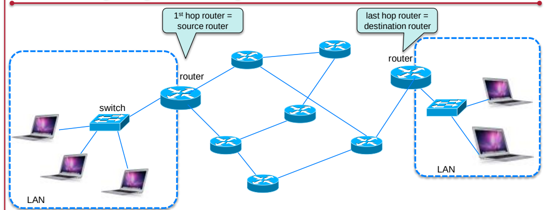
Internet Technology

08. Routing

Paul Krzyzanowski
Rutgers University
Spring 2016

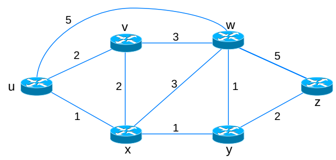
Routing algorithm goal



Routing algorithm: given routers connected with links, what is a good (best?) path from a source to a destination router

good = least cost
cost = time or money

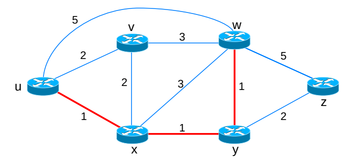
Routing graphs, neighbors, and cost



Graph $G = (N, E)$ N = set of nodes (routers)
E = set of edges (links)
Each edge = pair of connected nodes in N
Node y is a neighbor of node x if $(x, y) \in E$

Cost Each edge has a value representing the cost of the link
$c(x, y)$ = cost of edge between nodes x & y
if $(x, y) \notin E$, then $c(x, y) = \infty$
We will assume $c(x, y) = c(y, x)$

Path cost, least-cost path, & shortest path



A path in a graph $G = (N, E)$ is a sequence of nodes $(x_1, x_2, \ldots, x_p)$ such that each of the pairs $(x_1, x_2), (x_2, x_3), \ldots, (x_{p-1}, x_p)$ are edges in E.
The cost of a path is the sum of edge costs: $c(x_1, x_2), c(x_2, x_3), \ldots, c(x_{p-1}, x_p)$

There could be multiple paths between two nodes, each with a different cost.
One or more of these is a least-cost path.

Example: the least-cost path between u and w is $(u, x, y, w) \Rightarrow c(u, x, y, w) = 3$
If all edges have the same cost, then least-cost path = shortest path

Algorithm classifications

Global routing algorithms
- Compute the least-cost path using complete knowledge of the network
- The algorithm knows the connectivity between all nodes & costs
- Centralized algorithm
- These are link-state (LS) algorithms

Decentralized routing algorithms
- No node has complete information about the costs of all links
- A node initially knows only its direct links
- Iterative process: calculate & exchange info with neighbors
  - Eventually calculate the least-cost path to a destination
- Distance-Vector (DV) algorithm

Additional algorithm classifications

- Static routing algorithms
  - Routes change very slowly over time
- Dynamic routing algorithms
  - Change routing paths as network traffic loads or topology change

- Load-sensitive algorithms
  - Link costs vary to reflect the current level of congestion
- Load-insensitive algorithms
  - Ignore current or recent levels of congestion

Link-State (LS): Dijkstra's Algorithm

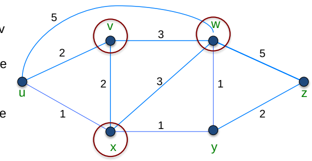
- Assumption:
  Entire network topology & link costs are known
  - Each node broadcasts link-state packets to all other nodes
  - All nodes have an identical, complete view of the network
- Compute least-cost path from one node to all other nodes in the network
- Iterative algorithm
  - After *k* iterations, least-cost paths are known to *k* nodes

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



Initialize:
N' = current node
N' = { *u* }

for all nodes *v*
  if *v* is a neighbor of *u*
    D(*v*) = *c*(*u*, *v*)
  else
    D(*v*) = ∞

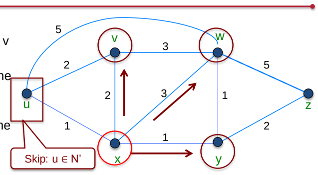
step	*N'*	*D(v), p(v)*	*D(w), p(w)*	*D(x),p(x)*	*D(y),p(y)*	*D(z),p(z)*
0	u	2,u	5,u	1,u	∞	∞

