Network Layer
- Control plane -

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Network-layer Functions

Recall: two network-layer functions:

- *forwarding:* move packets from router's input to appropriate router output

data plane

- *routing: determine route taken by packets from source to destination*

control plane

Two approaches to structuring network control plane:

- per-router control (traditional)
- logically centralized control (software defined networking)

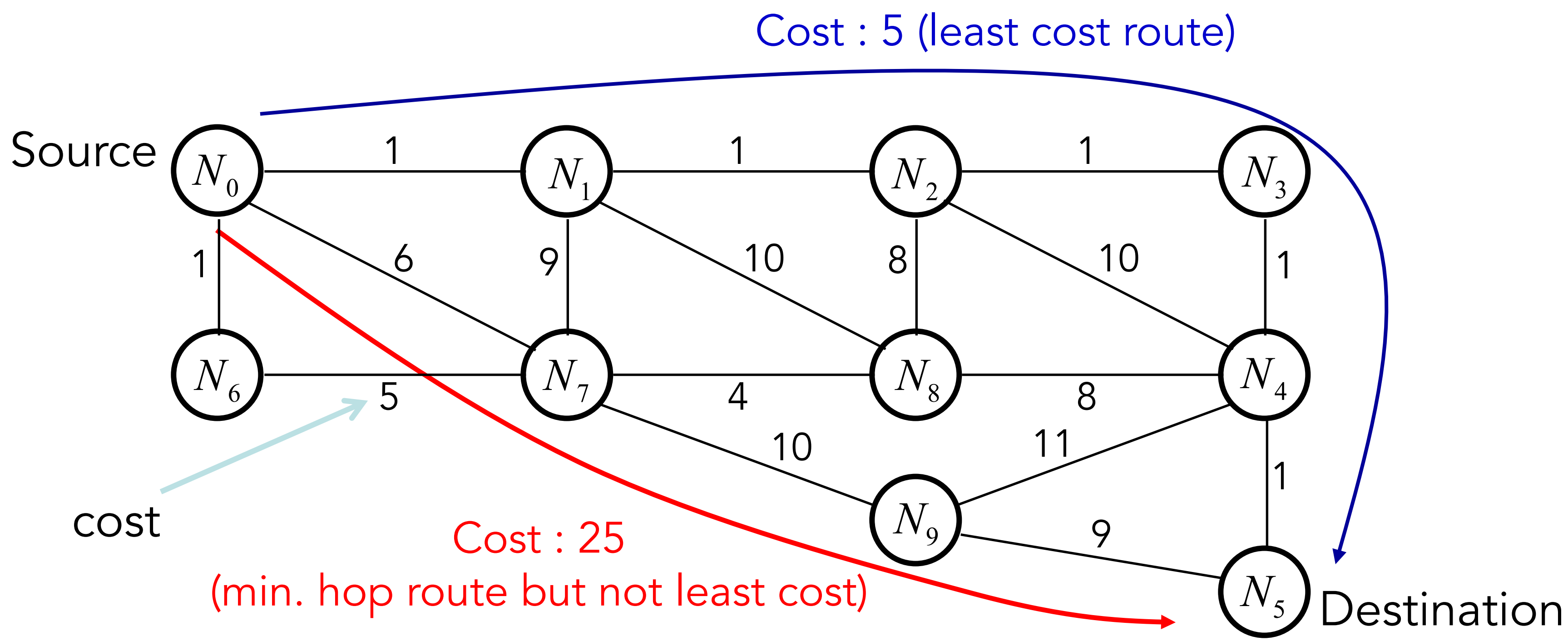
Routing protocols

Routing protocol goal: determine "good" paths (equivalently, routes), from sending hosts to receiving host, through network of routers

- path: sequence of routers packets will traverse in going from given initial source host to given final destination host
- "good": least "cost", "fastest", "least congested"
- routing: a "top-10" networking challenge!

Shortest Path Routing Problem

- The cost of a path = the sum of all the link costs on the path
- Find the path with the least cost between a pair of source and destination



What is the Cost?

- Some fixed quantity:
  - Link length or hop count
  - Speed or bandwidth
  - Propagation delay
  - Some combination of the above
- Possibly, variable quantity:
  - Average traffic expected at a given time
  - Buffer occupancy (queueing)
  - Processing delay (e.g., DPI)
  - Error conditions

User requirements

- Different users prefer different routing paths
  - File transfer -- high bandwidth path
  - Interactive communication (e.g., VoIP) -- low delay path (avoid satellite links!)
  - Important Information (e.g., money transfer) -- secure data path.

