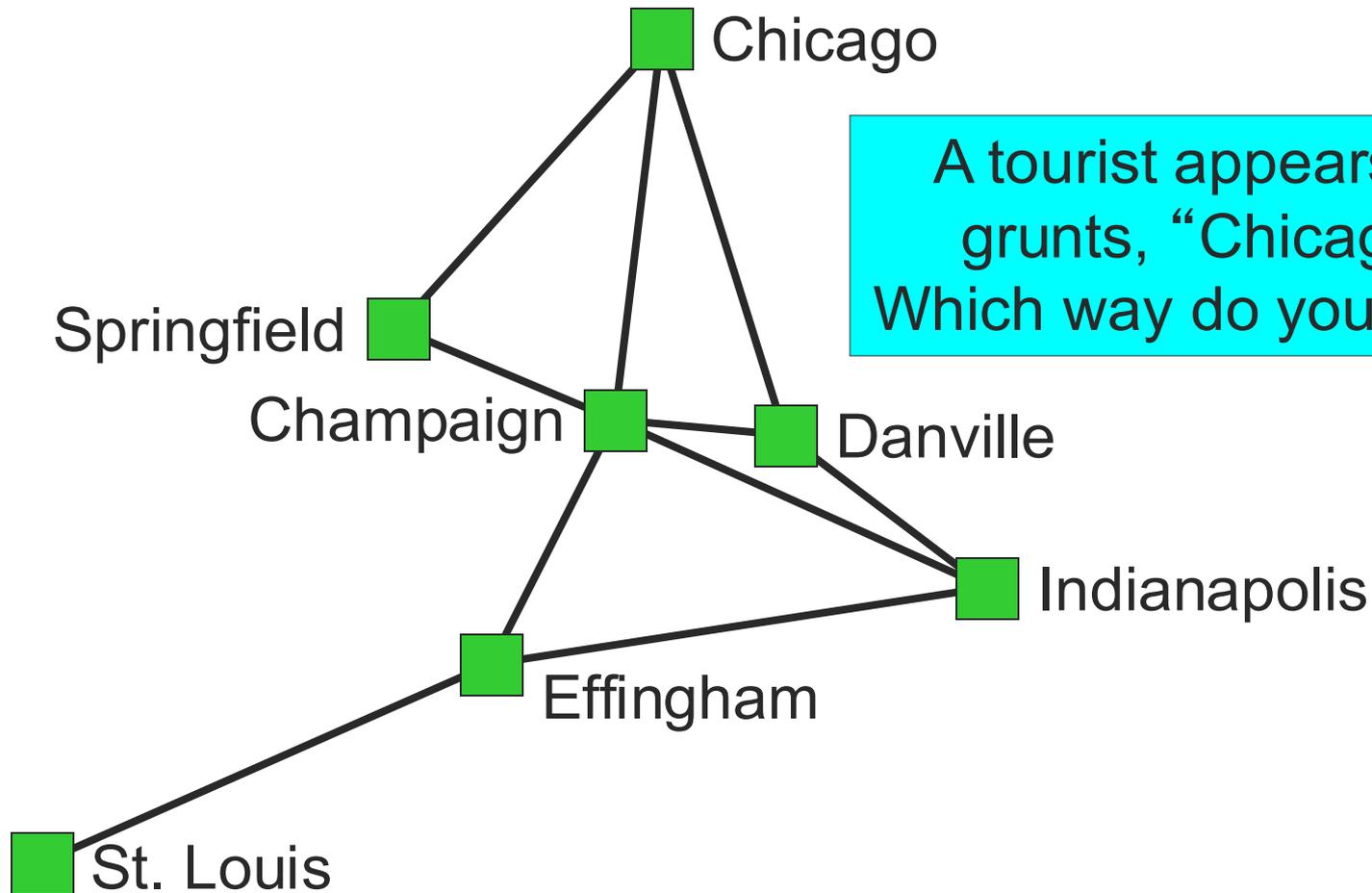




# Routing

# [ Routing ]

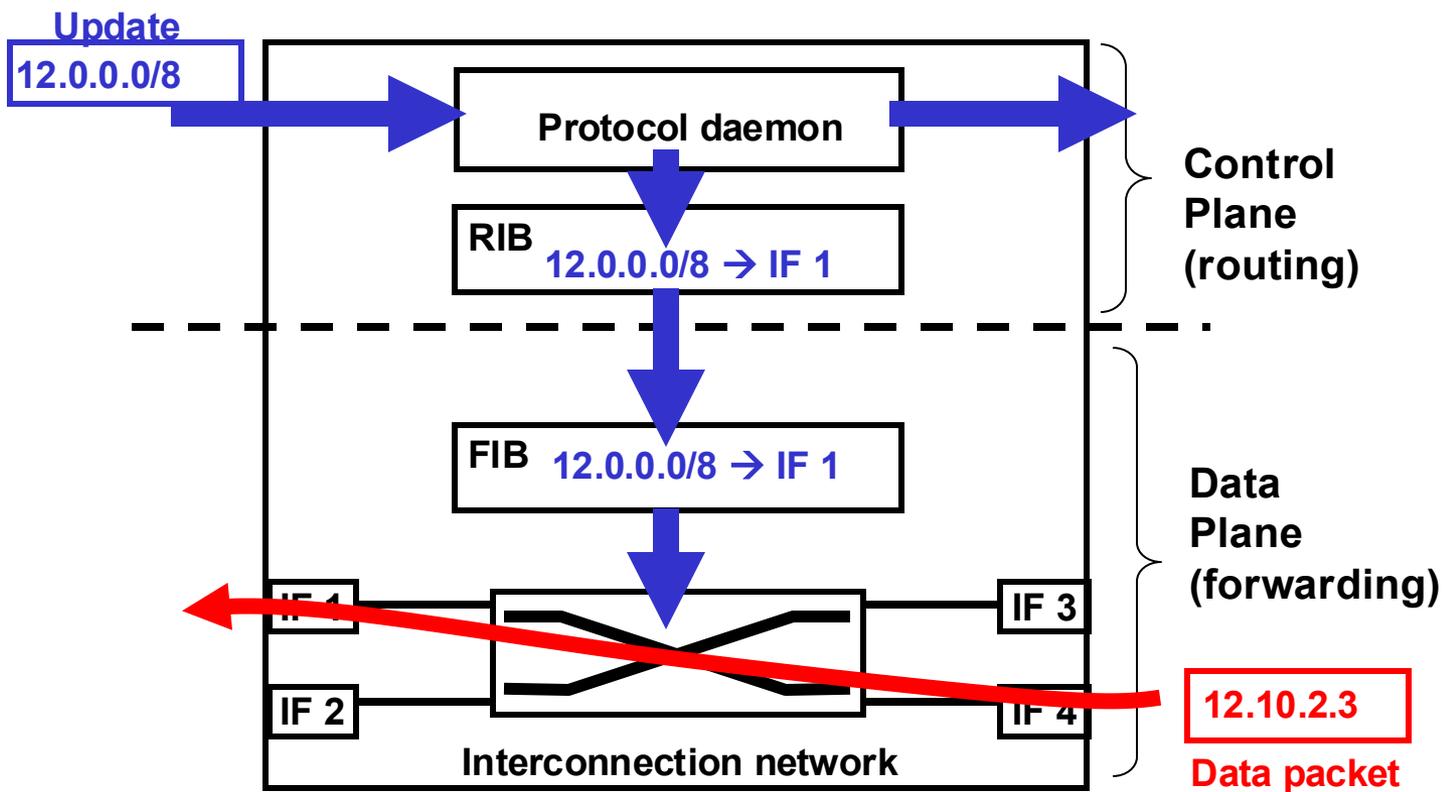


A tourist appears and grunts, "Chicago?"  
Which way do you point?



# Network Routing

Constructing and maintaining forwarding information in hosts or routers



# [ Routing ]

## ■ Goals

- Capture the notion of “best” routes
- Propagate changes effectively
- Require limited information exchange

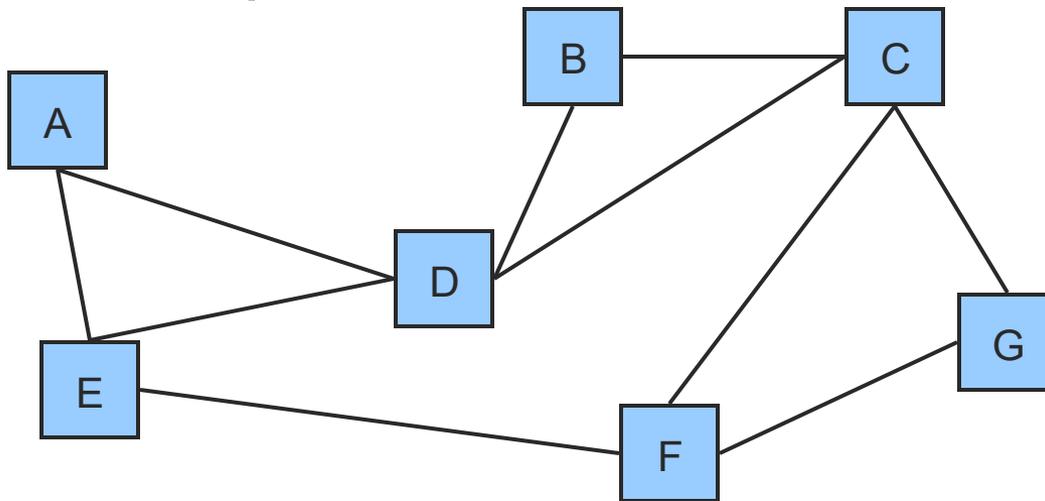
## ■ Conceptually

- A network can be represented as a graph where each host/router is a node and each physical connection is a link



# Routing: Ideal Approach

- Maintain information about each link
- Calculate fastest path between each directed pair



For each direction, maintain:

- Bandwidth
- Latency
- Queueing delay



# [ Routing: Ideal Approach ]

- Problems
  - Unbounded amount of information
  - Queueing delay can change rapidly
  - Graph connectivity can change rapidly
- Solution
  - Dynamic
    - Periodically recalculate routes
  - Distributed
    - No single point of failure
    - Reduced computation per node
  - Abstract Metric
    - “Distance” may combine many factors
    - Use heuristics



# [ Routing Overview ]

- Algorithms
  - Static shortest path algorithms
    - Bellman-Ford
      - Based on local iterations
    - Dijkstra's algorithm
      - Build tree from source
  - Distributed, dynamic routing algorithms
    - Distance vector routing
      - Distributed Bellman-Ford
    - Link state routing
      - Implement Dijkstra's algorithm at each node



# Bellman-Ford Algorithm

- Concept
  - Static centralized algorithm
- Given
  - Directed graph with edge costs and destination node
- Finds
  - Least cost path from each node to destination
- Multiple nodes
  - To find shortest paths for multiple destination nodes, run entire Bellman-Ford algorithm once per destination

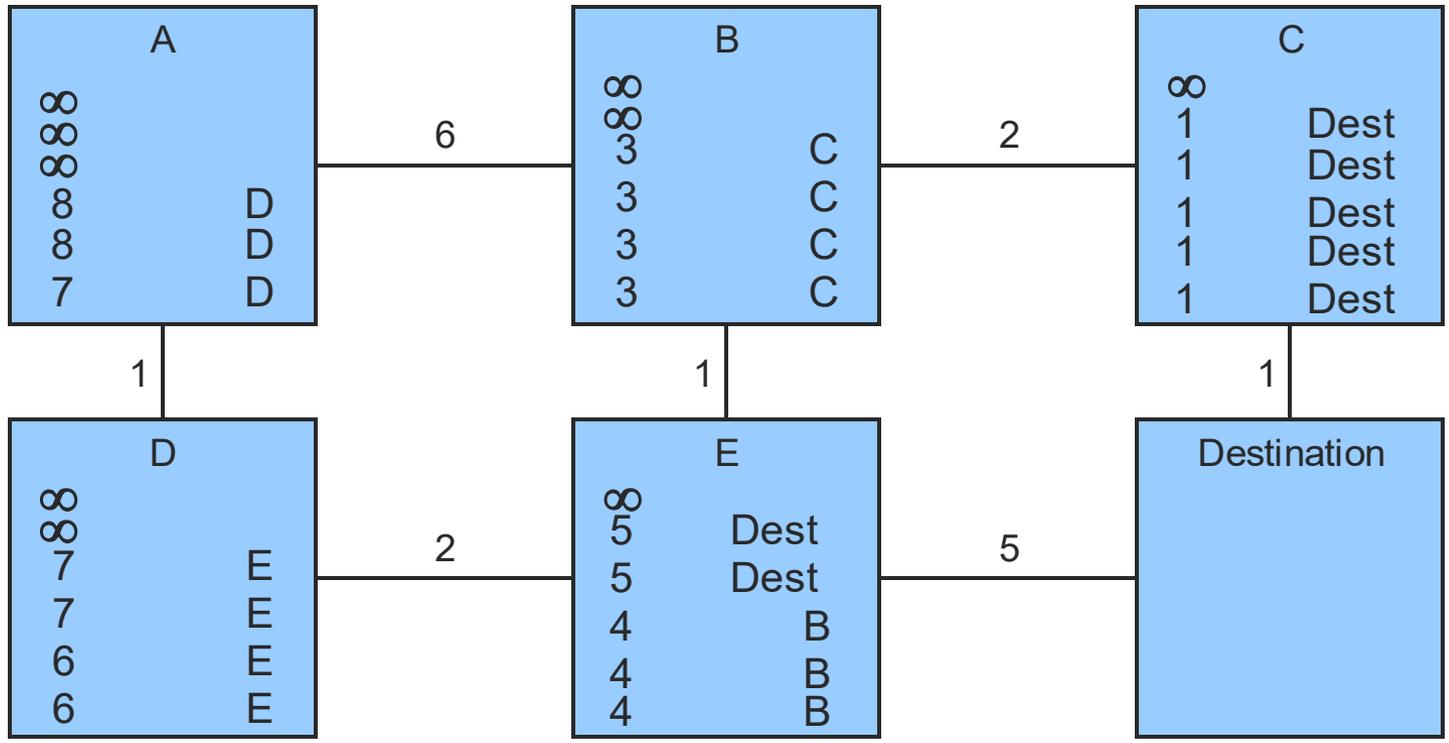


# [ Bellman-Ford Algorithm ]

- Based on repetition of iterations
  - For every node A and every neighbor B of A
    - Is the cost of the path (A → B → → → destination) smaller than the currently known cost from A to destination?
      - If YES
        - Make B the successor node for A
        - Update cost from A to destination
  - Can run iterations synchronously or all at once



# Bellman-Ford Algorithm



# Distance Vector Routing

- Distributed dynamic version of Bellman-Ford
- Each node maintains a table of
  - *<destination, distance, successor>*
- Information acquisition
  - Assume nodes initially know cost to immediate neighbor
  - Nodes send *<destination, distance>* vectors to all immediate neighbors
    - Periodically – seconds, minutes
    - Whenever vector changes – triggered update



# Distance Vector Routing

- When a route changes
  - Local failure detection
    - Control message not acknowledged
    - Timeout on periodic route update
  - Current route disappears
  - Newly advertised route is shorter than previous route
- Used in
  - Original ARPANET (until 1979)
  - Early Internet: Routing Information Protocol (RIP)
  - Early versions of DECnet and Novell IPX



# Distance vector: update propagation

D tells B: I am D, and I can reach F via 1 hop

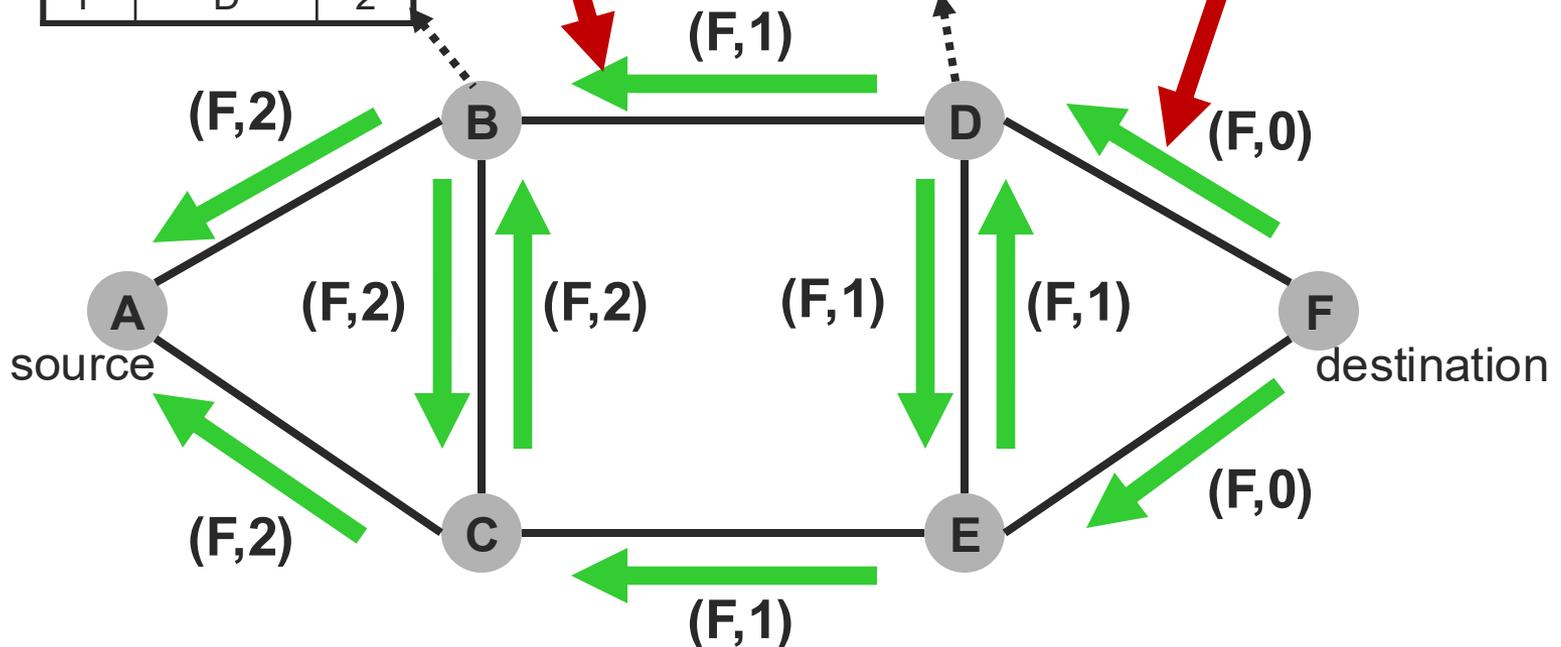
B's forwarding table

| Dest | NextHop | Dist |
|------|---------|------|
| F    | D       | 2    |

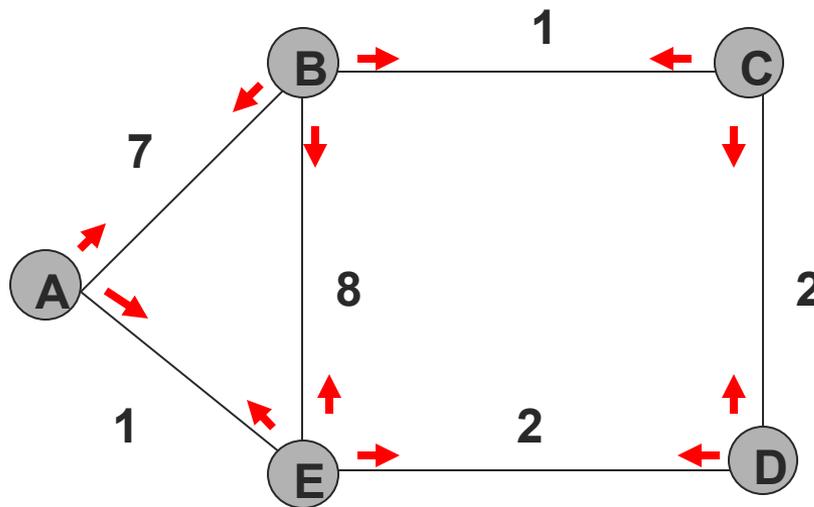
D's forwarding table

| Dest | NextHop | Dist |
|------|---------|------|
| F    | F       | 1    |

F tells D: I am F, and I can reach F via 0 hops



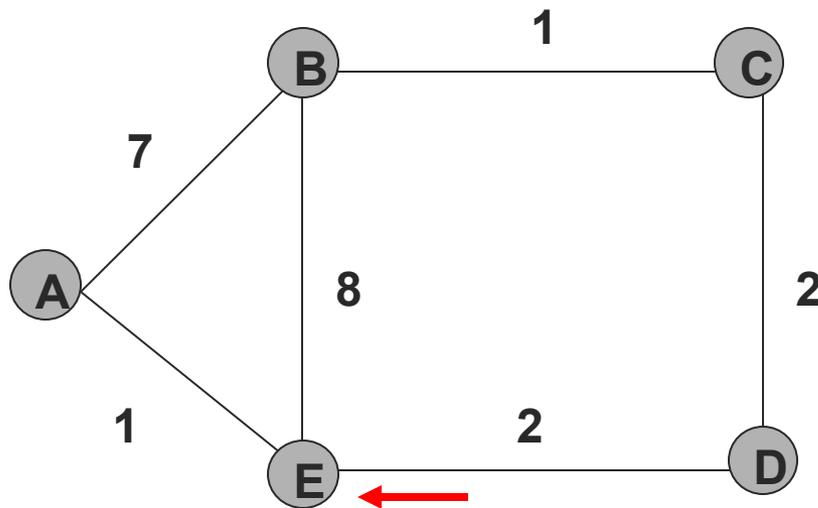
# [ Example - Initial Distances ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 7 | ~ | ~ | 1 |
| B            | 7                | 0 | 1 | ~ | 8 |
| C            | ~                | 1 | 0 | 2 | ~ |
| D            | ~                | ~ | 2 | 0 | 2 |
| E            | 1                | 8 | ~ | 2 | 0 |



