1 GSW… Routeing Algorithms

One possible point of confusion to sort out first: although operating at the internet (network) layer[1], IP is not a routeing protocol, it is just a routable protocol: the IP header contains the address of the destination for the packet, the address that the routers look up in their routeing tables to work out where to send the packet.

Routeing protocols have the job of working out the contents of the routeing tables in the routers. This chapter (the first of two) is about the basic problem that routeing protocols try to solve, and introduces two of the most common algorithms that routeing protocols use to solve it: the Bellman-Ford and Dijkstra algorithms. I won't talk about any details of the operation of the protocols themselves: that's the subject of the next chapter.

1.1 Methods of Routeing

By means of an example, consider a small network connected in a mesh topology, as shown below:

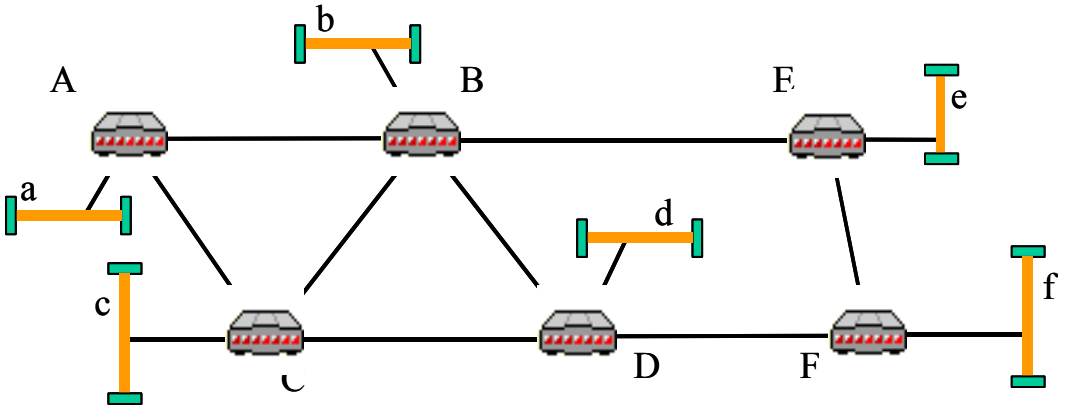


Figure 1-1 – Sample Network for Routeing

This network connects six subnetworks (**a** to **f**) with the aid of six routers (**A** to **F**). The task of the routeing protocol is to get enough information to each router so that it can work out where to send packets destined for any of the subnetworks. There are a few very simple options:

- **Fixed Routeing tables**. Get the administrator to tell each router where to send packets to every network by manually typing the addresses into each router[2]. There are two main problems with this approach: firstly it's a huge managerial task on a big network, and packets can easily get lost if a mistake is made; and secondly, if a router or a link between routers breaks down, packets are lost until human intervention can sort things out.

[1] Another point of potential confusion: the layer responsible for routeing packets from the source node to the destination node across a network is known as the network layer in the ISO OSI protocol stack, and as the Internet layer in the TCP/IP protocol stack.

[2] This might seem like an impossibly impractical way to run a network; but on small and stable networks it has some benefits (simplicity), and it was how routeing was done on the Internet when it first started (although that was before it was called the Internet).

- **Flooding**. Every packet arriving at any router is sent out of all other ports. Hugely wasteful of network resources, it also requires a receiver intelligent enough to delete unwanted copies of packets (usually several copies arrive at the destination). It is however extremely robust to routers or links breaking down: if it is possible to deliver the message, flooding will succeed in delivering it. Another problem is that the protocol requires the ability to delete packets, otherwise if there is any possibility of a loop in the network, packets could end up going round in circles forever. It's a useful technique on very dynamic mobile networks, where nodes move around faster than any routeing protocol can hope to keep up.

- **Random**. Pick a random output from each router, and send the packet out of that one. Statistically, eventually the packet should make it to the right destination. This technique uses a much lower total amount of traffic than flooding, it's still useful on very dynamic networks, and is still immune to some router breakdown, but the delay across the network can be long and variable, and packets can (and often do) overtake each other and arrive in a random order. The receiver needs to be intelligent enough to put them back into the right order again.

None of these simple methods are very satisfactory. More intelligent adaptive routeing strategies would clearly be better, ones that allow the routers themselves to work out the best route for each packet to take, and to store this information.

