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

\textbf{good} = least cost
\textbf{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 $c(x, y) = c(y, x)$
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
    - $D(v) = \infty$

---

**Example Table**

<table>
<thead>
<tr>
<th>step</th>
<th>$N'$</th>
<th>$D(v)$, $p(v)$</th>
<th>$D(w)$, $p(w)$</th>
<th>$D(x)$, $p(x)$</th>
<th>$D(y)$, $p(y)$</th>
<th>$D(z)$, $p(z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u</td>
<td>2,u</td>
<td>5,u</td>
<td>1,u</td>
<td>\infty</td>
<td>\infty</td>
</tr>
</tbody>
</table>

March 21, 2016
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 \(p(m) = n\)
**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 $p(m) = n$

---

<table>
<thead>
<tr>
<th>step</th>
<th>$N'$</th>
<th>$D(v)$, $p(v)$</th>
<th>$D(w)$, $p(w)$</th>
<th>$D(x)$, $p(x)$</th>
<th>$D(y)$, $p(y)$</th>
<th>$D(z)$, $p(z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$u$</td>
<td>2,$u$</td>
<td>5,$u$</td>
<td>1,$u$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>1</td>
<td>$ux$</td>
<td>2,$u$</td>
<td>4,$x$</td>
<td></td>
<td></td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>$uxy$</td>
<td>2,$u$</td>
<td>3,$y$</td>
<td></td>
<td></td>
<td>4,$y$</td>
</tr>
</tbody>
</table>

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

We now have a path to $z$

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

Least cost path
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 \( p(m) = n \)
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 \( p(m) = n \)

---

<table>
<thead>
<tr>
<th>step</th>
<th>( N' )</th>
<th>( D(v), p(v) )</th>
<th>( D(w), p(w) )</th>
<th>( D(x), p(x) )</th>
<th>( D(y), p(y) )</th>
<th>( D(z), p(z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u</td>
<td>2, u</td>
<td>5, u</td>
<td>1, u</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>1</td>
<td>ux</td>
<td>2, u</td>
<td>4, x</td>
<td>2, x</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>2</td>
<td>uxy</td>
<td>2, u</td>
<td>3, y</td>
<td>4, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uxyv</td>
<td>3, y</td>
<td>4, y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>uxyvw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( 4, y )</td>
</tr>
</tbody>
</table>

No improvement \((3+5) \not< 4\)
Dijkstra’s Algorithm

\( D(v) \):  
\text{cost of least-cost path from source to } v

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

\( N' \):  
\text{subset of nodes for which we found the least-cost path}

\text{Loop until } N' = N: 
\text{find } n \text{ not in } N' \text{ such that } D(n) \text{ is a minimum}

\rightarrow \text{Node } z \text{ is the only one left!}

\text{add } n \text{ to } N'

\text{\( N' = \{ u, x, y, v, w, z \} \)}

\text{for each neighbor } m \text{ of } n \text{ not in } N':

\text{\( There \ are \ no \ neighbors \ not \ in \ N'! \)}

\text{\( We're \ done \)}

<table>
<thead>
<tr>
<th>step</th>
<th>( N' )</th>
<th>( D(v), p(v) )</th>
<th>( D(w), p(w) )</th>
<th>( D(x), p(x) )</th>
<th>( D(y), p(y) )</th>
<th>( D(z), p(z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u</td>
<td>2, u</td>
<td>5, u</td>
<td>1, u</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>1</td>
<td>ux</td>
<td>2, u</td>
<td>4, x</td>
<td>2, x</td>
<td>( \infty )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>uxy</td>
<td>2, u</td>
<td>3, y</td>
<td>4, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uxyv</td>
<td>3, y</td>
<td></td>
<td>4, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>uxyvw</td>
<td></td>
<td></td>
<td></td>
<td>4, y</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>uxyvwz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 \( u \rightarrow y \) is \( u \rightarrow x \rightarrow y \)

<table>
<thead>
<tr>
<th>step</th>
<th>( N' )</th>
<th>( D(v), p(v) )</th>
<th>( D(w), p(w) )</th>
<th>( D(x), p(x) )</th>
<th>( D(y), p(y) )</th>
<th>( D(z), p(z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u</td>
<td>2,u</td>
<td>5,u</td>
<td>1,u</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>1</td>
<td>ux</td>
<td>2,u</td>
<td>4,x</td>
<td>2,x</td>
<td>( \infty )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>uxy</td>
<td>2,u</td>
<td>3,y</td>
<td></td>
<td>4,y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uxyv</td>
<td></td>
<td>3,y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>uxyvw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>uxyvwz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Step</th>
<th>(N')</th>
<th>(D(v), p(v))</th>
<th>(D(w), p(w))</th>
<th>(D(x), p(x))</th>
<th>(D(y), p(y))</th>
<th>(D(z), p(z))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(u)</td>
<td>2,(u)</td>
<td>5,(u)</td>
<td>1,(u)</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
<tr>
<td>1</td>
<td>(ux)</td>
<td>2,(u)</td>
<td>4,(x)</td>
<td>2,(x)</td>
<td>(\infty)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(uxy)</td>
<td>2,(u)</td>
<td>3,(y)</td>
<td></td>
<td>4,(y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(uxyv)</td>
<td></td>
<td>3,(y)</td>
<td></td>
<td>4,(y)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(uxyvw)</td>
<td></td>
<td></td>
<td></td>
<td>4,(y)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(uxyvwz)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Forwarding table for node \(u\)

