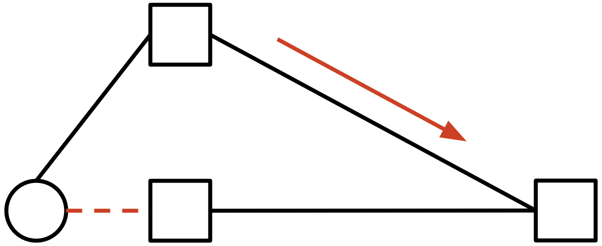


Lectures 5 and 6 (Routing 2)

Distance-Vector Protocols

CS 168, Spring 2025 @ UC Berkeley

Slides credit: Sylvia Ratnasamy, Rob Shakir, Peyrin Kao, Iuniana Oprescu

Distance-Vector Algorithm Sketch

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

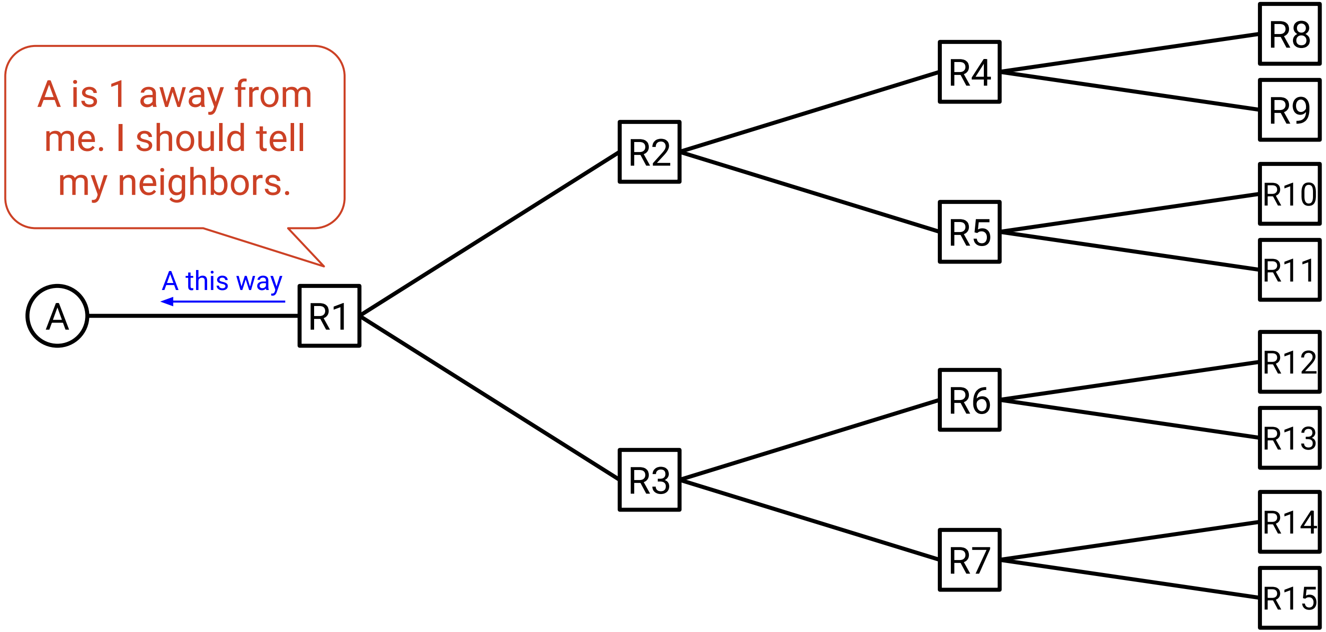
- **Algorithm Sketch**
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

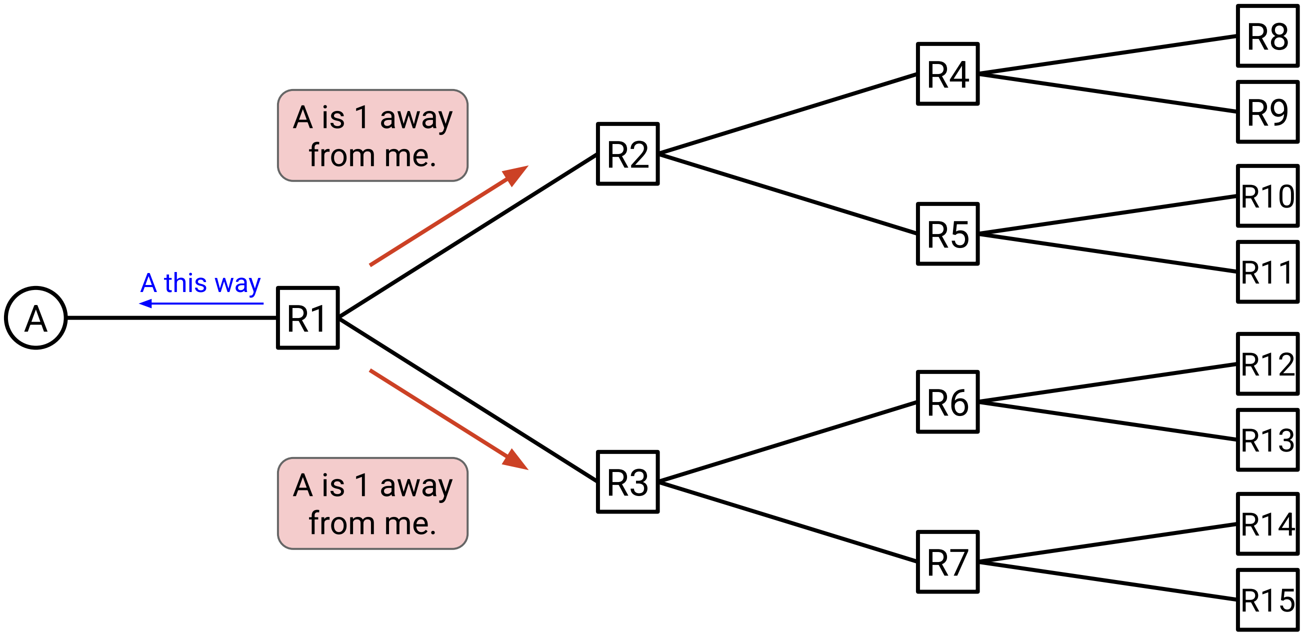
Distance-Vector Algorithm Sketch

One-line algorithm: If you hear about a path to a destination, tell all your neighbors.



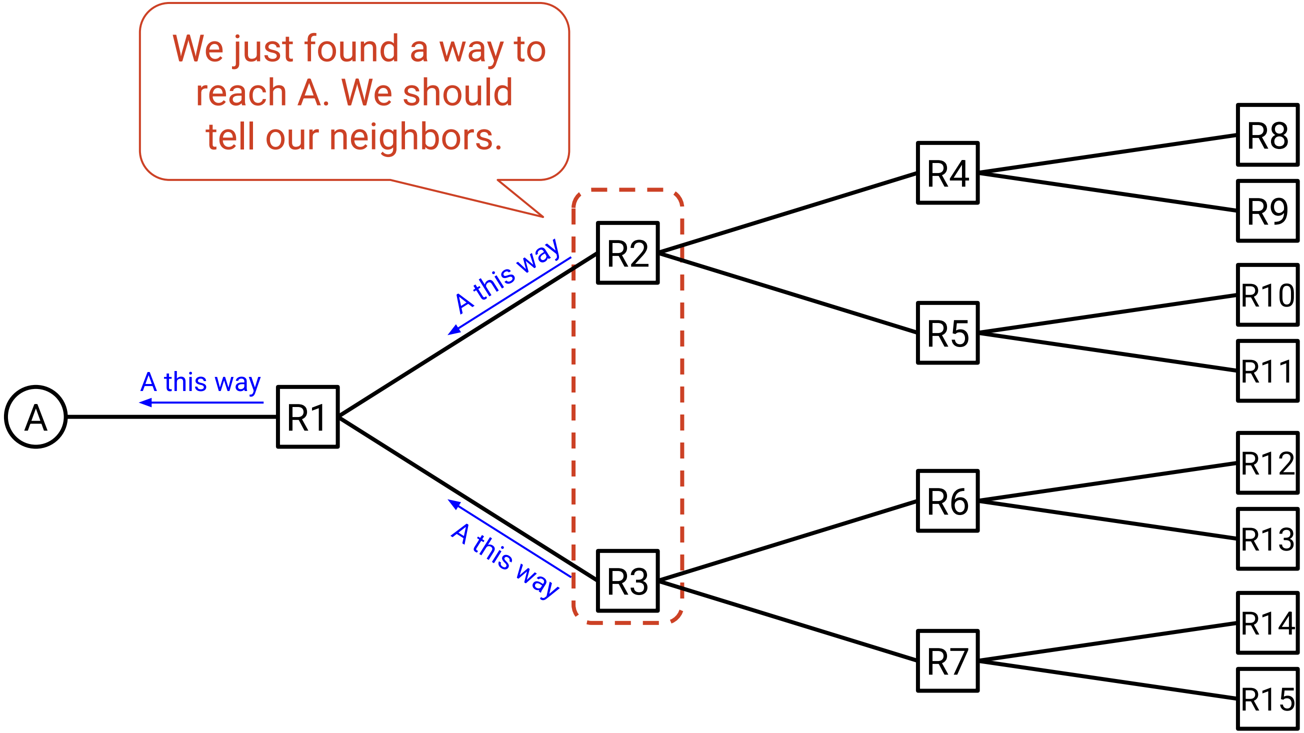
Distance-Vector Algorithm Sketch

One-line algorithm: If you hear about a path to a destination, tell all your neighbors.



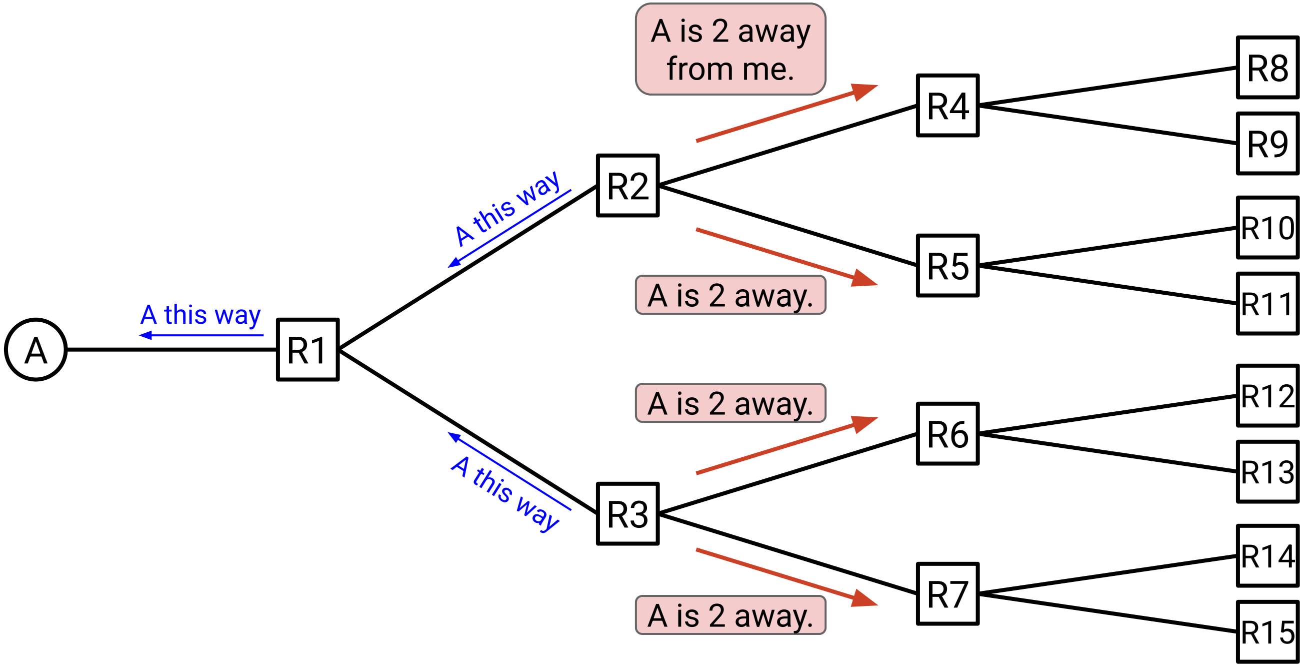
Distance-Vector Algorithm Sketch

One-line algorithm: If you hear about a path to a destination, tell all your neighbors.



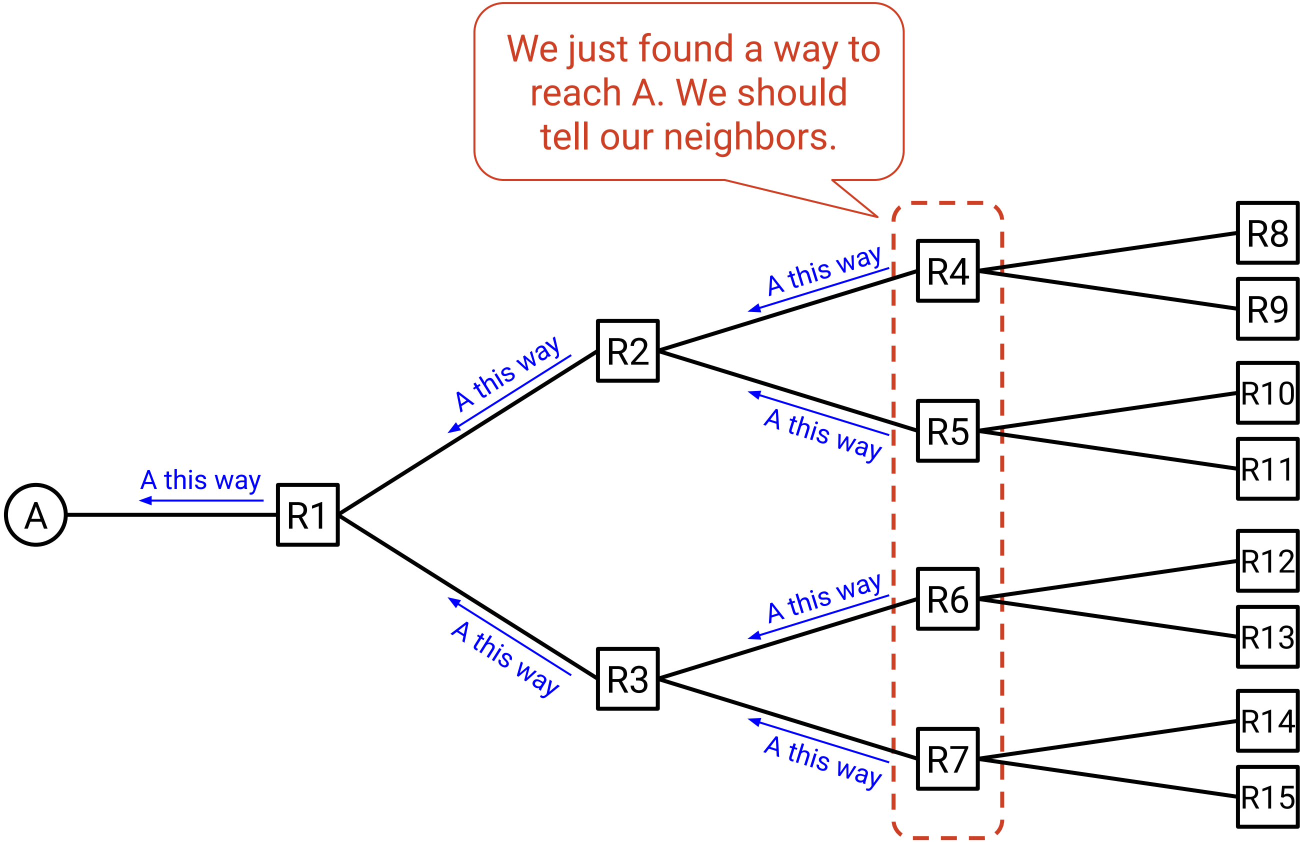
Distance-Vector Algorithm Sketch

One-line algorithm: If you hear about a path to a destination, tell all your neighbors.