1.2 Routeing Tables

Making any more intelligent decisions about where to send packets requires the routers to have and maintain a routeing table[3]. These tables contain information about which port to send packets out from, and where to send it to. A typical routeing table entry has five fields: an IP address prefix, a subnet mask, a cost metric, the interface (port) to send the packet out from, and the address of the next hop. A simple example might look something like this:

Destination Network Address	Destination Network Bit Mask	Output Interface	Next Hop	Cost
192.168.0.0	255.255.254.0	Ethernet0	Direct	1
192.168.2.0	255.255.254.0	Ethernet1	Direct	1
0.0.0.0	0.0.0.0	Serial1	Router 192.168.4.31	3

When a packet arrives at a router, it searches down the routeing table to find entries whose IP addresses and network masks match the destination of the incoming packet. To match an entry, the packet's destination IP address ANDed with the network mask must equal the network number. For example, an IPv4 address prefix of 192.168.0.0/23 would match any

[3] Computers have routeing tables too, although their tables are usually very simple: if the packet is destined for this computer, then receive it; if it's destined for someone else on the same local area network send it directly there; otherwise send it to some default router. If you've got a PC handy, try bringing up a command window, and entering the command "route print". That should print out the current contents of the routeing table.

IPv4 destination address with the first 23 bits equal to the first 23 bits of 192.168.0.0: in other words, any IPv4 address between 192.168.0.0 and 192.168.1.255.

If no routeing table entries match a packet's Destination Address, the packet is discarded as undeliverable (possibly with an ICMP 'destination unreachable' packet sent back to the sender). If multiple routeing tables entries match, the entry with the longest network mask is preferred[4] (that is, the entry with the most 1 bits in its routeing mask).

To avoid needing routeing entries for every possible Internet destination, most hosts and routers use a default route with a network address and bit mask of 0.0.0.0/0. In other words, it matches every IPv4 address, but since there are no '1' bits in its routeing mask, any other valid match would be selected by the "use the entry with the longest match" rule. The default route will only be used if there are no other matches in the routeing table; hence its name.

The output interface and next hop fields in the routeing table tell the router which port to send the packet out of, and which node to send it to ('direct' in the example table above means send it directly to the final destination).

1.3 Routeing Protocols

The large number of routeing protocols in use can be divided into three general types: distance-vector protocols, link-state protocols and path-vector protocols.

1.3.1 Distance-Vector Routeing Protocols

A *distance-vector* protocol gives each hop (from a router to another router, or from a router to a subnetwork attached to a router) a *cost* or *metric*. This cost is sometimes known as a *distance*, in the sense that it is a measure of how far away the router is from the destination. If there are two ways to get a packet to a destination, it is sent by the route with the lowest cost.

With distance-vector protocols, routers periodically broadcast their costs (distances) from all the subnetworks they currently know about to all the other routers they can directly talk to. Eventually, information about all the subnetworks spreads out across the network, and all the routers store the lowest-cost route to each destination (but that's all)[5].

Distance-vector protocols are simple, but if something goes wrong, they can take a long time to recover.

1.3.2 Link-State Routeing Protocols

Link-state protocols are more complex, but more efficient. Again, they operate by routers telling each other about their local environment, so that gradually information about the entire network spreads throughout the network. However, rather than just storing the lowest cost route to a destination subnetwork, routers running link-state protocols attempt to build up an

[4] See the chapter on IPv4 Addressing for an example of how this rule can help simplify routeing tables.

[5] Some real-world protocols based on distance-vector routeing do store more information than this (for example a back-up route with the second-lowest cost as well), but a 'plain vanilla' distance vector protocol does not, it only stores one route to each destination subnetwork: the route with the lowest cost.

internal map of the whole network, and then make intelligent decisions about where to send packets based on looking at this map.

If something goes wrong in the network, a link-state protocol just has to inform all the routers than the faulty link is no longer available, and the routers can adapt their internal maps, and quickly work out how to route the packets to avoid the bad link.

1.3.3 Path-Vector Routeing Protocols

Path-vector protocols do not attempt to store the topology of the entire network (unlike link-state protocols), but they do store information about the whole path to each destination networks (unlike distance-vector protocols, which only store information about the next hop).

Storing information about the entire path to the destination helps prevents *routeing loops* (situations in which a set of routers thinks that the best route to a destination is via each other, and the packets go around in circles until they time out), since each router can look and see if it is part of the path to a final destination that a neighbouring router tells it about. (If so, it knows that the path is useless, and ignores it.)

Path-vector protocols are more immune to routeing loops than distance-vector protocols, but require more information to be passed between neighbouring routers (the entire path, not just the next hop). They don't reconfigure as fast as link-state protocols, but they require much less processing at each router.

1.4 The Bellman-Ford Algorithm

The Bellman-Ford algorithm is one method of automatically generating routeing tables, and is widely used in distance-vector protocols. It works by an iterative process, with each router sending its current routeing tables to its neighbouring routers.