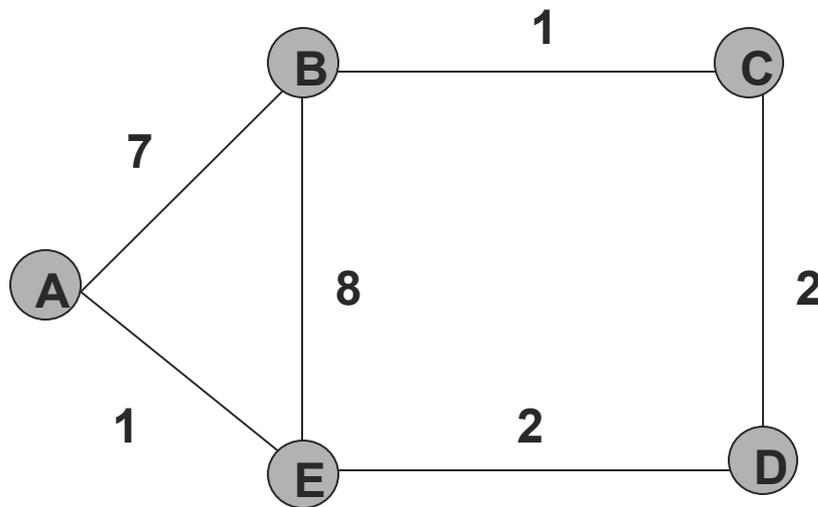
# [ E Receives D's Routes ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 7 | ~ | ~ | 1 |
| B            | 7                | 0 | 1 | ~ | 8 |
| C            | ~                | 1 | 0 | 2 | ~ |
| D            | ~                | ~ | 2 | 0 | 2 |
| E            | 1                | 8 | ~ | 2 | 0 |



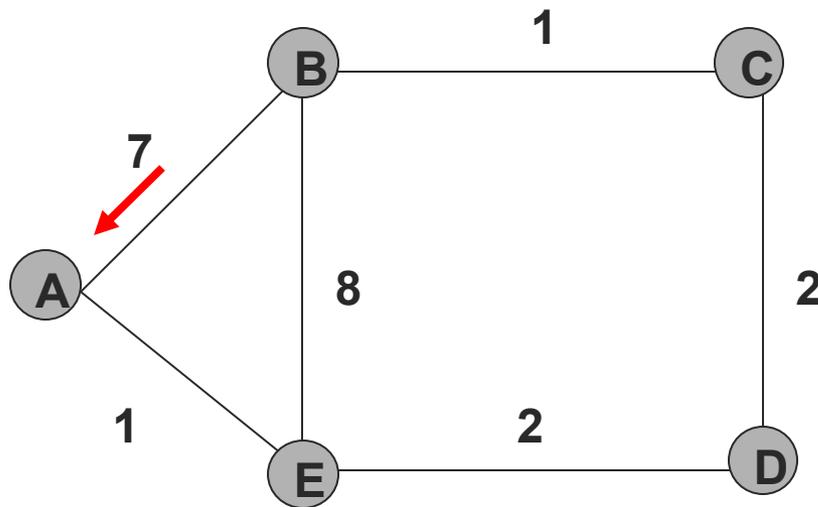
# [ E Updates Cost to C ]



| Info at node | Distance to node |   |          |   |   |
|--------------|------------------|---|----------|---|---|
|              | A                | B | C        | D | E |
| A            | 0                | 7 | ~        | ~ | 1 |
| B            | 7                | 0 | 1        | ~ | 8 |
| C            | ~                | 1 | 0        | 2 | ~ |
| D            | ~                | ~ | 2        | 0 | 2 |
| E            | 1                | 8 | <b>4</b> | 2 | 0 |



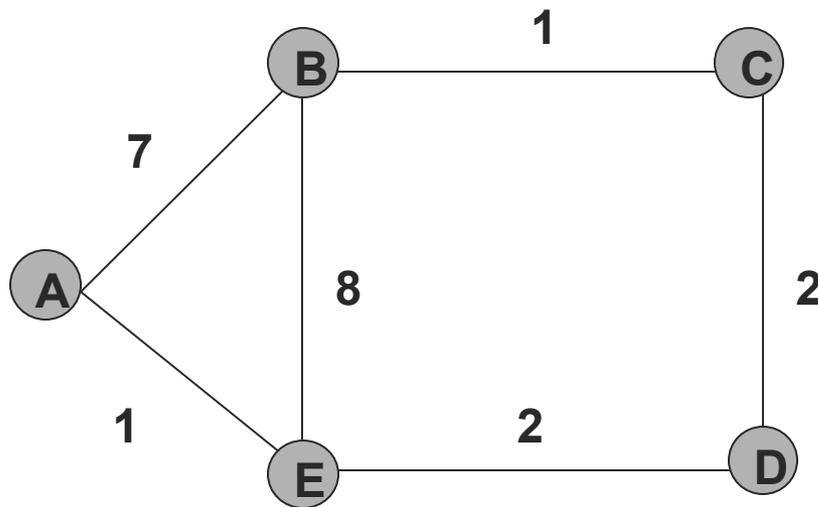
# [ A Receives B's Routes ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 7 | ~ | ~ | 1 |
| B            | 7                | 0 | 1 | ~ | 8 |
| C            | ~                | 1 | 0 | 2 | ~ |
| D            | ~                | ~ | 2 | 0 | 2 |
| E            | 1                | 8 | 4 | 2 | 0 |



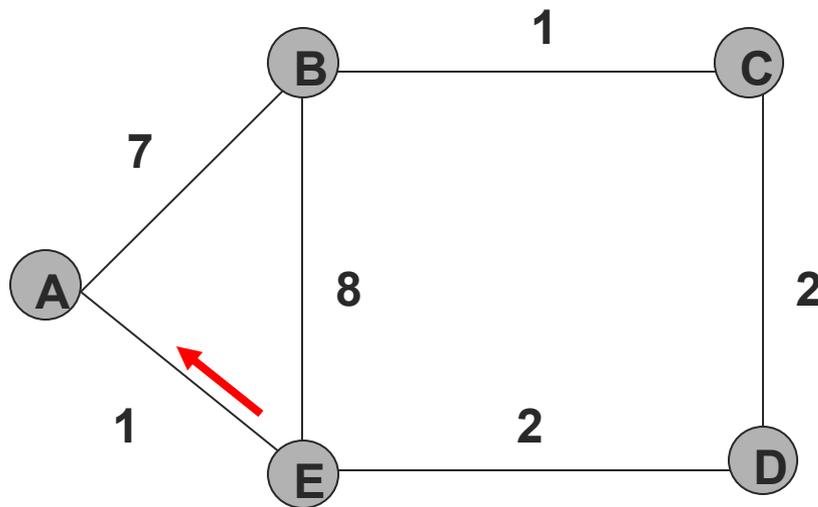
# [ A Updates Cost to C ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 7 | 8 | ~ | 1 |
| B            | 7                | 0 | 1 | ~ | 8 |
| C            | ~                | 1 | 0 | 2 | ~ |
| D            | ~                | ~ | 2 | 0 | 2 |
| E            | 1                | 8 | 4 | 2 | 0 |



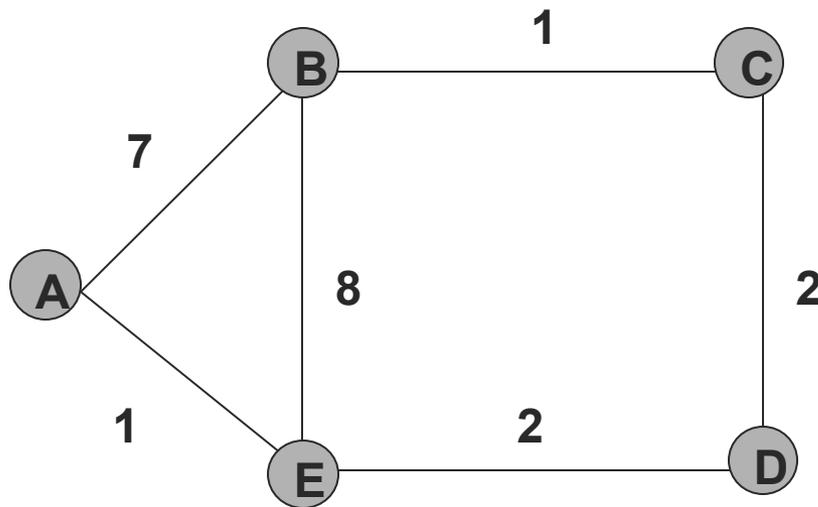
# [ A Receives E's Routes ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 7 | 8 | ~ | 1 |
| B            | 7                | 0 | 1 | ~ | 8 |
| C            | ~                | 1 | 0 | 2 | ~ |
| D            | ~                | ~ | 2 | 0 | 2 |
| E            | 1                | 8 | 4 | 2 | 0 |



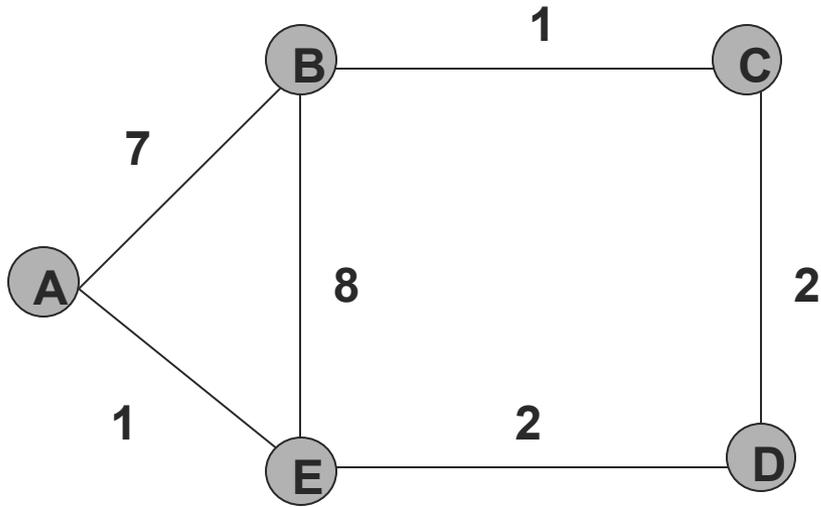
# [ A Updates Cost to C and D ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 7 | 5 | 3 | 1 |
| B            | 7                | 0 | 1 | ~ | 8 |
| C            | ~                | 1 | 0 | 2 | ~ |
| D            | ~                | ~ | 2 | 0 | 2 |
| E            | 1                | 8 | 4 | 2 | 0 |



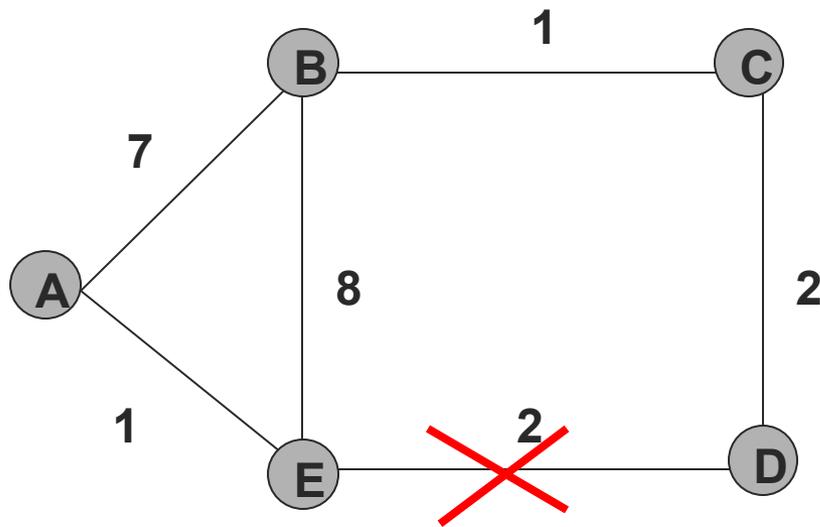
# [ Final Distances ]



| Info at node | Distance to node |   |   |   |   |
|--------------|------------------|---|---|---|---|
|              | A                | B | C | D | E |
| A            | 0                | 6 | 5 | 3 | 1 |
| B            | 6                | 0 | 1 | 3 | 5 |
| C            | 5                | 1 | 0 | 2 | 4 |
| D            | 3                | 3 | 2 | 0 | 2 |
| E            | 1                | 5 | 4 | 2 | 0 |



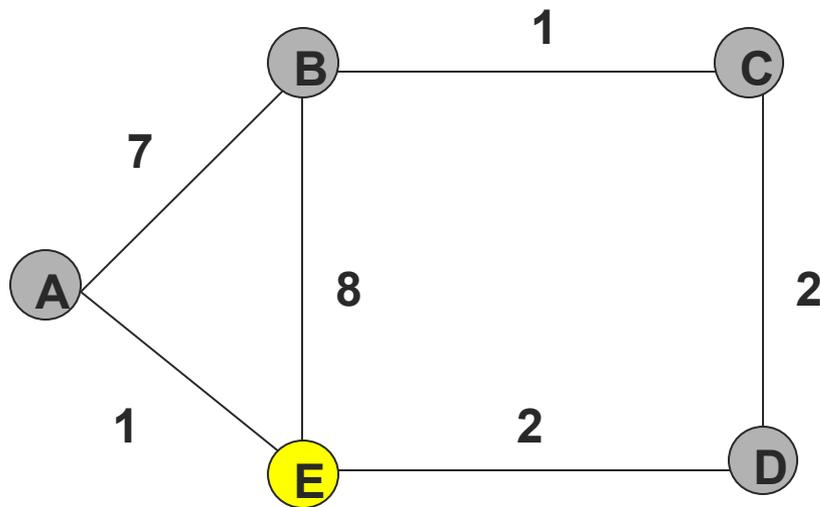
# Final Distances After Link Failure



| Info at node | Distance to node |   |   |    |    |
|--------------|------------------|---|---|----|----|
|              | A                | B | C | D  | E  |
| A            | 0                | 7 | 8 | 10 | 1  |
| B            | 7                | 0 | 1 | 3  | 8  |
| C            | 8                | 1 | 0 | 2  | 9  |
| D            | 10               | 3 | 2 | 0  | 11 |
| E            | 1                | 8 | 9 | 11 | 0  |



# View From a Node



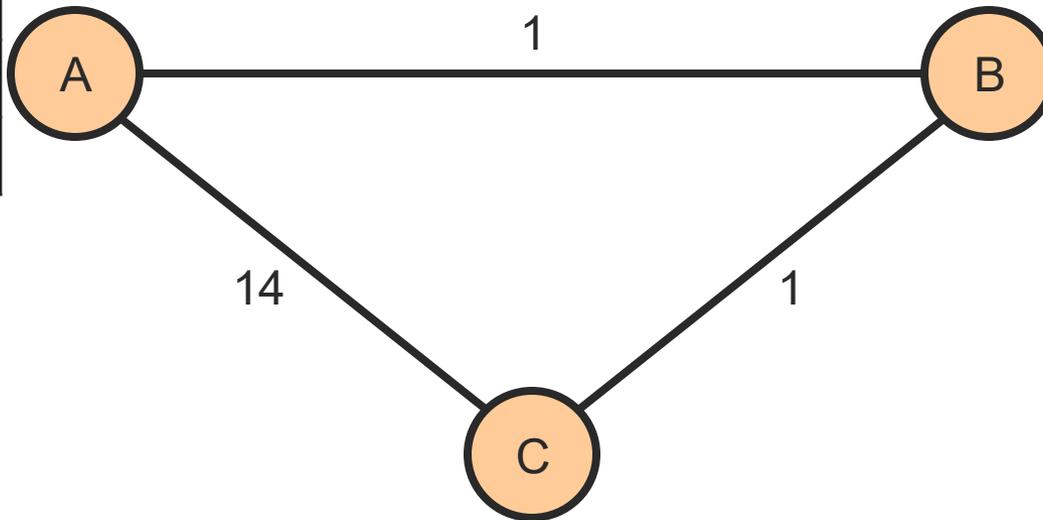
E's routing table

| dest | Next hop |    |   |
|------|----------|----|---|
|      | A        | B  | D |
| A    | 1        | 14 | 5 |
| B    | 7        | 8  | 5 |
| C    | 6        | 9  | 4 |
| D    | 4        | 11 | 2 |



# What happens after a failure?

| dest | cost |
|------|------|
| B    | 1    |
| C    | 2    |



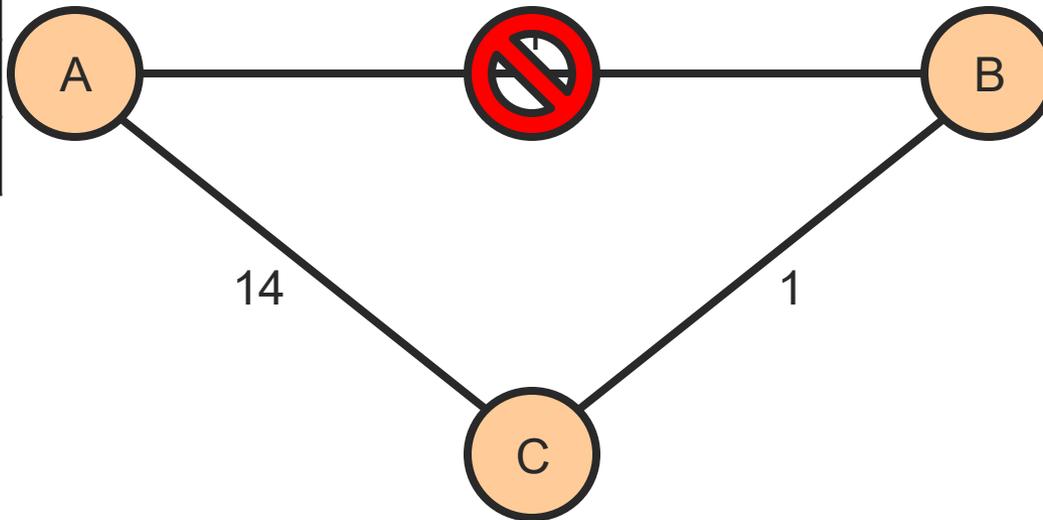
| dest | cost |
|------|------|
| A    | 1    |
| C    | 1    |

| dest | cost |
|------|------|
| A    | 2    |
| B    | 1    |



# Count-to-Infinity Problem

| dest | cost |
|------|------|
| B    | 1    |
| C    | 2    |



| dest | cost |
|------|------|
| A    | 1    |
| C    | 1    |

| dest | cost |
|------|------|
| A    | 2    |
| B    | 1    |

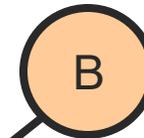
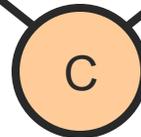