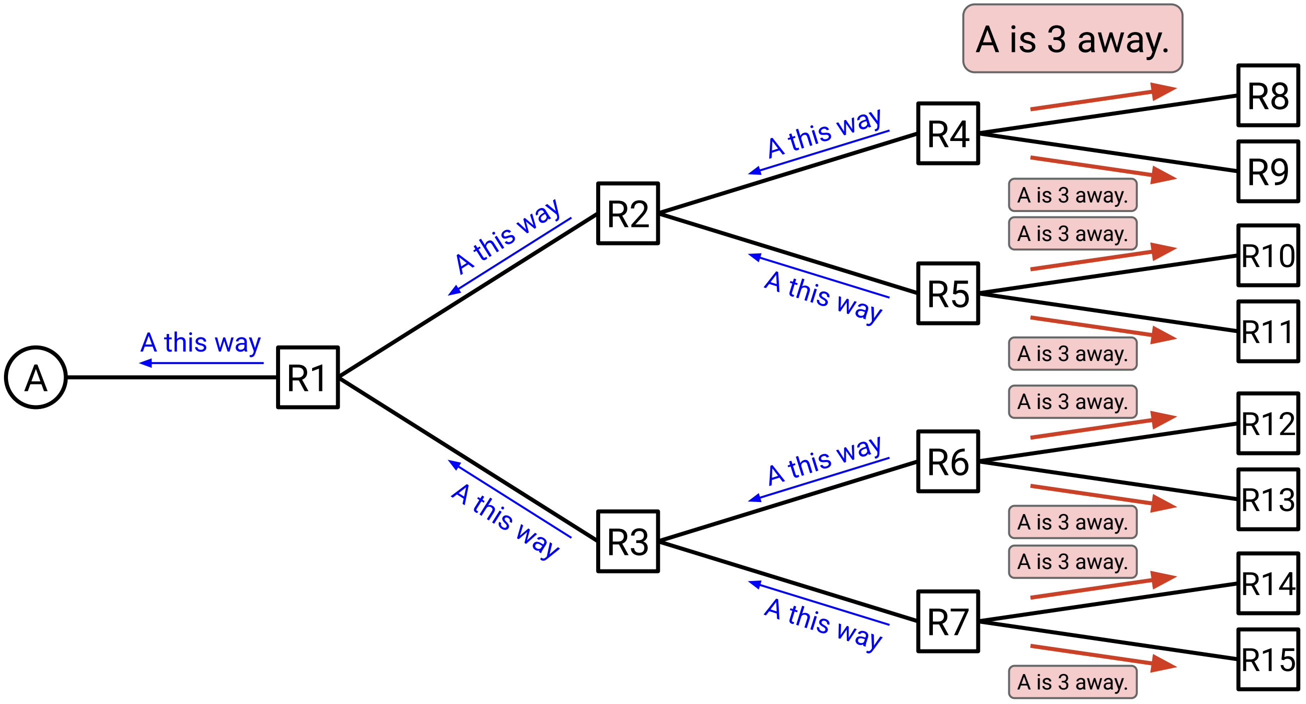
Distance-Vector Algorithm Sketch

One-line algorithm: If you hear about a path to a destination, tell all your neighbors.



Distance-Vector Algorithm Sketch

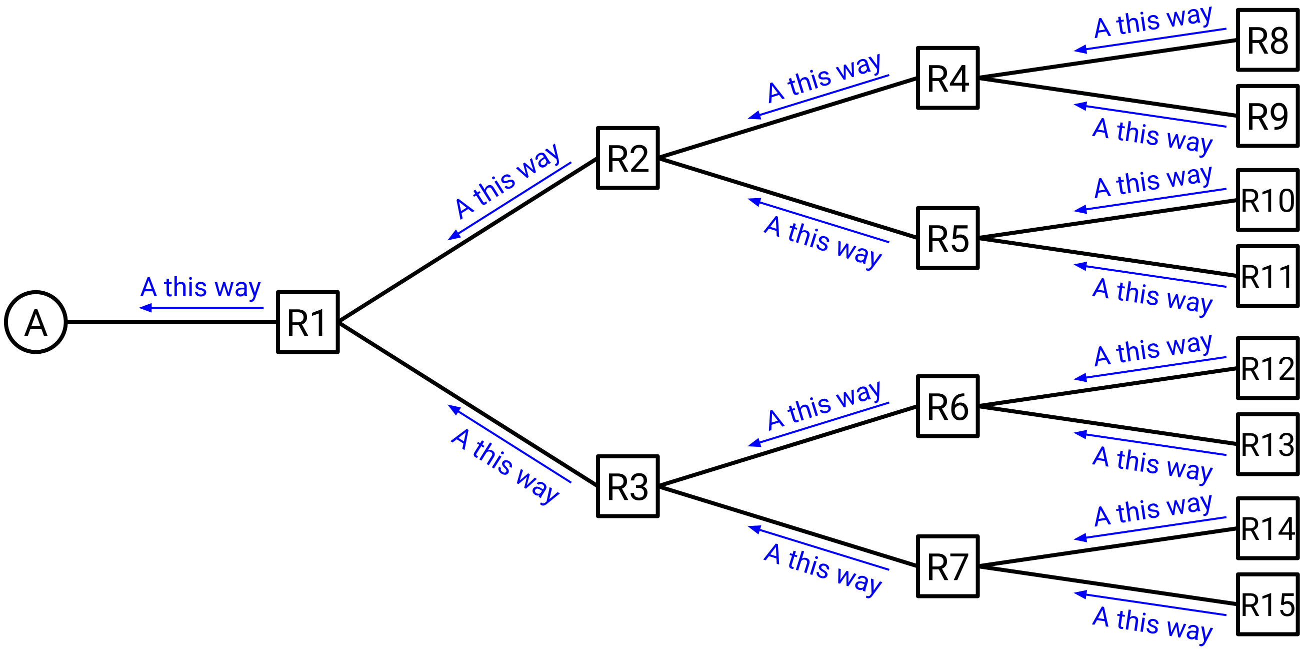
One-line algorithm: If you hear about a path to a destination, tell all your neighbors.



Distance-Vector Algorithm Sketch

One-line algorithm: If you hear about a path to a destination, tell all your neighbors.

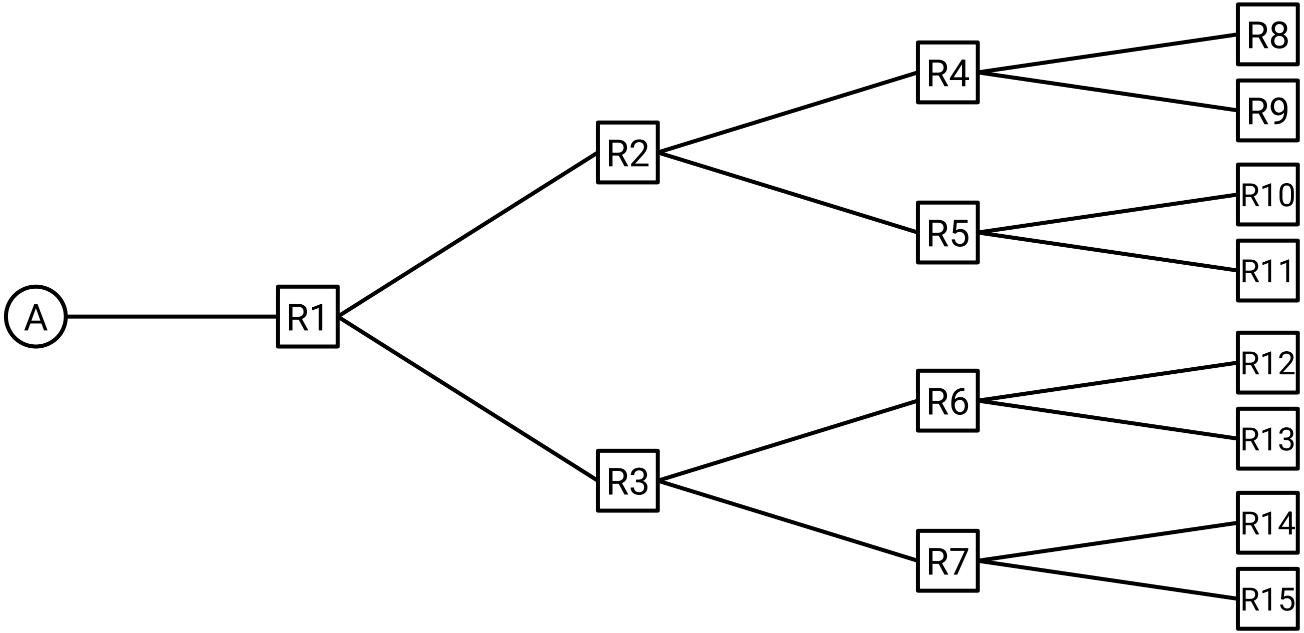
We did it! Everybody knows the next-hop to A now.



Distance-Vector Algorithm Sketch – Routing vs. Forwarding

Routing announcements ("I can reach A") propagated *outward*, away from A.

When **forwarding** packets toward A, packets travel *inward*, toward A.



Distance-Vector Algorithm Sketch – Multiple Destinations

What if there are multiple destinations?

- Run the same path propagation algorithm, once per destination.
- Routers use **forwarding tables** to keep track of the next-hop of each destination.

We'll focus on a single destination for simplicity.

- But the protocol can extend to multiple destinations.

Rule 1: Bellman-Ford Updates

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- **Rule 1: Bellman-Ford Updates**
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

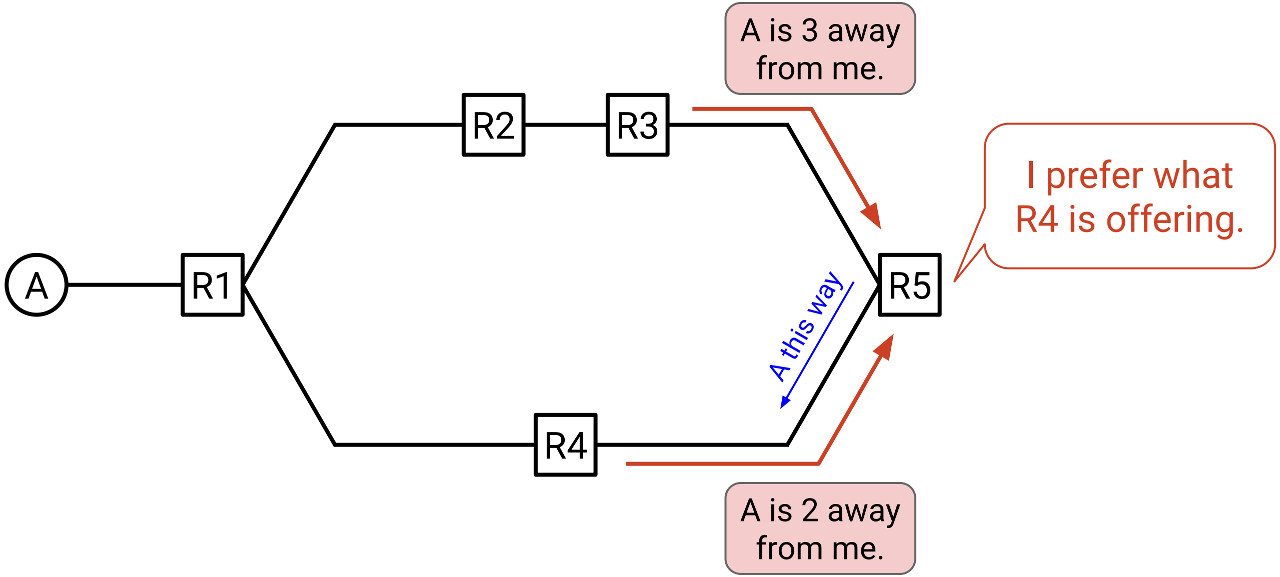
Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

Multiple Paths Advertised

What if you hear about multiple paths to a single destination?

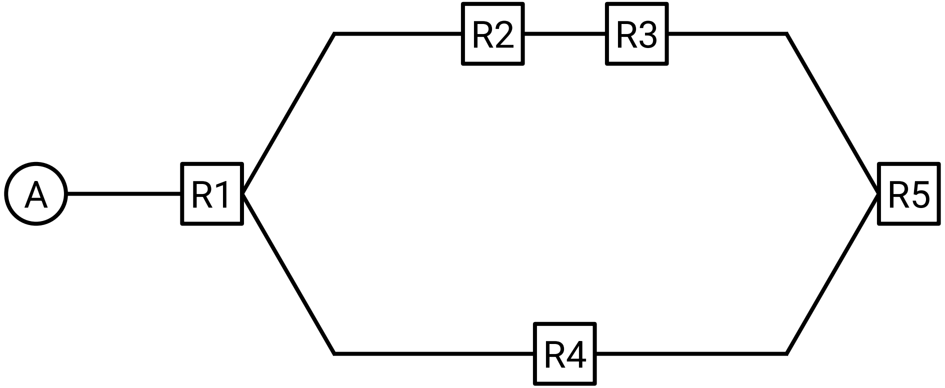
- Accept the shorter path.



Multiple Paths Advertised

What if you hear about multiple paths to a single destination?

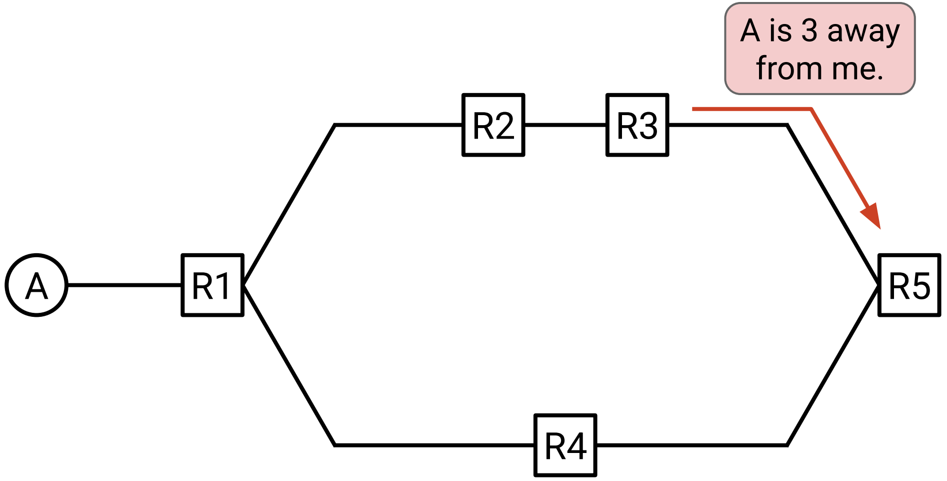
- Accept the shorter path.



