Discussion 4, Routing I 

CS 168, Fall 2024 @ UC Berkeley

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

Logistics

- Project 1a and 1b due last week
- Project 2 released, due on October 4th
- Homework 1 released, due on September 30th

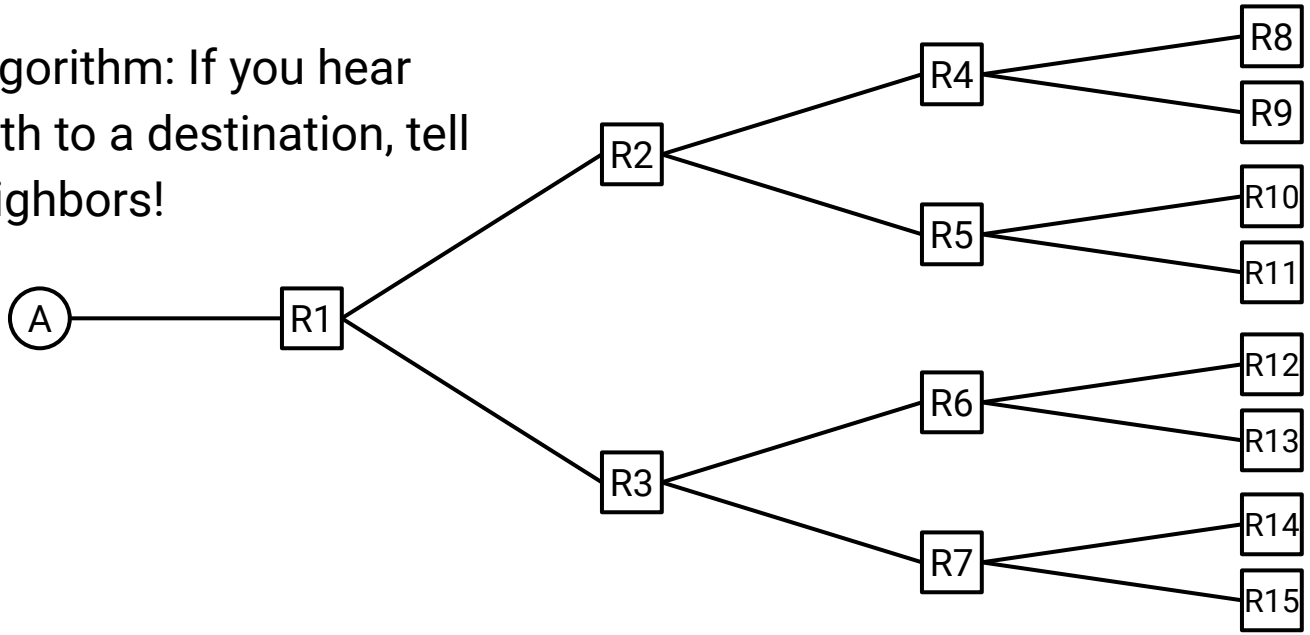
Distance-Vector Algorithm Sketch

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.