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



Loop until *N'* = *N*:
Find a node *n* not in *N'* such that *D*(*n*) is a minimum
→ Node *x* has minimum *D*(*n*)

add *n* to *N'*
N' = {*u, x*}
for each neighbor *m* of *n* not in *N'*:
  for each neighbor of node x
  D(*m*) = min(*D*(*m*), *D*(*n*) + *c*(*n*, *m*))
  new cost = old cost or cost through x
  if D(*m*) *changed, set p*(*m*) *= n*

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	∞

Least cost path

Cost to v is not better through x

Cost to w is better through x

Ignore x; it is in N'

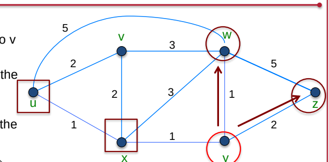
We now have a path to y

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



Loop until *N'* = *N*:
find *n* not in *N'* such that *D*(*n*) is a minimum
→ Nodes *v* & *y* have minimum *D(n)*
  Pick any one: we choose *y*

add *n* to *N'*
N' = {*u, x, y*}
for each neighbor *m* of *n* not in *N'*:
  for each neighbor of node y
  D(*m*) = min(*D*(*m*), *D*(*n*) + *c*(*n*, *m*))
  new cost = old cost or cost through x
  if D(*m*) *changed, set p*(*m*) *= n*

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

Least cost path

Cost to *w* is even better through *y*

Skip: *x* and *y* are in *N'*

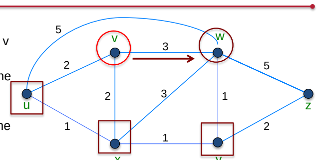
We now have a path to *z*

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



Loop until *N'* = *N*:
find *n* not in *N'* such that *D*(*n*) is a minimum
→ Node *v* has minimum *D(n)*

add *n* to *N'*
N' = {*u, x, y, v*}
for each neighbor *m* of *n* not in *N'*:
  for each neighbor of node v
  D(*m*) = min(*D*(*m*), *D*(*n*) + *c*(*n*,*m*))
  new cost = old cost or cost through x
  if D(*m*) *changed, set p*(*m*) *= n*

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

No improvement (2+3) ≮ 3

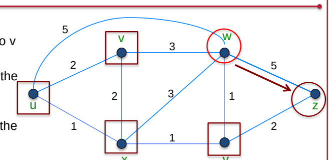
No change: *z* is not a neighbor

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



Loop until *N'* = *N*:
find *n* not in *N'* such that *D*(*n*) is a minimum
→ Node *w* has minimum *D(n)*

add *n* to *N'*
N' = {*u, x, y, v, w*}
for each neighbor m of n not in N':
  for each neighbor of node w
  D(*m*) = min(*D*(*m*), *D*(*n*) + *c*(*n*,*m*))
  new cost = old cost or cost through x
  if D(*m*) *changed, set p*(*m*) *= n*

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

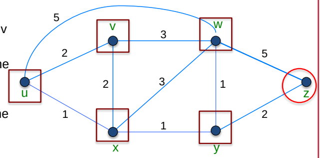
No improvement (3+5) ≮ 4

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



Loop until *N'* = *N*:
find *n* not in *N'* such that *D*(*n*) is a minimum
→ Node *z* is the only one left!

add *n* to *N'*
N' = {*u, x, y, v, w, z*}
for each neighbor m of n not in N':
There are no neighbors not in N'!
We're done

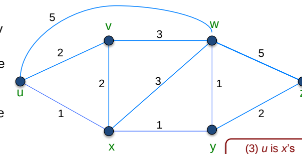
step	N'	D(v), p(v)	D(w), p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



***N'* = *N*:**
All nodes are in *N'*

For each node, we have the total cost from the source and the predecessor along that path.

We can look up the predecessor to find its predecessor
E.g., least-cost path from *u* → *y* is *u* → *x* → *y*

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	uxyvw					4,y
5	uxyvwz					

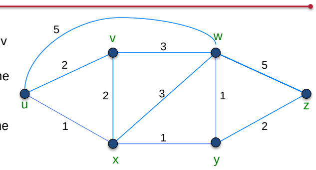
(3) *u* is *x*'s predecessor
(2) *x* is *y*'s predecessor
(1) *y* is *w*'s predecessor

Dijkstra's Algorithm

D(v):
cost of least-cost path from source to v

p(v):
previous node (neighbor of v) along the least-cost path to v

N':
subset of nodes for which we found the least-cost path



We can create a forwarding table that stores the next hop on the least-cost route