Multiple Paths Advertised

You might not hear about both paths simultaneously.

- In the forwarding table, record the best-known cost to a destination.
- If your table doesn't have a path to a destination, accept any path you hear about.

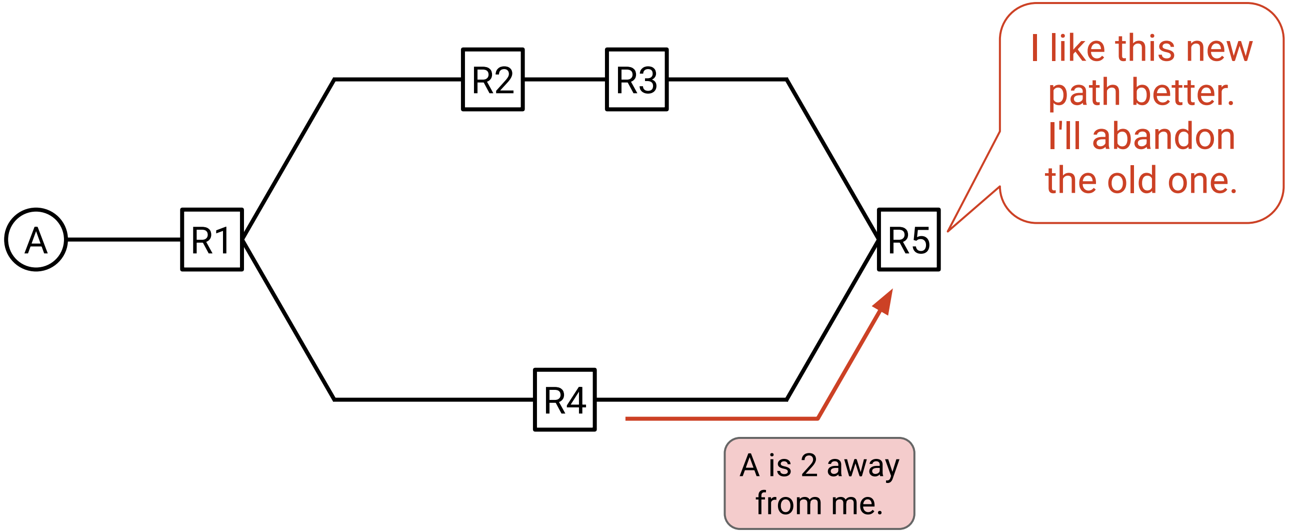


R5's Table		
To:	Via:	Cost:
A	R3	4

Multiple Paths Advertised

You might not hear about both paths simultaneously.

- In the forwarding table, record the best-known cost to a destination.
- If your table doesn't have a path to a destination, accept any path you hear about.
- If you hear about a better path later, update the table (next-hop and cost).



R5's Table		
To:	Via:	Cost:
A	~~R3~~ **R4**	~~4~~ **3**

The Distance-Vector Algorithm So Far

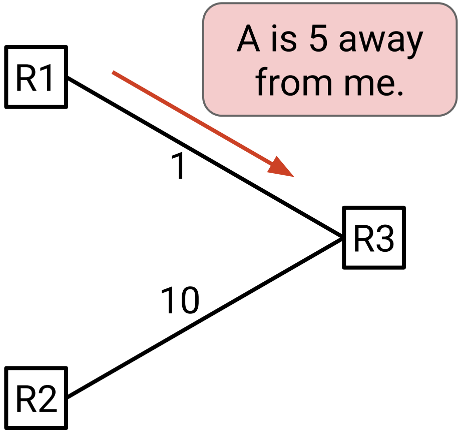
For each destination:

- If you hear about a path to that destination, update table if:
    - The destination isn't in the table.
    - The advertised cost is better than best-known cost.
- Then, tell all your neighbors.

Unequal Costs

Not all link costs are 1.

- When a neighbor advertises a path, the cost via that path is the sum of:
  - Link cost from you to the neighbor.
  - Cost from neighbor to destination (as advertised by neighbor).

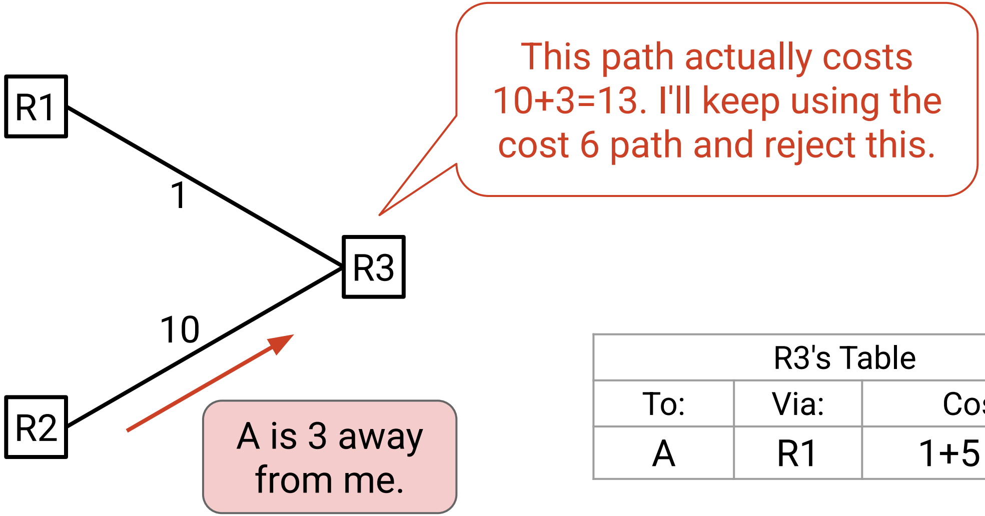


R3's Table		
To:	Via:	Cost:
A	R1	1+5 = 6

Unequal Costs

Not all link costs are 1.

- When a neighbor advertises a path, the cost via that path is the sum of:
  - Link cost from you to the neighbor.
  - Cost from neighbor to destination (as advertised by neighbor).



This path actually costs 10+3=13. I'll keep using the cost 6 path and reject this.

A is 3 away from me.

R3's Table		
To:	Via:	Cost:
A	R1	1+5 = 6

The Distance-Vector Algorithm So Far

For each destination:

- If you hear about a path to that destination, update table if:
  - The destination isn't in the table.
  - Advertised cost + link cost to neighbor < best-known cost. (#1)
- Then, tell all your neighbors.

Distributed Bellman-Ford Algorithm

Careful viewers might have noticed this operation looks familiar.

- "If cost to neighbor + cost from neighbor to destination < best-known cost, accept update."
- This is the relaxation operation in Dijkstra's shortest path algorithm!

Bellman-Ford is another relaxation-based shortest path algorithm.

- Relax every edge repeatedly until we get shortest paths.
- Unlike Dijkstra's, does not require relaxing the edges in any specific order.

Distance-vector algorithms are a *distributed*, *asynchronous* version of Bellman-Ford.

- Distributed: Each router relaxes its own links. No global mastermind.
- Asynchronous: Nobody is syncing when the routers do relaxations.

Distributed Bellman-Ford Algorithm

The centralized Bellman-Ford algorithm for a single destination:

```
def bellman_ford(dst, routers, links):
    distance = {};   nexthop  = {}

    for r in routers:
        distance[r] = INFINITY
        nexthop[r]  = None
    distance[dst] = 0
    for _ in range(len(routers)-1):
        for (r1, r2, linkcost) in links:
            if distance[r1] + linkcost < distance[r2]:
                distance[r2] = distance[r1] + linkcost
                nexthop[r2] = r1

    return distance, nexthop
```

Everyone starts infinity away from the destination, except for the destination itself (0 away).

Bellman-Ford loops through nodes and relaxes repeatedly.

In distance-vector, each router relaxes in parallel, with no order between routers.

The relaxation operation.

Bellman-Ford Demo

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- **Bellman-Ford Demo**
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

The Distance-Vector Algorithm So Far

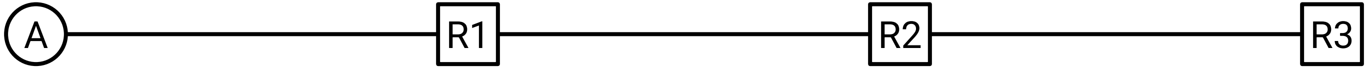
For each destination:

- If you hear an advertisement, update table if:
    - The destination isn't in the table.
    - Advertised cost + link cost to neighbor < best-known cost. (#1)
- Then, advertise to all your neighbors.

Terminology note:

- Sending "I'm R1, and I can reach A with cost 5" is called **announcing** or **advertising** a route.

Bellman-Ford Demo

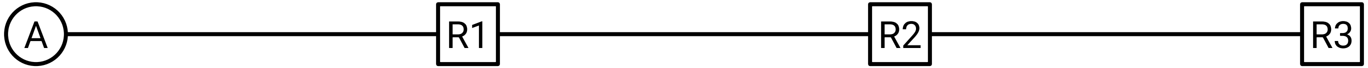


R1's Table		
To:	Via:	Cost:

R2's Table		
To:	Via:	Cost:

R3's Table		
To:	Via:	Cost:

Bellman-Ford Demo



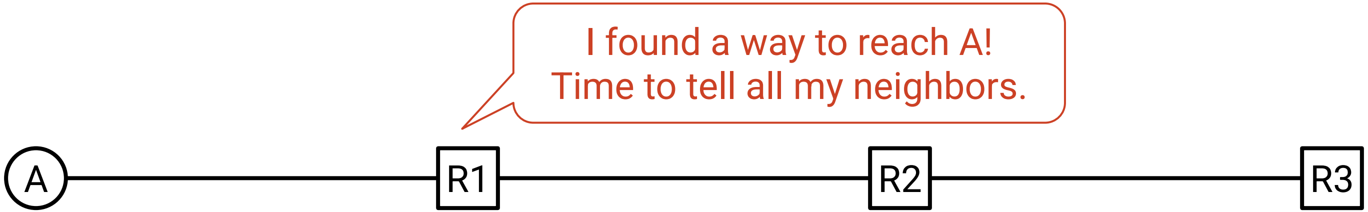
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:

R3's Table		
To:	Via:	Cost:

Static routing: Someone hard-codes R1's table to say it can reach A.

Bellman-Ford Demo

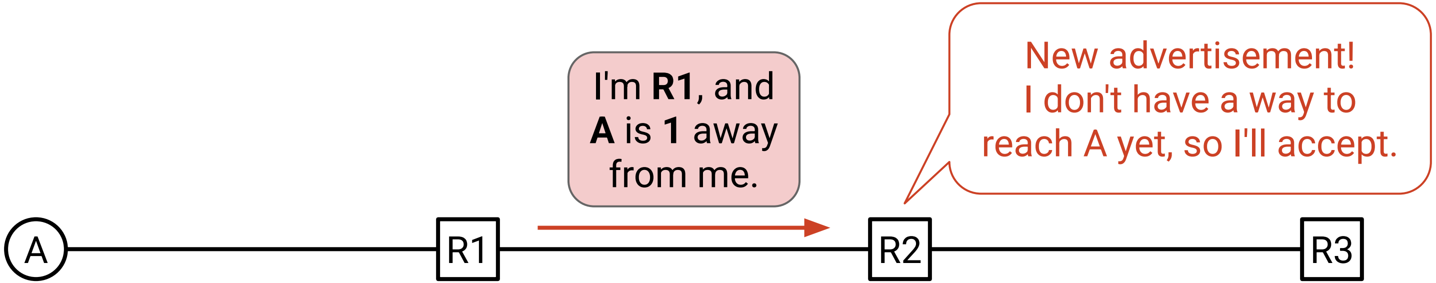


R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:

R3's Table		
To:	Via:	Cost:

Bellman-Ford Demo



I'm **R1**, and **A** is **1** away from me.

New advertisement!
I don't have a way to reach A yet, so I'll accept.

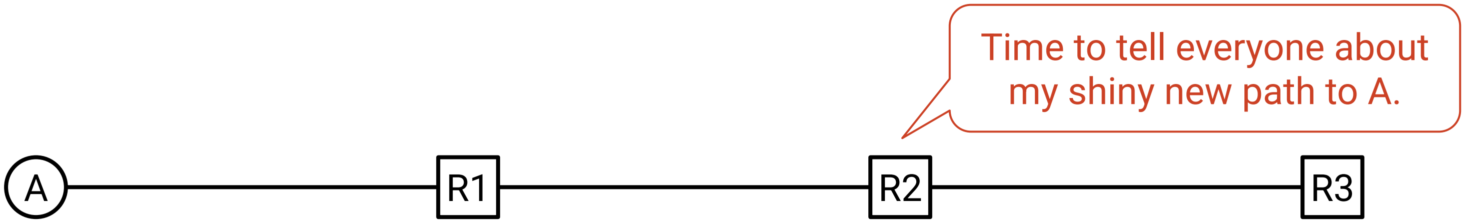
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	**R1**	1+**1**=2

R3's Table		
To:	Via:	Cost:

1 (link cost to R1), plus
1 (advertised cost).

Bellman-Ford Demo



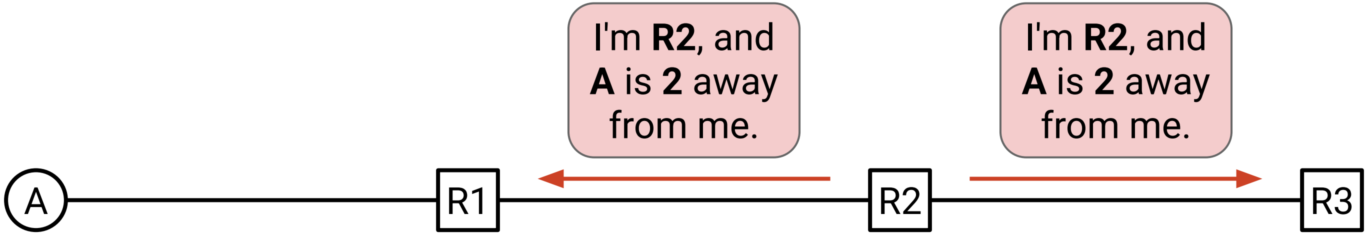
Time to tell everyone about my shiny new path to A.