# Count-to-Infinity Problem

| dest | cost |
|------|------|
| B    | 1    |
| C    | 2    |



14



| dest | cost |
|------|------|
| A    | 1    |
| C    | 1    |

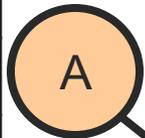
1

| dest | cost |
|------|------|
| A    | 2    |
| B    | 1    |

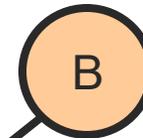
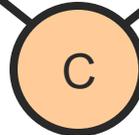


# Count-to-Infinity Problem

| dest | cost |
|------|------|
| B    | 1    |
| C    | 2    |



14



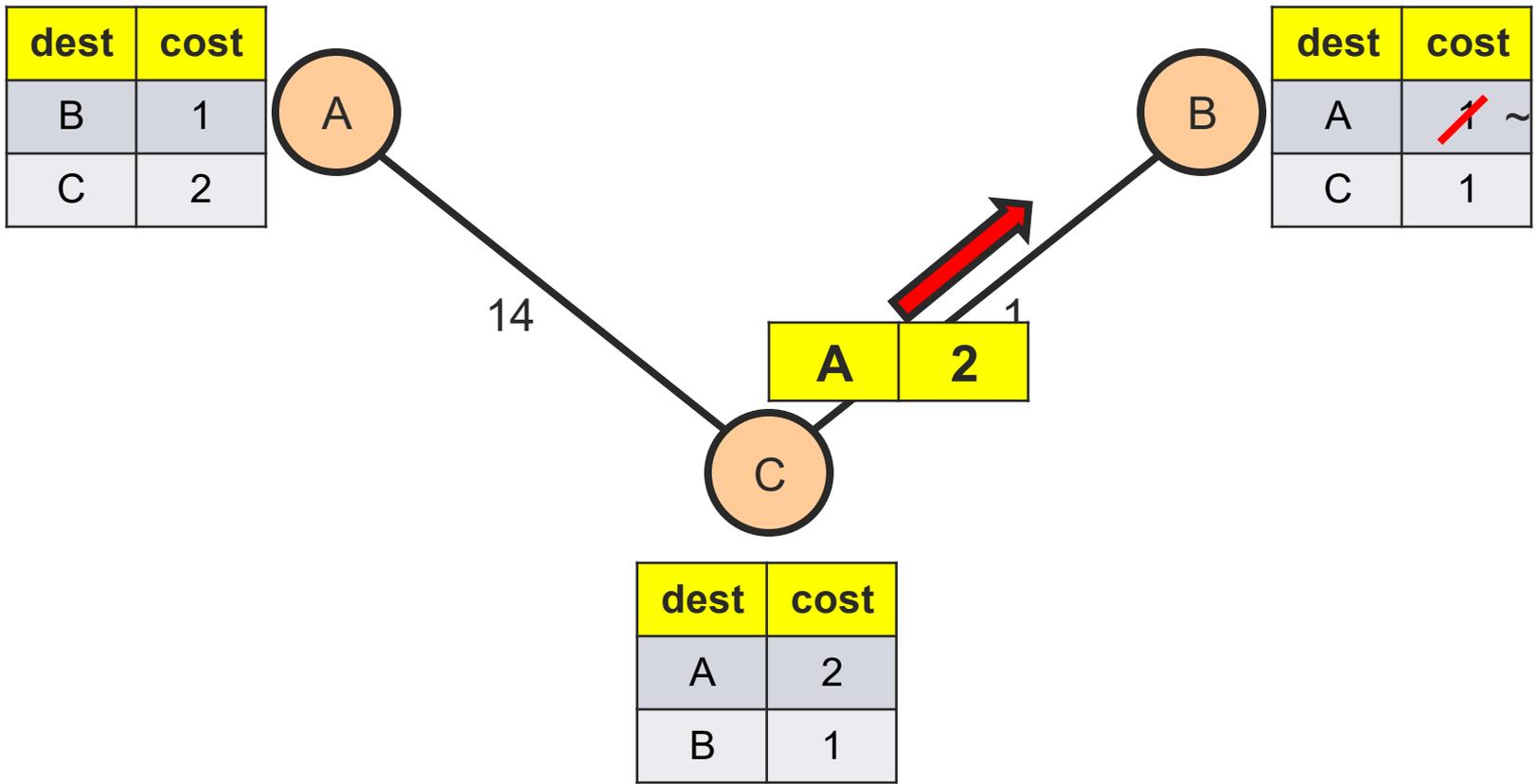
| dest | cost           |
|------|----------------|
| A    | <del>1</del> ~ |
| C    | 1              |

1

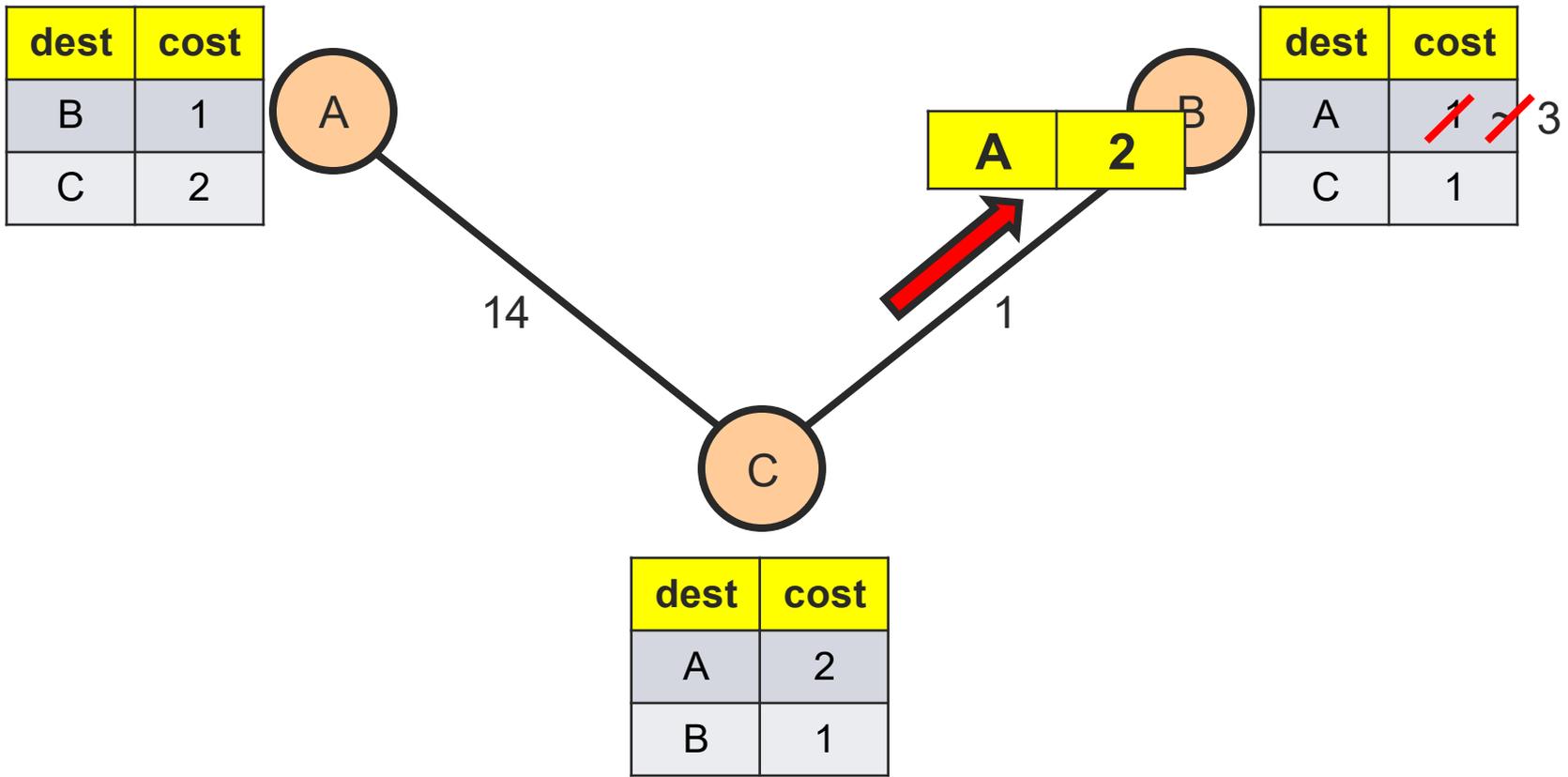
| dest | cost |
|------|------|
| A    | 2    |
| B    | 1    |



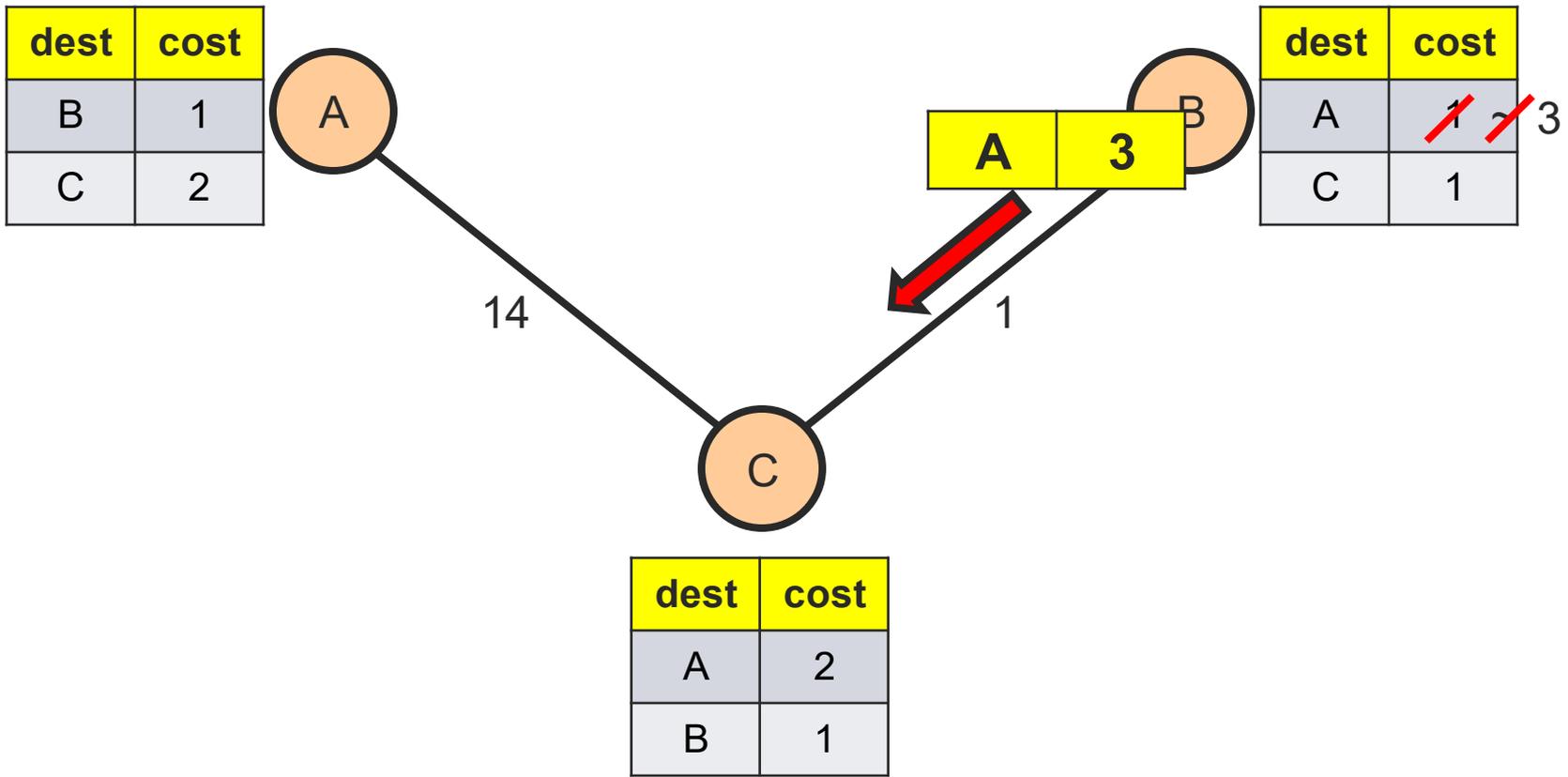
# Count-to-Infinity Problem



# Count-to-Infinity Problem



# Count-to-Infinity Problem



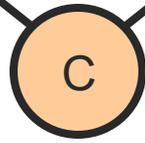
# Count-to-Infinity Problem

Really through B

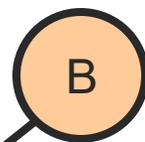
| dest | cost |
|------|------|
| B    | 1    |
| C    | 2    |



14



| A | 3 |
|---|---|
|---|---|

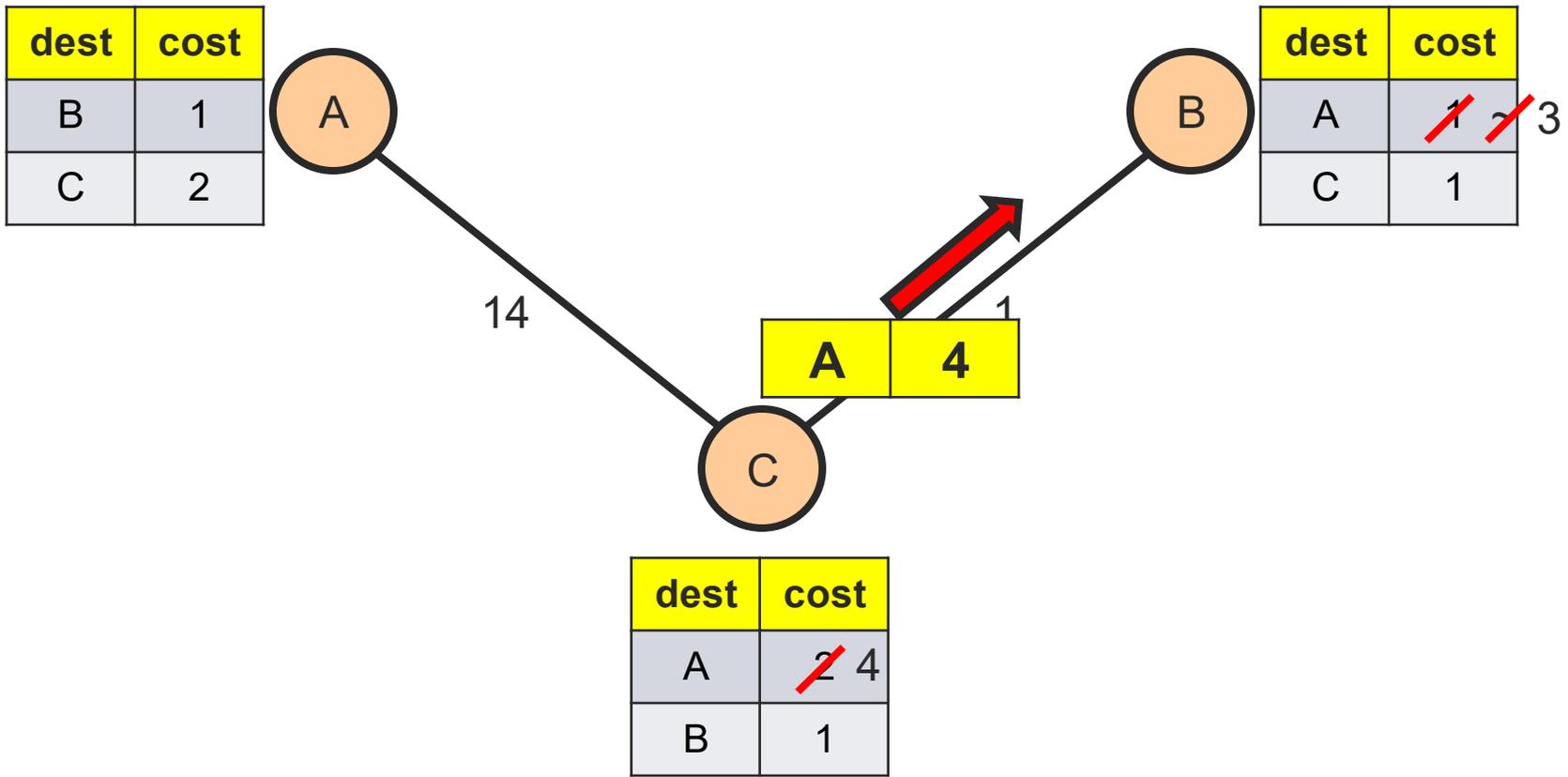


| dest | cost           |
|------|----------------|
| A    | <del>1</del> 3 |
| C    | 1              |

| dest | cost           |
|------|----------------|
| A    | <del>2</del> 4 |
| B    | 1              |

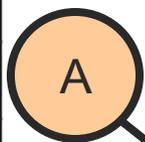


# Count-to-Infinity Problem

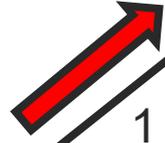
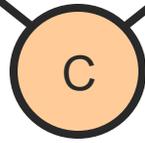


# Count-to-Infinity Problem

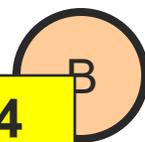
| dest | cost |
|------|------|
| B    | 1    |
| C    | 2    |



14



1



|   |   |
|---|---|
| A | 4 |
|---|---|

| dest | cost                                     |
|------|--|
| A    | <del>1</del> <del>2</del> <del>3</del> 5 |
| C    | 1  |

Also called the bouncing effect

| dest | cost           |
|------|----------------|
| A    | <del>2</del> 4 |
| B    | 1              |



# [ Distance Vector Routing ]

## ■ Problem

- Node **X** notices that its link to **Y** is broken
- Other nodes believe that the route through **X** is still good
- Mutual deception!



# How Are These Loops Caused?

- Observation 1:
  - B's metric **increases**
- Observation 2:
  - C picks B as next hop to A
  - But, the **implicit path** from C to A includes itself!