The idea in brief: each router transmits the details of every network and subnetwork it knows about together with the costs to send a packet there, to every other router it can communicate with directly. Each router then calculates the lowest-cost route to the destination subnetworks, and maintains this data in the routeing table.

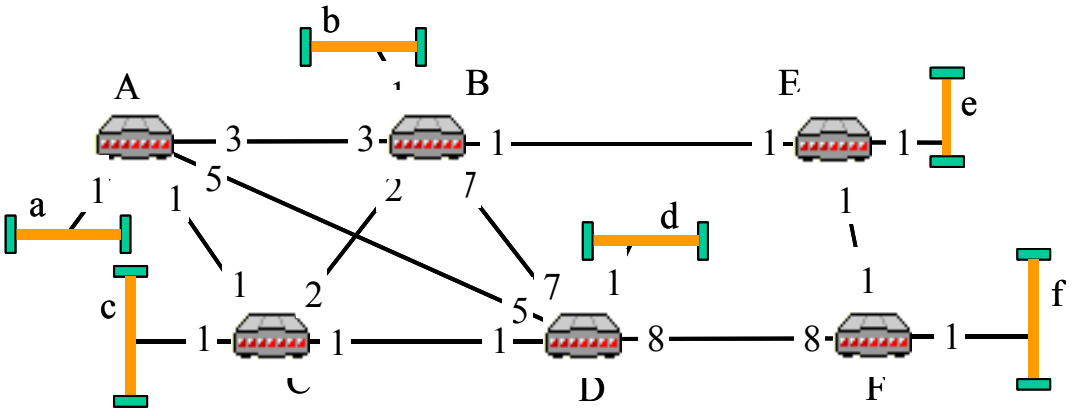


Figure 1-2 Sample Network with Costs on Inter-Router Links

This should be much clearer with an example. Consider the network shown above, where the costs of sending a packet out from a certain port on each router are shown. Note that the cost of sending out a packet onto a local Ethernet (direct to the final destination) is always set to one: there's no point in setting the cost any higher, any packet destined for any node on an Ethernet will have to go out of its attached router's port.

Just after switching the whole network on, the initial routeing tables (with cost values) could be very simple[6]. For example, for router A, the initial table could look like this[7]:

Destination Network / Bit Mask	Output Interface	Next Hop	Cost
a	Local Ethernet	Direct to destination	1

(The cost of going from Router **A** to subnetwork **a** is assumed to be one.)

There would be similar tables at each of the routers. A little while later, router **A** would receive copies of the tables from routers **B**, **C** and **D**, and it would find out about the subnetworks **b**,**c** and **d**, and add these subnetworks to its routeing table, and compute the

---

[6] In practice, it's usually more complex than this for two reasons. Firstly, since the router has the ability to receive packets itself (otherwise it couldn't receive the routeing tables from its neighbouring routers), there must also be an entry in the routeing table with the IP address of the router, directing all packets sent to the router's IP address to a local interface for the processor in the router.

The second reason is that I've ignored entries for the links between the routers themselves. In practice, these are subnetworks too, with their own network addresses and bitmasks, and the ports on the routers have IP addresses. A complete routeing table for router **A** might look something more like this:

Destination Network / Mask	Output Interface	Next Hop	Cost
a	Local Ethernet	Direct to destination	1
AB / 30	Serial Link to **B**	Direct to destination	3
AC / 30	Serial Link to **C**	Direct to destination	1
AD / 30	Serial Link to **D**	Direct to destination	5
port on **a** / 32	local	none	1
port on **AB** / 32	local	none	1
port on **AC** / 32	local	none	1
port on **AD** / 32	local	none	1

Here **AB** represents the subnetwork of the serial link between router A and B, which I've given a bitmask of / 30 to, since there are only two nodes on this subnetwork (one at each end). The four ports of this router represent the four IP addresses this router has (one for each port), and they are all received locally by the router itself.

I'll won't bother writing out these complete routeing tables here, since it's only the entries that point to the other subnetworks that are really interesting from the point of view of understanding the Bellman-Ford algorithm.

[7] Note I've combined the 'destination network address' and 'destination network bit mask' fields into one field just indicating the subnetwork, to save a bit of typing. A real routeing table would have both.

current value of the 'Cost' field with a new value formed from the sum of the 'Cost' field in the incoming routeing table, plus the cost of the link the frame came in from.

Just after the first set of these frames had been received, the routeing table for router **A** would look like this:

Destination Network / Mask	Output Interface	Next Hop	Cost
a	Local Ethernet	Direct to destination	1
b	link to **B**	Send to **B**	4
c	link to **C**	Send to **C**	2
d	link to **D**	Send to **D**	6

Note that router **A** still hasn't heard of subnetworks **e** or **f**, since when the routeing tables left routers **B**, **C** and **D**, these routers had not received routeing tables from **E** or **F**[8].