Routing algorithm classification

Q: global or decentralized information?

global:

- all routers have complete topology, link cost info
- "link state" algorithms

decentralized:

- router knows physically-connected neighbors, link costs to neighbors
- iterative process of computation, exchange of info with neighbors
- "distance vector" algorithms

Q: static or dynamic?

static:

- routes change slowly over time

dynamic:

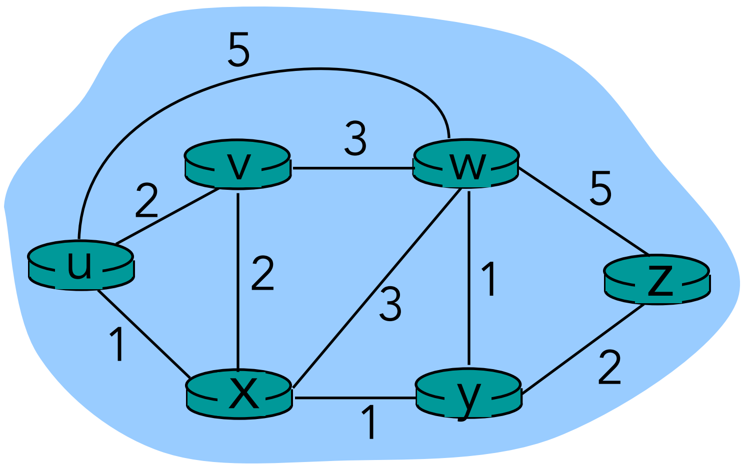
- routes change more quickly
  - periodic update
  - in response to link cost changes

Centralized vs. Decentralized

- Centralized Routing:
  - A central entity calculates all paths between source and destination nodes, and
  - Then, distributes routing information to all the nodes
  - **Problems:** single point of failure, complexity, etc.

- Decentralized Routing:
  - Each node exchanges cost and routing information
    - Keep exchanging with its neighbors until routing table converges
  - **Problems:** Convergence and sub-optimality (due to delayed information).

Graph abstraction of the network



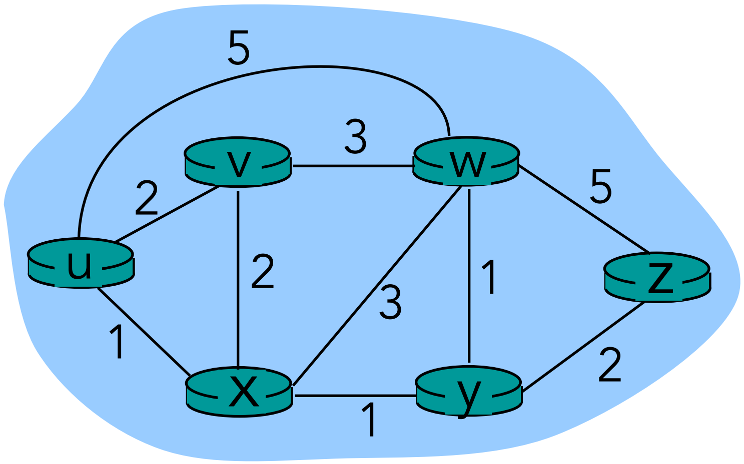
graph: G = (N,E) where N: nodes, E: edges

N = set of routers = { u, v, w, x, y, z }

E = set of links ={ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }

aside: graph abstraction is useful in other network contexts, e.g., P2P, where *N* is set of peers and *E* is set of TCP connections

Graph abstraction: costs



$c(x,y)$ = cost of link (x,y)
(e.g., $c(w,z) = 5$)

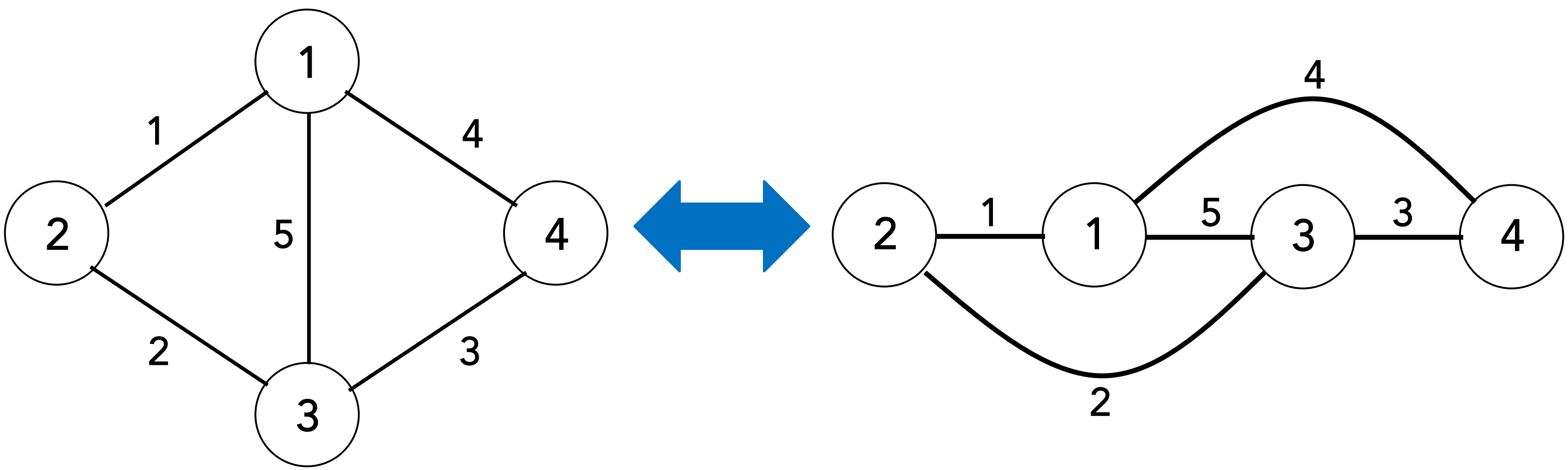
cost could always be 1, or inversely related to bandwidth, or inversely related to congestion

cost of path $(x_1, x_2, x_3,\ldots, x_p) = c(x_1,x_2) + c(x_2,x_3) + \ldots + c(x_{p-1},x_p)$

key question: what is the least-cost path between u and z ?
routing algorithm: algorithm that finds that least cost path