R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:

Bellman-Ford Demo


I'm **R2**, and **A** is **2** away from me.
I'm **R2**, and **A** is **2** away from me.
A
R1
R2
R3

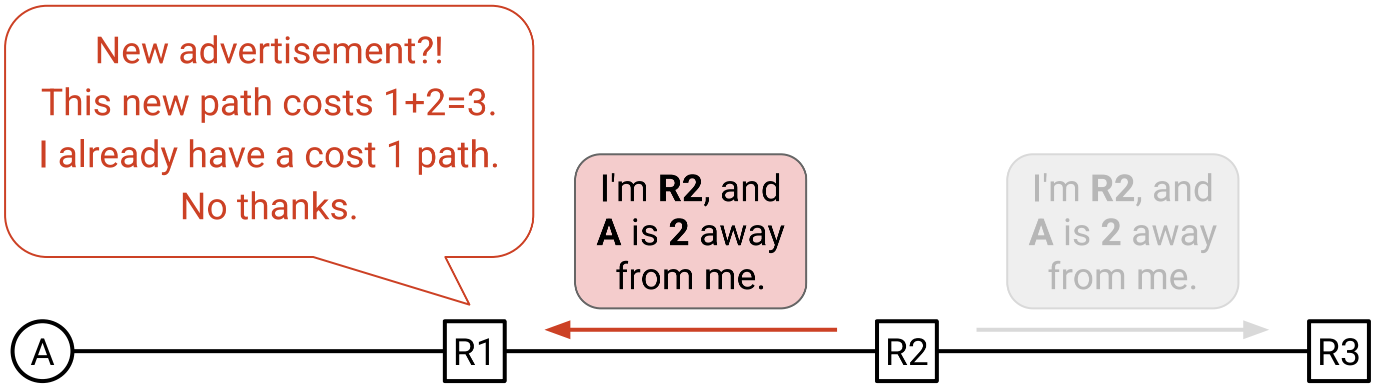
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:

Notice: R2's announcement doesn't include the next-hop.
Nobody else cares *how* R2 reaches A, just that R2 *can* reach A.

Bellman-Ford Demo

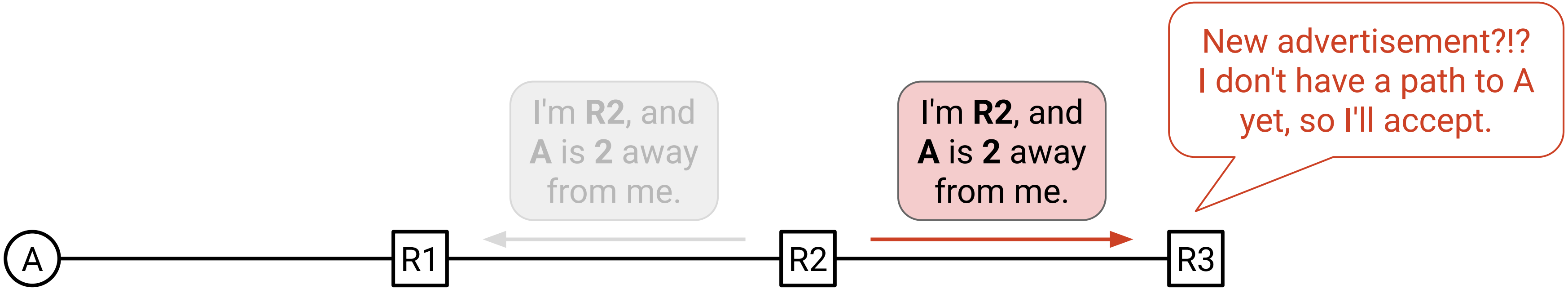

New advertisement?!
This new path costs 1+2=3.
I already have a cost 1 path.
No thanks.
I'm **R2**, and **A** is **2** away from me.
I'm **R2**, and **A** is **2** away from me.
A
R1
R2
R3

R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:

Bellman-Ford Demo



I'm **R2**, and **A** is **2** away from me.

I'm **R2**, and **A** is **2** away from me.

New advertisement?!? I don't have a path to A yet, so I'll accept.

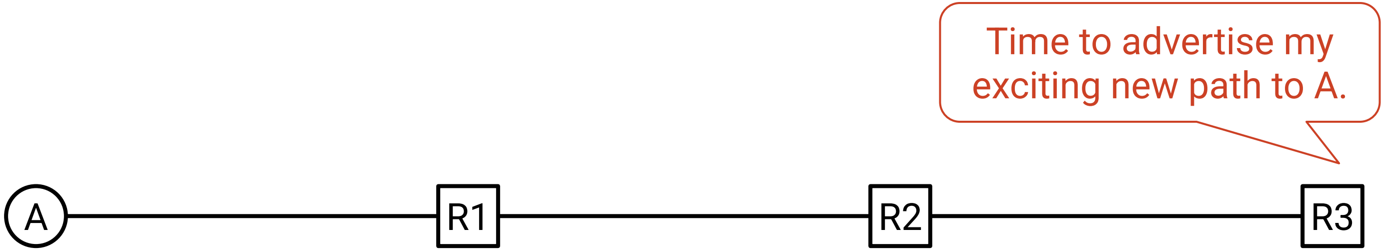
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	**R2**	1+**2**=3

1 (link cost to R2), plus
1 (advertised cost).

Bellman-Ford Demo



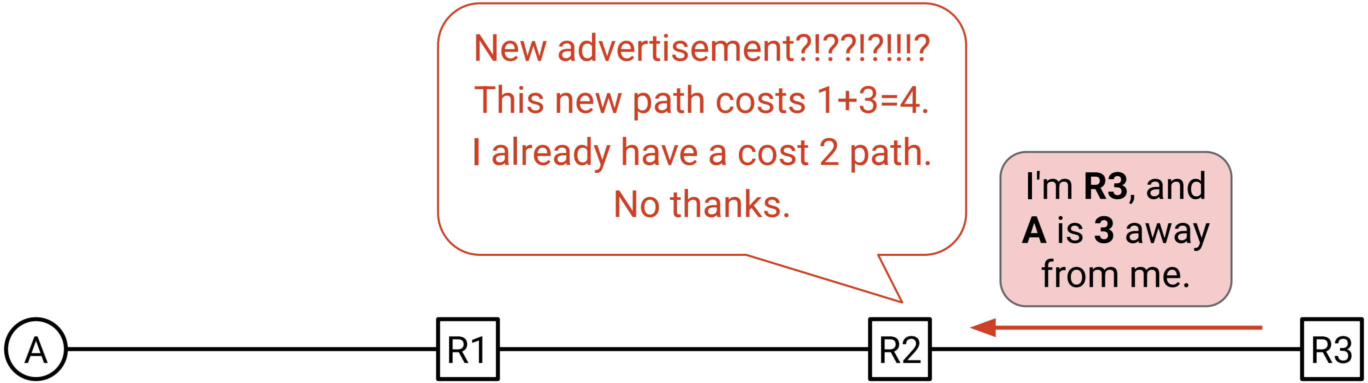
Time to advertise my exciting new path to A.

R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	R2	3

Bellman-Ford Demo


New advertisement?!??!?!!!?
This new path costs 1+3=4.
I already have a cost 2 path.
No thanks.

I'm **R3**, and
A is **3** away
from me.

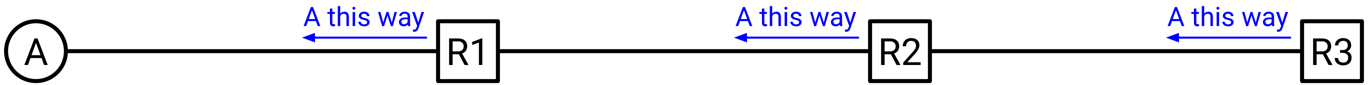
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	R2	3

Bellman-Ford Demo

We did it! Everybody has a way to reach A now.



R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	R2	3

Rule 2: Updates from Next-Hop

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- **Rule 2: Updates From Next-Hop**
- Rule 3: Resending
- Rule 4: Expiring

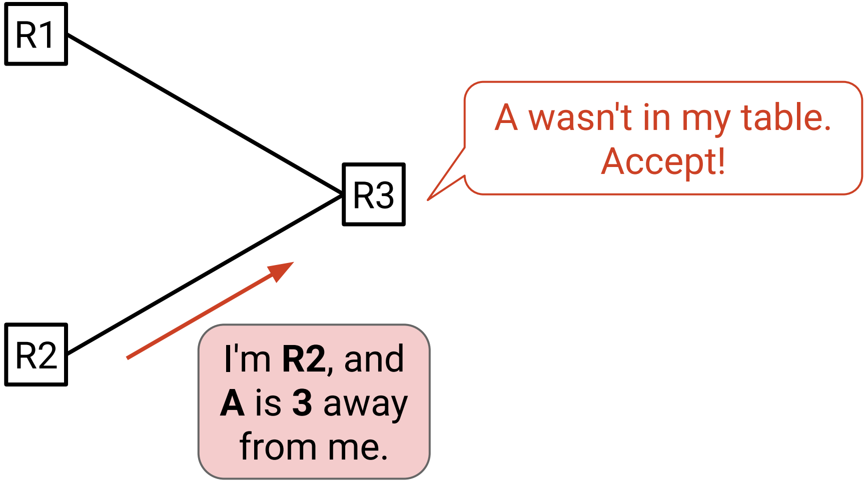
Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

Updates From the Current Next-Hop

Recall our routing challenges: Topology can change.

- So far: We update if we get a better path (or if we didn't have a path before).
- Fix: If our current next hop sends us an announcement, accept it, *even if the path is worse*.
- This lets the next-hop notify us if the topology changed.



A wasn't in my table. Accept!

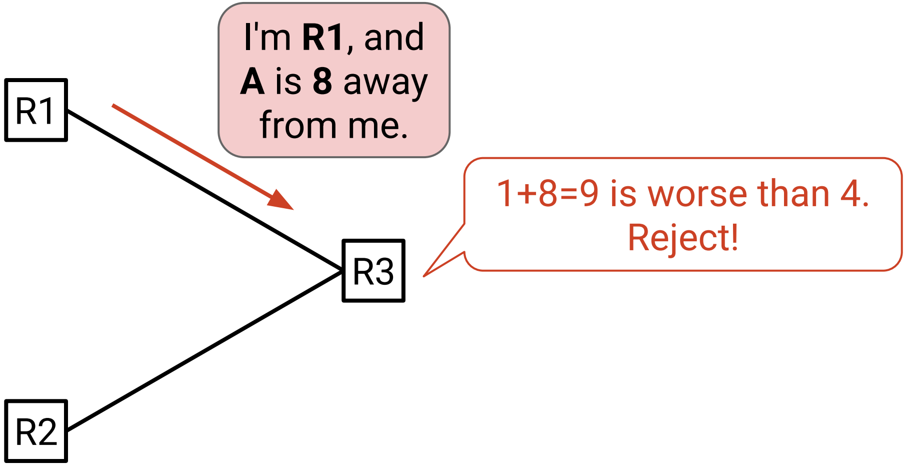
I'm **R2**, and **A** is **3** away from me.

R3's Table		
To:	Via:	Cost:
A	**R2**	1+**3** = 4

Updates From the Current Next-Hop

Recall our routing challenges: Topology can change.

- So far: We update if we get a better path (or if we didn't have a path before).
- Fix: If our current next hop sends us an announcement, accept it, *even if the path is worse*.
- This lets the next-hop notify us if the topology changed.



I'm **R1**, and **A** is **8** away from me.

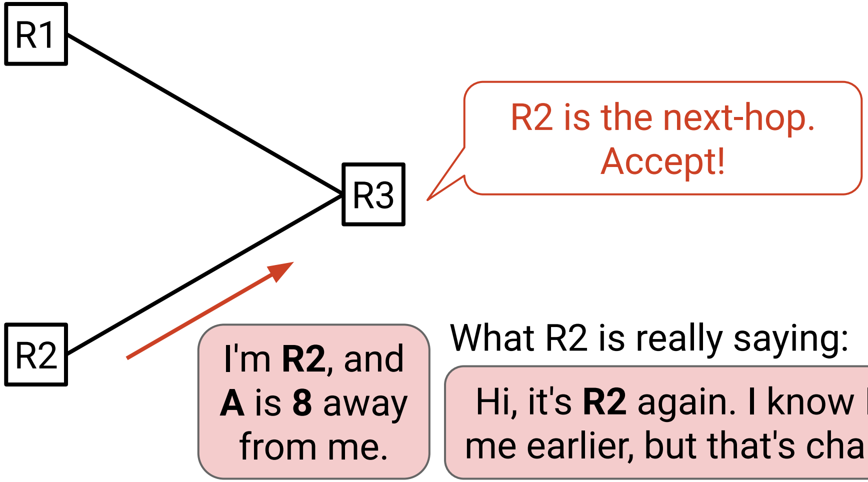
1+8=9 is worse than 4. Reject!

R3's Table		
To:	Via:	Cost:
A	R2	4

Updates From the Current Next-Hop

Recall our routing challenges: Topology can change.

- So far: We update if we get a better path (or if we didn't have a path before).
- Fix: If our current next hop sends us an announcement, accept it, *even if the path is worse*.
- This lets the next-hop notify us if the topology changed.



R2 is the next-hop. Accept!

I'm **R2**, and **A** is **8** away from me.

R3's Table		
To:	Via:	Cost:
A	**R2**	1+**8**=9

What R2 is really saying:

Hi, it's **R2** again. I know I said A is 3 away from me earlier, but that's changed. Now **A** is **8** away.

Convergence

If the network never changes:

- After running this protocol for some time, it will **converge**.
- Everyone's forwarding table has the least-cost next hop.
- All future announcements will be rejected.

If a change happens (e.g. a link goes down):

- Some new announcements are sent.
- Some forwarding tables are updated.
- Eventually, we converge again to the new routing state.

The network topology is constantly changing, so routers run the protocol *indefinitely*.

- **Steady-state** occurs when the network has converged.
- In steady-state, everything stays the same until the next topology change.

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table if:
    - The destination isn't in the table.
    - Advertised cost + link cost to neighbor < best-known cost. (#1)
    - The advertisement is from current next-hop. (#2)
- Then, advertise to all your neighbors.

Rule 3: Resending

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- **Rule 3: Resending**
- Rule 4: Expiring

Distance-Vector Enhancements

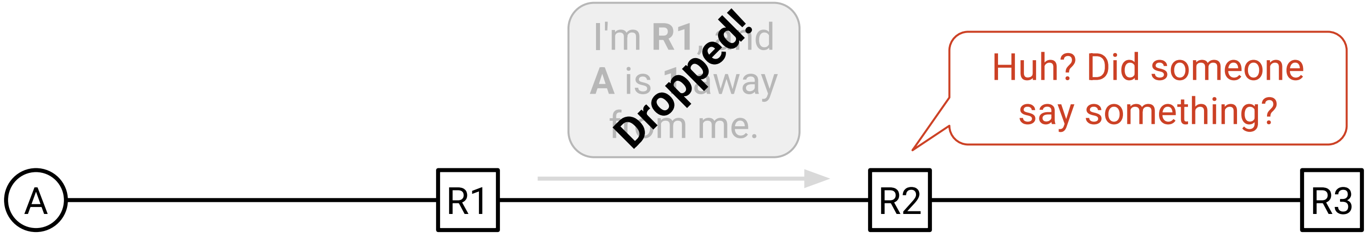
- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

Ensuring Reliability

Recall our routing challenges: Packets can get dropped.

Solution: Resend advertisements every *X* seconds.

- *X* is the "advertisement interval."
- This should work eventually, assuming the link is functional (>0% delivery rate).