Meanwhile, routers **B**, **C** and **D** would have their routeing tables updated as well, as they will receive copies of the routeing tables from all routers directly connected to them.

A little time later, all the routers will pass on their updated routeing tables to each other. Now, router **A** will find out about subnetworks **e** and **f**, since router **B** will have found out about subnetwork **e**, and router **D** will know about subnetwork **f**. Router **A**'s routeing table now looks like this:

Destination Network / Mask	Output Interface	Next Hop	Cost
a	Local Ethernet	Direct to destination	1
b	link to **B**	Send to **B**	4
c	link to **C**	Send to **C**	2
d	link to **C**	Send to **C**	3
e	link to **B**	Send to **B**	5
f	link to **D**	Send to **D**	14

Three interesting things have just happened:

- The route (and the cost) from Router **A** to subnetwork **d** has changed: router **A** has just received information from router **C** that it had a route of cost 2 to network **d**, and that makes the total cost of going to network **d** via router **C** equal to **3** (a cost of one to get to router **C**, then a cost of two from router **C** to subnetwork **d**). That's less than the current total cost (of 6, going direct via router **D**). So the table is updated, and from now on router **A** sends all packets destined for network **d** via router **C**.

- The router can now reach everywhere in the network, but not necessarily by the optimum routes. It has only learned about routes with two hops. It will send

[8] I'm assuming here that all routers are switched on simultaneously, and routeing tables are transmitted by all routers at the same time. This is admittedly very unlikely in practice, but it does make the explanation of the Bellman-Ford algorithm a bit easier. The algorithm still works, no matter what order routeing tables are exchanged.

packets to subnetwork **f** to router **D**, despite the fact that it already knows that a better route to subnetwork **d** exists (via router **C**). However, router **C** doesn't know about subnetwork **f** yet, so subnetwork **f** was not in router **C**'s routeing table. The only router that told router **A** about subnetwork **f** was router **D**.

- The route to subnetwork **b** is still via router **B**. In fact, router **A** would have received information about subnetwork **b** from both routers **B** and **C**, and would have worked out that it could get a packet to subnetwork **b** via both routers for the same cost: three. Which does it chose? Usually, it would chose the one it found out about most recently, on the grounds that this is most likely to be an up-to-date route. Here, I've assumed that router **B**'s routeing table arrived slightly after router **C**'s routeing table, so the route via router **B** was chosen.

In this way, each router iteratively discovers the lowest cost route to any subnetwork in the network. After one iteration, all routers know the best one-hop paths to the subnetworks, after two iterations, all routers know the best two-hop paths, and so on.

1.4.1 Problems with Distance-Vector Protocols

The Bellman-Ford algorithm is often used in distance-vector routeing protocols. It will always find the optimum routes in a fixed network topology; however there can be problems with all such routeing algorithms when the topology is not fixed. Consider the network shown below (again, assume that Router **A** is connected to network **a**, etc, and costs of going both ways along the links are identical).

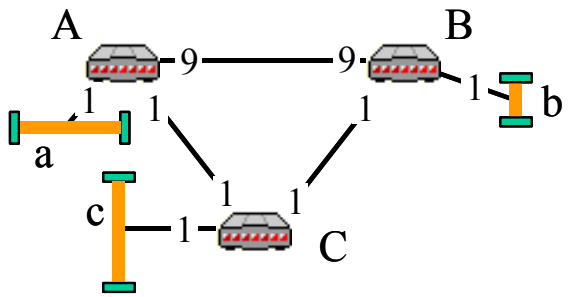


Figure 1-3 – Sample Network Illustrating Loops

The Bellman-Ford algorithm will quickly work out what the costs are to get to each network from each router. Now suppose that the link from **C** to **B** goes down. What we would like is that the network instantly re-configures to use the higher cost link from **C** to **B** via **A**. But consider what actually happens: just before the break, **A** thinks it can get to **b** via **C** for a cost of three. **C** thinks it can get to **b** via **B** for a cost of two. And **B** knows it can get to **b** directly, for a cost of one. So we can write up the costs and routes for these three routers as follows:

Going to b from Router:	Next Router	Cost
A	**C**	3
B	-	1
C	**B**	2

(Please note this is not a routeing table: it's a simplified composite of entries from three routeing tables.)

