorithm like Distance-Vector or Link-State will never complete when applied to the large number of routers in the Internet.
2. Any routing algorithm will have to pass around too much data, flooding all the links in the Internet.
3. The forwarding tables, assuming that they ever get created, will be huge! Remember we need one entry for each network. How will we store them?

We stop here and start next time with discussing the solutions to these problems.

Thank You!