# [ Solution 1: Holddowns ]

- If metric increases, delay propagating information
  - in our example, B delays advertising route
  - C eventually thinks B's route is gone, picks its own route
  - B then selects C as next hop
- Adversely affects convergence



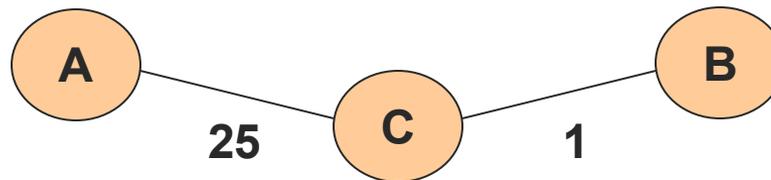
# Heuristics for breaking loops

- Set infinity to 16
  - Small limit allows fast completion of “counting to infinity”
  - Limits the size of the network
- Split horizon
  - Avoid counting to infinity by solving “mutual deception” problem
- Split horizon with poisoned reverse
  - “Poison” the routes sent to you by your neighbors
- Sequence numbers on delay estimates



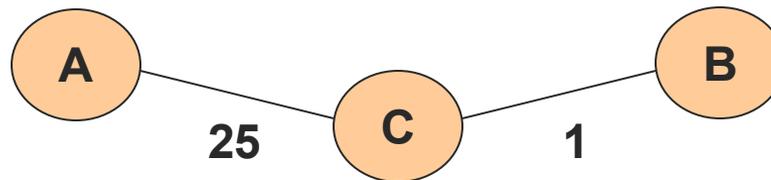
# [ Split Horizon ]

- Avoid counting to infinity by solving “mutual deception” problem
- Distance Vector with split horizon:
  - when sending an update to node **X**, do not include destinations that you would route through **X**
  - If **X** thinks route is not through you, no effect
  - If **X** thinks route is through you, **X** will timeout route



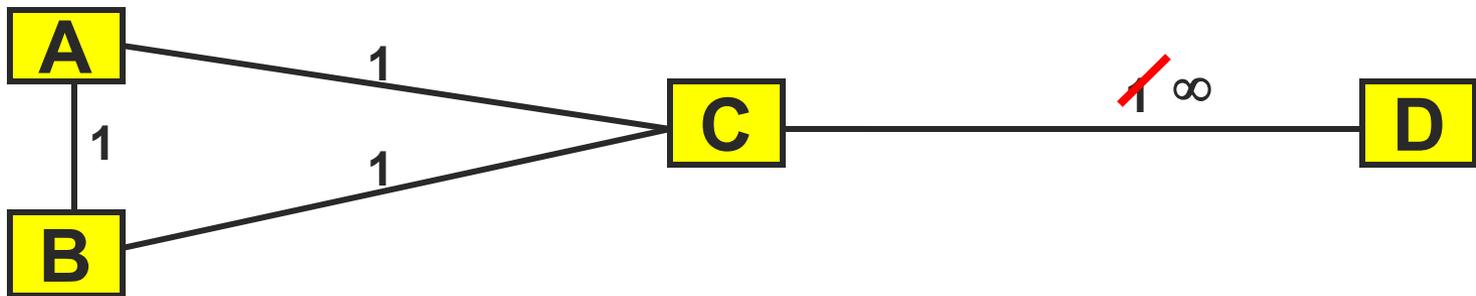
# Split Horizon and Poisoned Reverse

- Distance Vector with Split Horizon and Poisoned Reverse:
  - When sending update to node **X**, include destinations that you would route through **X** with distance set to infinity
  - Don't need to wait for **X** to timeout
- Problem:
  - still doesn't fix loops of 3+ hops!



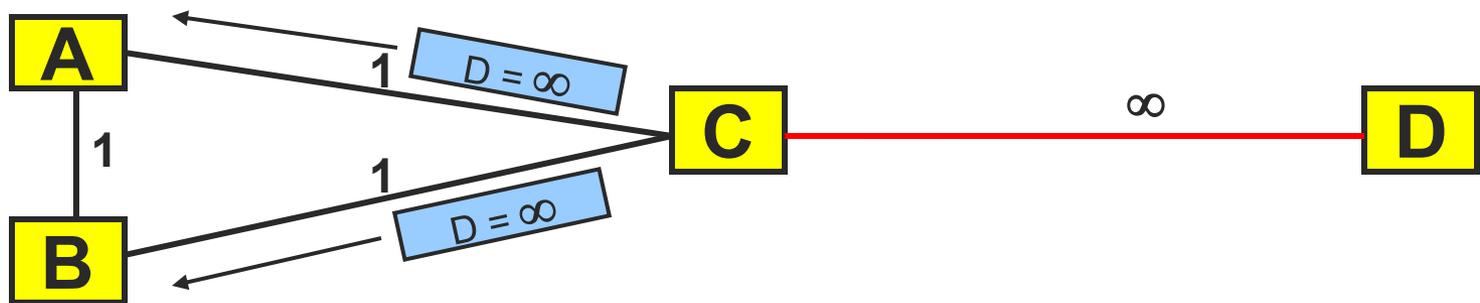
# [ Split Horizon ]

- Split Horizon (with or without poisoned reverse) may still allow some routing loops and counting to infinity
  - guarantees no 2-node loops
  - can still be fooled by 3-node (or larger) loops
- Consider link failure from **C** to **D**



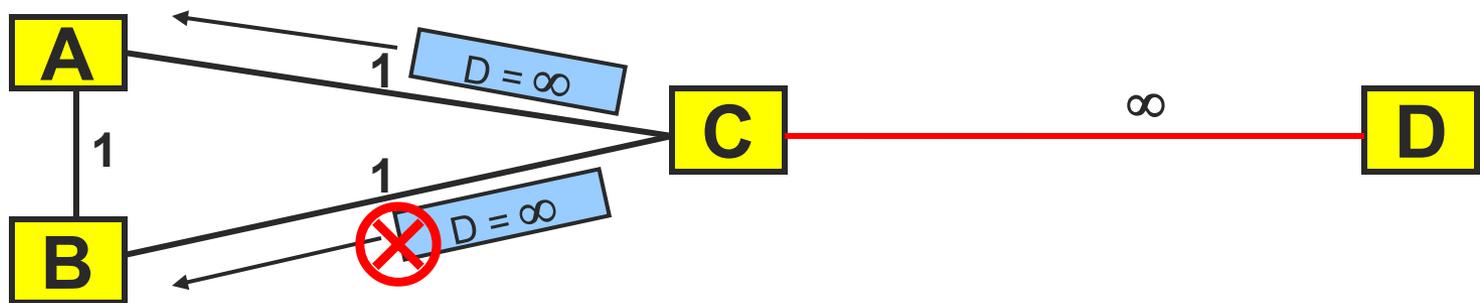
# [ Split Horizon ]

- Initial routing table entries for route to **D**:
  - A** 2 via **C**
  - B** 2 via **C**
  - C** 1
- **C** notices link failure and changes to infinity
- Now **C** sends updates to **A** and **B**:
  - to **A**: infinity
  - to **B**: infinity



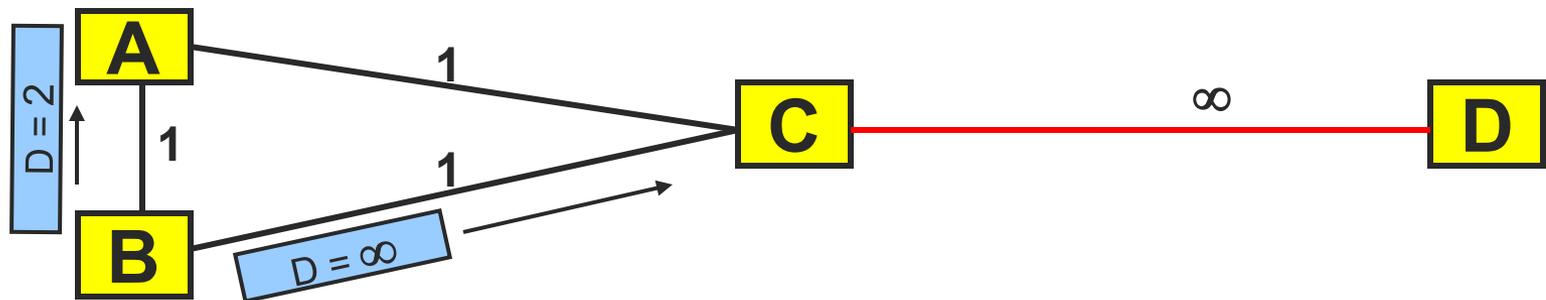
# [ Split Horizon ]

- Suppose update to **B** is lost
- New tables:  
A unreachable  
B 2 via C  
C unreachable



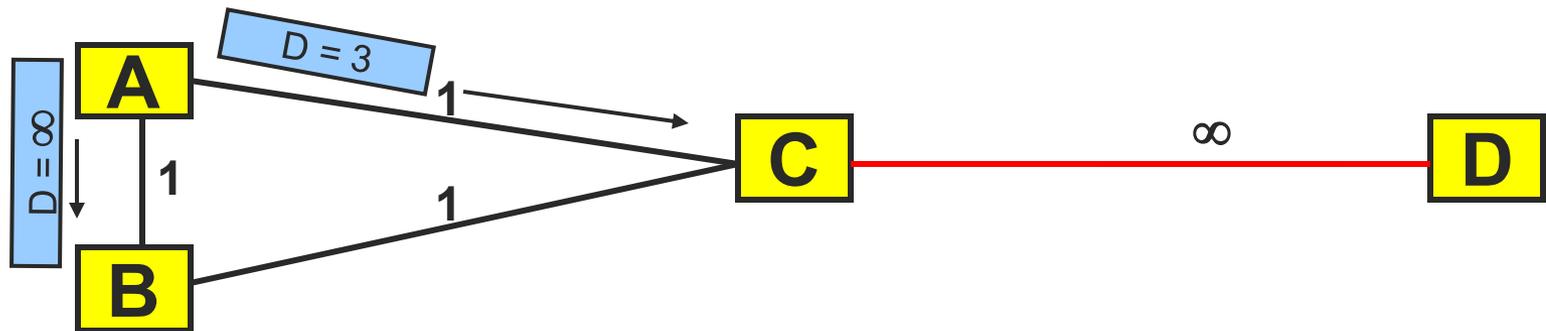
# [ Split Horizon ]

- Suppose update to **B** is lost
- New tables:
  - A** unreachable
  - B** 2 via **C**
  - C** unreachable
- Now **B** sends its periodic routing update:
  - to **C**: infinity (poisoned reverse)
  - to **A**: 2



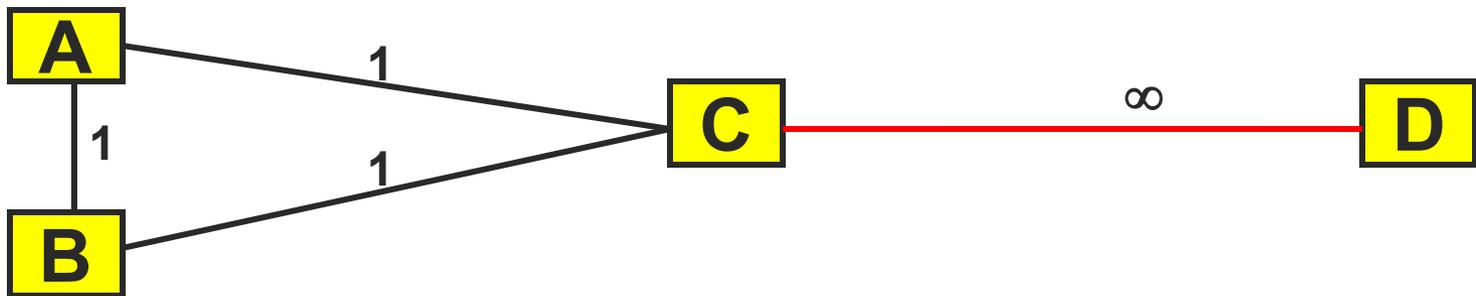
# [ Split Horizon ]

- New tables for route to **D**:
  - A** 3 via **B**
  - B** 2 via **C**
  - C** unreachable
- Finally **A** sends its periodic routing update:
  - to **B**: infinity (poisoned reverse)
  - to **C**: 3



# [ Split Horizon ]

- New tables for route to **D**:
  - A** 3 via **B**
  - B** 2 via **C**
  - C** 4 via **A**
- A, B and C will still continue to count to infinity



# Avoiding the Counting to Infinity Problem

- Select loop-free **paths**
- One way of doing this:
  - Each route advertisement carries entire path instead of just distance
  - If router sees itself in path, reject route
  - $\Rightarrow$  called Path-Vector routing
- BGP does it this way
- Space proportional to diameter



# Loop Freedom at Every Instant

- Have we now avoided all loops?
  - No! **Transient** loops are still possible
  - Why? Implicit path information may be stale
- Many approaches to fix this
  - Maintain backup paths in case you get stuck
  - Use multiple paths
  - Source routing
  - Keep packets flowing or queued during convergence
  - ...and much more current research



# Distance Vector in Practice

- RIP and RIP2
  - uses split-horizon/poison reverse
- BGP/IDRP
  - propagates entire path
  - path also used for affecting policies
- AODV
  - “on-demand” protocol for wireless networks
  - Only maintain distance vectors along paths to destinations that you need to reach



# [ Routing So Far ... ]

- Problem
  - Information propagates slowly
    - One period per hop for new routes
    - Count to infinity to detect lost routes



# [ Dijkstra's Algorithm ]

- Given
  - Directed graph with edge weights (distances)
- Calculate
  - Shortest paths from one node to all others

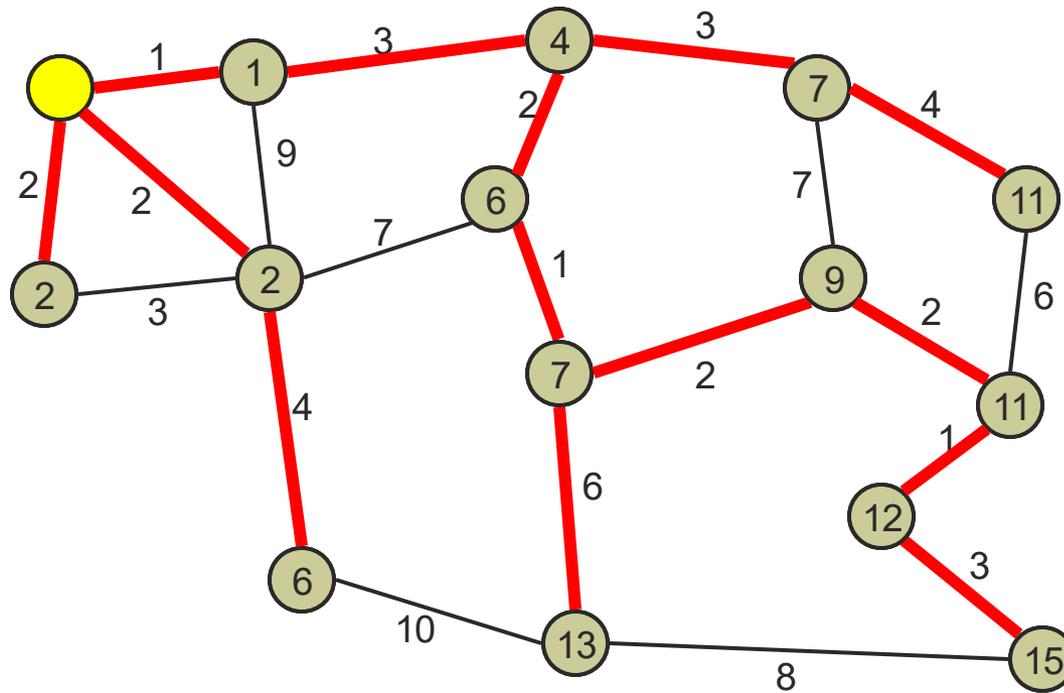


# Dijkstra's Algorithm

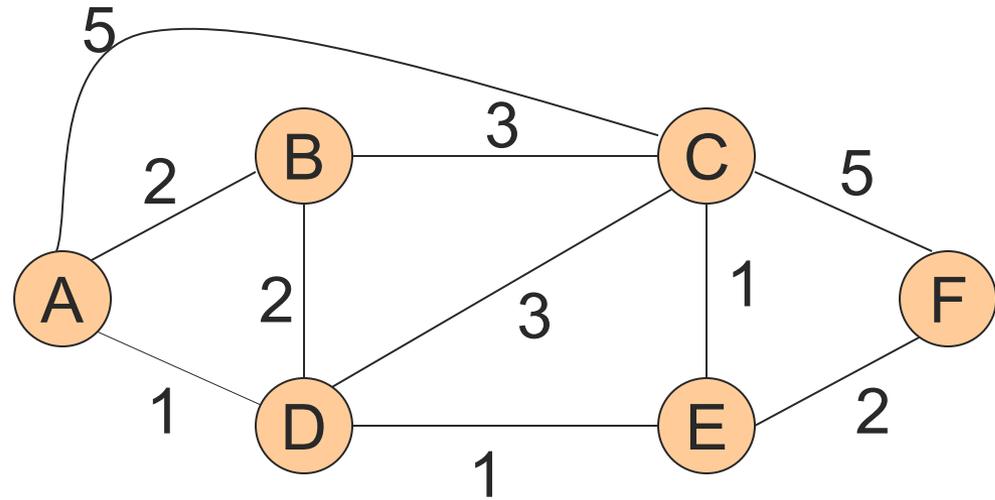
- Greedily grow set C of confirmed least cost paths
- Initially  $C = \{\text{source}\}$
- Loop N-1 times
  - Determine the node M outside C that is closest to the source
  - Add M to C and update costs for each node P outside C
    - Is the path (source  $\rightarrow \dots \rightarrow M \rightarrow P$ ) better than the previously known path for (source  $\rightarrow P$ )?
    - If YES
      - Update cost to reach P



# [ Dijkstra's Algorithm ]



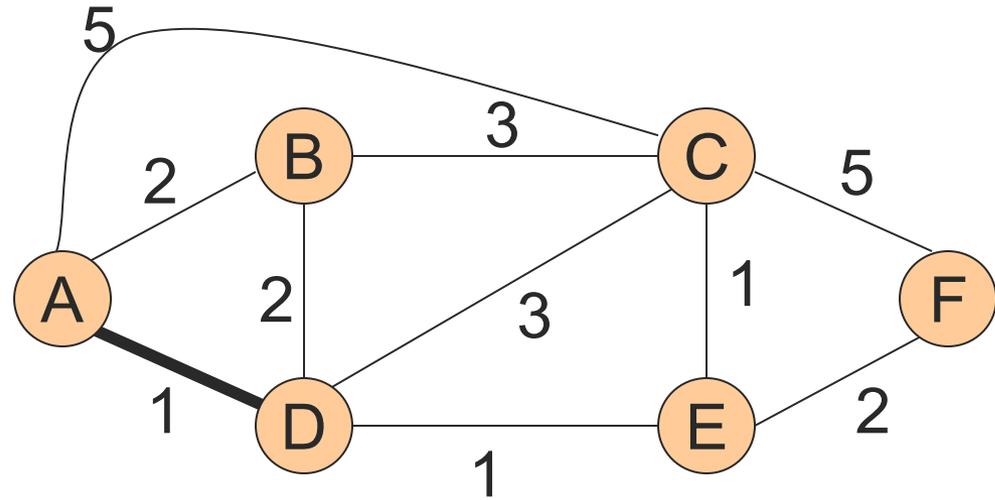
# [ Example ]



|      |     | B          | C          | D          | E          | F          |
|------|-----|------------|------------|------------|------------|------------|
| step | SPT | D(b), P(b) | D(c), P(c) | D(d), P(d) | D(e), P(e) | D(f), P(f) |
| 0    | A   | 2, A       | 5, A       | 1, A       | ~          | ~          |



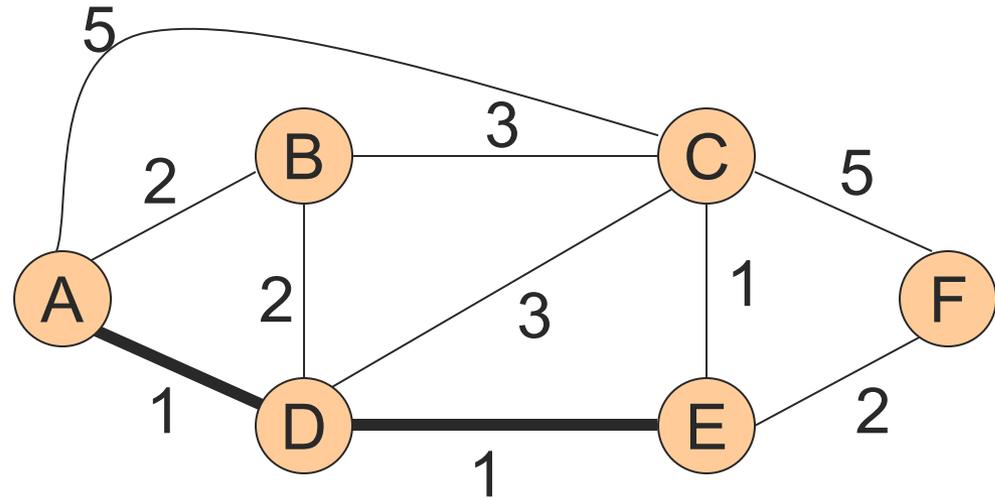
# [ Example ]



|      |     | B          | C          | D           | E          | F          |
|------|-----|------------|------------|-------------|------------|------------|
| step | SPT | D(b), P(b) | D(c), P(c) | D(d), P(d)  | D(e), P(e) | D(f), P(f) |
| 0    | A   | 2, A       | 5, A       | <b>1, A</b> | ~          | ~          |
| 1    | AD  | 2, A       | 4, D       |             | 2, D       | ~          |