As soon as the link between **B** and **C** breaks, **C** will notice it can no longer get to **B**, but **A** won't:

Going to b from Router:	Next Router	Cost
A	**C**	3
B	-	1
C	-	∞

Now at the next update scheduled time, **A** will get a message from **B** that the cost-9 route direct to **B** is still available, and from **C** that it doesn't know how to get to **b** anymore. So **A** will try and get to **b** via the direct route. **C** will get a message from **A** saying it can get to **b** for a cost of three, so it will send packets to **A**:

Going to b from Router:	Next Router	Cost
A	**B**	10
B	-	1
C	**A**	4

and everything works again. **C** thinks the route to **b** is easier than it really is, but that's OK, the packets will still get there. But consider what happens after the next update time:

Going to b from Router:	Next Router	Cost
A	**C**	5
B	-	1
C	**A**	11

Oh dear. Any packet that **A** or **C** receives to send to a node in network **b** will oscillate between the two routers **A** and **C** until they time out. This is known as a *routeing loop*. What happens in the next update time? Does this situation continue indefinitely?

Going to b from Router:	Next Router	Cost
A	**B**	10
B	-	1
C	**A**	6

Fixed, although router **C** still has the wrong cost. Next update:

Going to b from Router:	Next Router	Cost
A	**C**	7
B	-	1
C	**A**	11

and we've got the routeing loop back. If we carry on like this, we'll note that the cost from **A** or **C** to subnetwork **B** is slowly increasing. Only when it reaches a greater number than the cost to send a packet from **A** to **B** via the high-cost direct link will the network finally achieve a stable state again. This can take some time.

1.4.2 Split Horizons with Poisoned Reverse

There's a simple solution to this problem: never advertise 'reachability' to a subnetwork to the router through which you would forward the packets for that network. There's no point, the other router should have more up-to-date information about the routes to those subnetworks than you do, since it's closer to them.

This implies that the routeing table being sent out of each port of the router is no longer the same one that the router uses: it has the total cost of paths to all routes out through that port set

to infinity. This is known as *split horizons with poisoned reverse*. (One could just omit these routes from the routeing tables sent out: that would be just *split horizons,* but that would mean that invalid routes would have to time out, rather than being set to an infinite cost and hence eliminated immediately).

Consider the previous example with the use of split horizons with poisoned reverse, we start with the same routeing table entries of:

Going to b from Router:	Next Router	Cost
A	**C**	3
B	-	1
C	**B**	2

Now just as before the link between **B** and **C** breaks, and **B** will notice it can no longer get to **C**:

Going to b from Router:	Next Router	Cost
A	**C**	3
B	-	1
C	-	∞

and again, **A** will get a message from **B** that the cost-9 route is still available, and from **C** that it doesn't know how to get to **b** anymore. **A** will now try and get to **b** via the direct route[9].

Now the difference: **C** will no longer get a message from **A** saying it can get to **b** for a cost of three. That route from **A** would go through router **C**, so **A** will not advertise that route to router **C**. There's no point: if this route was available, router **C** would already know about it. Net result: Router **C** still has no idea how to get to subnetwork **b**. We've got:

Going to b from Router:	Next Router	Cost
A	**B**	10
B	-	1
C	-	∞

Then, when the next update comes from router **A** to router **C**, router **C** will find out about the route from **A** to **b** via router **B** for a cost of 10: this route is advertised to router **C** since it does not go through router **C**, it goes directly to router **B**. We end up with:

Going to b from Router:	Next Router	Cost
A	**B**	10
B	-	1
C	**A**	11

[9] There's a subtlety here. The sequence of events described assumes that the routeing update from router **C** arrives at router B before the routeing update from router **B**. If they had arrived in the other order, router **A** would first have heard about the route of cost 10 via router **B**, and ignored it, since it thinks there's a perfectly valid route via router **C**. Then, when the update from router **C** arrives, it would say "oh dear", and delete the route from the routeing table. Since basic Bellman-Ford algorithms only store the lowest-cost router, router **A** would then have no idea how to get to subnetwork **b**, and packets arriving at router **A** destined for subnetwork **b** would be deleted. This makes router **A** into what's known as a routeing *black hole*.

and everything has adjusted back to a stable state. Great. It's a bit more complex, since different versions of the routeing tables have been sent out of the different ports, but it avoids this infinite loop occurring.

Have we fixed the problem in all cases? No.