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



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

Distance-Vector Correctness

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

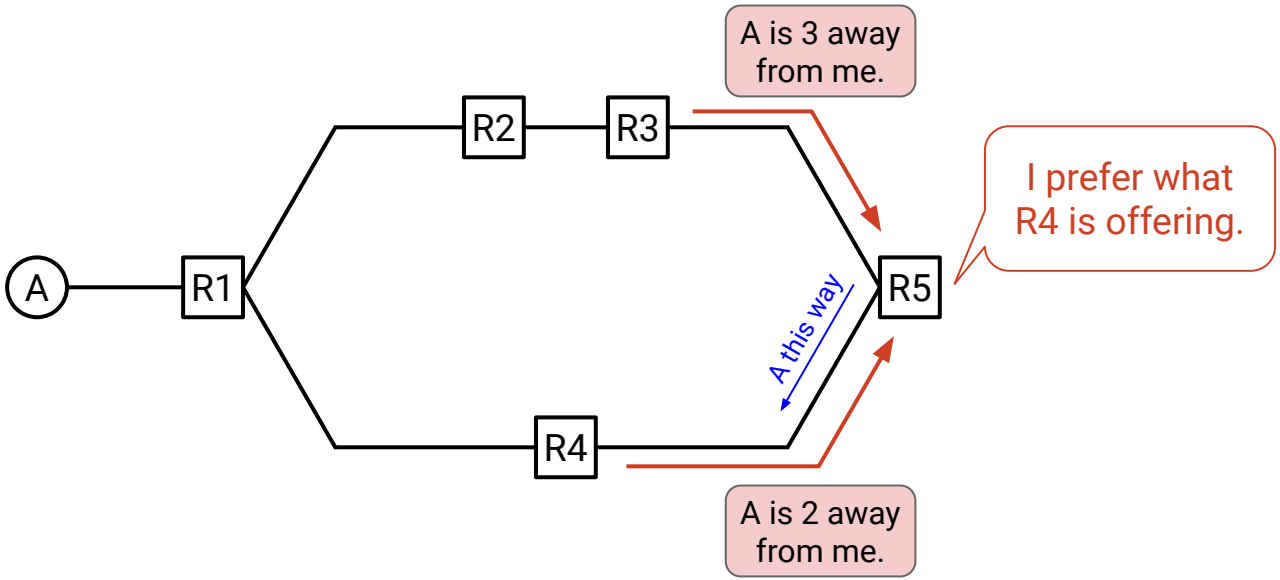
Distance-Vector Enhancements

- Rule 4: Poison Expired Routes
- Rule 5A: Split Horizon
- Rule 5B: Poison Reverse
- Rule 6: Count To Infinity
- Eventful Updates

Multiple Paths Advertised

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

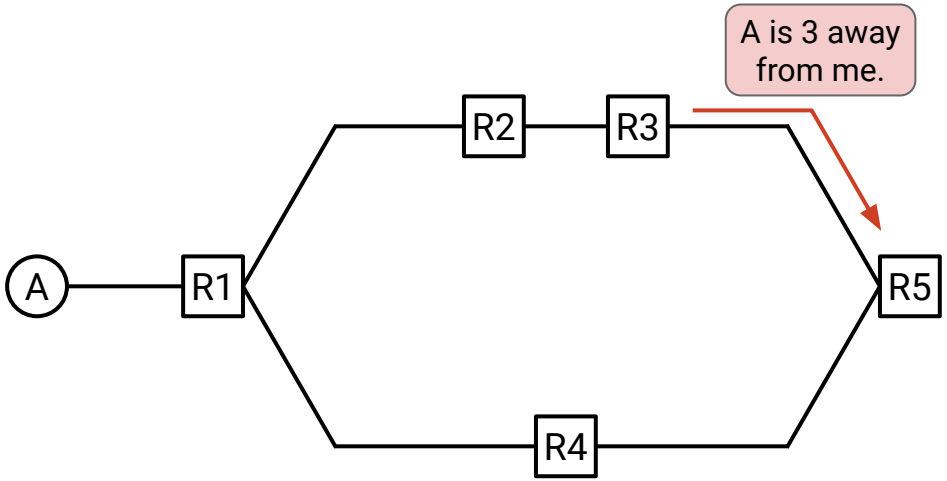
- Accept the shorter path.



Multiple Paths Advertised

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.