I'm **R1**, and **A** is 1 away from me.
Dropped!
Huh? Did someone say something?
A
R1
R2
R3

R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:

R3's Table		
To:	Via:	Cost:

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table if:
    - The destination isn't in the table.
    - Advertised cost + link cost to neighbor < best-known cost. (#1)
    - The advertisement is from current next-hop. (#2)
- Advertise to all your neighbors when the table updates, and periodically. (#3)

Rule 4: Expiring

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- **Rule 4: Expiring**

Distance-Vector Enhancements

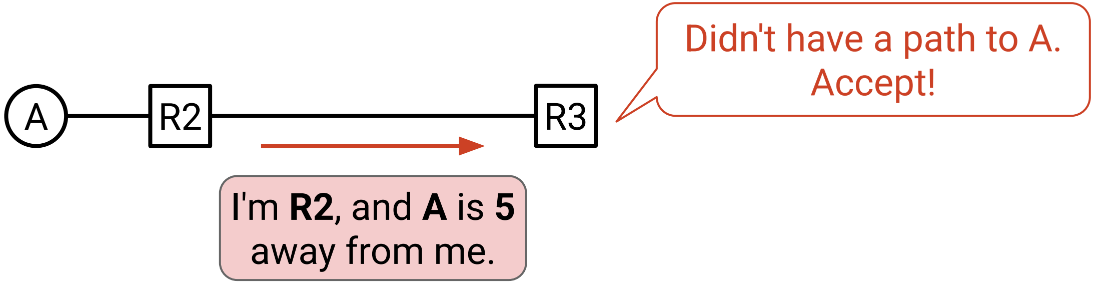
- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

Handling Failures

Recall our routing challenges: Links and routers can fail.

Solution: Each route has a finite **time to live (TTL)**.

- Periodic advertisements help us confirm that a route still exists.
  - When we get an advertisement, reset ("recharge") the TTL.
- If a link goes down, we stop getting periodic updates, and the TTL will expire.
  - If the TTL expires, delete the entry from the table.


A
R2
R3
Didn't have a path to A. Accept!
I'm **R2**, and **A** is **5** away from me.

t = 0

R3's Table			
To:	Via:	Cost:	TTL:
A	**R2**	6	11

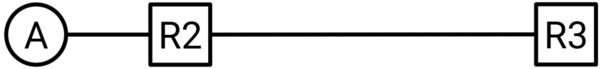
Need another confirmation of this route in the next 11 seconds.

Handling Failures

Recall our routing challenges: Links and routers can fail.

Solution: Each route has a finite **time to live (TTL)**.

- Periodic advertisements help us confirm that a route still exists.
  - When we get an advertisement, reset ("recharge") the TTL.
- If a link goes down, we stop getting periodic updates, and the TTL will expire.
  - If the TTL expires, delete the entry from the table.



t = **4**

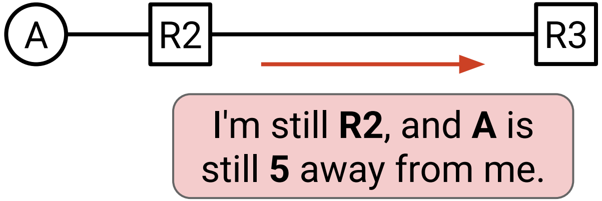
R3's Table			
To:	Via:	Cost:	TTL:
A	R2	6	**7**

Handling Failures

Recall our routing challenges: Links and routers can fail.

Solution: Each route has a finite **time to live (TTL)**.

- Periodic advertisements help us confirm that a route still exists.
    - When we get an advertisement, reset ("recharge") the TTL.
- If a link goes down, we stop getting periodic updates, and the TTL will expire.
    - If the TTL expires, delete the entry from the table.



I'm still **R2**, and **A** is still **5** away from me.

t = **5**

R3's Table

To:	Via:	Cost:	TTL:
A	R2	6	**11**

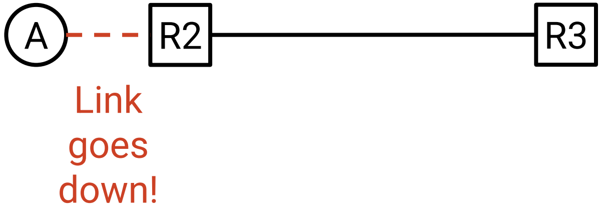
We got a confirmation!
Reset TTL back to 11.

Handling Failures

Recall our routing challenges: Links and routers can fail.

Solution: Each route has a finite **time to live (TTL)**.

- Periodic advertisements help us confirm that a route still exists.
  - When we get an advertisement, reset ("recharge") the TTL.
- If a link goes down, we stop getting periodic updates, and the TTL will expire.
  - If the TTL expires, delete the entry from the table.



t = **15**

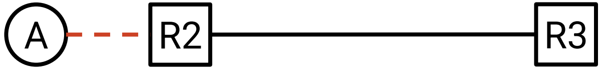
R3's Table			
To:	Via:	Cost:	TTL:
A	R2	6	**1**

Handling Failures

Recall our routing challenges: Links and routers can fail.

Solution: Each route has a finite **time to live (TTL)**.

- Periodic advertisements help us confirm that a route still exists.
  - When we get an advertisement, reset ("recharge") the TTL.
- If a link goes down, we stop getting periodic updates, and the TTL will expire.
  - If the TTL expires, delete the entry from the table.



Link goes down!

t = **16**

R3's Table			
To:	Via:	Cost:	TTL:
~~A~~	~~R2~~	~~6~~	~~**0**~~

Timeout! Delete expired entry.

Timers

Routers maintain multiple timers:

- Advertisement interval: How long before we advertise routes to neighbors.
  - Usually one timer for all entries in the table.
- TTL: How long before we expire a route.
  - Each table entry has its own TTL.

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table and reset TTL if:
  - The destination isn't in the table.
  - Advertised cost + link cost to neighbor < best-known cost. (#1)
  - The advertisement is from current next-hop. (#2)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
- If a table entry expires, delete it. (#4)

This is a mostly-functional protocol now.

Let's add some optimizations for faster convergence.

Rule 5: Poison Expired Routes

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

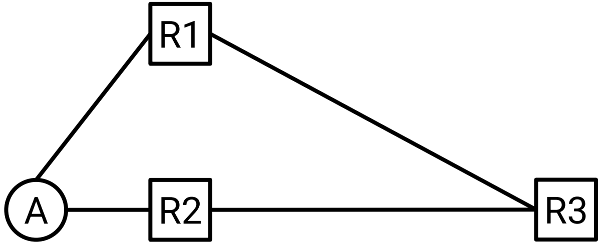
Distance-Vector Enhancements

- **Rule 5: Poison Expired Routes**
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

Route Expiry is Slow

Waiting for routes to expire is slow. Let's watch the demo again.

- R1 is offering a new path to A, but R3 has to wait for the old busted route to expire before accepting the new path.



Assume that by t=3, R3 knows a route to A.

t = **5**

R3's Table			
To:	Via:	Cost:	TTL:
A	R2	2	**11**

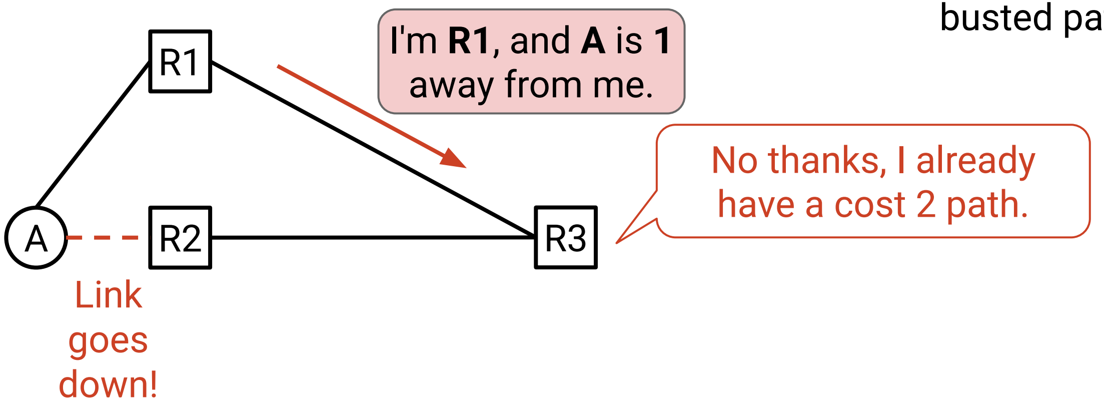
Route Expiry is Slow

Waiting for routes to expire is slow. Let's watch the demo again.

- R1 is offering a new path to A, but R3 has to wait for the old busted route to expire before accepting the new path.

Pause right here.

- At this point, we know the path via R2 is busted.
- But R3 won't know until the timeout 10s later.
- If R3 knew now, it could accept the new path. Instead, R3 rejects the new path, thinking the busted path is still valid.



R1
A
R2
R3
I'm **R1**, and **A** is **1** away from me.
No thanks, I already have a cost 2 path.
Link goes down!

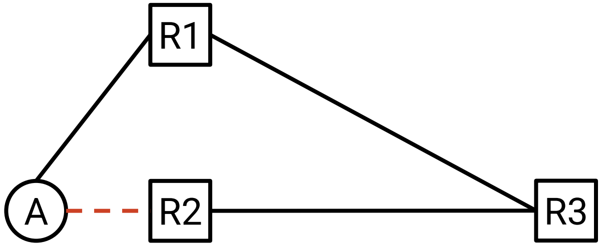
t = **6**

R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	**10**

Route Expiry is Slow

Waiting for routes to expire is slow. Let's watch the demo again.

- R1 is offering a new path to A, but R3 has to wait for the old busted route to expire before accepting the new path.



t = **10**

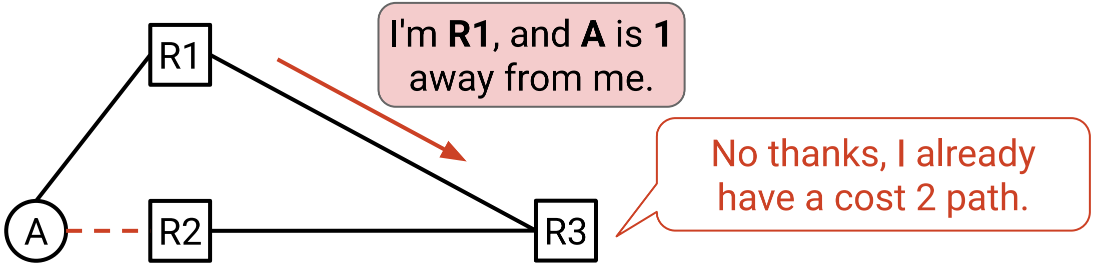
R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	**6**

Route Expiry is Slow

Waiting for routes to expire is slow. Let's watch the demo again.

- R1 is offering a new path to A, but R3 has to wait for the old busted route to expire before accepting the new path.

Again, R3 is forced to reject this new path, because it's still waiting for the busted path to time out.



I'm **R1**, and **A** is **1** away from me.

No thanks, I already have a cost 2 path.

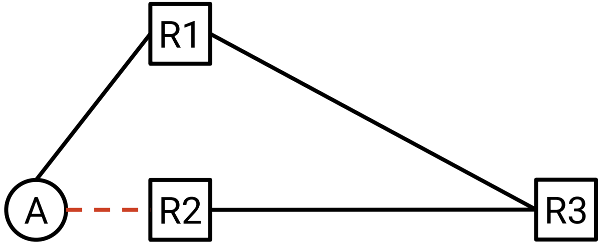
t = **11**

R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	**5**

Route Expiry is Slow

Waiting for routes to expire is slow. Let's watch the demo again.

- R1 is offering a new path to A, but R3 has to wait for the old busted route to expire before accepting the new path.



t = **15**

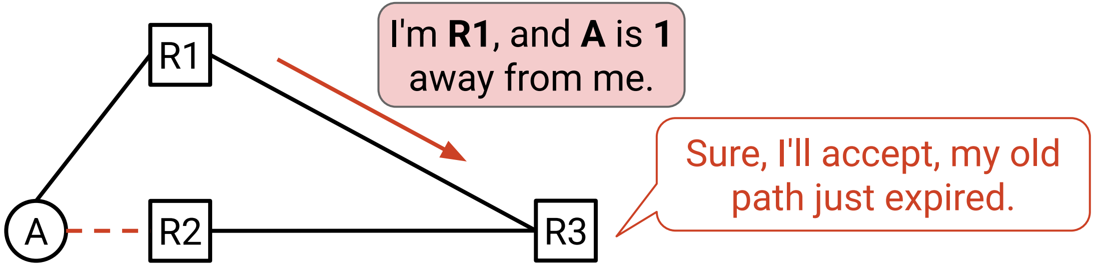
R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	**1**

Route Expiry is Slow

Waiting for routes to expire is slow. Let's watch the demo again.

- R1 is offering a new path to A, but R3 has to wait for the old busted route to expire before accepting the new path.

Can we alert R3 of the failure *sooner*, so it can delete the old busted path earlier (and start accepting new paths)?



I'm **R1**, and **A** is **1** away from me.

Sure, I'll accept, my old path just expired.

t = **16**

R3's Table			
To:	Via:	Cost:	TTL:
~~R2~~	~~A~~	~~2~~	~~**0**~~

Timeout! Delete expired entry.

Poison for Fast Route Expiry

Waiting for routes to expire is slow.

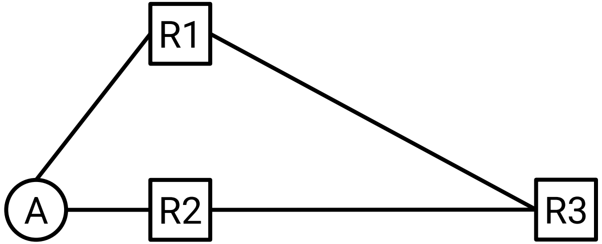
- You keep a busted path in the forwarding table for a long time.
  - Packets might get lost during this time.
  - You might advertise that busted route to other people.
- You might reject new paths, thinking the busted path is still valid.
  - Could have converged on a better path earlier.
- Key problem: When something fails, nobody's reporting it.

Solution: **Poison**.

- Explicitly advertise that a path is busted.
- A path with cost infinity represents a busted path.
- This path propagates just like any other path.
  - Routers accept the poison path to invalidate the route.
- Can be much faster than waiting for timeouts!

Poison for Fast Route Expiry

Poison lets us detect busted routes faster. Let's watch the demo again.



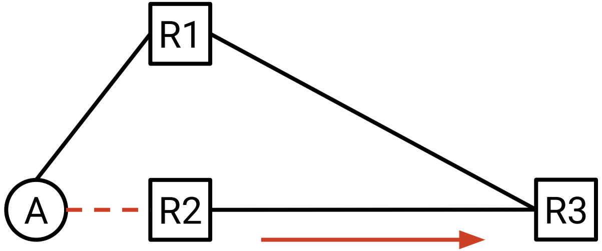
Assume that by t=3, R3 knows a route to A.

t = **5**

R3's Table			
To:	Via:	Cost:	TTL:
A	R2	2	**11**

Poison for Fast Route Expiry

Poison lets us detect busted routes faster. Let's watch the demo again.



Link goes down!

I'm **R2**, and **A** is ∞ away from me.

