The Control Plane: Distance Vector - Routing Information Protocol

In this lesson, we'll discuss the Routing Information Protocol, a popular distance-vector algorithm, based on the famous Bellman-Ford algorithm.

We'll cover the following

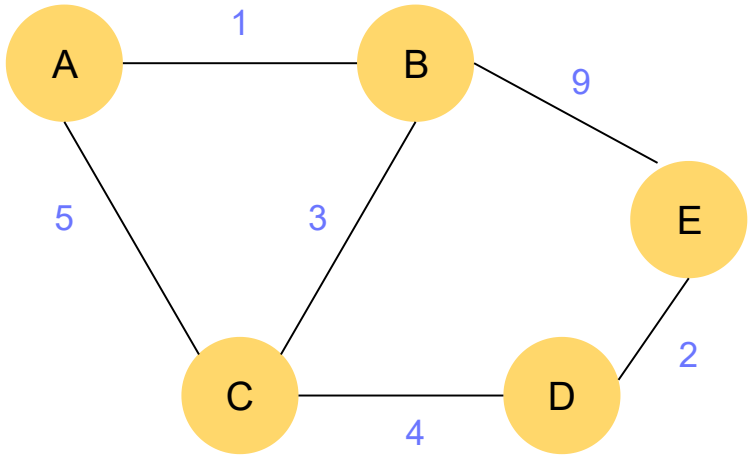
Introduction

The **Routing Information Protocol (RIP)** based on the famous **Belmman-Ford** algorithm belongs to the distance vector class of routing algorithms and was used in ARPANET. While it used to be incredibly popular, it’s not used very much now. There are other distance vector routing algorithms too such as Ford-Fulkerson.

Initial State

Each router or ‘node,’ maintains a **routing table** that initially contains the *estimated* cost to each of its neighbors.

Consider the following example of a small network where the yellow circles represent nodes, the black lines represent links, and the purple numbers represent the cost of each link.



Sample Network

Initial Routing Table

What would the initial routing table look like at node **C** for example?

Destination	Cost	Next Hop
A	5	A
B	3	B
D	4	D

initial routing table at C

The table contains:

1. The names of the **destination nodes** which are the neighbors in this case.
2. The **initial cost** of the link to each of **C**'s neighbors,
3. The **'next hop' node**, i.e. the node that **C** would have to send a packet to in order for it to reach its destination. In this case, the next hop and the destination are the same since the destinations are all **C**'s neighbors.

Every node receives all of its neighbors' routing tables in two cases:

1. When a **trigger** happens such as a router or link failure happens.
2. Every N seconds, where N is a configurable parameter.

The Algorithm

Let's look at *how* the distance vector routing algorithm would arrive at the table above.

When a node **x** receives a routing table, `routing_table_y` from a neighbor **y**, the node applies the **Bellman Equation** in order to calculate/update the cost to reach each of **y**'s neighbors. Note that `routing_table_x` is the routing table of the node **x**.

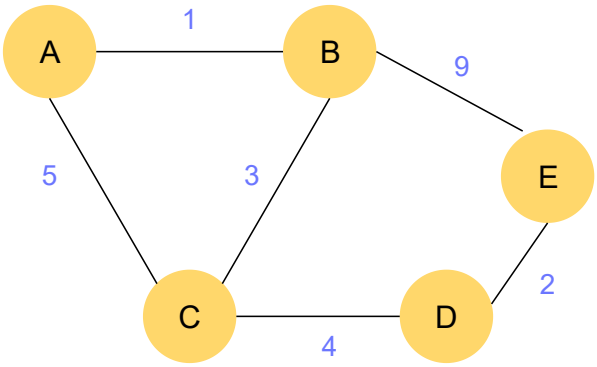
1. The Bellman Equation returns the **minimum** of the current estimated cost to reach a node, and the cost to reach a node *via* **y**.
2. If **x** does not have an estimated cost to reach a certain node, the cost to reach it is entered as the cost to reach the node *via* **y**.