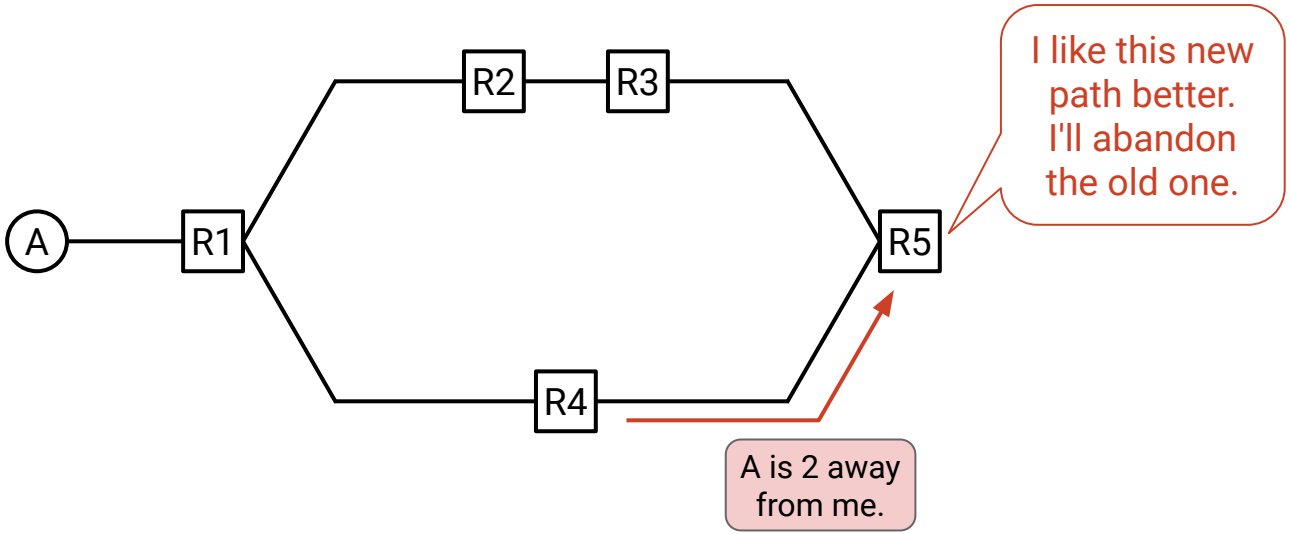


R5's Table		
To:	Via:	Cost:
A	R3	4

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

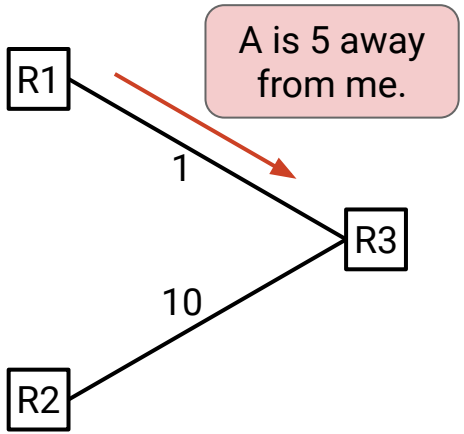
For each destination:

- 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).

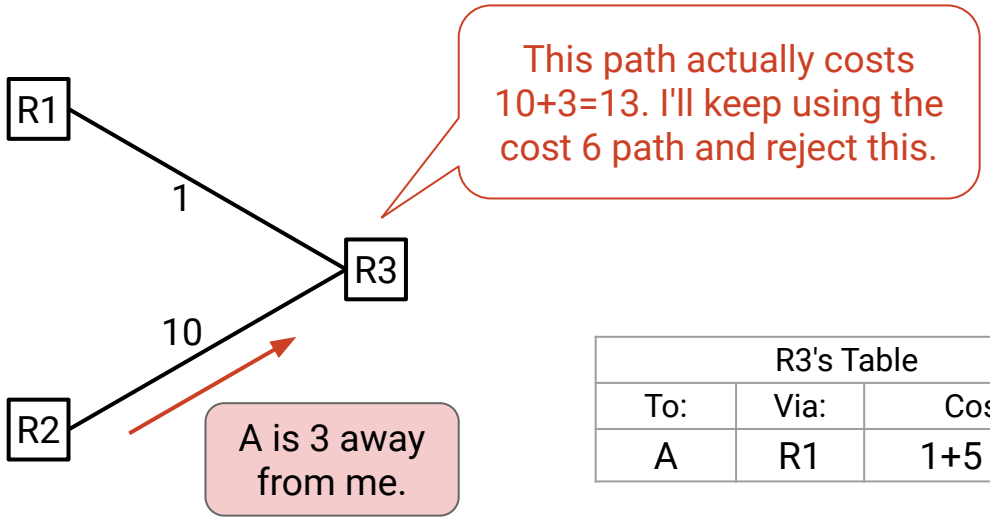


R3's Table		
To:	Via:	Cost:
A	R1	1+5 = 6

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).



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.

Rule 2: Updates from Next-Hop

Distance-Vector Correctness

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

Distance-Vector Enhancements

- Rule 4: Poison Expired Routes
- Rule 5A: Split Horizon
- Rule 5B: Poison Reverse
- Rule 6: 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.