Looks like the path via R2 is busted.

t = **6**

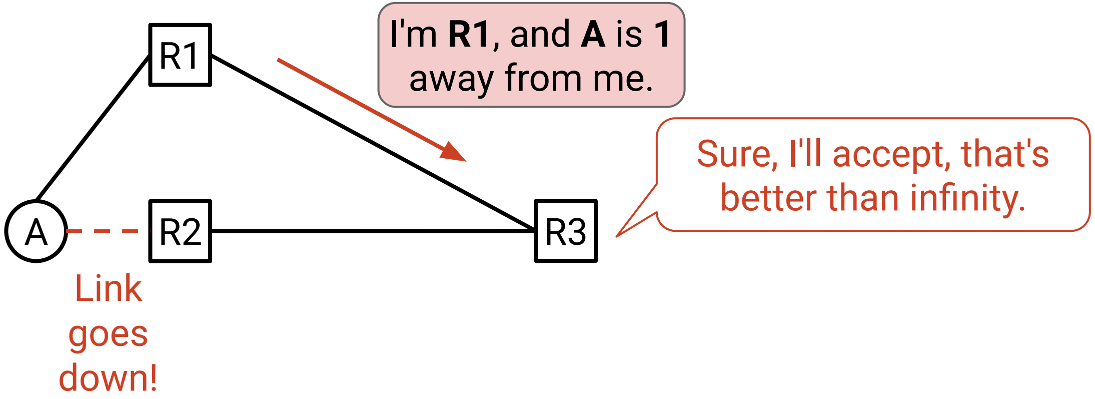
R3's Table			
To:	Via:	Cost:	TTL:
A	**R2**	**∞**	**16**

R3 updates the table to indicate the path is busted.
TTL recharges, just like any other update.

Poison for Fast Route Expiry

Poison lets us detect busted routes faster. Let's watch the demo again.

R3 was able to accept the new route way earlier!
t=6 with poison, t=16 without poison.


R1
A
R2
R3
I'm **R1**, and **A** is **1** away from me.
Sure, I'll accept, that's better than infinity.
Link goes down!

t = **6**

R3's Table			
To:	Via:	Cost:	TTL:
A	**R1**	**2**	**16**

Accepting and Advertising Poison

Where does poison come from?

- One of your routes times out.
- You notice a local failure, e.g. one of your links goes down.

When one of those occurs:

- Poison the entry: Set cost to infinity, reset TTL.
- Advertise the poison to your neighbors.

Accepting and Advertising Poison

When you get a poison advertisement from the current next-hop:

- Accept it, even if you have a better path.
  - Because the next-hop is telling you that the route no longer exists.
  - Similar to Rule #2: accept worse paths from current next-hop.

When you update the table with a poison route:

- Reset the TTL, just like any other table update.
- Advertise the poison to your neighbors, so they also know about the busted route.

Don't forward packets along a poisoned route.

To:	Via:	Cost:
A	R1	∞

← Don't forward to R1.

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table and reset TTL if:
  - The destination isn't in the table.
  - Advertised cost + link cost to neighbor < best-known cost. (#1)
  - The advertisement is from current next-hop. (#2)

    Includes poison advertisements. (#5)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
- If a table entry expires, make the entry poison and advertise it. (#4, #5)

Rule 6A: Split Horizon

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

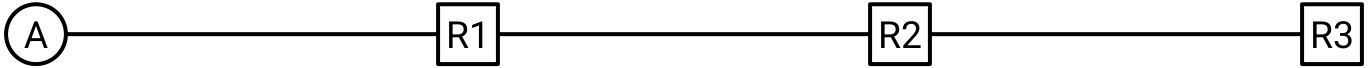
- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- **Rule 6A: Split Horizon**
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- Eventful Updates

Split Horizon – The Problem

We ran the algorithm for some time, and we converged to this steady-state.
All subsequent advertisements will be rejected.

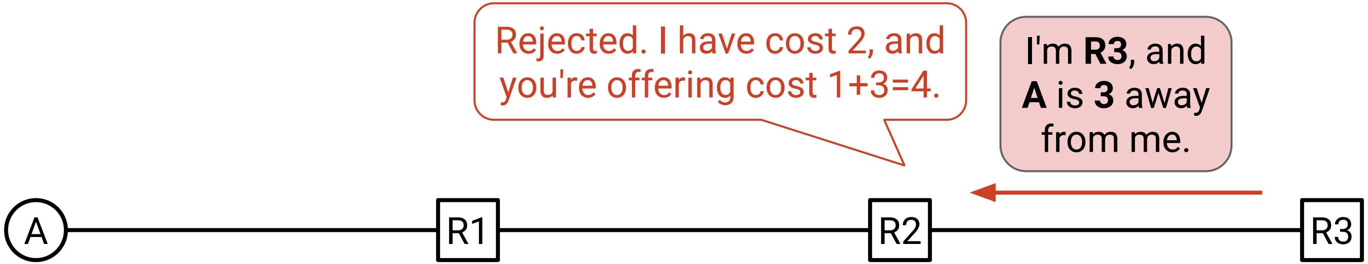


R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	R2	3

Split Horizon – The Problem



Rejected. I have cost 2, and you're offering cost 1+3=4.

I'm **R3**, and **A** is **3** away from me.

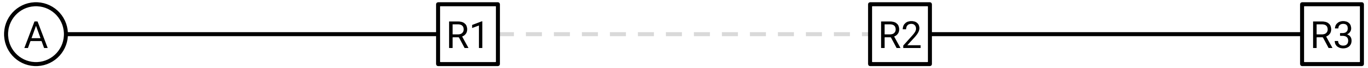
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	R2	3

Split Horizon – The Problem

A link goes down, and R2's entry expires (no more updates from R1).
What happens now?

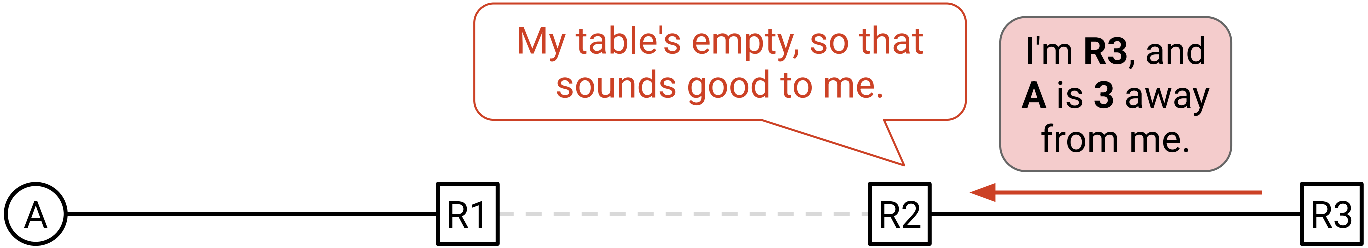


R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:

R3's Table		
To:	Via:	Cost:
A	R2	3

Split Horizon – The Problem



My table's empty, so that sounds good to me.

I'm **R3**, and **A** is **3** away from me.

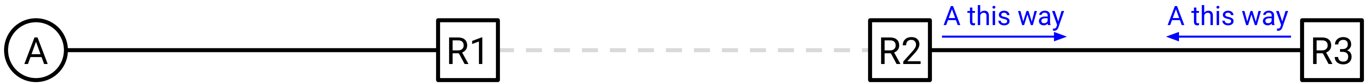
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R3	4

R3's Table		
To:	Via:	Cost:
A	R2	3

Split Horizon – The Problem

We made a routing loop!



R1's Table		
To:	Via:	Cost:
A	Direct	1

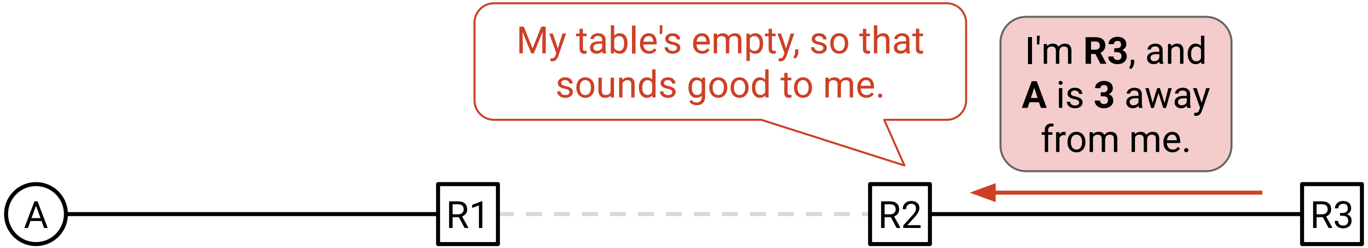
R2's Table		
To:	Via:	Cost:
A	R3	4

R3's Table		
To:	Via:	Cost:
A	R2	3

Split Horizon – The Problem

Problem ("me" = R2):

- I gave R3 a path via me, and R3 accepted.
- Then, R3 turned around and gave me that same path.
- I'm being offered a path that goes through myself!
- Normally, I would never accept, because a path with a loop is longer.
- But if I lost my earlier route, I might accept and create a loop.


My table's empty, so that sounds good to me.
I'm **R3**, and **A** is **3** away from me.
A
R1
R2
R3

R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R3	4

R3's Table		
To:	Via:	Cost:
A	R2	3

Split Horizon – The Problem

The split horizon problem: When I give someone a path, they advertise it back to me.

- Path goes from me → them → me.
- Path with extra loop is always longer, so I'd never accept.
- But if I lost my earlier routes, I might accept.
- I might not realize the path is going through me.

Solution: Don't advertise a path back to the person who gave it to you.

- A advertises a route to B.
- B can advertise that route to everybody *except A*.
- More generally: Don't advertise paths back to the next-hop (the person who gave you the path).

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table and reset TTL if:
  - The destination isn't in the table.
  - Advertised cost + link cost to neighbor < best-known cost. (#1)
  - The advertisement is from current next-hop. (#2)
    Includes poison advertisements. (#5)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
  - But don't advertise back to the next-hop. (#6A)
- If a table entry expires, make the entry poison and advertise it. (#4, #5)

Rule 6B: Poison Reverse

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- **Rule 6B: Poison Reverse**
- Rule 7: Count To Infinity
- Eventful Updates

Poison Reverse

Split horizon: If R1 gives me a route, don't advertise it to R1.

- Don't tell R1 anything.

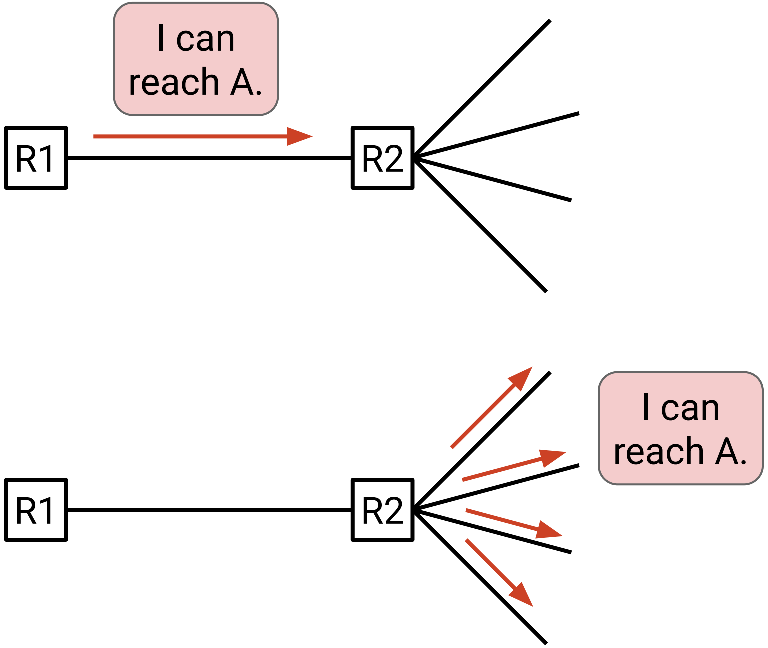
Poison reverse: If R1 gives me a route, advertise poison back to R1.

- Explicitly tell R1: "Do not forward packets to me."

Poison reverse is an alternative way to avoid routing loops.

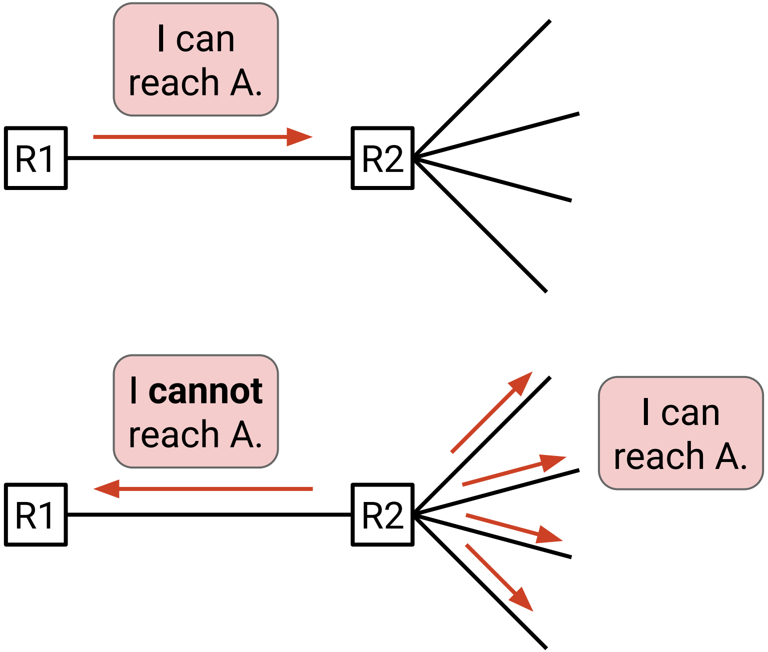
Poison Reverse vs. Split Horizon

Split Horizon:



Don't advertise anything back to R1.

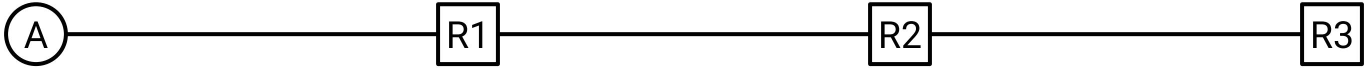
Poison Reverse:



Explicitly advertise poison back to R1.

Poison Reverse

Let's watch the demo again, but with poison reverse this time.
As before, we first reach steady state.



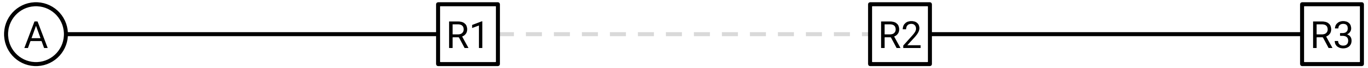
R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	R1	2

R3's Table		
To:	Via:	Cost:
A	R2	3

Poison Reverse

A link goes down, and R2's entry expires (no more updates from R1).
What happens now?



R1's Table		
To:	Via:	Cost:
A	Direct	1

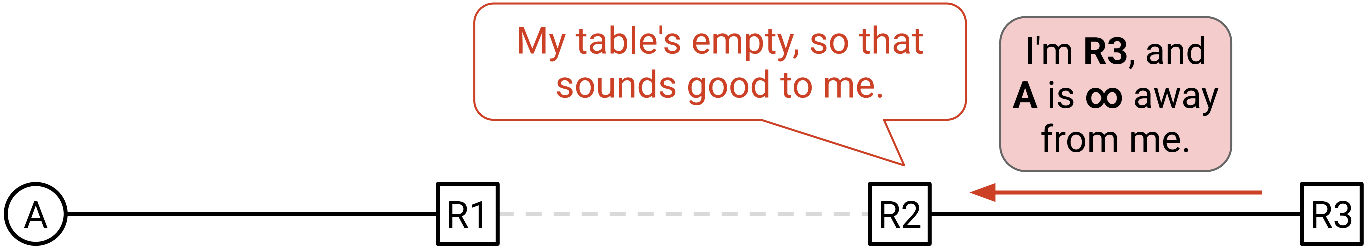
R2's Table		
To:	Via:	Cost:

R3's Table		
To:	Via:	Cost:
A	R2	3

Poison Reverse

R2's table now explicitly says: Do not send packets to R3.