Here's some pseudocode to clarify the picture.

```
for node in routing_table_y:
  if (node in routing_table_x):
   # Existing route, is the new one better?
    if (routing_table_x[node].cost > routing_table_x[y].cost + node.cost): # If current cost
      routing_table_x[node].cost = routing_table_x[y].cost + node.cost # Update
  else:
    # New route
    routing_table_x[node].cost = routing_table_x[y].cost + node.cost # Add entry
```

The estimated cost will finally converge to the optimal cost after a series of these message exchanges. Have a look at the following slides for an example:

Example



Destination	Cost	Next Hop
A	5	A
B	3	B
D	4	D

Routing table of **C**

Destination	Cost	Next Hop
B	1	B
C	5	C

Routing table of **A**

Node **C** receives a routing table from node **A**

Distance Vector Routing: Table Upgrades

1 of 11

That being said, this algorithm has some issues.

Count to Infinity Problem

Consider the following scenario based on the given graph below:

1. Suppose the link between **D** and **E** fails.
2. **D** corrects its routing table such that the cost to reach **E** is infinity ∞.
3. The network will be triggered to exchange routing tables.
4. Suppose **C** gets a chance to advertise its routing table first and sends it over to **D**.
5. Since **C** does not yet know about the link failure between **D** and **E**, its routing table has an entry for a route to **E** with the cost of 6 via **D**.
6. **D** will notice that **C** has a route to **E** and will update its routing table with a route to **E** via **C**. The cost of the route will be the sum of the cost of **D** to

C, and the cost C has to reach **E**: $4 + 6 = 9$.

7. When **C** receives **D**'s routing table, it will notice that **D** has changed its cost to reach **E** from 2 to 9 and will update its table accordingly to $4 + 9 = 13$. Then it advertises it to other neighbors.

8. Things will eventually converge at C with a cost of 12 to E and D will learn that it can reach E at a cost of 16.

9. So, the 'infinity', in this case is not very high. The problem is that this convergence to takes quite a while, and until the routers converge, there is a forwarding loop. Packets at C destined to E would go around in circles until their TTL expires.

There are a few improvements we can look at:

Fix #1: Split Horizon

This count to infinity problem occurs because a node **C** advertises to its neighbor **B** a route that it's learned from the neighbor **B** itself.

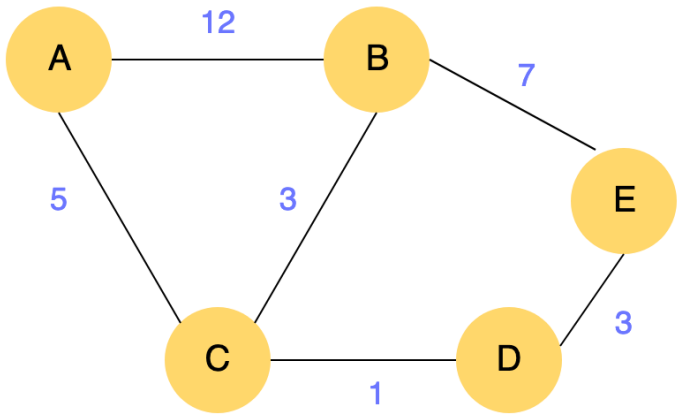
A possible solution to avoid this problem could be to change how a router creates its routing table. Instead of computing one routing table and sending it to all its neighbors, a router could create a routing table that's specific to each neighbor and **only contains the routes that have not been learned via this neighbor**. This technique is called **Split Horizon**.

Fix #2: Split Horizon with Poison Reverse

Another variant of this is called **Split-Horizon with Poison Reverse**. Nodes using this variant advertise a cost of ∞ for the destinations that they reach via the node to which they send their routing tables. Split Horizon with poison reverse converges faster than regular split horizon.

Quick Quiz!

Q What would the initial routing table look like at B for the following example with distance vector routing?



A)

Destination	Count	Next Hop
A	5	A
B	3	B
D	4	D

B)

Destination	Count	Next Hop
A	12	A
C	3	C
E	7	E

C)

Destination	Count	Next Hop

A	5	A
C	3	B
E	4	D

The most popular routing algorithms today use **link state routing**, which we'll look at in the next lesson.