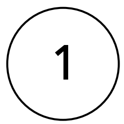
Basic Graph Theoretic Notations

- Definitions:
  - A *graph* (or undirected graph) $G = (N, A)$ is defined to be a finite *non-empty* set N of *nodes* and a collection A of pairs of distinct nodes from N
  - Each pair of nodes in A is called an *arc* (or link, edge)
  - Same graph with very different pictorial representation:



Basic Graph Theoretic Notations

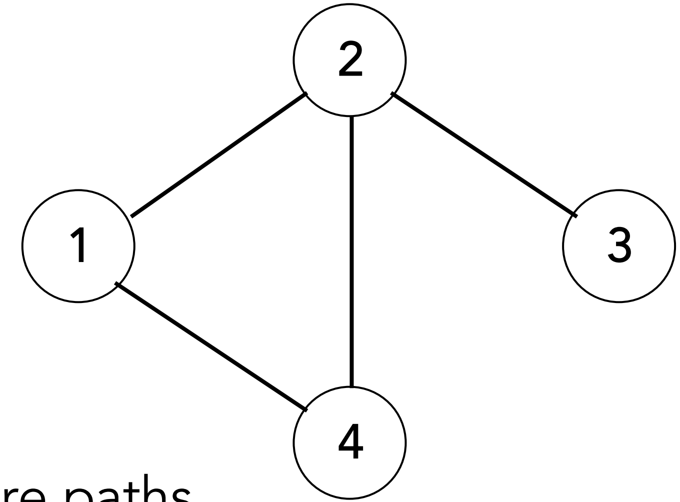
- Can the following be a graph G(N,A)?



- More definitions:
  - Walk
    - A walk in a graph G is a sequence of nodes $(n_1, n_2, \ldots, n_L)$ such that the pairs $(n_1, n_2), (n_2, n_3), \ldots., (n_{L-1}, n_L)$ are arcs of G
  - Path
    - A walk with no repeated nodes is called a path
  - Cycle
    - A walk $(n_1, n_2, \ldots, n_L)$ with $n_1 = n_L$, $L \geq 4$, and no repeated nodes other than $n_1 = n_L$ is called a cycle.

Examples

- Graph with a net of nodes
  N={1, 2, 3, 4},
  and a set of arcs
  A={ (1,2), (2,3), (2,4), (4,1) }

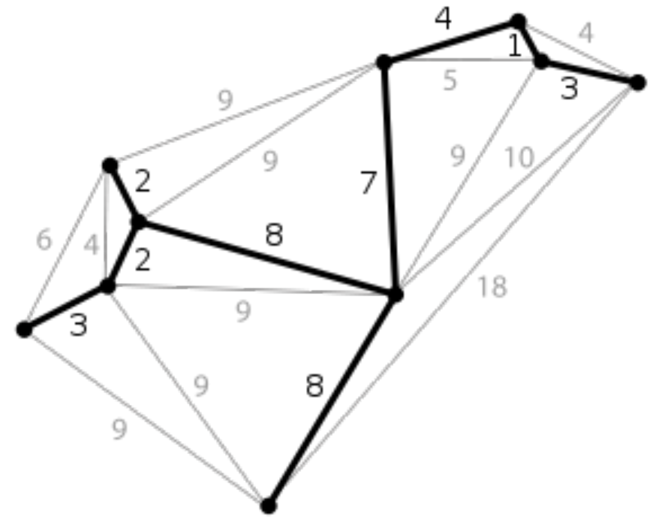


  - The sequence (1, 4, 2, 3) and (2) are paths
  - The sequence (1, 4, 2, 1) is a cycle
  - Sequences (1, 4, 2, 3), (1, 4, 2, 1), (1, 4, 2, 1, 4, 1), (2, 3, 2) and (2) are all walks

(Note : (2, 3, 2) and (2) are not considered cycles)

Basic Graph Theoretic Notations

- More definitions:
  - A graph is *connected* if for each node i, there is a path (i = n_1, n_2, …, n_L = j) to every other node j
  - A *tree* is a connected graph that contains no cycles
  - A *spanning tree* of a connected graph G is a subgraph of G that contains all the nodes in G and is also a tree



spanning tree
(figure from http://en.wikipedia.org)

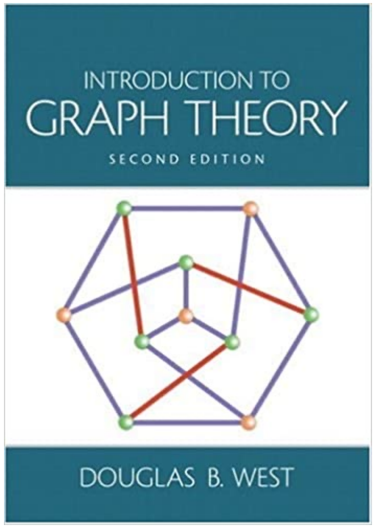
Basic Graph Theoretic Notations

- Proposition:

  Let G be a connected graph (N, A), then,

  1) G contains a spanning tree
  2) $|A| \geq |N| - 1$
  3) G is a tree if and only if $|A| = |N| - 1$

  - Proof can be done by induction.