- Because R3 would just send the packet back to R2.



My table's empty, so that sounds good to me.

I'm **R3**, and **A** is **∞** away from me.

A

R1

R2

R3

R1's Table		
To:	Via:	Cost:
A	Direct	1

R2's Table		
To:	Via:	Cost:
A	**R3**	**∞**

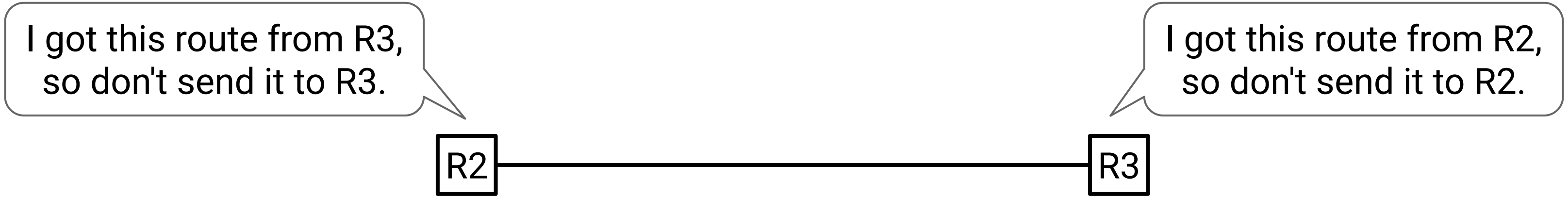
R3's Table		
To:	Via:	Cost:
A	R2	3

Poison Reverse vs. Split Horizon

Suppose we end up with a routing loop somehow.

Split horizon: No poison is sent.

- Loop stays until the routes expire.



I got this route from R3, so don't send it to R3.

I got this route from R2, so don't send it to R2.

R2's Table		
To:	Via:	Cost:
A	R3	4

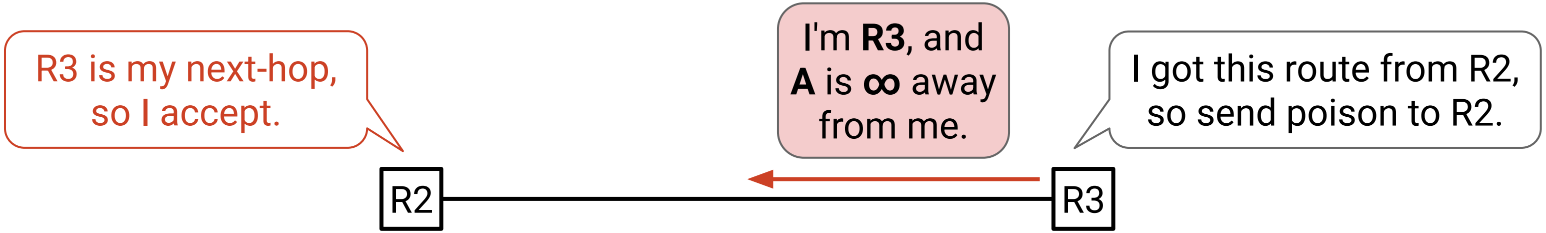
R3's Table		
To:	Via:	Cost:
A	R2	3

Poison Reverse vs. Split Horizon

Suppose we end up with a routing loop somehow.

Poison reverse: R3 explicitly sends poison back to R2.

- Loop is immediately eliminated!
- Faster than split horizon.



R3 is my next-hop, so I accept.

I'm **R3**, and **A** is **∞** away from me.

I got this route from R2, so send poison to R2.

R2's Table		
To:	Via:	Cost:
A	**R3**	**∞**

R3's Table		
To:	Via:	Cost:
A	R2	3

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table and reset TTL if:
  - The destination isn't in the table.
  - Advertised cost + link cost to neighbor < best-known cost. (#1)
  - The advertisement is from current next-hop. (#2)
    Includes poison advertisements. (#5)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
  - But don't advertise back to the next-hop. (#6A)
  - ...Or, advertise poison back to the next-hop. (#6B)
- If a table entry expires, make the entry poison and advertise it. (#4, #5)

Rule 7: Count to Infinity

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- **Rule 7: Count To Infinity**
- Eventful Updates

Count to Infinity – The Problem

Split horizon (or poison reverse) helps us avoid length-2 loops.

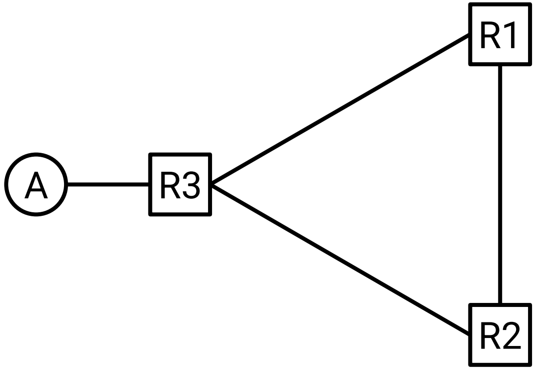
- R1 forwards to R2.
- R2 forwards to R1.

But we can still get routing loops with 3 or more routers.

Count to Infinity – The Problem

Suppose the tables reach steady-state.

R3's Table		
To:	Via:	Cost:
A	Direct	1



R1's Table		
To:	Via:	Cost:
A	R3	2

R2's Table		
To:	Via:	Cost:
A	R3	2

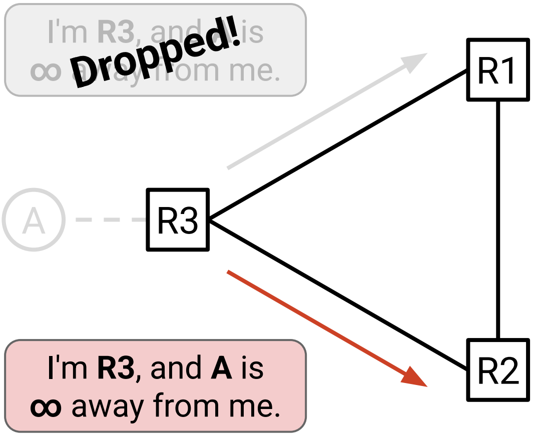
Count to Infinity – The Problem

Link goes down! A now unreachable.

R3 updates table and sends poison.

Poison reaches R2, but not R1!

R3's Table		
To:	Via:	Cost:
A	**Direct**	**∞**



I'm **R3**, and **A** is ∞ away from me.

Dropped!

I'm **R3**, and **A** is ∞ away from me.

R1's Table		
To:	Via:	Cost:
A	R3	2

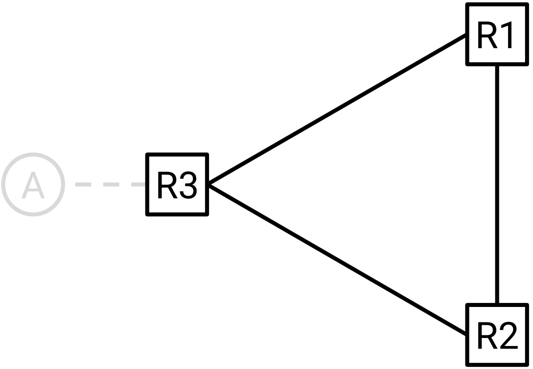
R2's Table		
To:	Via:	Cost:
A	**R3**	**∞**

Count to Infinity – The Problem

At this point, R3 and R2 know A is unreachable.

But R1 still thinks there's a path to A!

R3's Table		
To:	Via:	Cost:
A	Direct	∞



R1's Table		
To:	Via:	Cost:
A	R3	2

R2's Table		
To:	Via:	Cost:
A	R3	∞

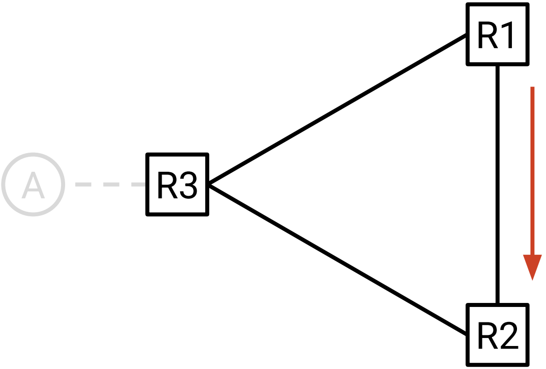
Count to Infinity – The Problem

R1 announces it can reach A.

Split horizon: R1's path came from R3, so don't tell R3.

R3's Table

To:	Via:	Cost:
A	Direct	∞



I'm **R1**, and **A** is **2** away from me.

R1's Table

To:	Via:	Cost:
A	R3	2

R2's Table

To:	Via:	Cost:
A	**R1**	**3**

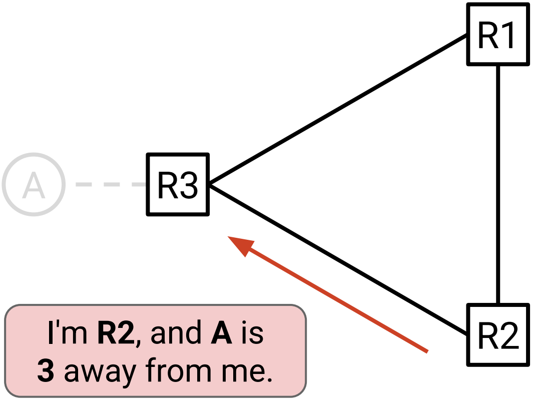
Count to Infinity – The Problem

R2 announces it can reach A.

Split horizon: R2's path came from R1, so don't tell R1.

R3's Table

To:	Via:	Cost:
A	**R2**	**4**



I'm **R2**, and **A** is **3** away from me.

R1's Table

To:	Via:	Cost:
A	R3	2

R2's Table

To:	Via:	Cost:
A	R1	3

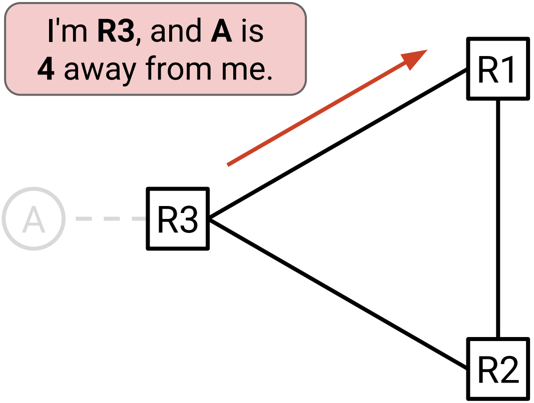
Count to Infinity – The Problem

R3 announces it can reach A.

Split horizon: R3's path came from R2, so don't tell R2.

R3's Table

To:	Via:	Cost:
A	R2	4



I'm **R3**, and **A** is **4** away from me.

R1's Table

To:	Via:	Cost:
A	**R3**	**5**

R2's Table

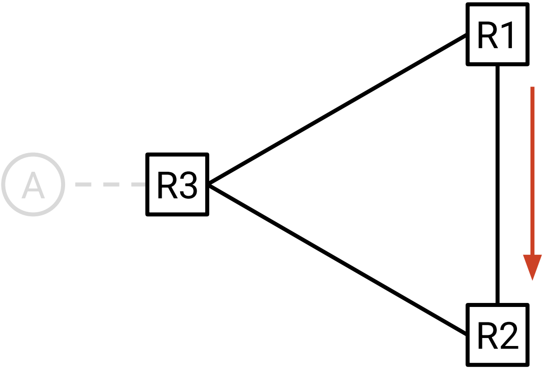
To:	Via:	Cost:
A	R1	3

Count to Infinity – The Problem

We keep advertising in a cycle, and costs keep increasing!

Split horizon can't save us.

R3's Table		
To:	Via:	Cost:
A	R2	4



I'm **R1**, and **A** is **5** away from me.

R1's Table		
To:	Via:	Cost:
A	R3	5

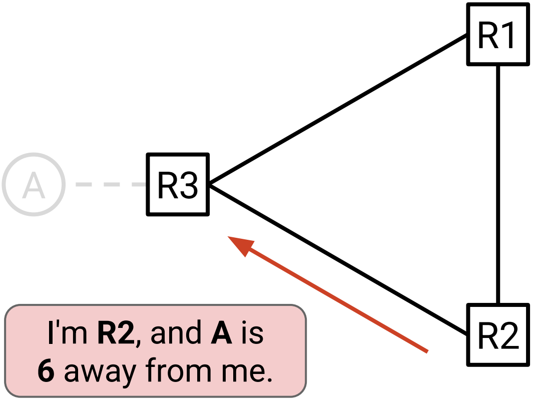
R2's Table		
To:	Via:	Cost:
A	**R1**	**6**

Count to Infinity – The Problem

We keep advertising in a cycle, and costs keep increasing!

Split horizon can't save us.

R3's Table		
To:	Via:	Cost:
A	**R2**	**7**



I'm **R2**, and **A** is **6** away from me.

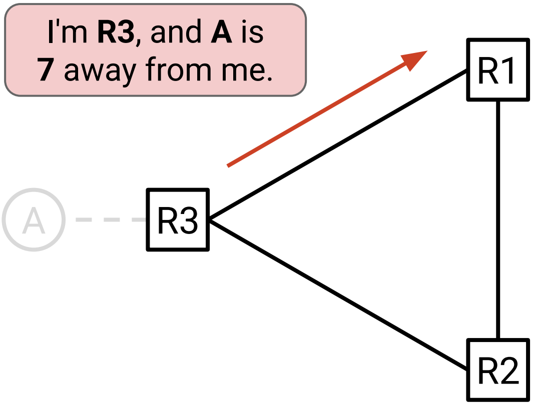
R1's Table		
To:	Via:	Cost:
A	R3	5

R2's Table		
To:	Via:	Cost:
A	R1	6

Count to Infinity – The Problem

We keep advertising in a cycle, and costs keep increasing!

Split horizon can't save us.



R3's Table		
To:	Via:	Cost:
A	R2	7

R1's Table		
To:	Via:	Cost:
A	**R3**	**8**

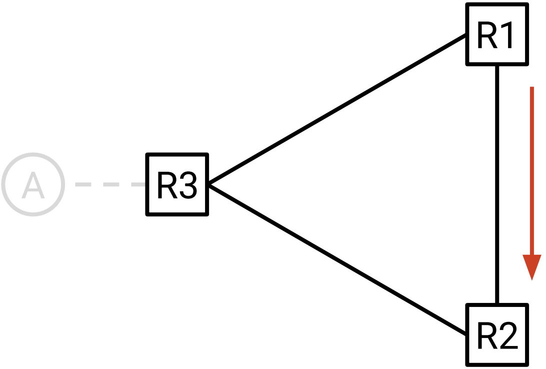
R2's Table		
To:	Via:	Cost:
A	R1	6

Count to Infinity – The Problem

We keep advertising in a cycle, and costs keep increasing!

