Routing Algorithms

Graph abstraction

Graph: \( G = (N, E) \)

- \( N \) = 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)\}

Remark: Graph abstraction is useful in other network contexts

Example: P2P, where \( N \) is set of peers and \( E \) is set of TCP connections

Graph abstraction: costs

- \( c(x, x') \) = cost of link \((x, x')\)
  - e.g., \( c(w, z) = 5 \)

- Cost could always be 1, or inversely related to bandwidth, or inversely related to congestion

Cost of path \((x_1, x_2, x_3, ..., x_p)\) = \( c(x_1, x_2) + c(x_2, x_3) + ... + c(x_{p-1}, x_p) \)

What’s the least-cost path between \( u \) and \( z \)?

Routing algorithm: algorithm that finds least-cost path

Placing routing into context
Routing Algorithm classification

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

Static or dynamic?
- Static:
  - routes change slowly over time
- Dynamic:
  - routes change more quickly
  - periodic update
  - in response to link cost changes

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

A Link-State Routing Algorithm: example

Dijkstra’s algorithm: example

Dijsktra’s Algorithm

1 Initialization:
2 \( N' = \{ u \} \)
3 for all nodes v
4 if v adjacent to u
5 then \( D(v) = c(u,v) \)
6 else \( D(v) = \infty \)
7 Loop
8 find w not in \( N' \) such that \( D(w) \) is a minimum
9 add w to \( N' \)
10 update \( D(v) \) for all v adjacent to w and not in \( N' \):
11 \( D(v) = \min( D(v), D(w) + c(w,v) ) \)
12 new cost to v is either old cost to v or known
13 shortest path cost to w plus cost from w to v + y
14 until all nodes in \( N' \)
Dijkstra’s algorithm: an example

Resulting shortest-path tree from u:

Resulting forwarding table in u:

<table>
<thead>
<tr>
<th>destination</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>(u,v)</td>
</tr>
<tr>
<td>x</td>
<td>(u,x)</td>
</tr>
<tr>
<td>y</td>
<td>(u,x)</td>
</tr>
<tr>
<td>w</td>
<td>(u,x)</td>
</tr>
<tr>
<td>z</td>
<td>(u,x)</td>
</tr>
</tbody>
</table>

Distance Vector Algorithm

Bellman-Ford Equation

Define

\[ d_x(y) := \text{cost of least-cost path from } x \text{ to } y \]

Then

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

where min is taken over all neighbors \( v \) of \( x \)

Bellman-Ford example

Clearly, \( d_u(z) = 5, d_v(z) = 3, d_w(z) = 3 \)

B-F equation says:

\[ d_x(z) = \min \left\{ c(u,v) + d_v(z), c(u,x) + d_x(z), c(u,w) + d_w(z) \right\} \]

\[ = \min \left( 2 + 5, 1 + 3, 5 + 3 \right) = 4 \]

Node that achieves minimum is next hop in shortest path \( \rightarrow \) forwarding table

Distance Vector Algorithm

- \( D_x(y) = \text{estimate of least cost from } x \text{ to } y \)
- Node \( x \) knows cost to each neighbor \( v: c(x,v) \)
- Node \( x \) maintains distance vector \( D_x = [D_x(y): y \in N ] \)
- Node \( x \) also maintains its neighbors’ distance vectors
  - For each neighbor \( v, x \) maintains
  \( D_v = [D_v(y): y \in N ] \)
Distance vector algorithm

- Each node periodically sends its own distance vector estimate to neighbors.
- When a node \( x \) receives new DV estimate from neighbor, it updates its own DV using B-F equation:
  \[
  D_x(y) \leftarrow \min_v \{c(x,v) + D_v(y)\} \quad \text{for each node } y \in N
  \]
- 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:

1. wait for (change in local link cost or msg from neighbor)
2. recompute estimates
3. if DV to any dest has changed, notify neighbors

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

<table>
<thead>
<tr>
<th>start</th>
<th>time</th>
<th>end</th>
<th>start</th>
<th>time</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>15</td>
<td>2</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>20</td>
<td>7</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

\[
D_x(y) = \min \{c(x,y) \cdot D_y(y), c(x,z) \cdot D_z(y)\}
\]

\[
D_x(z) = \min \{c(x,y) + D_y(z), c(x,z) + D_z(z)\}
\]
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”

At time $t_0$, $y$ detects the link-cost change, updates its DV, and informs its neighbors.

At time $t_1$, $z$ receives the update from $y$ and updates its table. It computes a new least cost to $x$ and sends its neighbors its DV.

At time $t_2$, $y$ receives $z$’s update and updates its distance table. $y$’s least costs do not change and hence $y$ does not send any message to $z$.

Distance Vector: link cost changes

**Link cost changes:**
- good news travels fast
- 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?