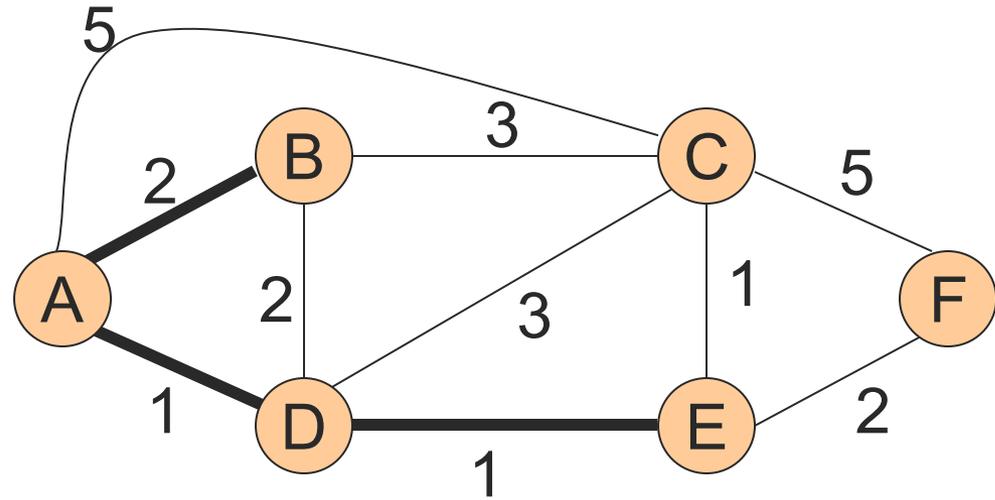
# [ Example ]



|      |     | B          | C          | D           | E           | F          |
|------|-----|------------|------------|-------------|-------------|------------|
| step | SPT | D(b), P(b) | D(c), P(c) | D(d), P(d)  | D(e), P(e)  | D(f), P(f) |
| 0    | A   | 2, A       | 5, A       | <b>1, A</b> | ~           | ~          |
| 1    | AD  | 2, A       | 4, D       |             | <b>2, D</b> | ~          |
| 2    | ADE | 2, A       | 3, E       |             |             | 4, E       |



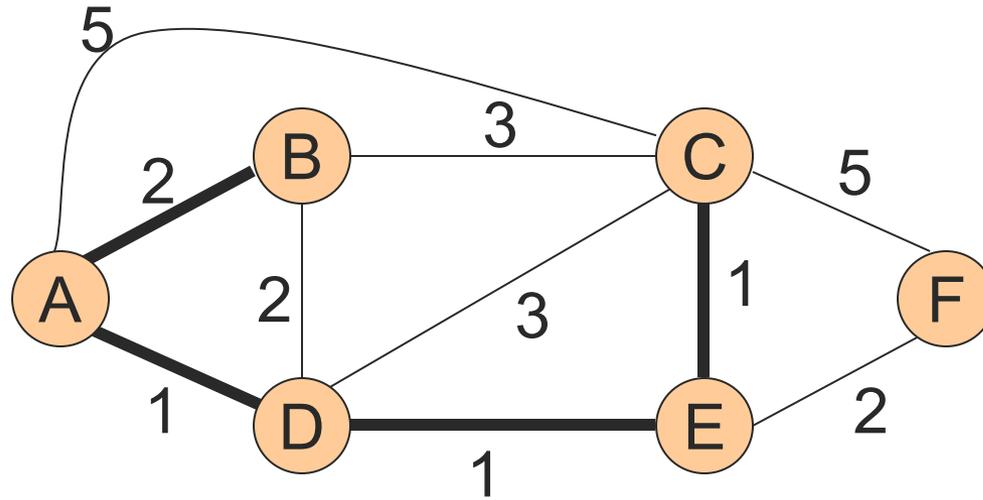
# [ Example ]



|      |      | B           | C          | D           | E           | F          |
|------|------|-------------|------------|-------------|-------------|------------|
| step | SPT  | D(b), P(b)  | D(c), P(c) | D(d), P(d)  | D(e), P(e)  | D(f), P(f) |
| 0    | A    | 2, A        | 5, A       | <b>1, A</b> | ~           | ~          |
| 1    | AD   | 2, A        | 4, D       |             | <b>2, D</b> | ~          |
| 2    | ADE  | <b>2, A</b> | 3, E       |             |             | 4, E       |
| 3    | ADEB |             | 3, E       |             |             | 4, E       |



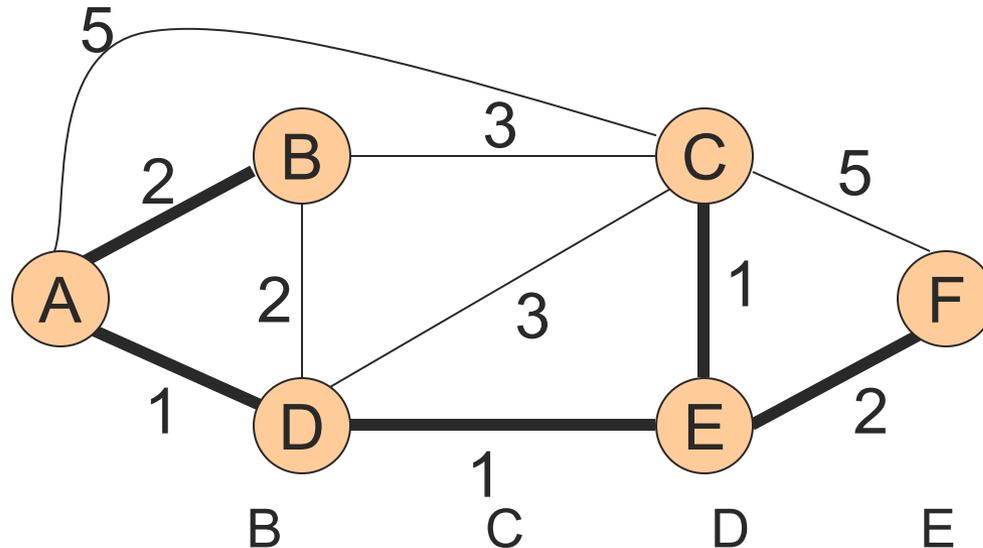
# [ Example ]



|      |       | B           | C           | D           | E           | F          |
|------|-------|-------------|-------------|-------------|-------------|------------|
| step | SPT   | D(b), P(b)  | D(c), P(c)  | D(d), P(d)  | D(e), P(e)  | D(f), P(f) |
| 0    | A     | 2, A        | 5, A        | <b>1, A</b> | ~           | ~          |
| 1    | AD    | 2, A        | 4, D        |             | <b>2, D</b> | ~          |
| 2    | ADE   | <b>2, A</b> | 3, E        |             |             | 4, E       |
| 3    | ADEB  |             | <b>3, E</b> |             |             | 4, E       |
| 4    | ADEBC |             |             |             |             | 4, E       |



# Example



| step | SPT   | B           | C           | D           | E           | F           |
|------|-------|-------------|-------------|-------------|-------------|-------------|
|      | SPT   | D(b), P(b)  | D(c), P(c)  | D(d), P(d)  | D(e), P(e)  | D(f), P(f)  |
| 0    | A     | 2, A        | 5, A        | <b>1, A</b> | ~           | ~           |
| 1    | AD    | 2, A        | 4, D        |             | <b>2, D</b> | ~           |
| 2    | ADE   | <b>2, A</b> | 3, E        |             |             | 4, E        |
| 3    | ADEB  |             | <b>3, E</b> |             |             | 4, E        |
| 4    | ADEBC |             |             |             |             | <b>4, E</b> |

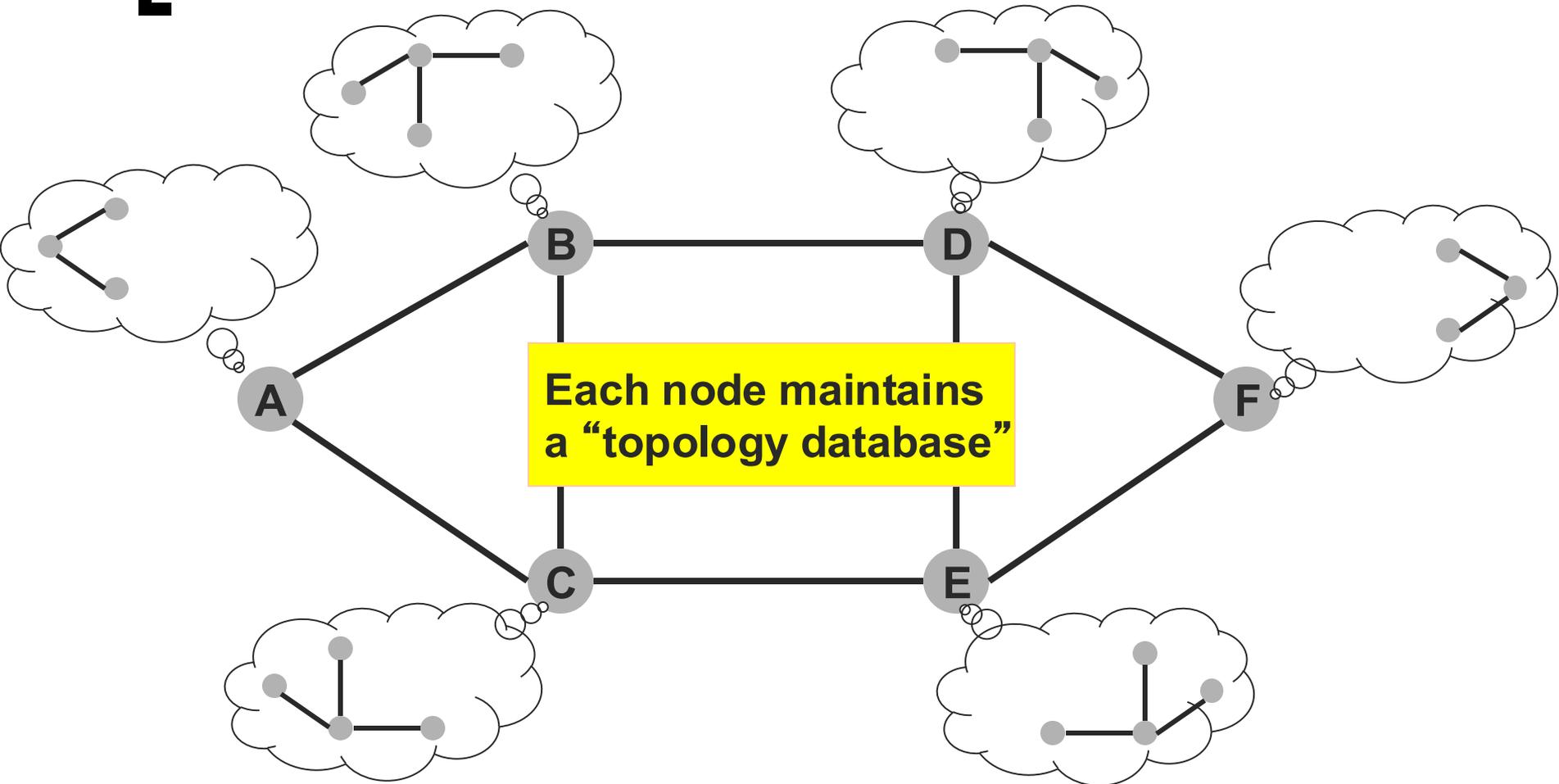


# [ Link State Routing ]

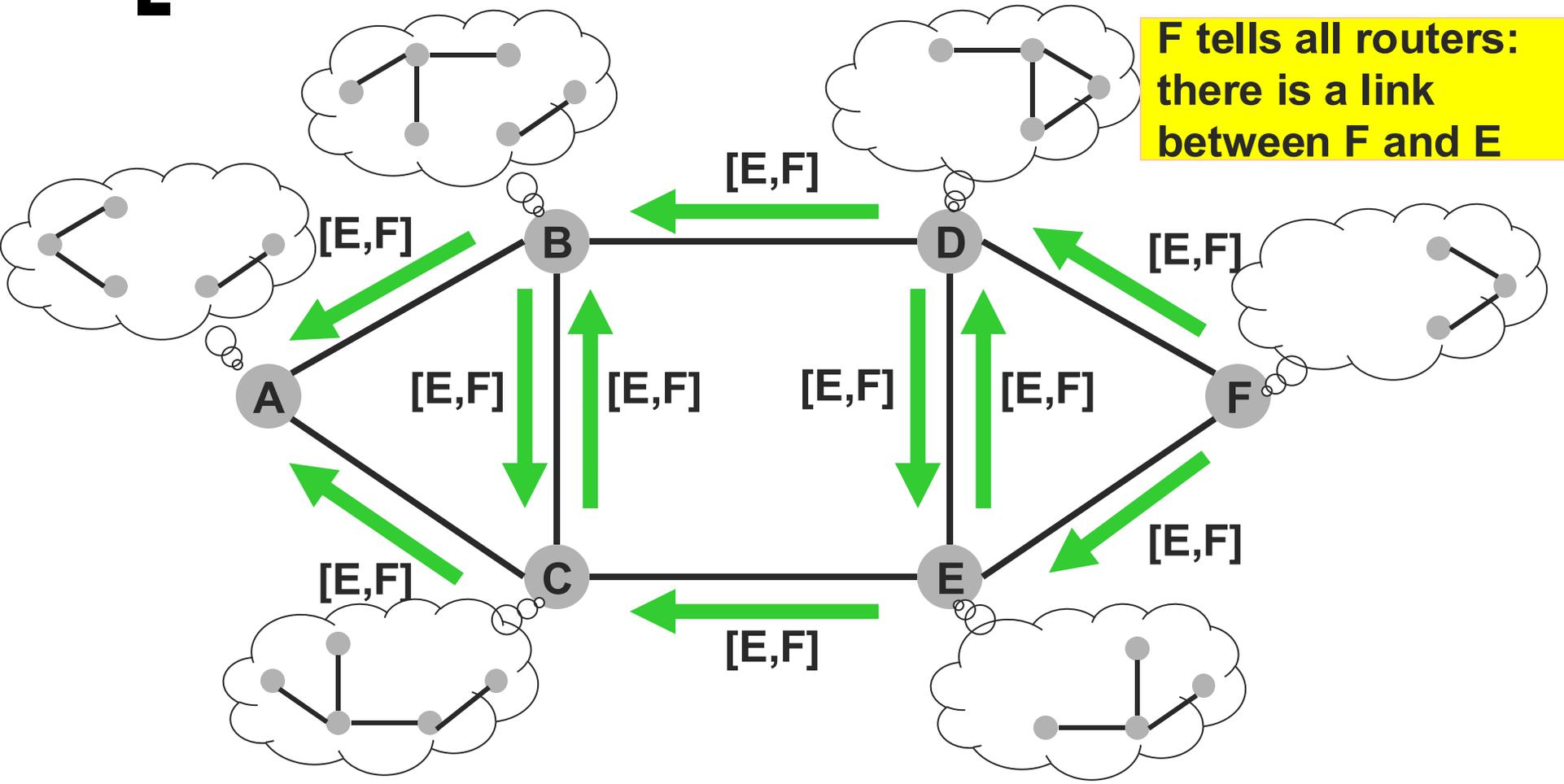
- Strategy
  - Send all nodes information about directly connected links
  - Status of links is flooded in link state packets (LSPs)
- Each LSP carries
  - ID of node that created the LSP
  - Vector of <neighbor, cost of link to neighbor> pairs for the node that created the LSP
  - Sequence number
  - Time-to-live (TTL)
- Each node maintains a list of (ideally all) LSP's and runs Dijkstra's algorithm on the list



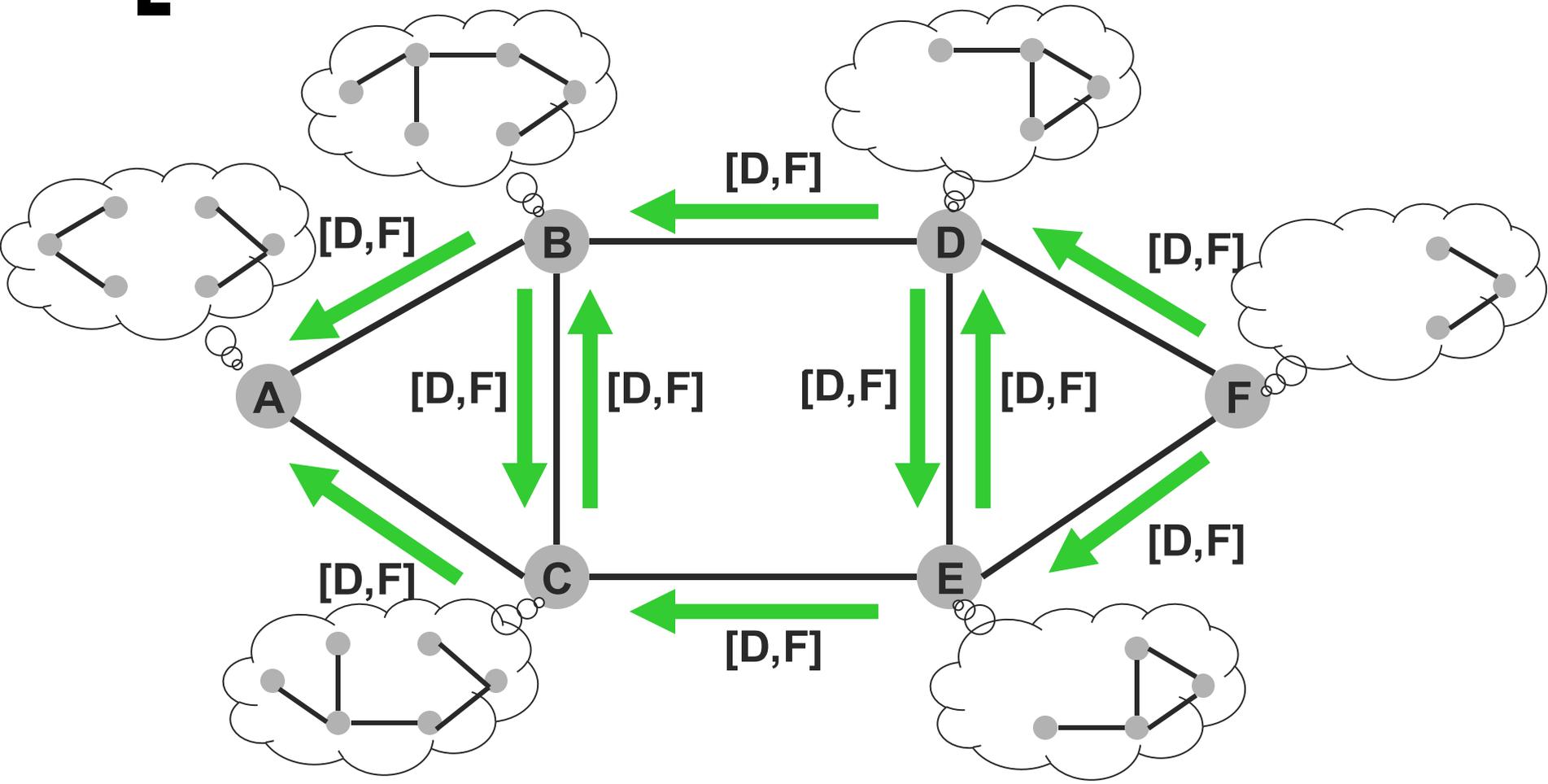
# [ Link state: update propagation ]