Split horizon can't save us.

R3's Table		
To:	Via:	Cost:
A	R2	7



I'm **R1**, and **A** is **8** away from me.

R1's Table		
To:	Via:	Cost:
A	R3	8

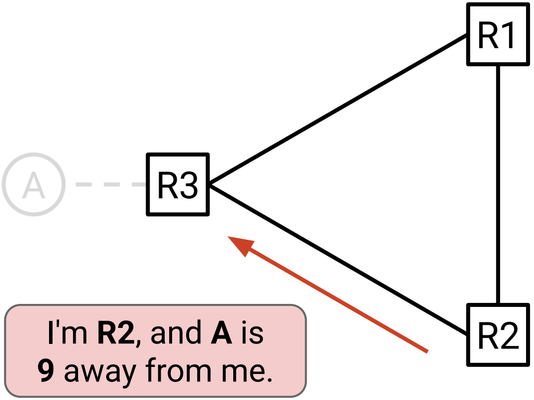
R2's Table		
To:	Via:	Cost:
A	**R1**	**9**

Count to Infinity – The Problem

We keep advertising in a cycle, and costs keep increasing!

Split horizon can't save us.

R3's Table		
To:	Via:	Cost:
A	**R2**	**10**



I'm **R2**, and **A** is **9** away from me.

R1's Table		
To:	Via:	Cost:
A	R3	8

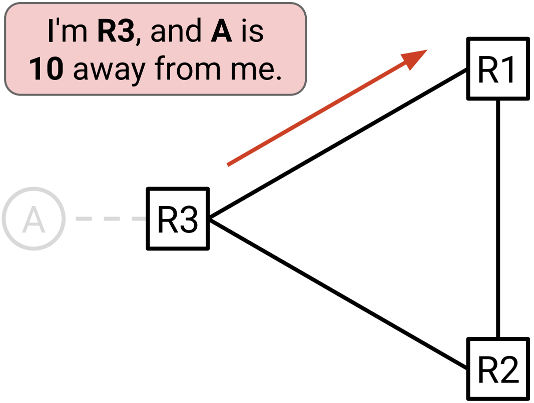
R2's Table		
To:	Via:	Cost:
A	R1	9

Count to Infinity – The Problem

We keep advertising in a cycle, and costs keep increasing!

Split horizon can't save us.

R3's Table		
To:	Via:	Cost:
A	R2	10



I'm **R3**, and **A** is **10** away from me.

R1's Table		
To:	Via:	Cost:
A	**R3**	**11**

R2's Table		
To:	Via:	Cost:
A	R1	9

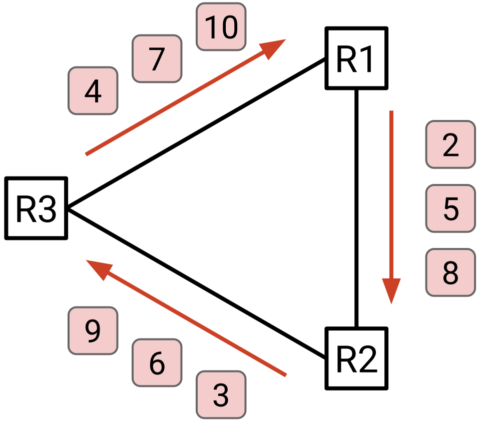
Count to Infinity – The Problem

The problem, restated:

- Poison wasn't propagated properly. A router had a busted path.
- Busted path is advertised in a loop.

Split horizon won't save us.

- We're never advertising a path back to the next-hop.



Count to Infinity – Solution

Solution: Enforce a maximum cost.

- 15 is a common choice.
- All numbers ≥ 16 are considered infinity.

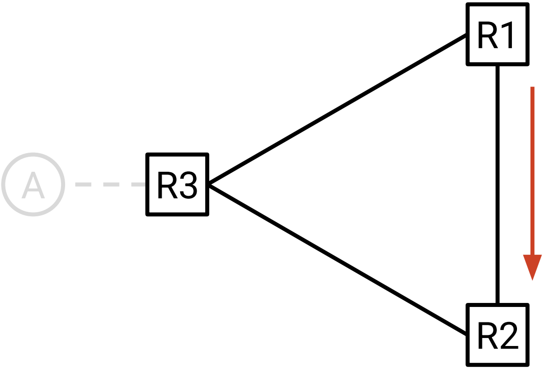
Result:

- Loop will stop when all costs reach 16.
- Busted path will expire, or get replaced by another non-infinite-cost path.

Count to Infinity – Solution

All numbers ≥ 16 are considered infinity.

R3's Table		
To:	Via:	Cost:
A	R2	10



R1's Table		
To:	Via:	Cost:
A	R3	11

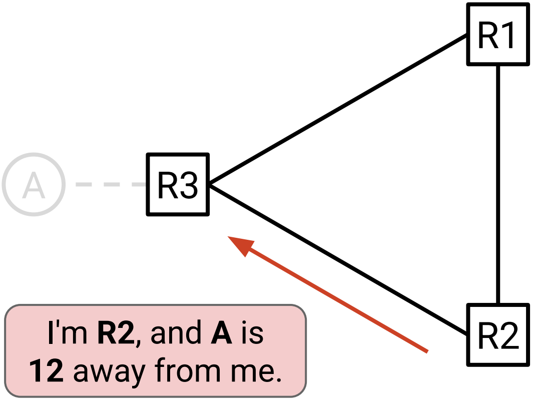
I'm **R1**, and **A** is **11** away from me.

R2's Table		
To:	Via:	Cost:
A	**R1**	**12**

Count to Infinity – Solution

All numbers ≥ 16 are considered infinity.

R3's Table		
To:	Via:	Cost:
A	**R2**	**13**



I'm **R2**, and **A** is **12** away from me.

R1's Table		
To:	Via:	Cost:
A	R3	11

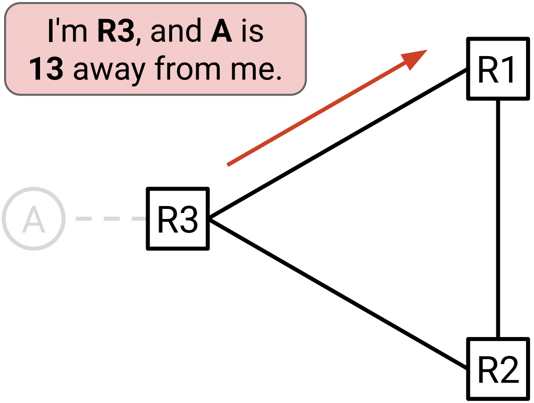
R2's Table		
To:	Via:	Cost:
A	R1	12

Count to Infinity – Solution

All numbers ≥ 16 are considered infinity.

R3's Table

To:	Via:	Cost:
A	R2	13



I'm **R3**, and **A** is **13** away from me.

R1's Table

To:	Via:	Cost:
A	**R3**	**14**

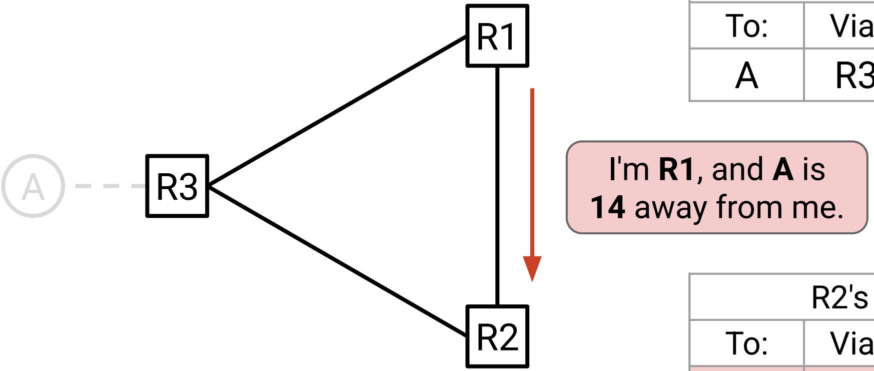
R2's Table

To:	Via:	Cost:
A	R1	12

Count to Infinity – Solution

All numbers ≥ 16 are considered infinity.

R3's Table		
To:	Via:	Cost:
A	R2	13



I'm **R1**, and **A** is **14** away from me.

R1's Table		
To:	Via:	Cost:
A	R3	14

R2's Table		
To:	Via:	Cost:
A	**R1**	**15**

Count to Infinity – Solution

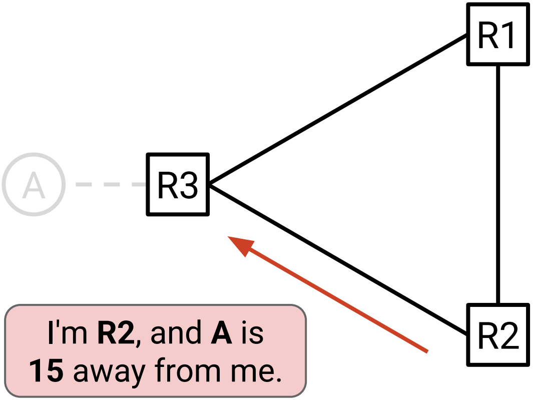
All numbers ≥ 16 are considered infinity.

R1's Table

To:	Via:	Cost:
A	R3	14

R3's Table

To:	Via:	Cost:
A	**R2**	~~**16**~~ **∞**



I'm **R2**, and **A** is **15** away from me.

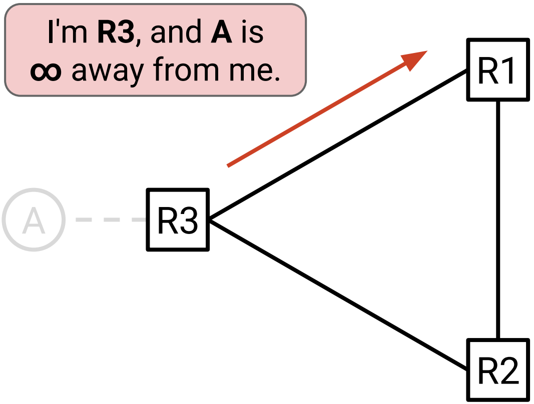
R2's Table

To:	Via:	Cost:
A	R1	15

Count to Infinity – Solution

All numbers ≥ 16 are considered infinity.

R3's Table		
To:	Via:	Cost:
A	R2	∞



I'm **R3**, and **A** is **∞** away from me.

R1's Table		
To:	Via:	Cost:
A	**R3**	**∞**

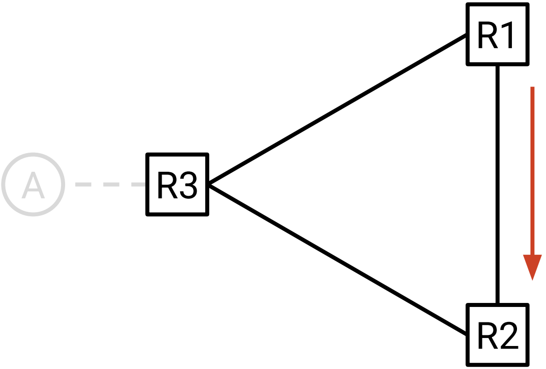
R2's Table		
To:	Via:	Cost:
A	R1	15

Count to Infinity – Solution

All numbers ≥ 16 are considered infinity.

R3's Table

To:	Via:	Cost:
A	R2	∞



I'm **R1**, and **A** is **∞** away from me.

R1's Table

To:	Via:	Cost:
A	R3	∞

R2's Table

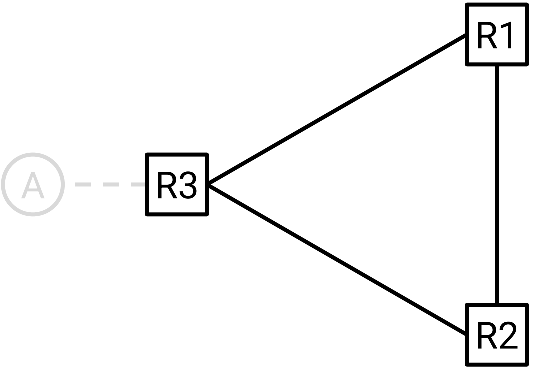
To:	Via:	Cost:
A	**R1**	**∞**

Count to Infinity – Solution

We've reached steady state!

- Future advertisements won't change the tables.
- Routes for A will soon expire.
    - Or, if another route to A appears, it'll replace the infinite-cost entry.

R3's Table		
To:	Via:	Cost:
A	R2	∞



R1's Table		
To:	Via:	Cost:
A	R3	∞

R2's Table		
To:	Via:	Cost:
A	R1	∞

The Distance-Vector Algorithm So Far

For each destination:

- If you hear an advertisement, update table and reset TTL if:
  - The destination isn't in the table.
  - Advertised cost + link cost to neighbor < best-known cost. (#1)
  - The advertisement is from current next-hop. (#2)

    Includes poison advertisements. (#5)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
  - But don't advertise back to the next-hop. (#6A)
  - ...Or, advertise poison back to the next-hop. (#6B)
  - Any cost ≥ 16 is advertised as ∞. (#7)
- If a table entry expires, make the entry poison and advertise it. (#4, #5)

Eventful Updates

Lecture 5, CS 168, Spring 2025

Distance-Vector Correctness

- Algorithm Sketch
- Rule 1: Bellman-Ford Updates
- Bellman-Ford Demo
- Rule 2: Updates From Next-Hop
- Rule 3: Resending
- Rule 4: Expiring

Distance-Vector Enhancements

- Rule 5: Poison Expired Routes
- Rule 6A: Split Horizon
- Rule 6B: Poison Reverse
- Rule 7: Count To Infinity
- **Eventful Updates**

Eventful Updates

When do we send advertisements?

- When the table changes (**triggered updates**).
  - When we accept a new advertisement.
  - When a new link is added. *(Add static routes and advertise them.)*
  - When a link goes down. *(Poison routes and advertise poison.)*
- Periodically (once every "advertisement interval").
- When a table entry expires.

Triggered updates are an optimization.

- Instead of advertising when the table changes, we could just wait for the interval. Protocol is still correct.
- Triggered updates help us converge faster.

Our Completed Distance-Vector Algorithm

For each destination:

- If you hear an advertisement, update table and reset TTL if:
    - The destination isn't in the table.
    - Advertised cost + link cost to neighbor < best-known cost. (#1)
    - The advertisement is from current next-hop. (#2)
      Includes poison advertisements. (#5)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
    - But don't advertise back to the next-hop. (#6A)
    - ...Or, advertise poison back to the next-hop. (#6B)
    - Any cost ≥ 16 is advertised as ∞. (#7)
- If a table entry expires, make the entry poison and advertise it. (#3, #5)

Summary: Distance-Vector Rules

1. **Bellman-Ford Updates**: Accept if advertised cost + link cost to neighbor < best-known cost.

2. **Updates From Next-Hop**: Accept if advertisement is from next hop.

3. **Resending**: Advertise periodically.

4. **Expiring**: Expire an entry if TTL runs out.

5. **Poison Expired Routes**: Send poison if an entry expires.

6A. **Split Horizon**: Don't advertise path back to the person who gave it to you.

6B. **Poison Reverse**: Send poison back to the person who gave you the path.

7. **Count To Infinity**: Any cost $\geq$ 16 is advertised as ∞.

This is now a pretty good routing protocol!