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 and Expiring

Distance-Vector Correctness

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

Distance-Vector Enhancements

- Rule 4: Poison Expired Routes
- Rule 5A: Split Horizon
- Rule 5B: Poison Reverse
- Rule 6: 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)
  - The advertisement is from current next-hop. (#2)
- Advertise to all your neighbors when the table updates, and periodically. (#3)

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.

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. (#3)

This is a mostly-functional protocol now.

Let's add some optimizations for faster convergence.

Rule 4: Poison Expired Routes

Distance-Vector Correctness

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

Distance-Vector Enhancements

- **Rule 4: Poison Expired Routes**
- Rule 5A: Split Horizon
- Rule 5B: Poison Reverse
- Rule 6: Count To Infinity
- Eventful Updates

Poison for Fast Route Expiry

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!

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

Rule 5A/5B: Split Horizon / Poison Reverse

Distance-Vector Correctness

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

Distance-Vector Enhancements

- Rule 4: Poison Expired Routes
- **Rule 5A: Split Horizon**
- **Rule 5B: Poison Reverse**
- Rule 6: Count To Infinity
- Eventful Updates

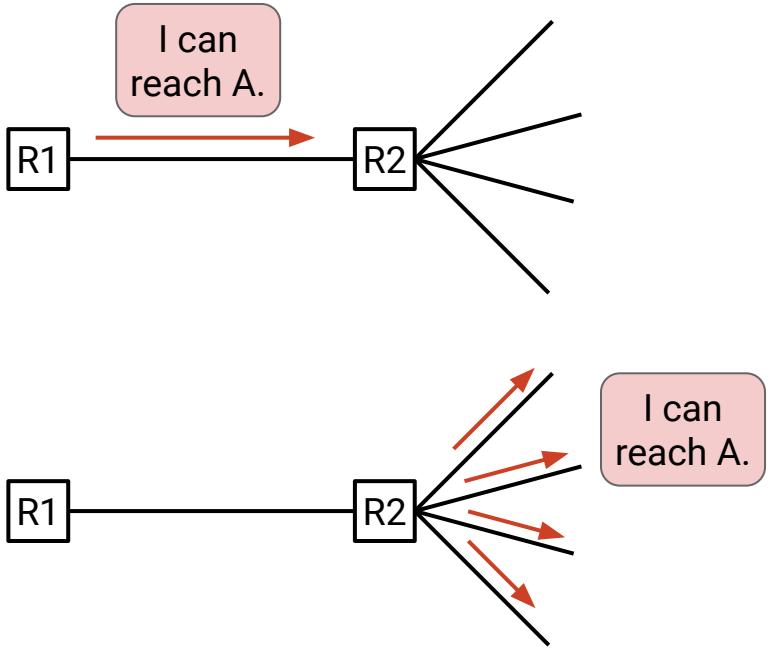
Split Horizon / Poison Reverse – The Problem

The 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.

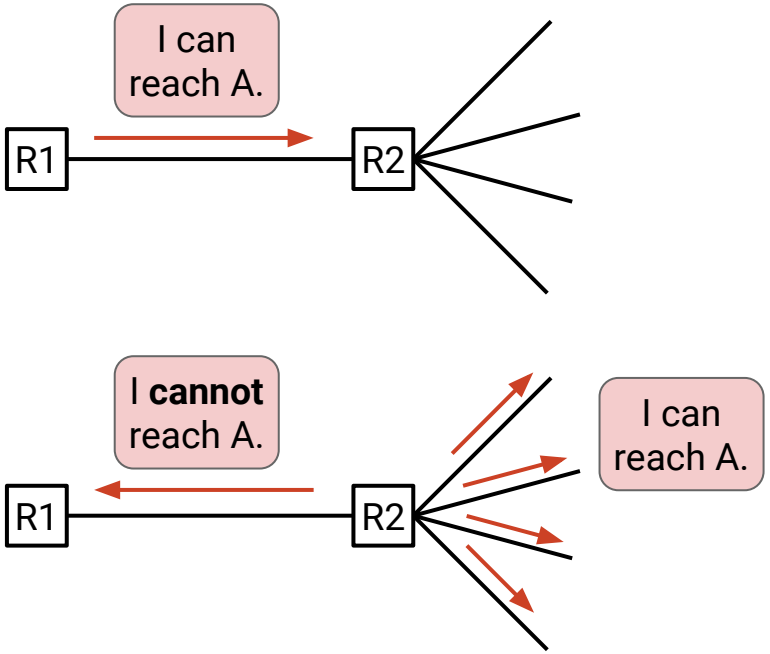
Poison Reverse vs. Split Horizon

Split Horizon:



Don't advertise anything back to R1.