<table>
<thead>
<tr>
<th>Destination</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v)</td>
<td>(uv)</td>
</tr>
<tr>
<td>(w)</td>
<td>(ux)</td>
</tr>
<tr>
<td>(x)</td>
<td>(ux)</td>
</tr>
<tr>
<td>(y)</td>
<td>(ux)</td>
</tr>
<tr>
<td>(z)</td>
<td>(ux)</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm

Computational cost

– 1\textsuperscript{st} iteration: search \( n \) nodes to find the minimum cost node
– 2\textsuperscript{nd} iteration: search \( n-1 \) nodes
– 3\textsuperscript{rd} iteration: search \( n-2 \) nodes
– \( n\textsuperscript{th} \) iteration: search 1 node

– Total of \( n \) iterations = \( n + (n - 1) + (n - 2) + \ldots + 1 = \sum_{i=0}^{n}(n - i) \)
  • We need to search \( n(n+1)/2 \) nodes
– Complexity = \( O(n^2) \)
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 $(y \rightarrow z \rightarrow w)$ cost is 1 compared to $(y \rightarrow x \rightarrow w)$ cost, which is $1 + e$
  - $x$ determines that $x \rightarrow y \rightarrow z \rightarrow w$ is a lower-cost path
After route updates, LS is run again

- $x$, $y$, and $z$ detect 0-cost path counterclockwise
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
    
    $$(\text{the cost of the first hop to } v) + (\text{the lowest cost from } v \text{ to } y)$$
    
    $$= c(x, v) + d_v(y)$$
  – 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 \( N \)
    
    \[
    D_x = [ D_x(y) : y \in N ]
    \]
    
    Least-cost estimates from \( x \) to all other nodes \( y \)
  
  – Distance vectors received from its neighbors
    
    \[
    D_v = [ D_v(y) : y \in N ]
    \]
    
    Set of least-cost estimates from each neighbor \( v \) to each node \( y \)
  
• Each node \( v \) periodically sends its distance vector, \( D_v \) to its neighbors
  
  – When a node receives a distance vector, it saves it and updates its own distance vector using the Bellman-Ford equation
    
    \[
    D_x(y) = \min_V \{ c(x, v) + D_v(y) \} \quad \text{for each node } y \in N
    \]
  
  – 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

<table>
<thead>
<tr>
<th></th>
<th>cost to</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>∞</td>
</tr>
<tr>
<td>z</td>
<td>∞</td>
</tr>
</tbody>
</table>

Node x DV table

<table>
<thead>
<tr>
<th></th>
<th>cost to</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>∞</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>∞</td>
</tr>
</tbody>
</table>

Node y DV table

<table>
<thead>
<tr>
<th></th>
<th>cost to</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>∞</td>
</tr>
<tr>
<td>y</td>
<td>∞</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
</tr>
</tbody>
</table>

Node z DV table

![Diagram of a triangle with nodes x, y, and z connected by edges with costs 2, 7, and 1]
Distance-Vector Example

Node $x$ sends its DV $\{0, 2, 7\}$ to nodes $y$ and $z$

<table>
<thead>
<tr>
<th></th>
<th>$cost$ to</th>
<th></th>
<th>$cost$ to</th>
<th></th>
<th>$cost$ to</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>$x$</td>
<td>$y$</td>
<td>$z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td>$0$</td>
<td></td>
<td></td>
<td>$2$</td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$0$</td>
</tr>
<tr>
<td>$z$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$cost$ to</th>
<th></th>
<th>$cost$ to</th>
<th></th>
<th>$cost$ to</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>$x$</td>
<td>$y$</td>
<td>$z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td>$0$</td>
<td></td>
<td></td>
<td>$2$</td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$0$</td>
</tr>
<tr>
<td>$z$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$cost$ to</th>
<th></th>
<th>$cost$ to</th>
<th></th>
<th>$cost$ to</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>$x$</td>
<td>$y$</td>
<td>$z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td>$0$</td>
<td></td>
<td></td>
<td>$2$</td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$0$</td>
</tr>
<tr>
<td>$z$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

$c(y, x) = 2$  
$c(z, x) = 7$
Distance-Vector Example

**Node x DV table**

<table>
<thead>
<tr>
<th>from</th>
<th>cost to</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
</tr>
</tbody>
</table>

**Node y DV table**

<table>
<thead>
<tr>
<th>from</th>
<th>cost to</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
</tr>
</tbody>
</table>

**Node z DV table**

<table>
<thead>
<tr>
<th>from</th>
<th>cost to</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>3</td>
</tr>
</tbody>
</table>

Node y sends its DV {2, 0, 1} to nodes x and z
Node z sends its DV {7, 1, 0} to nodes x and y

From y: \( c(y, z) = 1 \)  
\( c(x, z) = c(x, y) + c(y, z) = 2 + 1 = 3 \)  
Less than old value, 7

From y: \( c(y, x) = 2 \)  
\( c(z, x) = c(z, y) + c(y, x) = 1 + 2 = 3 \)  
Less than old value, 7

Every update to a node's DV also updates the forwarding table
Distance-Vector Example

Node x DV table

<table>
<thead>
<tr>
<th>from</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

cost to

Node y DV table

<table>
<thead>
<tr>
<th>from</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

cost to

Node z DV table

<table>
<thead>
<tr>
<th>from</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

cost to

Node x sends its DV {0, 2, 7} to nodes y and z
Node y’s vector did not change – it stays quiet
Node z sends its DV {2, 0, 1} to nodes x and y

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
Suppose we lose the link to C: \( c(B,C) = \infty \)

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: \( B \to A \to C \) 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