A link-state routing algorithm

Dijkstra's algorithm

- net topology, link costs known to all nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node (source) to all other nodes
  - gives *forwarding table* for that node
- iterative: after k iterations, know least cost path to k dest.'s

Notation:

- c(x,y): link cost from node x to y; c(x,y)= ∞ if not direct neighbors
- D(v): current value of cost of path from source to dest. v
- p(v): predecessor node along path from source to v
- N': set of nodes whose least cost path definitively known

Dijsktra's algorithm

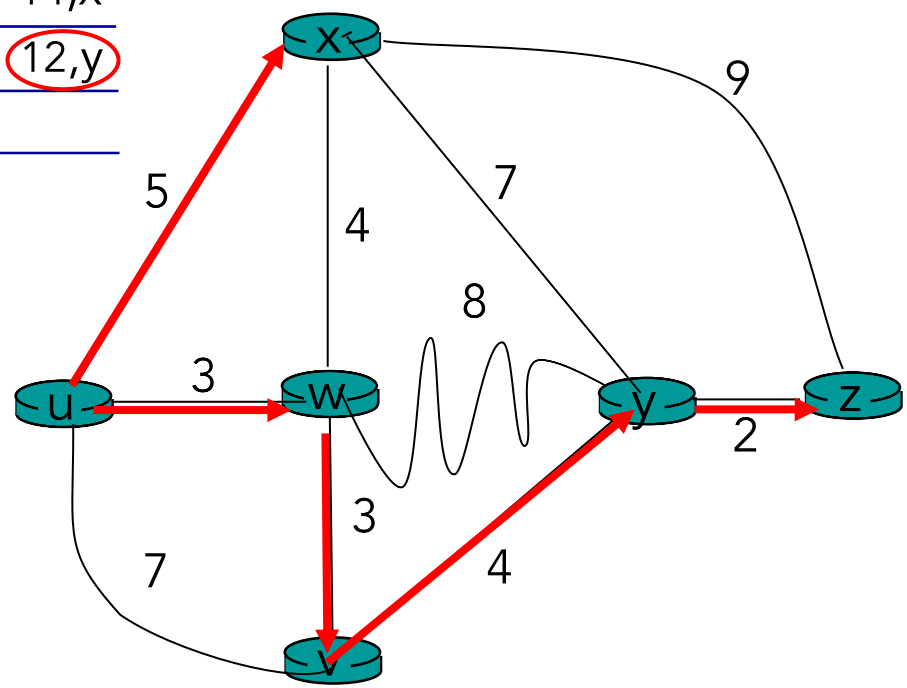
```
Initialization:
  N' = {u}
  for all nodes v
    if v adjacent to u
        then D(v) = c(u,v)
    else D(v) = ∞

Loop
   find w not in N' such that D(w) is a minimum
    add w to N'
    update D(v) for all v adjacent to w and not in N' :
       D(v) = min( D(v), D(w) + c(w,v) )
    /* new cost to v is either old cost to v or known
     shortest path cost to w plus cost from w to v */
 until all nodes in N'
```

Dijkstra's algorithm: example

Step	N'	D(v) p(v)	D(w) p(w)	D(x) p(x)	D(y) p(y)	D(z) p(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					

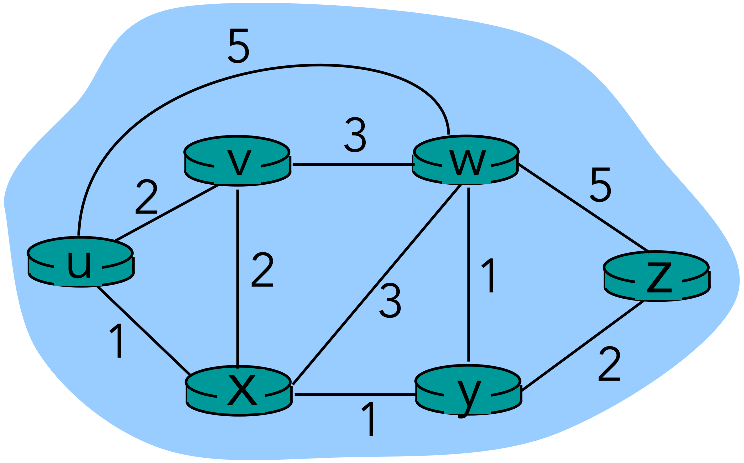
notes:

- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



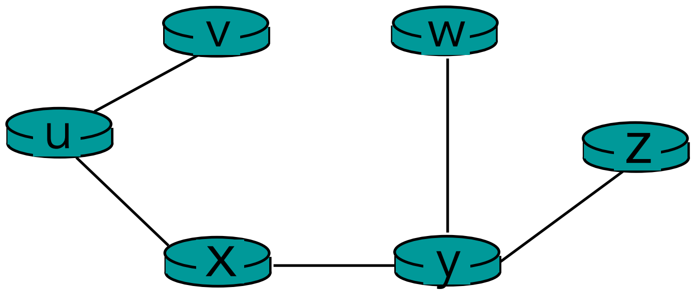
Dijkstra’s algorithm: another example

Step	N'	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					



Dijkstra’s algorithm: example (2)