Poison Reverse:



Explicitly advertise poison back to R1 (infinity).

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.

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

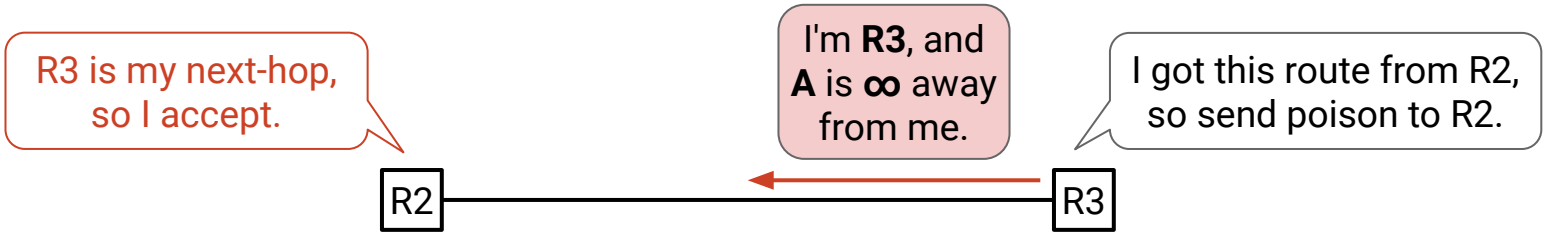
R3's Table		
To:	Via:	Cost:
A	R2	3

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.



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. (#4)
- Advertise to all your neighbors when the table updates, and periodically. (#3)
  - But don't advertise back to the next-hop. (#5A)
  - ...Or, advertise poison back to the next-hop. (#5B)
- If a table entry expires, make the entry poison and advertise it. (#3, #4)

Rule 6: Count to Infinity

Distance-Vector Correctness

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

Distance-Vector Enhancements

- Rule 4: Poison Expired Routes
- Rule 5A: Split Horizon
- Rule 5B: Poison Reverse
- **Rule 6: Count To Infinity**
- Eventful Updates

Count to Infinity – The Problem

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

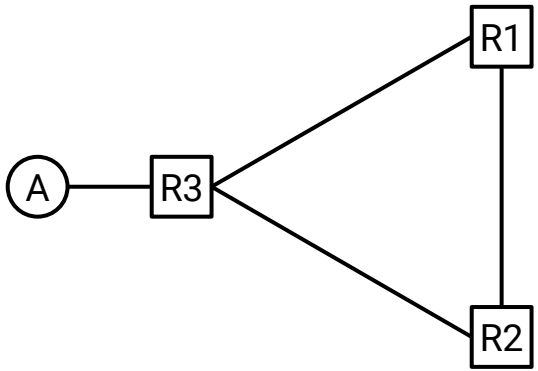
- R1 forwards to R2.
- R2 forwards to R1.