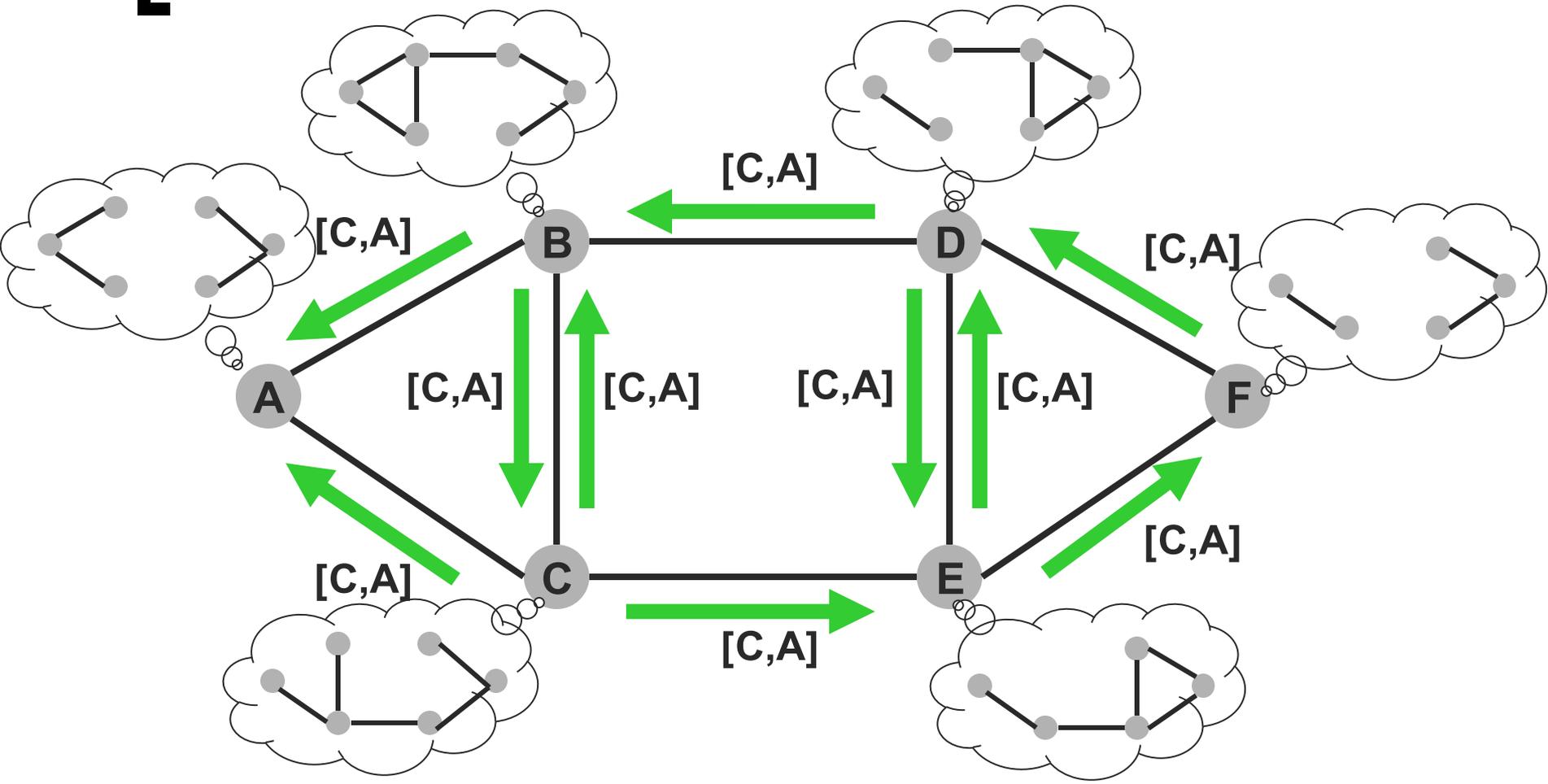
# [ Link state: update propagation ]



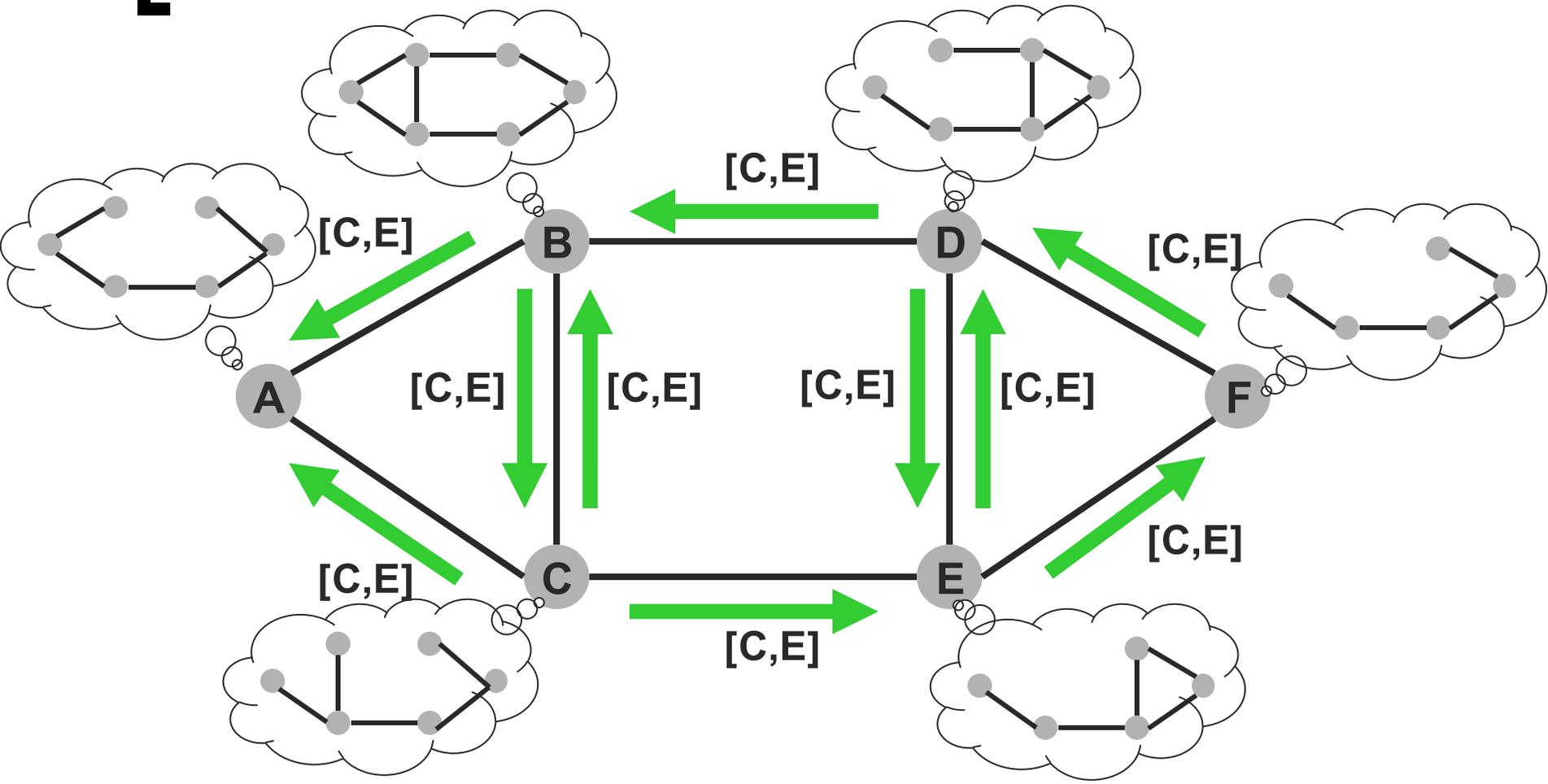
# Link state: update propagation



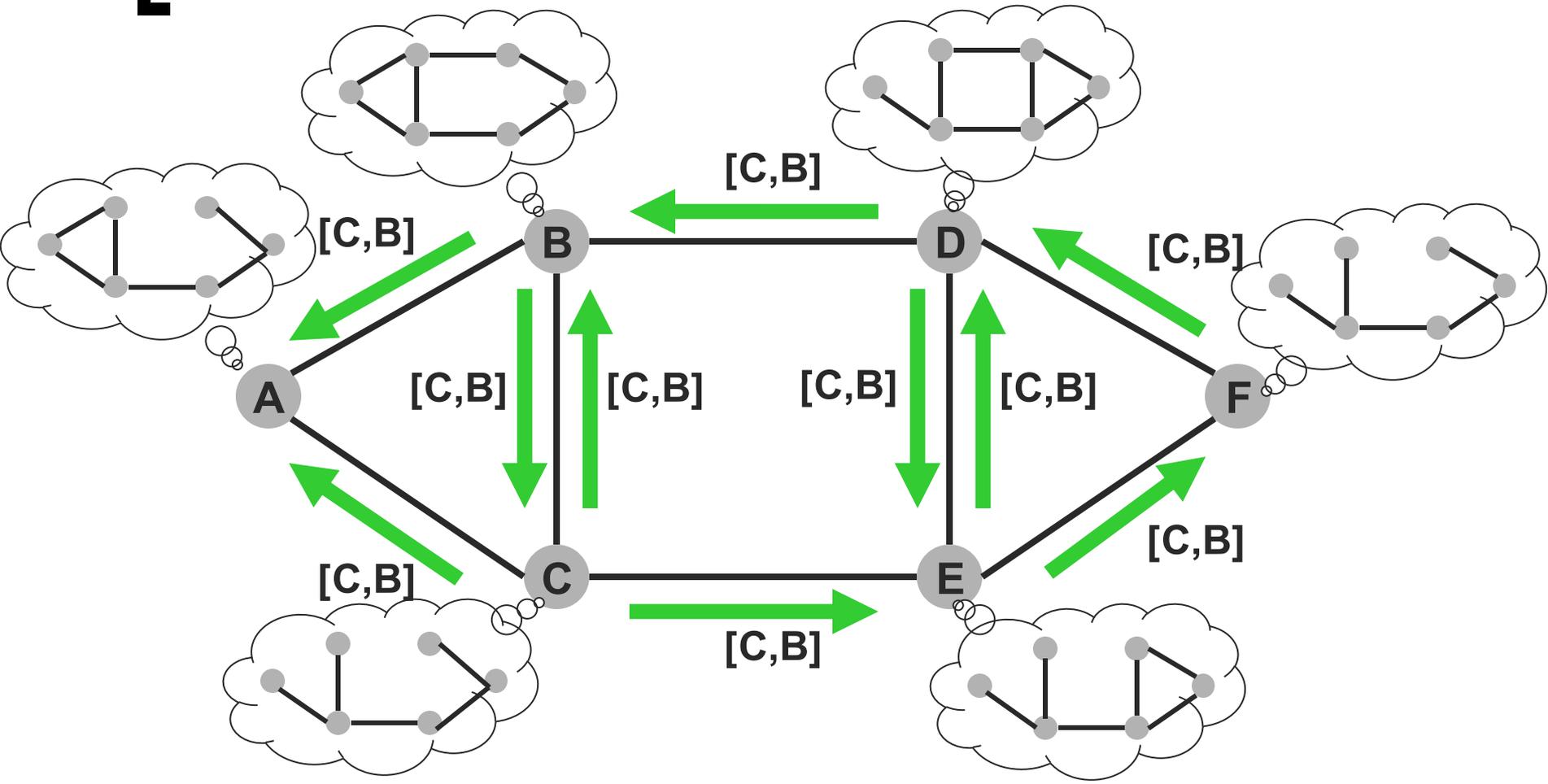
# [ Link state: update propagation ]