- resulting shortest-path tree from u:



- resulting forwarding table in u:

destination	link
v	(u,v)
x	(u,x)
y	(u,x)
w	(u,x)
z	(u,x)

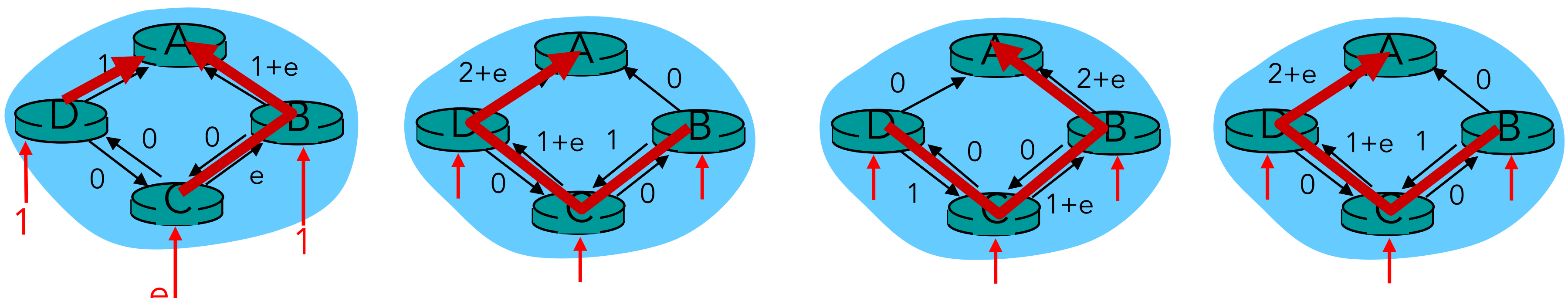
Dijkstra’s algorithm, discussion

algorithm complexity: n nodes

- each iteration: need to check all nodes, w, not in N
- n(n-1)/2 comparisons: $O(n^2)$
- more efficient implementations possible: O(nlogn)

oscillations possible:

- e.g., support link cost equals amount of carried traffic:



initially

given these costs, find new routing…. resulting in new costs

given these costs, find new routing…. resulting in new costs

given these costs, find new routing…. resulting in new costs

Distance vector algorithm

Bellman-Ford equation (dynamic programming)

Let

$d_x(y)$:= cost of least-cost path from x to y

then

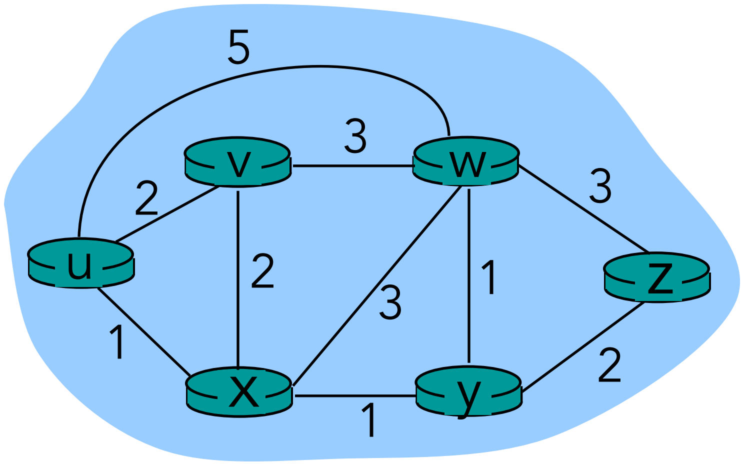
$$d_x(y) = \min_v \{c(x,v) + d_v(y)\}$$

cost from neighbor v to destination y

cost to neighbor v

min taken over all neighbors v of x

Bellman-Ford example



Clearly, $d_v(z) = 5$, $d_x(z) = 3$, $d_w(z) = 3$

B-F equation says:

$$\begin{aligned} d_u(z) = \min \{ & c(u,v) + d_v(z), \\ & c(u,x) + d_x(z), \\ & c(u,w) + d_w(z) \} \\ = \min \{ & 2 + 5, \\ & 1 + 3, \\ & 5 + 3\} = 4 \end{aligned}$$

node achieving minimum is next
hop in shortest path, used in forwarding table

Distance vector algorithm

- $D_x(y)$ = estimate of least cost from x to y
  - x maintains distance vector $D_x = [D_x(y): y \in N]$

- Node x:
  - knows cost to each neighbor v: $c(x,v)$
  - maintains its neighbors' distance vectors. For each neighbor v, x maintains $D_v = [D_v(y): y \in N]$

Distance vector algorithm

key idea:

- from time-to-time, each node sends its own distance vector estimate to neighbors
- when x receives new DV estimate from neighbor, it updates its own DV using B-F equation:

$$D_x(y) \leftarrow \min_v\{c(x,v) + D_v(y)\} \quad \textit{for each node } y \in N$$