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

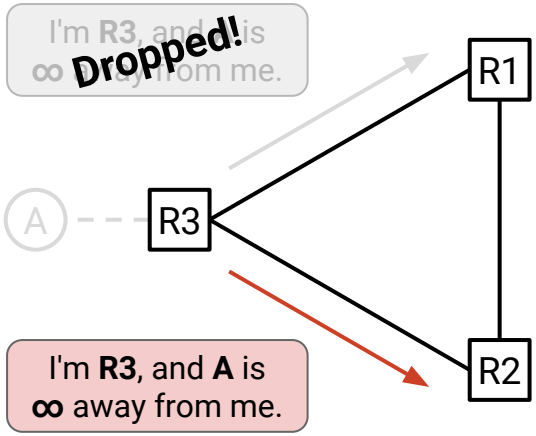
Count to Infinity – The Problem

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**	**∞**



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

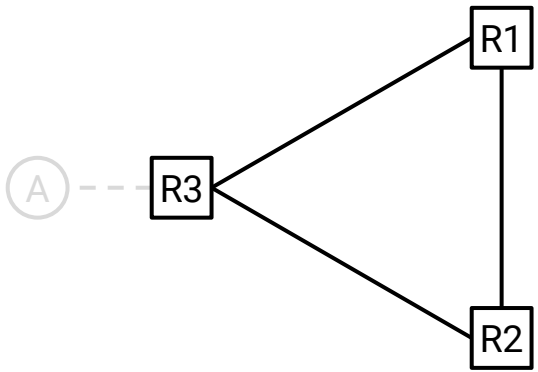
R2's Table		
To:	Via:	Cost:
A	**R3**	**∞**

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

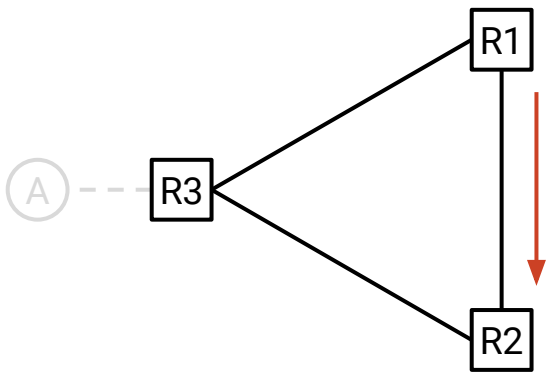
R2's Table		
To:	Via:	Cost:
A	R3	∞

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	∞



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

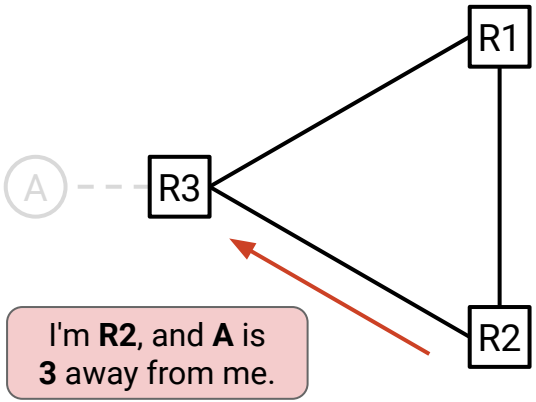
R2's Table		
To:	Via:	Cost:
A	**R1**	**3**

