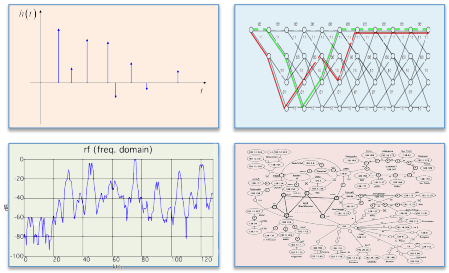



INTRODUCTION TO EECS II

# DIGITAL COMMUNICATION SYSTEMS

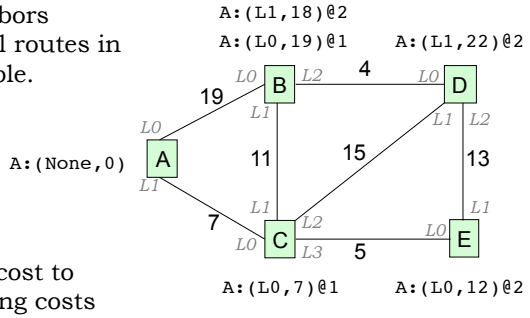
## 6.02 Spring 2011
## Lecture #20

- path-vector routing
- link-vector routing
- Dijkstra's shortest path algorithm
- hierarchical routing



## Distance-vector (DV) review

At each ADVERT interval, nodes tell neighbors (dest,cost) for all routes in their routing table.

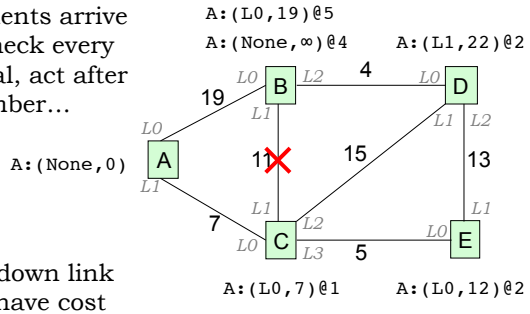



Nodes add link cost to neighbor's routing costs and keep their routing table up-to-date with shortest-path route.



## Link is Down at Time 4?

A link is considered down if no advertisements arrive over the link; check every ADVERT interval, act after some small number…

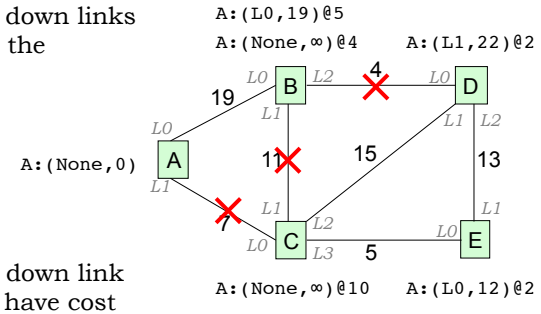



Routes using a down link are changed to have cost ∞, which will propagate to neighbors who then update their cost if they used you for their route.



## Paritioned Network at Time 10

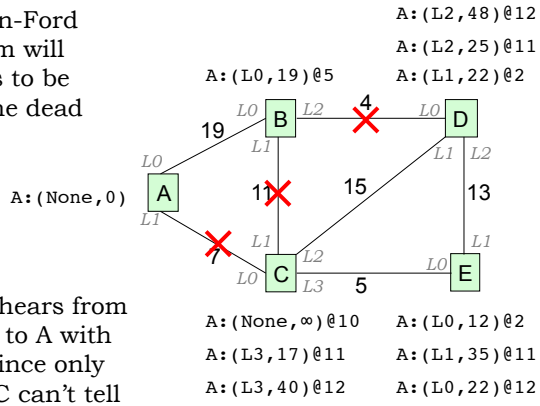
An unfortunate combination of down links might partition the network.




Routes using a down link are changed to have cost ∞, which will propagate to neighbors who then update their cost if they used you for their route.

# Count to Infinity

Now the Bellman-Ford update algorithm will cause new costs to be calculated for the dead routes.




For example, C hears from E about a route to A with total cost 17. Since only costs are kept, C can't tell that E was relying on *it* for its route of cost 12!

The costs spiral higher, eventually passing some bound, at which point they are recognized as ∞.