Forwarding table for node *u*

Destination	Link
v	*uv*
w	*ux*
x	*ux*
y	*ux*
z	*ux*

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

Computational cost

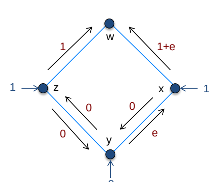
- 1^{st} iteration: search *n* nodes to find the minimum cost node
- 2^{nd} iteration: search *n*-1 nodes
- 3^{rd} iteration: search *n*-2 nodes
- n^{th} iteration: search 1 node

- Total of n iterations = *n* + (*n* - 1) + (*n* - 2) + … 1 = $\sum_{i=0}^{n}(n-i)$
  - We need to search n(n+1)/2 nodes
- Complexity = O(n^2)

Oscillations with congestion-based routing

If *link cost* = load carried on the link

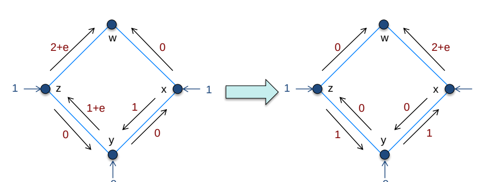
- Link costs are not symmetric
  - *c*(*u*, *v*) = *c*(*v*, *u*) only if the same load flows in both directions
- Example loads
  - Load of 1 comes into *z* for *w*
  - Load of 1 comes into *x* for *w*
  - Load of e comes into *y* for *w*
- When LS is run
  - *y* determines (*y*→*z*→*w*) cost is 1 compared to (*y*→*x*→*w*) cost, which is 1+e
  - x determines that *x*→*y*→*z*→*w* is a lower-cost path



Initial routing

Oscillations with congestion-based routing

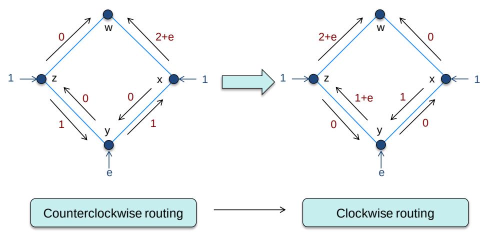
- After route updates, LS is run again
- *x*, *y*, and *z* detect 0-cost path counterclockwise



Clockwise routing → Counterclockwise routing

Oscillations with congestion-based routing

- After route updates, LS is run yet again
- *x*, *y*, and *z* now detect 0-cost path clockwise



Avoiding oscillations

- Ensure that not all routers run the LS algorithm at the same time
  - Avoid synchronized routers by randomizing the time when a router advertises its link state

Distance-Vector Routing Algorithm

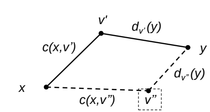
- Initial assumption
  - Each router (node) knows the cost to reach its directly-connected neighbors
- Iterative, asynchronous, distributed algorithm
  - Multiple iterations
    - Each iteration caused by local link cost change or distance vector update message from neighbor
  - Asynchronous
    - Does not require lockstep synchronization
  - Distributed
    - Each node receives information from one or more directly attached neighbors
    - Notifies neighbors only when its distance-vector changes

Bellman-Ford Equation

- What it says
  - If *x* is not directly connected to *y*, it needs to first hop to some neighbor *v*
  - The lowest cost is
    (the cost of the first hop to *v*) + (the lowest cost from *v* to *y*)
    $= c(x, v) + d_v(y)$
  - the least cost path from *x* to *y*, $d_x(y)$, is the minimum of the lowest cost of all of *x*'s neighbors

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

- The value of *v* that satisfies the equation is the forwarding table entry in *x*'s router for destination *y*



Distance-Vector Routing Algorithm

- At each node *x* we store:
  - $c(x, v)$ = cost for the direct link from *x* to *v* for each neighbor *v*
  - $D_x(y)$ = estimate of the cost of the least-cost path from x to y
  - Distance Vector is the set of $D_x(y)$ for all nodes *y* in *N*

    $\mathbf{D}_x = [\, D_x(y): y \in N \,]$ — Least-cost estimates from *x* to all other nodes *y*
  - Distance vectors received from its neighbors

    $\mathbf{D}_v = [\, D_v(y): y \in N \,]$ — Set of least-cost estimates from each neighbor *v* to each node *y*
- Each node *v* periodically sends its distance vector, $\mathbf{D}_v$ to its neighbors
  - When a node receives a distance vector, it saves it and updates its own distance vector using the Bellman-Ford equation