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**



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

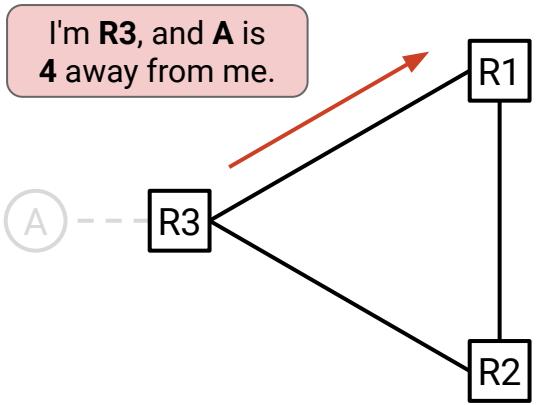
R2's Table		
To:	Via:	Cost:
A	R1	3

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



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

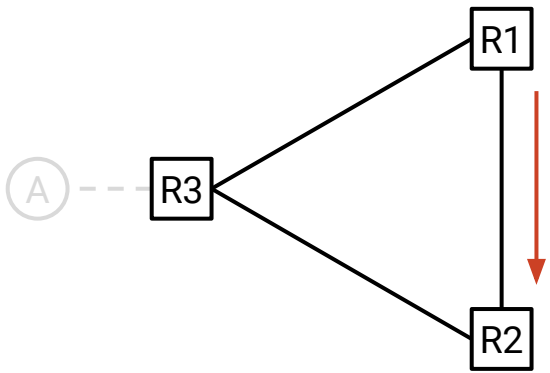
R2's Table		
To:	Via:	Cost:
A	R1	3

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



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

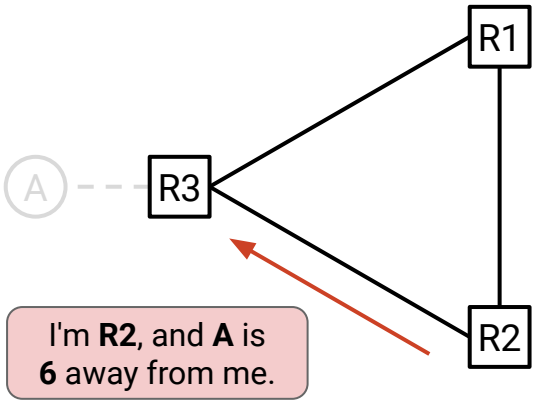
R2's Table		
To:	Via:	Cost:
A	**R1**	**6**

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	5

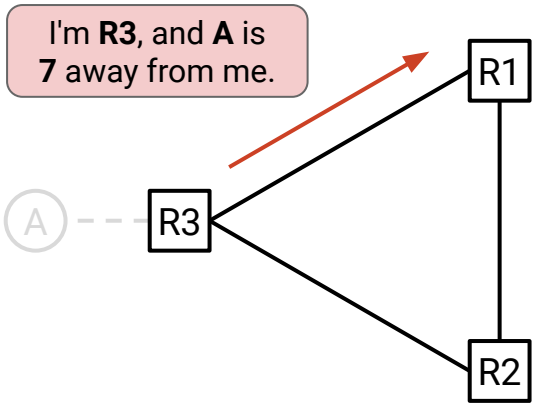
R2's Table		
To:	Via:	Cost:
A	R1	6

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**

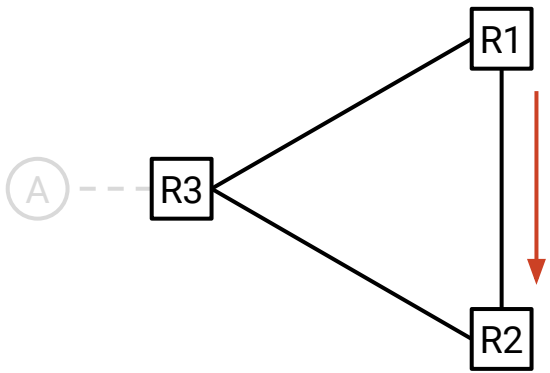
R2's Table		
To:	Via:	Cost:
A	R1	6

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



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

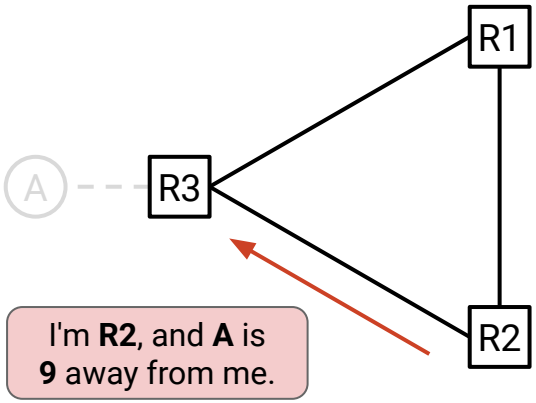
R2's Table		
To:	Via:	Cost:
A	**R1**	**9**

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**



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

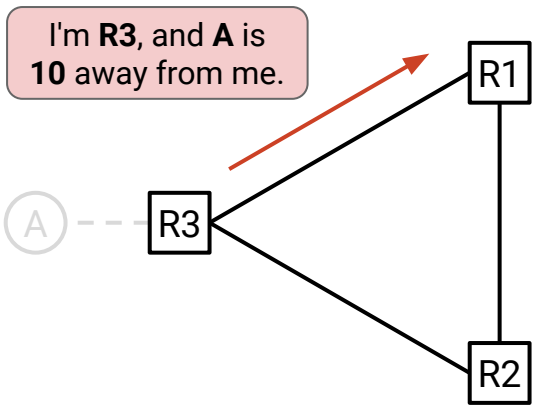
R2's Table		
To:	Via:	Cost:
A	R1	9

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



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

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

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.

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

Eventful Updates

Distance-Vector Correctness

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

Distance-Vector Enhancements

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

Feedback Form:
https://tinyurl.com/cs168-disc-fa24