1.4.3 More Problems with Distance-Based Algorithms

Consider the network shown below:

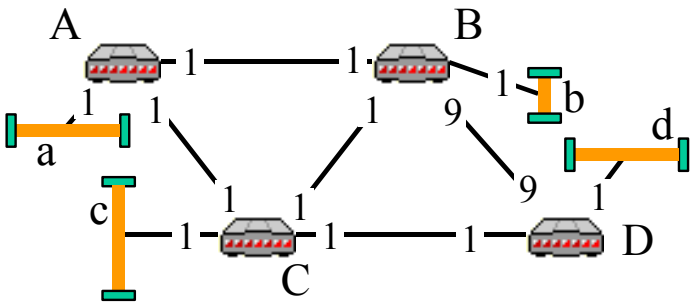


Figure 1-4 Another Sample Network with Routeing Loops

After all the routers have established their routeing tables, consider the various entries in the routeing tables for how to get to subnetwork **d**:

Going to d from Router:	Next Router	Cost
A	**C**	3
B	**C**	3
C	**D**	2
D	-	1

Now, consider what happens if the link between **C** and **D** goes down. After a short time, **C** would notice that it wasn't getting any packets back from **D**, and vice versa, and **C** would mark **D** as "unreachable", giving it an infinite cost to reach:

Going to d from Router:	Next Router	Cost
A	**C**	3
B	**C**	3
C	-	∞
D	-	1

At the next scheduled routeing update time, the fact that **C** could no longer reach **D** would propagate around the other routers, which would then adjust their paths to **d**:

Going to d from Router:	Next Router	Cost
A	**B**	4
B	**A**	4
C	-	∞
D	-	1

and now both routers **A** and **B** think they can get to **d** via the other. Packets oscillate between routers **A** and **B** until the next update time. Not desirable. It turns out to be difficult to stop this happening.

1.4.4 Triggered Updates

To help fix this sort of problem, another technique to speed up the reconfiguration of the network is used: *triggered updates*. Whenever a router sees a big jump in cost (for example a link goes down), it transmits the routeing table immediately, without waiting for the next scheduled update (which could be up to thirty seconds later). This speeds up the re-configuration process, so that at least the loops don't last for very long.

There's a slight problem with this: when the network is particularly unstable, large changes in the costs can happen frequently, and this could in theory lead to a very large amount of traffic being transmitted over the network as all the routers attempt to transmit their routeing tables every time they notice something has changed.

Two techniques are used to limit the amount of information transmitted. The first, called *holddown*, delays the transmission of a triggered update slightly (typically by a few seconds), so that a single routeing table is transmitted in response to what could be several triggering events over the previous few seconds (although this can slow down the reconfiguration of the network). The second is not to transmit the entire routeing table during a triggered update, but only the routes that have changed.

1.5 Dijkstra's Algorithm

One problem with all distance-vector algorithms is that it can take some time for changes in the network to propagate to all of the routers in the network, slowing down the reconfiguration time. A solution to this is for each router in the network to learn about the entire topology of the whole network, so that once informed of a change (perhaps a broken route) each router can work out the best way to update its routeing tables itself. Routeing protocols that do this are known as *link-state protocols*, and commonly use Dijkstra's algorithm (sometimes known as the Shortest Path First or SPF algorithm) to determine the best route to a destination.

Dijksta's algorithm works out the best route from one router to any sub-network, assuming each router has a complete knowledge of the topology and costs of every link in the network. (This is significantly different to the Bellman-Ford algorithm, which only requires a router to have knowledge of the costs of transmitting a packet out of its own ports.)

Once a router has constructed a map of the entire network, it divides the routers in the network into two groups: a sorted group (S) and an unsorted group (U). Initially, it places itself in the sorted group, and all the other routers in the network in the unsorted group. It then goes through the unsorted group, determining the total cost of transmitting a packet from each unsorted router back to itself. Whichever unsorted router has the lowest cost is then moved into the sorted group, and the lowest-cost link from the original root router to the newly sorted router is switched on. After N iterations (where N is the number of routers) the algorithm generates a *spanning tree*[10].

For example, consider the network shown below:

[10] A spanning tree is a set of connections between nodes that joins every node, but does not have any loops. There's more about spanning trees in the chapter on Bridging and Switching, since exactly the same issues come up in bridging too.

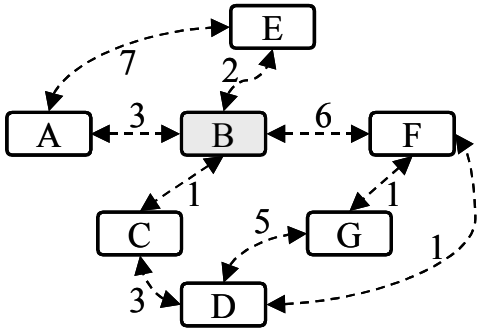


Figure 1-5 – Dijkstra's Algorithm Step One

Suppose that router **B** is the router that is trying to determine the best routes to use to the other routers in the network. At the initial stage of the algorithm, router **B** is in the sorted group (shaded), all other routers are unsorted. (I'm assuming here that the costs associated with each link are symmetric; this is not necessarily the case in practice.)