- under minor, natural conditions, the estimate $D_x(y)$ *converge to the actual least cost* $d_x(y)$

Distance vector algorithm

iterative, asynchronous:

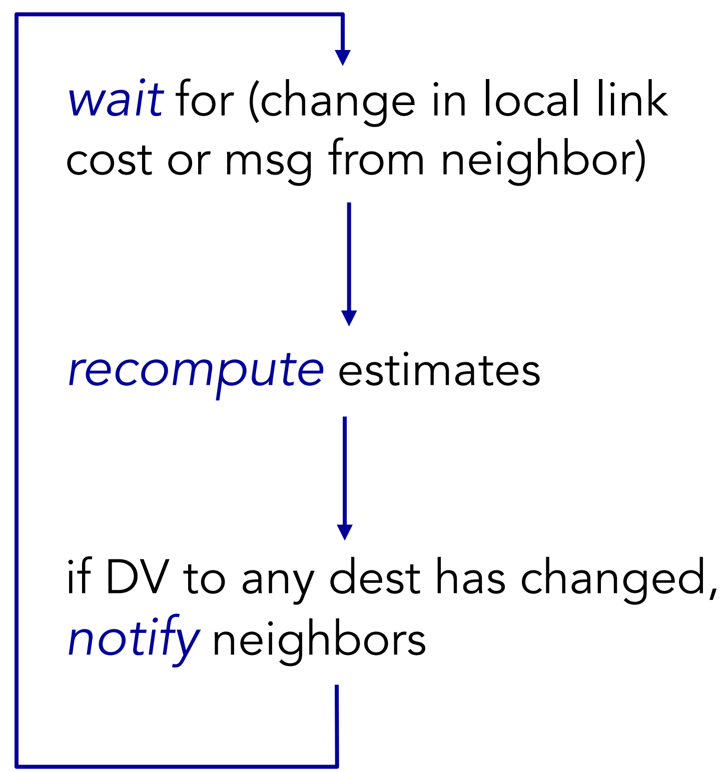
each local iteration caused by:

- □ local link cost change
- □ DV update message from neighbor

distributed:

- □ each node notifies neighbors *only* when its DV changes
  - ▪ neighbors then notify their neighbors if necessary

each node:



$$D_x(y) = \min\{c(x,y) + D_y(y),\ c(x,z) + D_z(y)\} = \min\{2+0\ ,\ 7+1\} = 2$$

$$D_x(z) = \min\{c(x,y) + D_y(z),\ c(x,z) + D_z(z)\} = \min\{2+1\ ,\ 7+0\} = 3$$

node x table

from \ cost to	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

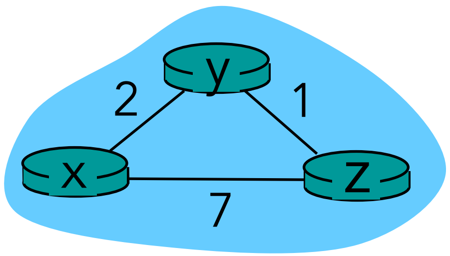
node y table

from \ cost to	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

node z table

from \ cost to	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

time



$D_x(y) = \min\{c(x,y) + D_y(y), c(x,z) + D_z(y)\}$
$= \min\{2+0 , 7+1\} = 2$

$D_x(z) = \min\{c(x,y) +$
$D_y(z), c(x,z) + D_z(z)\}$
$= \min\{2+1 , 7+0\} = 3$

node x table

from \ cost to	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

node y table

from \ cost to	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	7
y	2	0	1
z	7	1	0

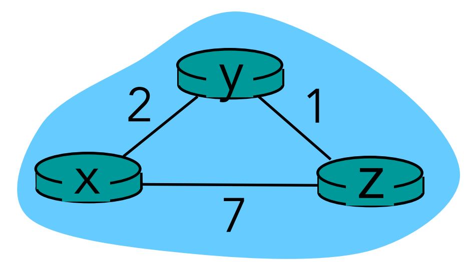
from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

node z table

from \ cost to	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

from \ cost to	x	y	z
x	0	2	7
y	2	0	1
z	3	1	0

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

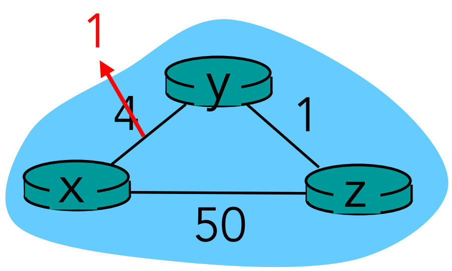


time

Distance vector: link cost changes

Link cost changes:

- node detects local link cost change
- updates routing info, recalculates distance vector
- if DV changes, notify neighbors



Good news travels fast

t_0 : *y* detects link-cost change, updates its DV, informs its neighbors.

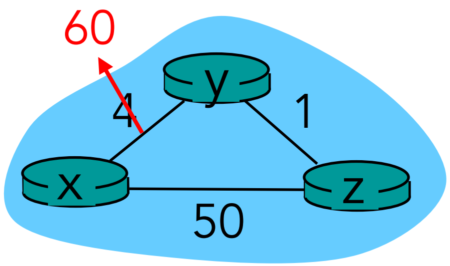
t_1 : *z* receives update from *y*, updates its table, computes new least cost to *x* , sends its neighbors its DV.

t_2 : *y* receives *z*'s update, updates its distance table. *y*'s least costs do *not* change, so *y* does *not* send a message to *z*.