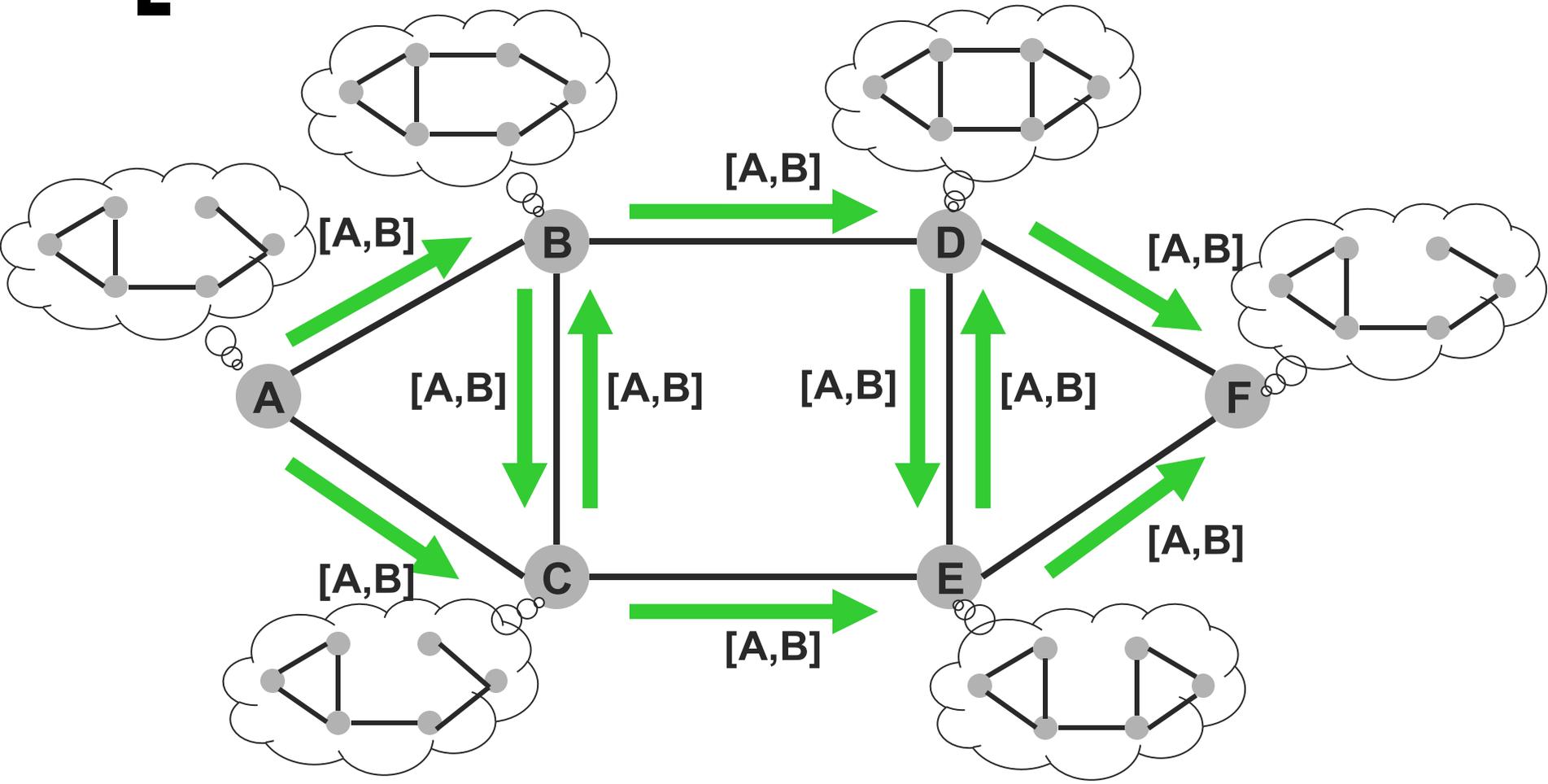
# Link state: update propagation



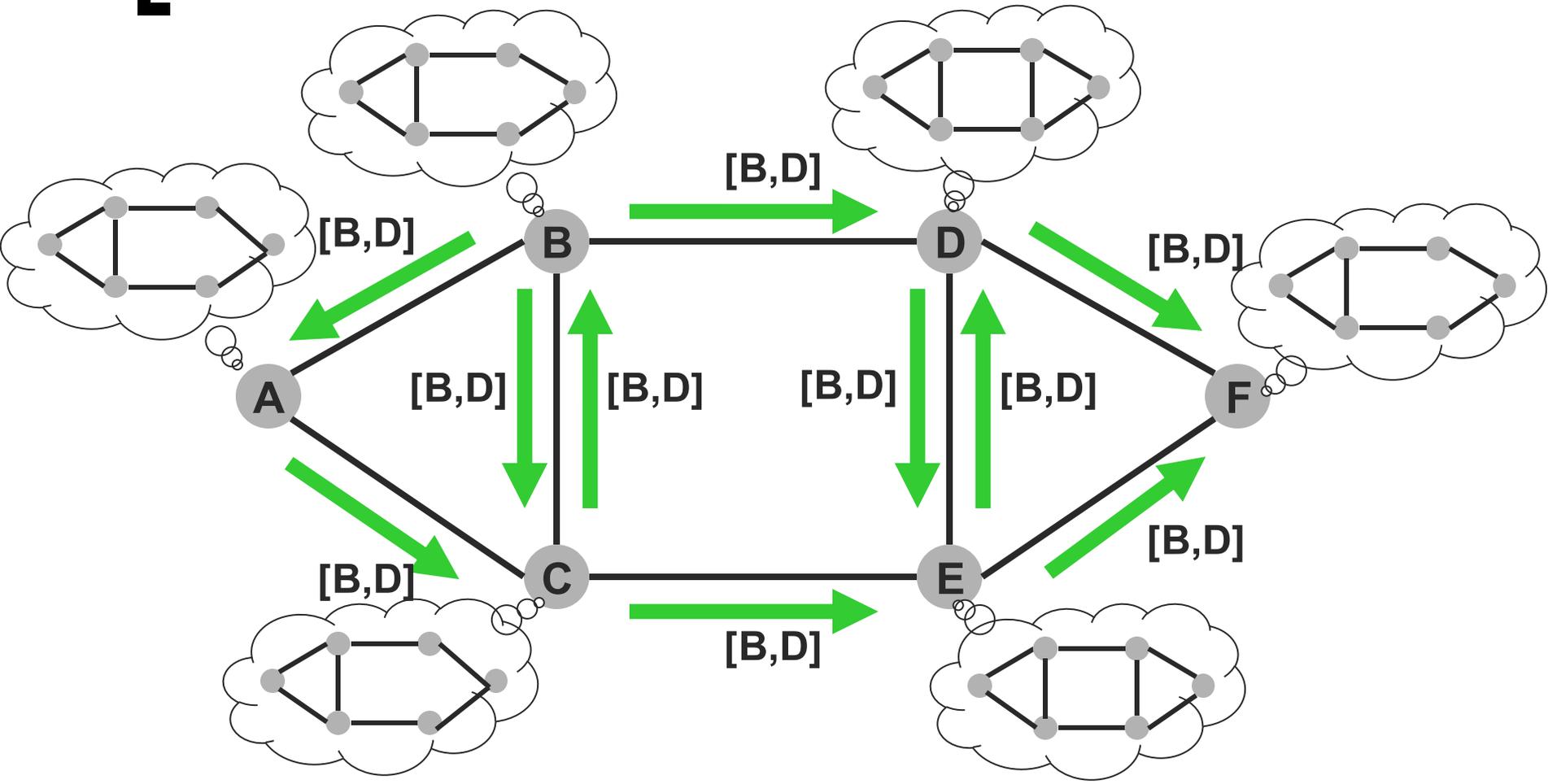
# Link state: update propagation



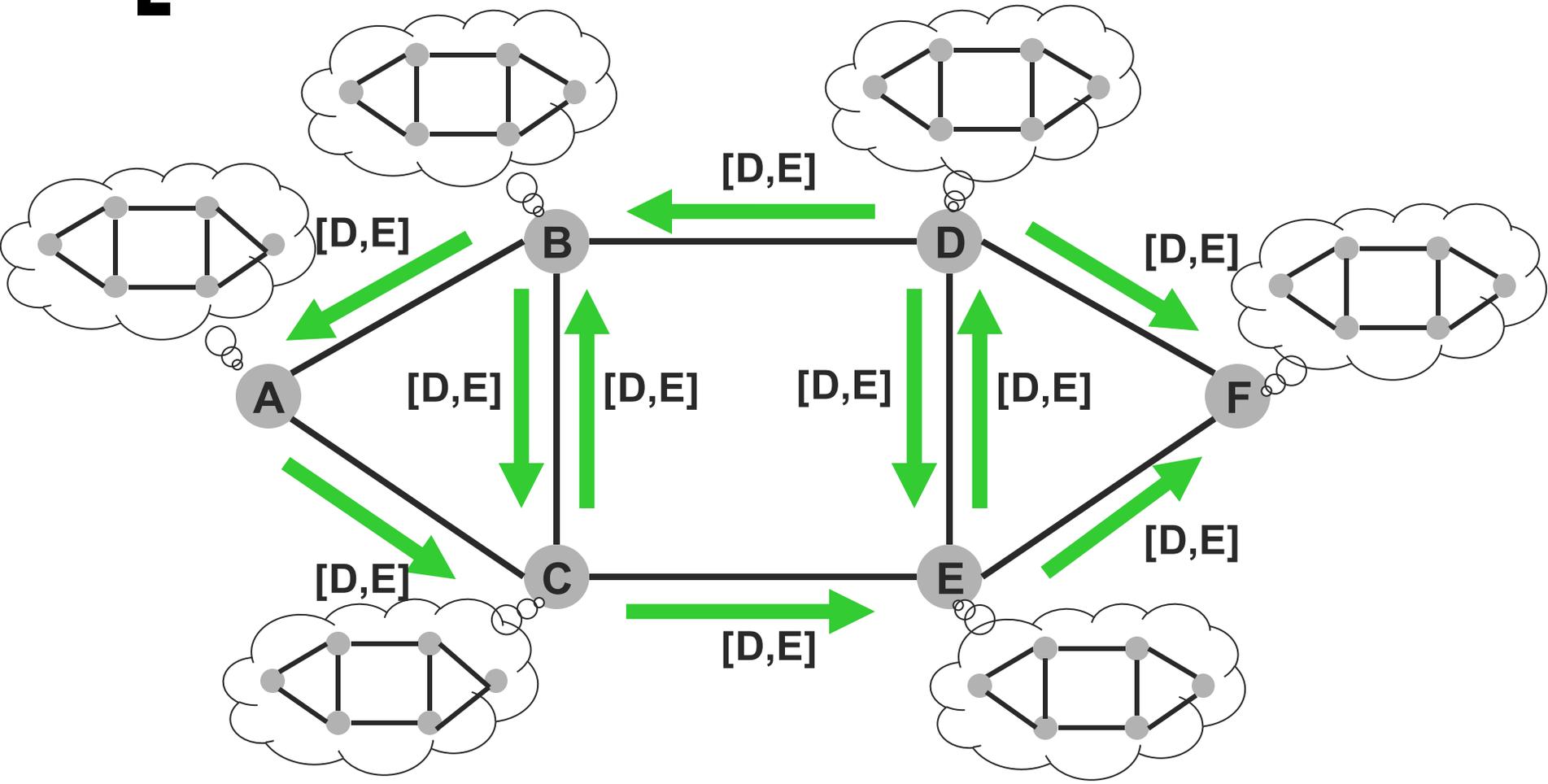
# Link state: update propagation



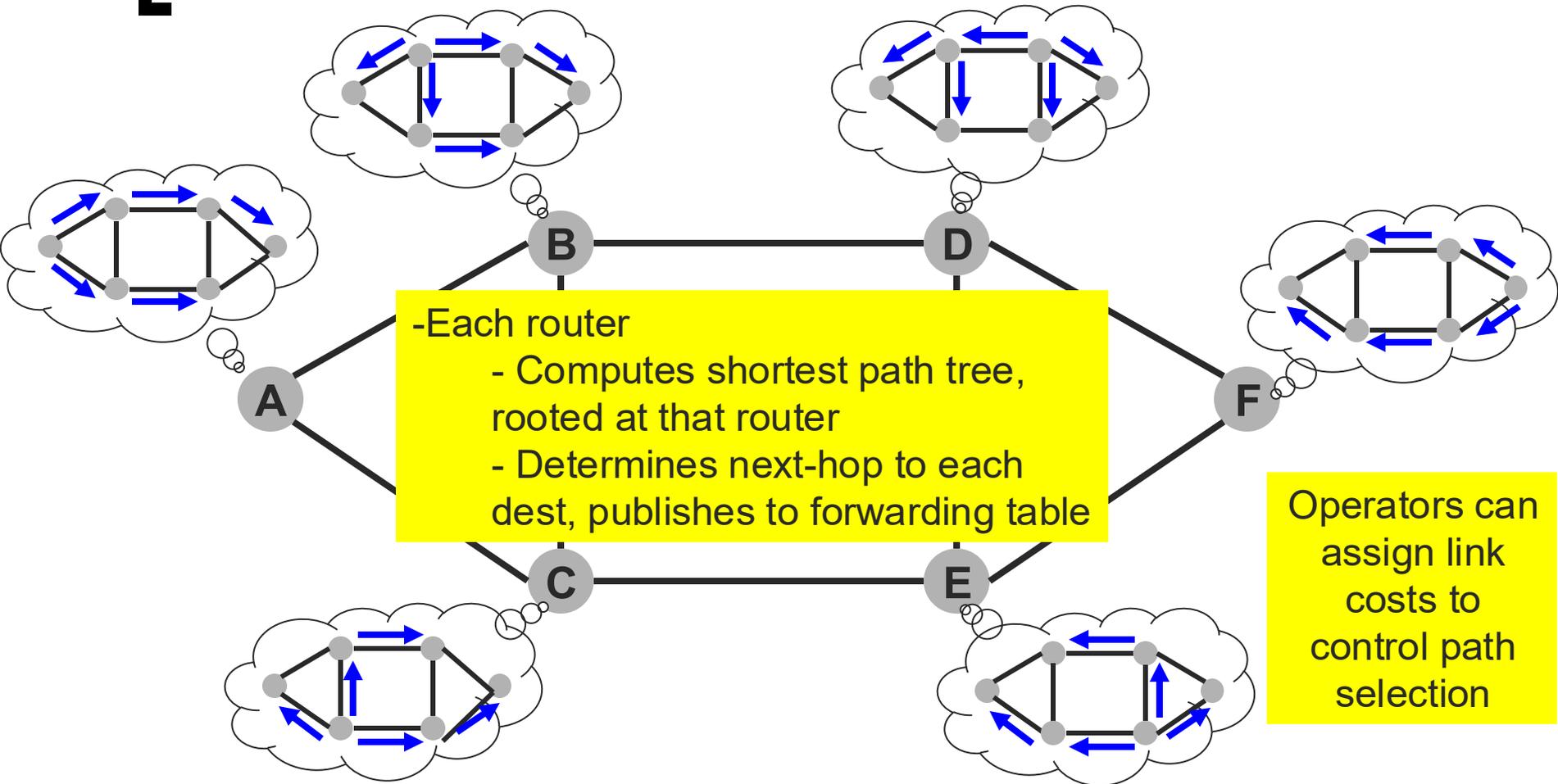
# [ Link state: update propagation ]



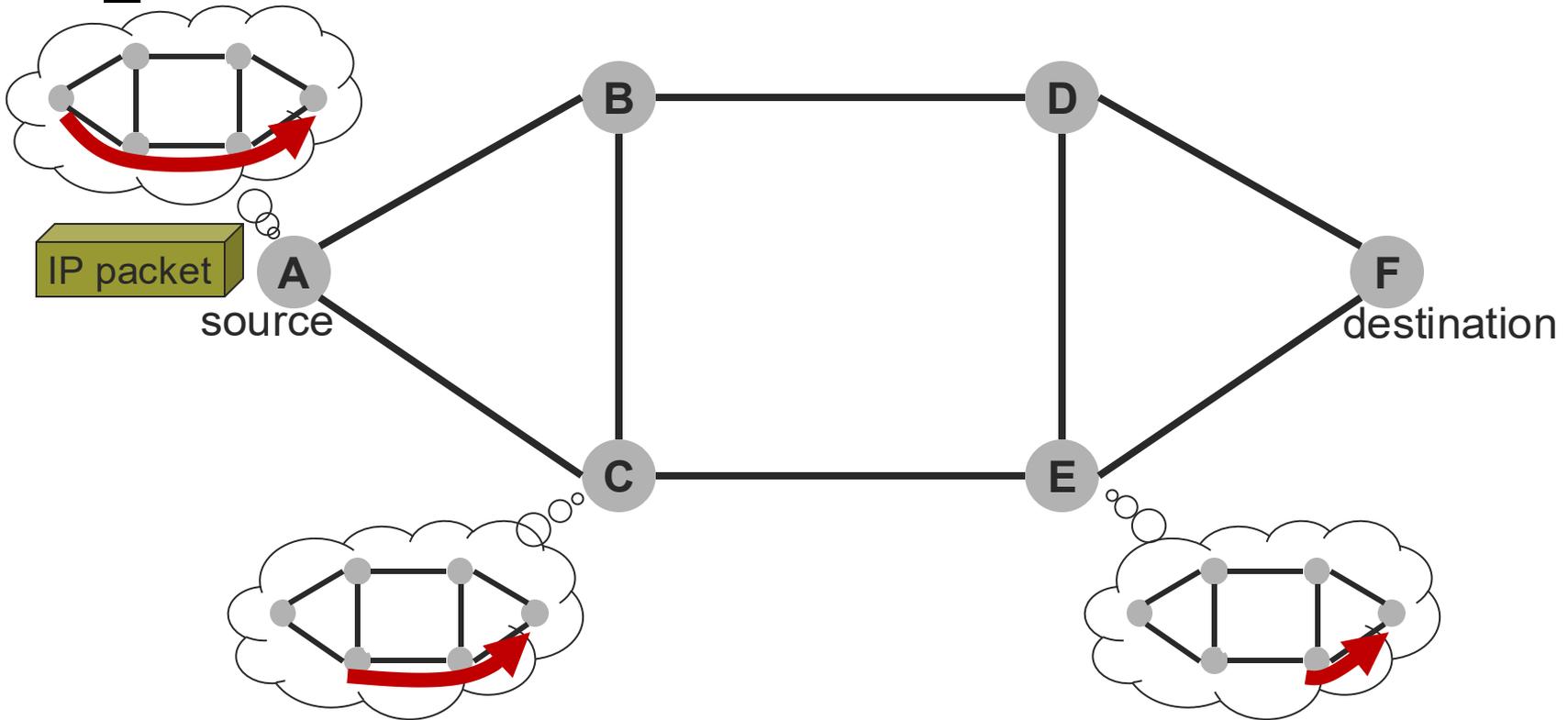
# Link state: update propagation



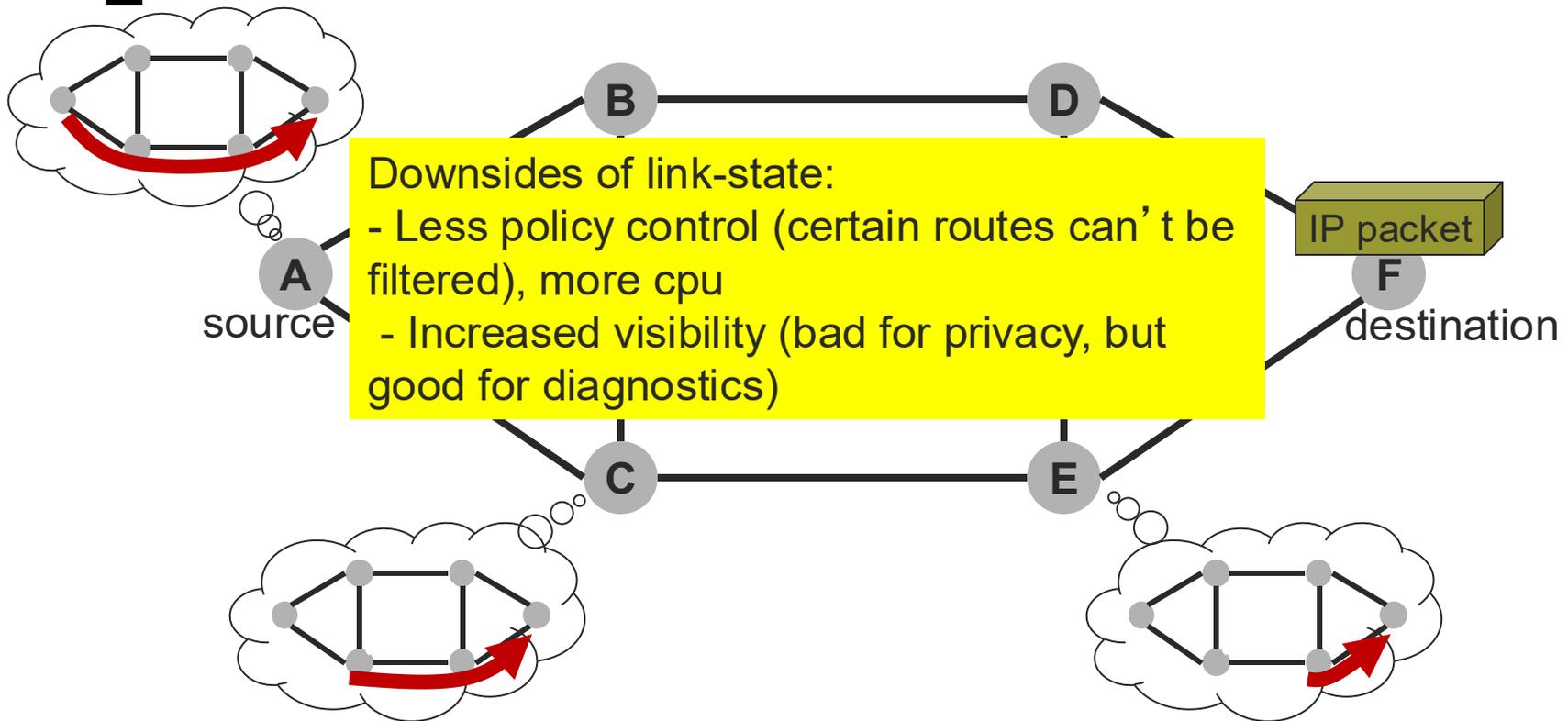
# Link state: route computation



# Link-state: packet forwarding



# Link-state: packet forwarding



# [ Link State Routing ]

- LSP must be delivered to all nodes
- Information acquisition via reliable flooding
  - Create local LSP periodically with increasing sequence number
  - Send local LSP to all immediate neighbors
  - Forward new LSP out on all other links
- What does “new” mean?
  - New sequence number
  - TTL accounts for wrapped sequence numbers
    - Decrement TTL for stored nodes



# [ Basic Steps ]

- Each node assumed to know state of links to its neighbors
- **Step 1:** Each node broadcasts its state to all other nodes
- **Step 2:** Each node locally computes shortest paths to all other nodes from global state



# [ Reliable Flooding ]

- When  $i$  receives LSP from  $j$ :
  - If LSP is the most recent LSP from  $j$  that  $i$  has seen so far
    - $i$  saves it in database and forwards a copy on all links except link LSP was received on
  - Otherwise, discard LSP

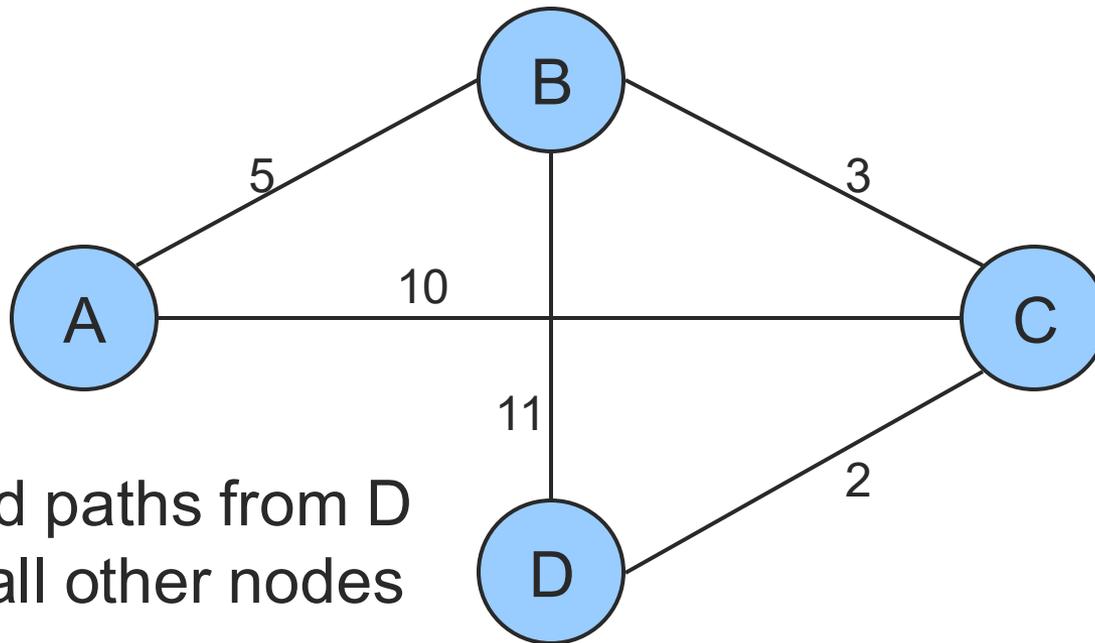


# [ Link State Routing ]

- At each router, perform a forward search algorithm
  - Variation of Dijkstra's
  - Variants to improve performance
    - e.g., incremental Dijkstra's
- Router maintains two lists
  - Tentative
  - Confirmed
- Each list contains triplets
  - <destination, cost, nexthop>



# [ Link State Routing ]

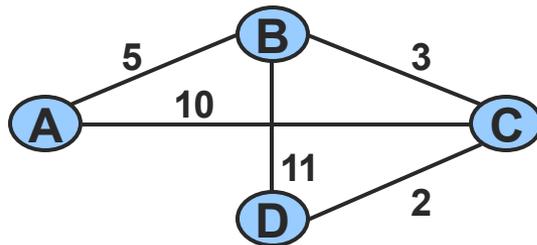


Find paths from D  
to all other nodes



# [ Link State Routing ]

| Step | Confirmed | Tentative |
|------|-----------|-----------|
| 1.   |           |           |
| 2.   |           |           |
| 3.   |           |           |
| 4.   |           |           |

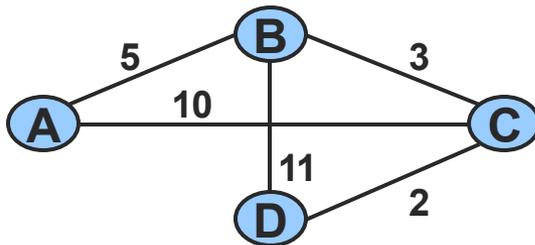


| Step | Confirmed | Tentative |
|------|-----------|-----------|
| 5    |           |           |
| 6    |           |           |
| 7    |           |           |



# Link State Routing

| Step | Confirmed          | Tentative           |
|------|--------------------|---------------------|
| 1.   | (D,0,-)            |                     |
| 2.   | (D,0,-)            | (B,11,B)<br>(C,2,C) |
| 3.   | (D,0,-)<br>(C,2,C) | (B,11,B)            |
| 4.   | (D,0,-)<br>(C,2,C) | (B,5,C)<br>(A,12,C) |

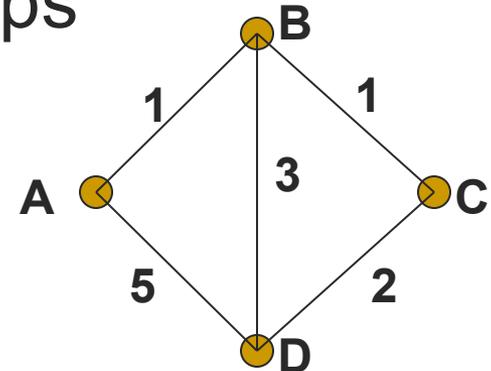


| Step | Confirmed                                 | Tentative |
|------|---|-----------|
| 5    | (D,0,-)<br>(C,2,C)<br>(B,5,C)             | (A,12,C)  |
| 6    | (D,0,-)<br>(C,2,C)<br>(B,5,C)             | (A,10,C)  |
| 7    | (D,0,-)<br>(C,2,C)<br>(B,5,C)<br>(A,10,C) |           |



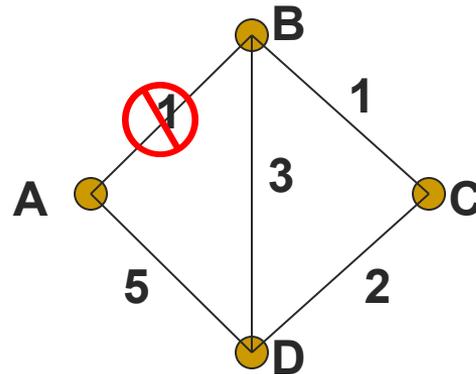
# Link State Characteristics

- With consistent LSDBs, all nodes compute consistent loop-free paths
- Limited by Dijkstra computation overhead, space requirements
- Can still have transient loops



# Link State Characteristics

- How could this cause loops?