In the first iteration, the lowest cost router from **B** is identified (as router **C**), the link between them made active, and router **C** moved into the sorted group. The next iteration sees router **E** (cost 2) added to the tree.

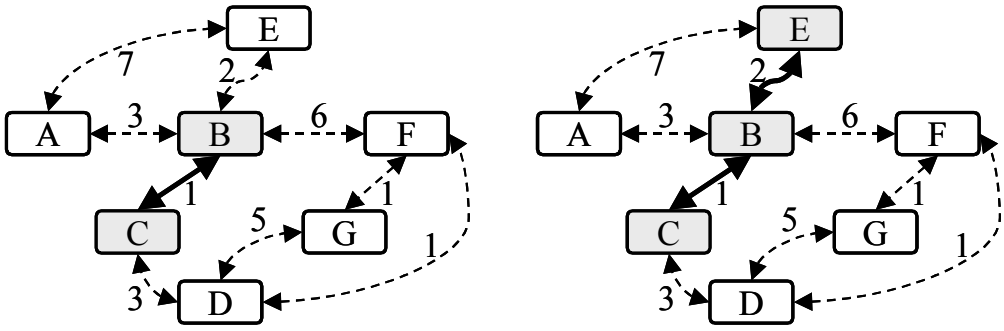


Figure 1-6 – Dijkstra's Algorithm Steps Two and Three

This process carries on until all the routers have been added to the tree: next are routers **A** (cost 3) and **D** (cost 4):

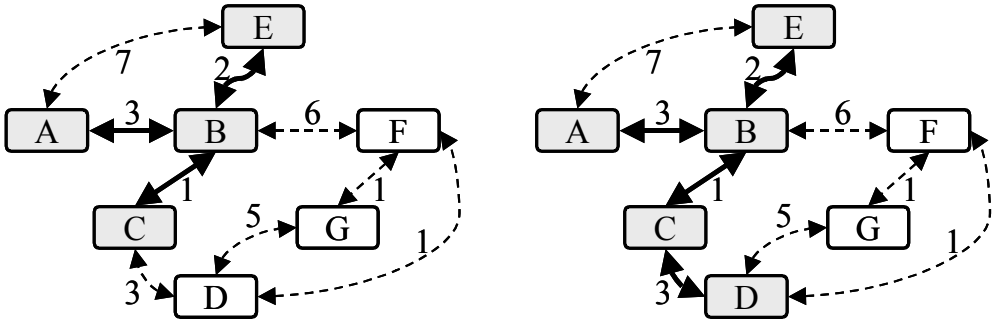


Figure 1-7 – Dijkstra's Algorithm Steps Four and Five

and finally routers **F** (cost 5) and **G** (cost 6).

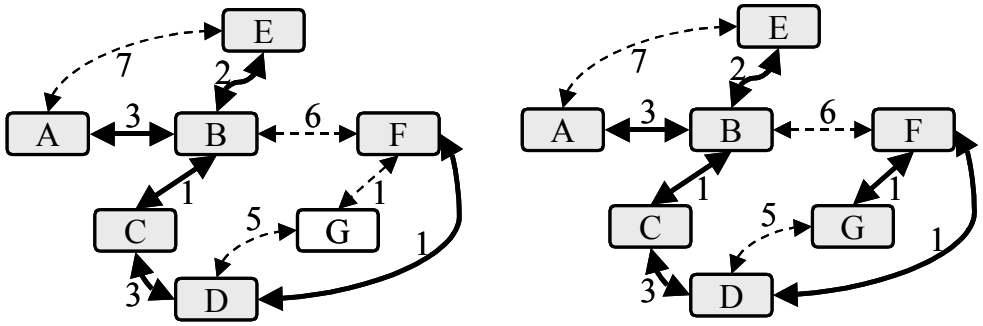


Figure 1-8 – Dijkstra's Algorithm Steps Six and Seven

Each router can then construct a routeing table based on the lowest-cost possible path from itself to the destination. The advantage of this method over Bellman-Ford is that it is much less likely to produce routeing loops during reconfiguration, and the amount of information required to be passed around the network is much lower. No longer is the entire routeing table (possibly modified by split horizons and poisoned reverse) sent out of each port of every router, now it is only the information about which routes are up and down that is required to be sent around the routers.