Distance vector: link cost changes

Link cost changes:

- node detects local link cost change
- *bad news travels slow* - "count to infinity" problem!
- 44 iterations before algorithm stabilizes: see text

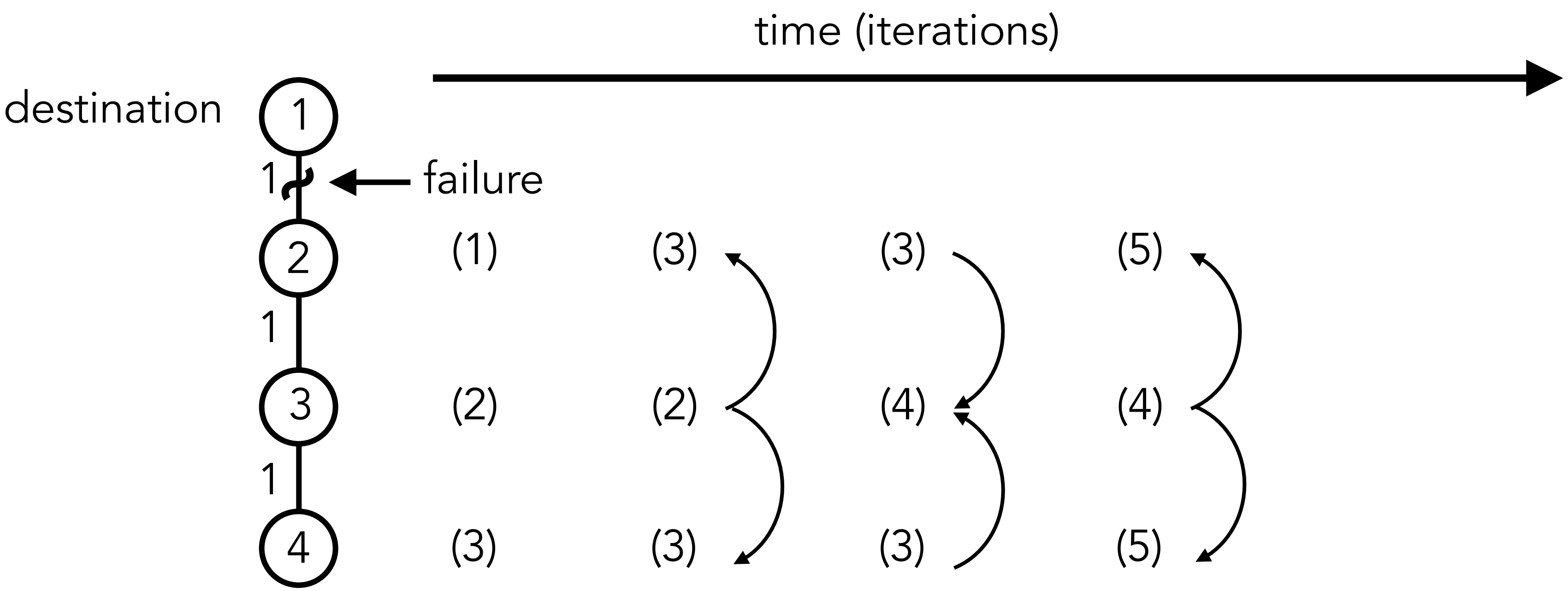


Poisoned reverse:

- If z routes through y to get to x :
  - z tells y its (z's) distance to x is infinite (so y won't route to x via z)
- will this completely solve count to infinity problem?