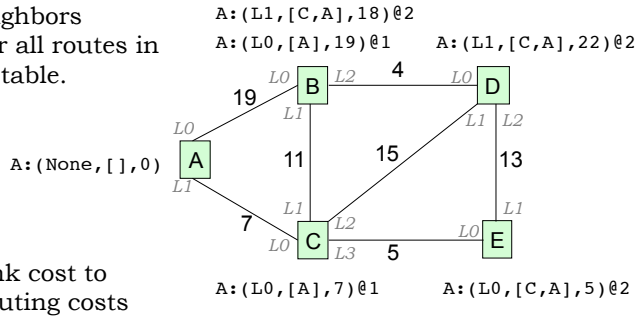
# Fixing "Count to Infinity"

- Problem
  - Node C's route to A breaks, C sets cost to ∞
  - But at next round of advertisements, hears of lower-cost routes from neighbors, not know the neighbor's routes used C itself to get to A.
- Solution
  - In addition to reporting costs in advertisements, also report routing path as discovered incrementally by Bellman-Ford
  - Called "path-vector"
  - Modify Bellman-Ford update with new rule: nodes should ignore advertised routes that contain itself in the routing path
  - Pros: count-to-infinity "problem" is solved (routing tables eliminate routes to unreachable nodes more quickly)
  - Cons: advertisement overhead is larger



# Path-vector (PV) routing

At each ADVERT interval, nodes tell neighbors (path,cost) for all routes in their routing table.

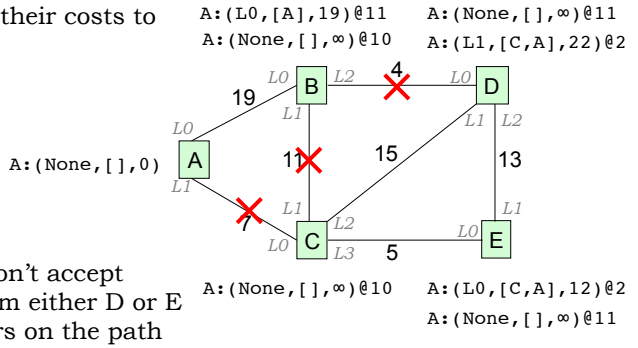



Nodes add link cost to neighbor's routing costs and keep their routing table up-to-date with shortest-path route.



# Paritioned Network at Time 10: PV

Nodes connected to down links change their costs to ∞.




Using PV, C won't accept routes to A from either D or E since C appears on the path they advertise. Unreachable nodes are quickly removed from tables.

Pros: simple, works well for small networks
Cons: only works for small networks

# Link-State Routing

- Advertisement step
  - Send information about its links to its neighbors (aka *link state advertisement* or LSA):

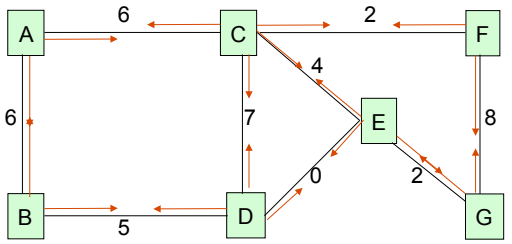
    [seq#, [(nbhr1, linkcost1), (nbhr2, linkcost2), …]

  - Do it periodically (liveness, recover from lost LSAs)
- Integration
  - If seq# in incoming LSA > seq# in saved LSA for source node: update LSA for node with new seq#, neighbor list rebroadcast LSA to neighbors (→ **flooding**)
  - Remove saved LSAs if seq# is too far out-of-date
  - Result: Each node discovers current map of the network
- Build routing table
  - Periodically each node runs the same *shortest path algorithm* over its map
  - If each node implements computation correctly and each node has the same map, then routing tables will be correct



# LSA Flooding

[F, seq, (G, 8), (C, 2)]




- LSA travels each link in each direction
  - Don't bother with figuring out which link LSA came from
- Termination: each node rebroadcasts LSA exactly once
- All reachable nodes eventually hear every LSA
  - Time required: number of links to cross network



# Dijkstra's Shortest Path Algorithm

- Initially
  - nodeset = [all nodes] = set of nodes we haven't processed
  - spcost = {me:0, all other nodes: ∞} # shortest path cost
  - routes = {me:--, all other nodes: ?} # routing table
- while nodeset isn't empty:
  - find u, the node in nodeset with smallest spcost
  - remove u from nodeset
  - for v in [u's neighbors]:
    - d = spcost(u) + cost(u,v) # distance to v via u
    - if d < spcost(v): # we found a shorter path!
      - spcost[v] = d
      - routes[v] = routes[u] (or if u == me, enter link from me to v)
- Complexity: N = number of nodes, L = number of links
  - Finding u (N times): linear search=O(N), using heapq=O(log N)
  - Updating spcost: O(L) since each link appears twice in neighbors