1.6 Key Points

At this point, you should:

- Know what Flooding, Random, and Fixed routeing strategies are, and why none of these strategies would work for the majority of the Internet;
- Know the difference between distance-vector, link-state and path-vector routeing protocols, and the advantages and disadvantages of each one;
- Be able to apply the Bellman-Ford algorithm to any given network, and work out how the routeing tables will configure themselves;
- Understand what *split horizons*, *triggered updates* and *poisoned reverse* mean in terms of routeing tables, and how they can help speed up re-configuration;
- Be able to apply the Dijkstra (Shortest Path First) algorithm to any given network, and work out how the routeing tables will configure themselves.

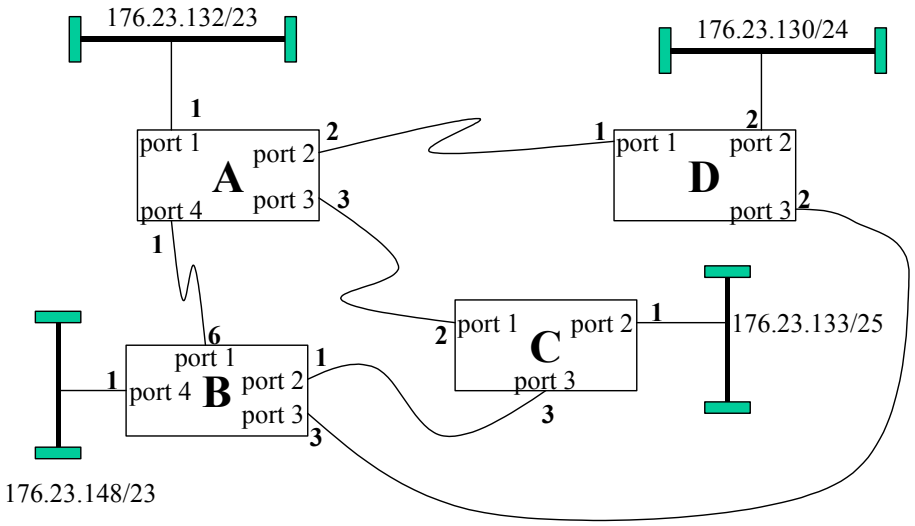
1.7 Tutorial Questions

1) Describe the key differences between distance-vector, path-vector and link-state routeing protocols.

2) Explain how the Bellman-Ford algorithm operates, and determines routeing tables. In particular, what is 'split horizons with poisoned reverse', and what problems does it help solve?

*3) The network shown below consists of four routers, with the costs associated with sending a packet out of each of their ports. Determine the contents of the routeing tables in each router after 1,2, and 3 iterations of the Bellman-Ford algorithm. (Neglect the routeing table entries

for the router ports associated with the links between routers, and assume all the routers are switched on at the same time, and transmit their routeing tables simultaneously.)



4) Given the network in question 2, are there any cases in which a packet going from one Ethernet to another takes a different route than the packets going in the opposite direction? If so, give an example.

Discuss under what circumstances this could happen in practice.

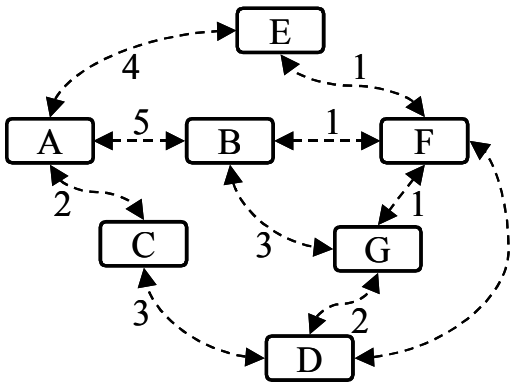
*5) Suppose the link between router A and router C was broken. Show how the Bellman-Ford algorithm would re-adjust the routeing tables in the nodes. How many iterations would be needed to re-establish a stable set of routeing tables?

**6) Consider the network given in section 1.4.3. How many update cycles does it take before the network finally removes all loops and black holes? How many cycles before the values in the routeing tables are stable?

**7) What is the disadvantage of using 'poisoned reverse'? Why might a network administrator want to turn it off, and just use 'split horizons'?

**8) A router running a routeing protocol based on the Bellman-Ford algorithm receives notification of a route with the same metric as it currently has for a given destination subnetwork, but from a different interface than its current route to that subnetwork. What's the best thing to do? Ignore this new route and retain the old one, or use the new route? Why?

9) A network contains 7 routers, as shown below. Describe how Dijkstra's algorithm works, and determine in what order the nodes would be added to the routeing table for routers **A** and **G**.



Are there any cases in which a packet going from a router **X to a router **Y** goes by a different route than one from router **Y** to router **X**? Is this possible with a link-state protocol running Dijkstra's algorithm?