    $D_x(y) = \min_v\{ c(x, v) + D_v(y) \}$ for each node $y \in N$
  - If this results in a change to *x*'s DV, it sends the new DV to its neighbors

Each cost estimate $D_x(y)$ converges to the actual least-cost $D_x(y)$

Distance-Vector Example

Node *x* DV table

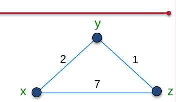
		cost to		
		x	*y*	*z*
from	*x*	0	2	7
	y	∞	∞	∞
	z	∞	∞	∞

Node *y* DV table

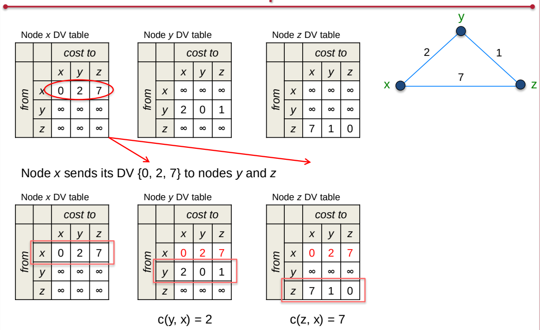
		cost to		
		x	*y*	*z*
from	*x*	∞	∞	∞
	y	2	0	1
	z	∞	∞	∞

Node *z* DV table

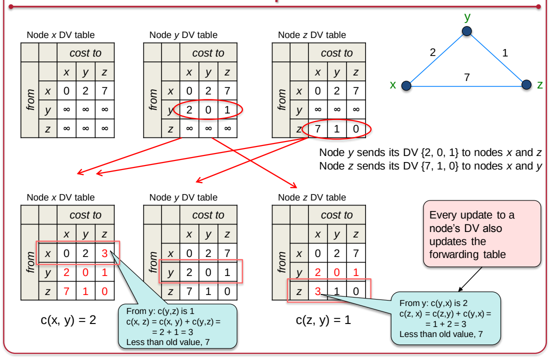
		cost to		
		x	*y*	*z*
from	*x*	∞	∞	∞
	y	∞	∞	∞
	z	7	1	0



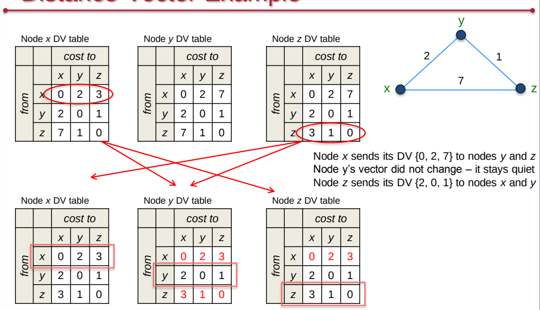
Distance-Vector Example



Distance-Vector Example



Distance-Vector Example

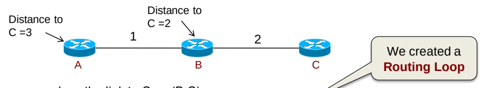


We converged. Everyone has the same view of the network. Nobody has updates to send.

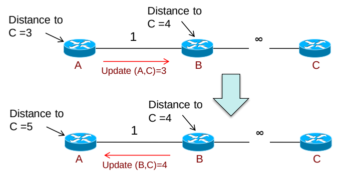
Link cost changes

- The DV algorithm remains quiet once it converges
  - … until some link cost changes
- If a node detects link cost change between itself and a neighbor
  - It updates its distance vector
  - If there is a change in the cost of any least-cost path it informs its neighbors of the new distance vector
  - Each neighbor computes a new least cost
    - If the value changed from its previous value, it sends its DV to its neighbors
    - Recompute until values converge

Link loss



Suppose we lose the link to C: c(B,C) = ∞
B will send an update to A but A thinks its cost to C is 3
B will think there is a route to C: B→A→C with a cost of (c(B,A) + 3) = 4



This continues ad infinitum!

Mitigation: Poison Reverse

- If A routes through B to get to C
  - A will advertise to B that its distance is infinity
  - B will then never attempt to route through A
- This does not work with loops involving 3 or more nodes!
- Other approaches
  - Limit size of network by setting a hop (cost) limit
  - Send full path information in route advertisement
    - Perform explicit queries for loops

The end