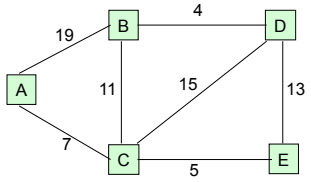


# Dijkstra Example

Finding shortest paths from A:

LSAs:

```
A: [(B,19), (C, 7)]
B: [(A,19), (C,11), (D, 4)]
C: [(A, 7), (B,11), (D,15), (E, 5)]
D: [(B, 4), (C,15), (E,13)]
E: [(C, 5), (D,13)]
```



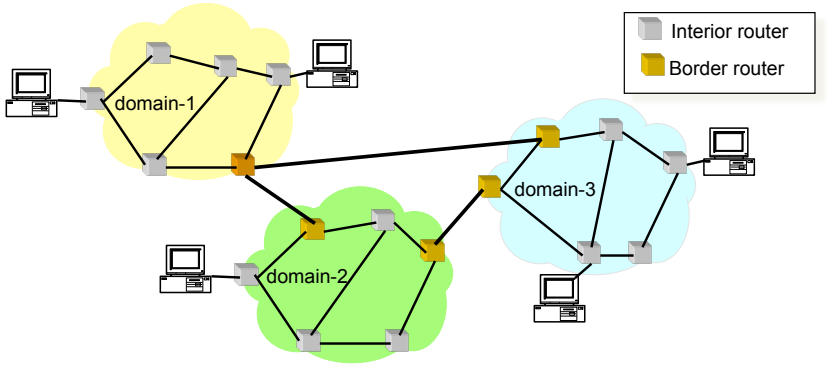

| *Step* | *u* | *Nodeset* | *spcost* | | | | | *route* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | *A* | *B* | *C* | *D* | *E* | *A* | *B* | *C* | *D* | *E* |
| 0 | | [A,B,C,D,E] | 0 | ∞ | ∞ | ∞ | ∞ | -- | ? | ? | ? | ? |
| 1 | A | [B,C,D,E] | 0 | 19 | 7 | ∞ | ∞ | -- | L0 | L1 | ? | ? |
| 2 | C | [B,D,E] | 0 | 18 | 7 | 22 | 12 | -- | L1 | L1 | L1 | L1 |
| 3 | E | [B,D] | 0 | 18 | 7 | 22 | 12 | -- | L1 | L1 | L1 | L1 |
| 4 | B | [D] | 0 | 18 | 7 | 22 | 12 | -- | L1 | L1 | L1 | L1 |
| 5 | D | [] | 0 | 18 | 7 | 22 | 12 | -- | L1 | L1 | L1 | L1 |

# Why is Network Routing Hard?

- Inherently distributed problem
  - Information about links and neighbors is local to each node, but we want global reach
- Efficiency: want reasonably good paths, and must find them without huge overhead
- Handling failures and "churn"
  - Must tolerate link, switch, and network faults
  - Failures and recovery could be arbitrarily timed, messages could be lost, etc.
- Scaling to large size very hard (later courses)
  - And on the Internet, many independent, competing organizations must cooperate
  - Mobility makes the problem harder



# Hierarchical Routing



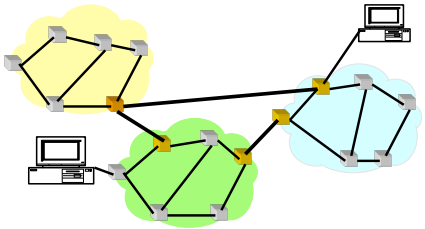

- Internet: collection of domains/networks
- Inside a domain: Route over a graph of routers
- Between domains: Route over a graph of domains
- Address: concatenation of "Domain Id", "Node Id"



# Pros and Cons

Advantages

- Scalable
  - Smaller tables
  - Smaller messages
- Delegation
  - Each domain can run its own routing protocol



Disadvantages

- Mobility is difficult
  - Address depends on geographic location
- Sup-optimal paths
  - E.g., in the figure, the shortest path between the two machines should traverse the yellow domain. But hierarchical routing goes directly between the green and blue domains, then finds the local destination → path traverses more routers.



# Summary

- The network layer implements the "glue" that achieves connectivity
  - Does addressing, forwarding, and routing
- Forwarding entails a routing table lookup; the table is built using routing protocol
- DV protocol: distributes route computation; each node advertises its best routes to neighbors
- LS protocol: distributes (floods) neighbor information; centralizes route computation using shortest-path algorithm