Count-to-Infinity Problem

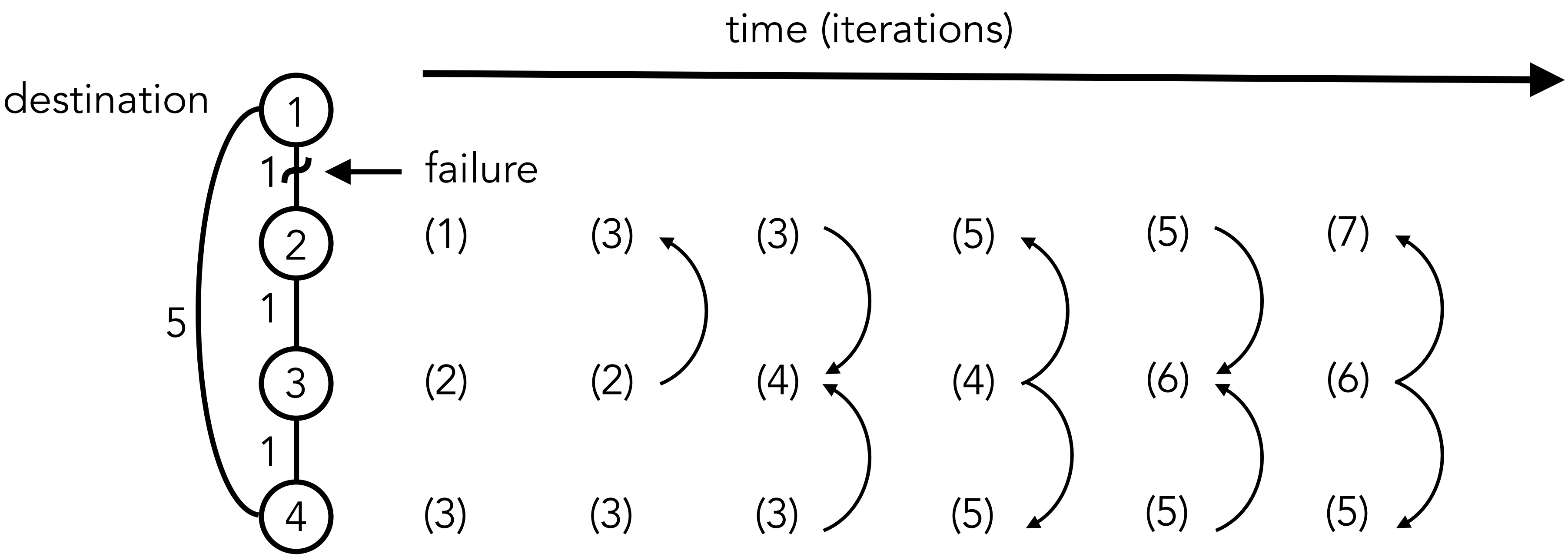
- Slow convergence under topology change
- An example



➔ Need to set a maximum cost

Count-to-Infinity Problem

- In general, the more unequal the link costs, the longer the convergence time is



Decentralized Routing

- Bellman-Ford algorithm can be readily adapted for decentralized or distributed routing
  - Algorithm requires very little information to be stored at the network nodes
    - Nodes do not require information of network topology
  - At each node: it suffices to have
    i) the length (cost) of its outgoing links,
    ii) the identity of every node, and
    iii) the cost (shortest path) of its immediate neighbors to the destination.

Decentralized Routing

- Each node maintains two tables
  - Distance table:
    - A vector of distances to all destinations, called a ***distance vector***
    - Neighboring nodes exchange distance vectors
  - Routing table:
    - Based on the distance vector, compute (or load) the next-hop node to go to for all destinations and the associated costs

Decentralized Routing

- Initialization (at *v* to destination *d*)
  - $D_v^{(0)} = \infty, v \neq d$
  - $D_d^{(0)} = 0$

- Iterative steps (at h-th step)
  - $D_d^{(h+1)} = 0$
  - $D_v^{(h+1)} = \min_{w \in N(v)} [D_w^{(h)} + l(v,w)], \text{ for each } v \neq d \qquad (*)$

    where $N(v)$ is the set of neighbors of node v

    $l(v,w) = \infty, \text{ if } w \text{ is not a neighbor of } v$

  - Algorithm is well suited for distributed computing, since (*) can be executed at each node *v* in parallel.

Decentralized Routing

- Synchronous Case:
  - (*) is executed at each node ***v*** in parallel (in lock steps), and ***simultaneously***
  - Exchange their results of computation with their neighbors
  - Execute again with index ***h*** incremented by 1
- Pros
  - The algorithm terminates in at most ***N***-1 iterations (*N*: # of all nodes)
- Cons:
  - How to make all the nodes to agree to start/stop each iteration (i.e., synchronization)
  - How to abort the algorithm and start a new version
    - For the case when a link status or cost changes, while it is running

Asynchronous Distributed Algorithm

- Basic Idea: The algorithm operates by executing the iteration from time to time at each node *v*

$$D_v = \min_{w \in N(v)} [l(v,w) + D_w] \qquad (+)$$

  - Each node uses the latest "D_w" received from its neighbors
  - No need of synchronization at all nodes
  - Only requirement is that a node *v* will
    - Eventually execute the B-F equation (+)
    - Eventually transmit this information to its neighbors.

Asynchronous Distributed Algorithm

- Advantages
  - No need for synchronization
  - Need not start with the normal B-F initial condition ($D_v^{(0)} = \infty$)
  - Eliminates the need for an algorithm initialization or restart

- Question: Does this asynchronous algorithm ***converge*** to a shortest path solution?
  - Answer: Yes, provided that each node ***v*** executes (+) and D_v changes are eventually transmitted to its neighbors, etc.

* It can be shown that if a number of link length (cost) changes occur up to some time t_0 and then no other changes occur subsequently, then within a finite amount of time (from t_0), the asynchronous algorithm will find the shortest distance for every node ***v*** (Bertsekas & Gallager).

Comparison of LS and DV algorithms

message complexity

- *LS:* with n nodes, E links, O(nE) msgs sent
- *DV:* exchange between neighbors only
  - convergence time varies

speed of convergence

- *LS:* $O(n^2)$ algorithm requires O(nE) msgs
  - may have oscillations
- *DV:* convergence time varies
  - may be routing loops
  - count-to-infinity problem

robustness: what happens if router malfunctions?

LS:

- node can advertise incorrect *link* cost
- each node computes only its *own* table

DV:

- node can advertise incorrect *path* cost
- each node's table used by others
  - error propagate thru network