Packet from C→A  
may loop around BDC



# [ Source Routing ]

- Variant of link state routing
  - Like link state, distribute network topology and compute shortest paths at source
  - ...but only at source, not every hop!



# [ Link State Routing ]

## ■ Pros

- Stabilizes quickly, does not generate much traffic, responds to topology changes or node failures

## ■ Cons

- Amount of information stored at each node is large



# Link State Routing in the Wild

- Intermediate System-Intermediate System (IS-IS)
  - Designed for DECnet
  - Adopted by ISO for connectionless network layer protocol (CNLP)
  - Used in NSFNET backbone
  - Used in some digital cellular systems
- ARPANET
  - Bad heuristics brought down the network in 1981
- Internet
  - Open shortest path first (OSPF)
  - Defined in RFC 5340
  - Used in some ISPs

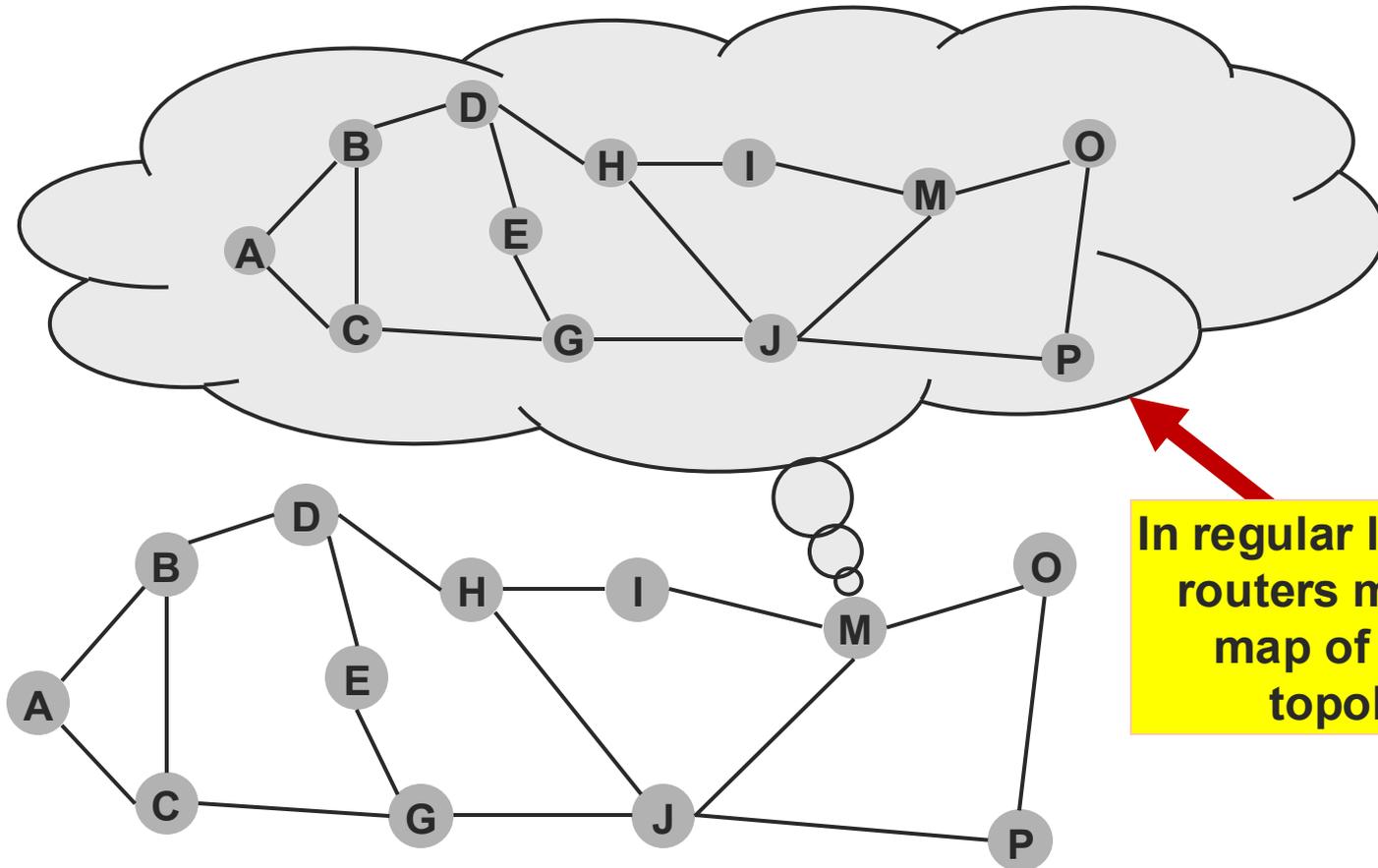


# [ OSPF ]

- Authentication of routing messages
  - Encrypted communication between routers
- Additional hierarchy
  - Domains are split into areas
  - Routers only need to know how to reach every node in a domain
  - Routers need to know how to get to the right area
  - Load balancing
    - Allows traffic to be distributed over multiple routes



# OSPF - Hierarchical routing



**In regular link-state, routers maintain map of entire topology**

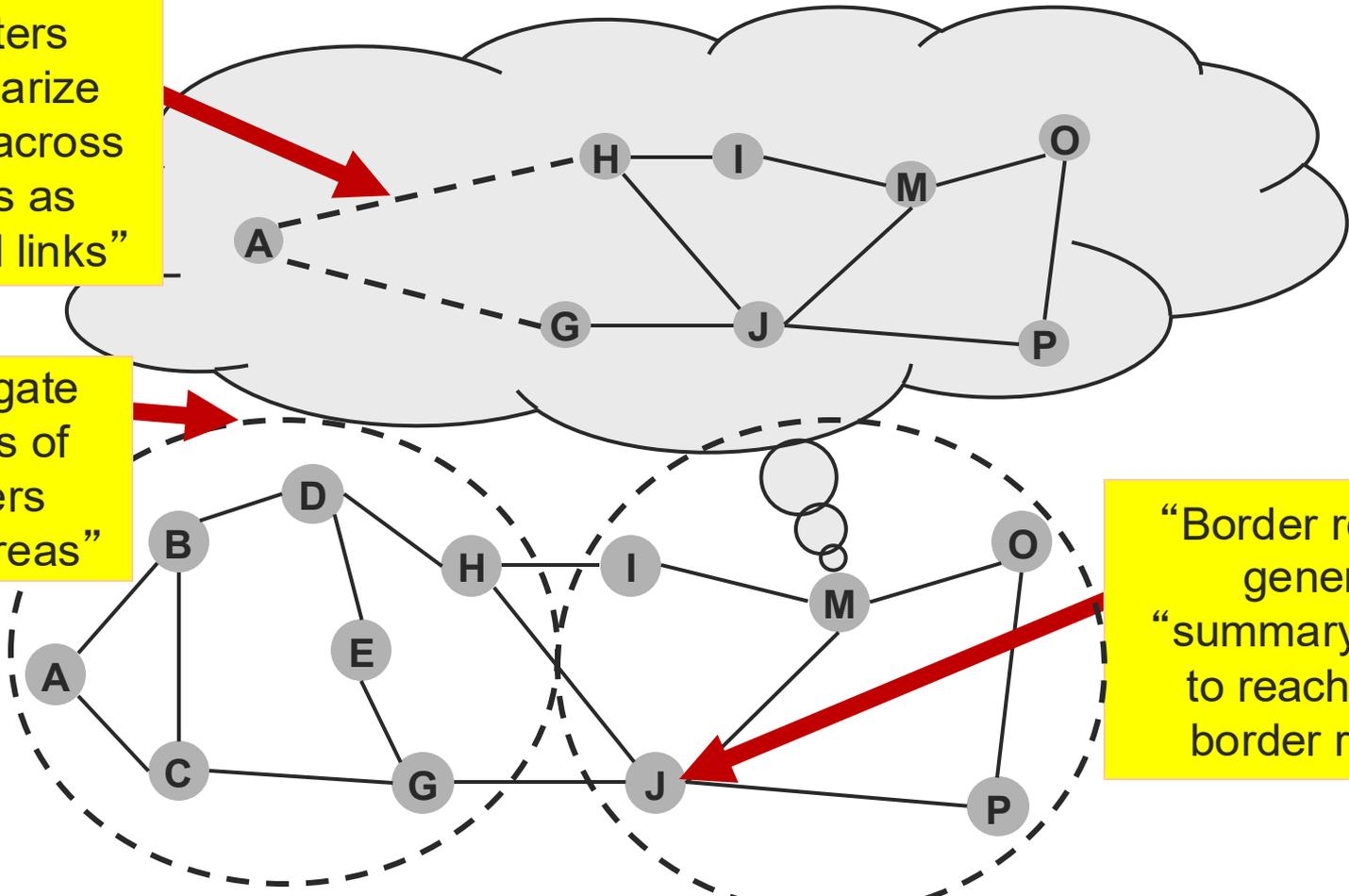


# OSPF - Hierarchical routing

Routers summarize paths across areas as “virtual links”

Aggregate groups of routers into “areas”

“Border routers” generate “summary LSPs” to reach other border routers



# Tradeoffs of hierarchical routing

- Advantages: scalability
  - Reduce size of link-state database
  - Isolate rest of network from changes/faults
- Disadvantages
  - Complexity
    - Extra configuration effort
    - Requires tight coupling with address assignment
  - Inefficiency
    - One link change may affect multiple path costs
    - Summarization hides shorter paths



# [ LS vs. DV ]

- DV
  - Send everything you know to your neighbors
- LS
  - Send info about your neighbors to everyone
- Message size
  - Small with LS
  - Potentially large with DV
- Message exchange
  - LS:  $O(nE)$
  - DV: only to neighbors



# [ LS vs. DV ]

- Convergence speed
  - LS: fast
  - DV: fast with triggered updates
- Space requirements
  - LS maintains entire topology
  - DV maintains only neighbor state



# [ LS vs. DV: Robustness ]

- LS can broadcast incorrect/corrupted LSP
  - localized problem
- DV can advertise incorrect paths to all destinations
  - incorrect calculation can spread to entire network
- Soft-state vs. Hard-state approaches
  - Should we periodically refresh? Or rely on routers to locally maintain their state correctly?



# [ LS vs. DV ]

- LS
  - Nodes must compute consistent routes independently
  - Must protect against LSDB corruption
- DV
  - Routes are computed relative to other nodes
- Bottom line
  - No clear winner, but we see more frequent use of LS in the Internet



# [ LS vs. DV ]

- LS typically used *within* ISPs because
  - Faster convergence (usually)
  - Simpler troubleshooting
- DV typically used *between* ISPs because
  - Can support more flexible policies
  - Can avoid exporting routes
  - Can hide private regions of topology





# Metrics

# Traffic engineering with routing protocols

- Load balancing
  - Some hosts/networks/paths are more popular than others
  - Need to shift traffic to avoid overrunning capacity
- Avoiding oscillations
  - What if metrics are a function of offered load?
  - Causes dependencies across paths



# [ Importance of Cost Metric ]

- Choice of link cost defines traffic load
  - Low cost = high probability link belongs to SPT
  - Will attract traffic, which increases cost
- Main problem: convergence
  - Avoid oscillations
  - Achieve good network utilization



# [ Metrics ]

- Capture a general notion of distance
- A heuristic combination of
  - Distance
  - Bandwidth
  - Average traffic
  - Queue length
  - Measured delay



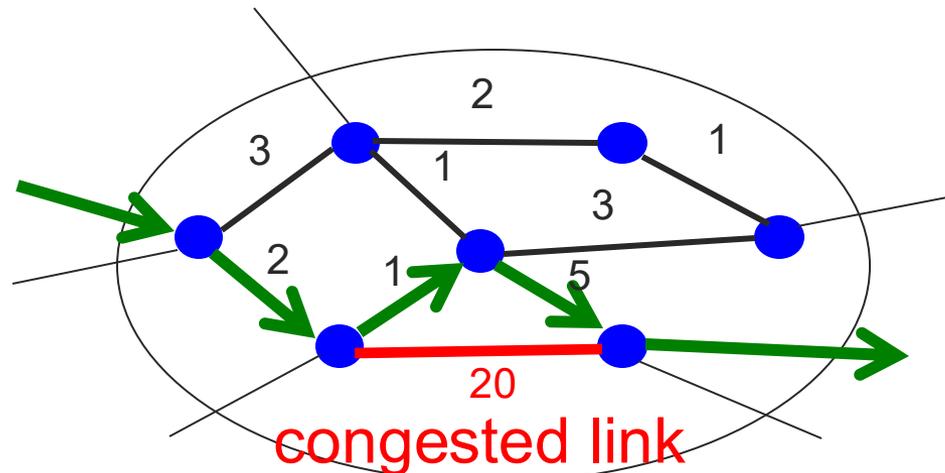
# [ Metric Choices ]

- Static metrics (e.g., hop count)
  - Good only if links are homogeneous
  - Definitely not the case in the Internet
- Static metrics do not take into account
  - Link delay
  - Link capacity
  - Link load (hard to measure)
- But, can improve stability



# Original ARPANET (1969)

- Distance vector routing
  - Routing tables exchanged every  $2/3$  seconds
- Use queue length as distance
  - Number of packets waiting to use a link
  - Instantaneous queue length as delay estimator



# Original ARPANET Algorithm

- Light load
  - Delay dominated by the constant part (transmission and propagation delay)
- Medium load
  - Queuing delay no longer negligible
  - Moderate traffic shifts to avoid congestion
- Heavy load
  - Very high metrics on congested links
  - Busy links look bad to all of the routers
  - All routers avoid the busy links
  - Routers may send packets on longer paths



# [ Original ARPANET ]

- Uniform 56 Kbps lines
  - Bandwidth equal on every line
  - Latency relatively unimportant
- Problems
  - Uniform bandwidth became an invalid assumption
  - Latency comparable to 1 KB transmission delay on 1.544 Mbps link



# [ New ARPANET(1979) ]

- Switch to link-state routing
- Routing updates only contain link cost information
- Link metric is measured delay
- Max time between updates = 50 sec



# [ New ARPANET(1979) ]

- Averaging of link metric over time
  - Old: Instantaneous delay fluctuates a lot
  - New: Averaging reduces the fluctuations
- Link-state protocol instead of DV
  - Old: DV led to loops
  - New: Flood metrics and let each router compute shortest paths
- Reduce frequency of updates
  - Old: Sending updates on each change is too much
  - New: Send updates if change passes a threshold



# [ Problem #2: Load balancing ]

- Conventional static metrics:
  - Proportional to physical distance
  - Inversely proportional to link capacity
- Conventional dynamic metrics:
  - Tune weights based on the offered traffic
  - Network-wide optimization of link-weights
  - Directly minimizes metrics like maximum link utilization



# Metrics: New Arpanet

- Captured delay, bandwidth and latency
- Queue delay
  - Timestamp packet arrival time (AT)
  - Also timestamp packet departure time (DT)
  - Only calculate when ACK received
  - Average DT- AT over packets and time
- Used fixed (per-link) measurements
  - Transmission time (bandwidth)
  - Latency
- Add three terms to find “distance” metric



# Metrics: New ARPANET

- Assumption
  - Measured delay = expected delay
- Worked well under light load
  - Static factors dominated cost
- Oscillated under heavy load
  - Heavily loaded link advertises high price
  - All traffic moves off
  - Then link advertises light load
  - All traffic returns
  - Repeat cycle

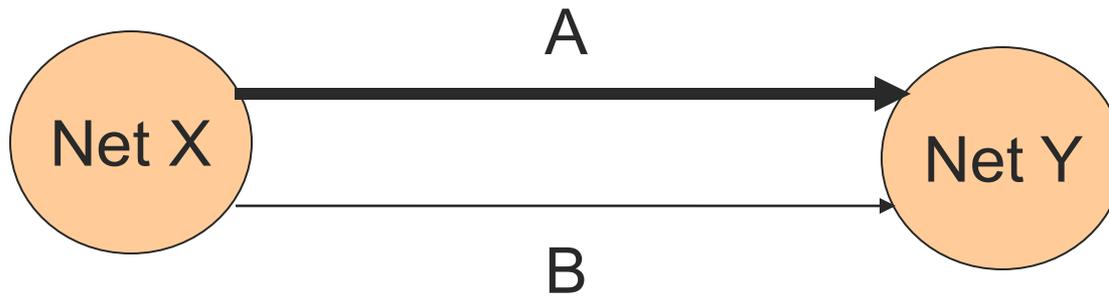


# [ Specific problems ]

- Range is too wide
  - 9.6 Kbps highly loaded link can appear 127 times costlier than 56 Kbps lightly loaded link.
  - Can make a 127-hop path look better than 1-hop.
- No limit in reported delay variation
- All nodes calculate routes simultaneously
  - Triggered by link update

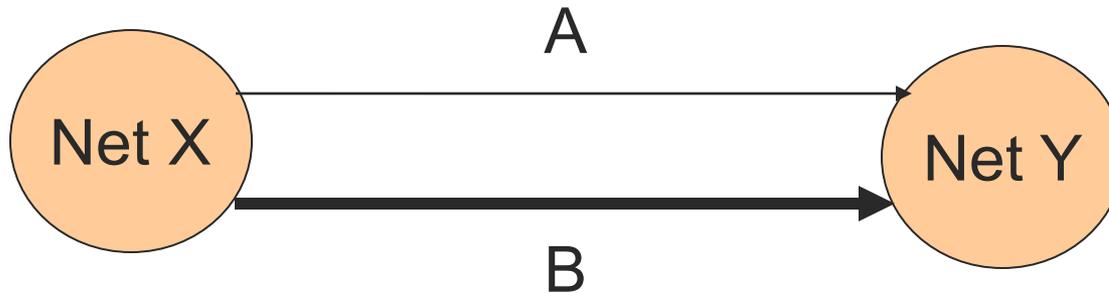


# [ Example ]



# [ Example ]

After everyone re-calculates routes:



.. Oscillations!



# [ Consequences ]

- Low network utilization (50% in example)
- Congestion can spread elsewhere
- Routes could oscillate between short and long paths
- Large swings lead to frequent route updates
  - More messages
  - Frequent SPF re-calculation



# [ Some Considerations ]

- Delay as absolute measure of path length
- Greedy approach to route selection
  - Each node chooses shortest path without regards for how it affects others
- Instead, routing should provide good path to average node
  - Some nodes get longer routes



# Metrics: Revised ARPANET

- Measure link utilization
- Feed measurement through function to restrict dynamic range
- Specific function chosen carefully based on bandwidth and latency
- Aspects of class of functions
  - Cost is constant at low to moderate utilization
  - Link cost is no more than 3 times idle link cost
  - Maximum cost (over all links) is no more than 7 times minimum cost (over all links)



# [ Reality of the Modern Internet ]

- Hierarchical routing used
  - Between different Autonomous Systems (e.g., a provider network), a standard protocol
  - Within each AS
    - Up to AS administrator
    - Usually a variant of link-state or distance-vector
- What metrics are really used?
  - Nothing involving load
  - Just too unstable

