Routing algorithm goal

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
- \( c(x, y) = \text{cost of edge between nodes } x \& y \)
- If \((x, y) \not\in E\), 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 \((x_i, x_{i+1})\) 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 \((u, x, y, w)\)

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
**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

**Iterative algorithm**
- After k iterations, least-cost paths are known to k nodes

---

**Dijkstra's Algorithm**

- **cost of least-cost path from source to 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:**
- \( N' = \text{current node} \)
- \( N' = \{ v \} \)
- for all nodes \( v \)
  - if \( v \) is a neighbor of \( u \)
    - \( D(v) = d(u, v) \)
    - \( D(v) = = \)

**Loop until \( N' = \text{all nodes} \)**
- Find node \( n \) not in \( N' \) such that \( D(n) \) is a minimum
  - Cost to \( n \) is not better through \( x \)
  - Cost to \( n \) through \( x \)
  - Cost to \( n \) is in \( N' \)
  - Cost to \( n \) is even better through \( x \)
  - Skip: \( x \) and \( y \) are in \( N' \)
  - We now have a path to \( n \)

---

**Link-State (LS): Dijkstra's Algorithm**

- **Assumption:**
- Each node broadcasts link-state packets to all other nodes
- All nodes have an identical, complete view of the network

- **Iterative algorithm**
- After k iterations, least-cost paths are known to k nodes

---

**Dijkstra's Algorithm**

- **cost of least-cost path from source to 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:**
- \( N' = \text{current node} \)
- \( N' = \{ v \} \)
- for all nodes \( v \)
  - if \( v \) is a neighbor of \( u \)
    - \( D(v) = d(u, v) \)
    - \( D(v) = = \)

**Loop until \( N' = \text{all nodes} \)**
- Find node \( n \) not in \( N' \) such that \( D(n) \) is a minimum
  - Cost to \( n \) is not better through \( x \)
  - Cost to \( n \) through \( x \)
  - Cost to \( n \) is in \( N' \)
  - Cost to \( n \) is even better through \( x \)
  - Skip: \( x \) and \( y \) are in \( N' \)
  - We now have a path to \( n \)
After route updates, LS is run again.

Load of 1 comes into Example loads.

When LS is run, we need to search $n(n+1)/2$ nodes.

Link costs are not symmetric.

c1 determines that $y ightarrow z$.

Dijkstra's Algorithm

Loop until $N'$ is $N$:
- find $n$ not in $N'$ such that $D(n)$ is a minimum.
- add $n$ to $N'$.
- $N' = \{u, x, y, w, z\}$ for each neighbor $m$ of $n$ not in $N'$.
- There are no neighbors not in $N'$.
- We're done.

Total of $n$ iterations.

Complexity = $O(n)$.

Load of $e$ comes into Link costs.

Dijkstra's Algorithm

Cost of least path from source to $v$.

Previous node (neighbor of $v$) along the least-cost path.

Subset of nodes for which we found the least-cost path.

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 the predecessor.

E.g., least-cost path from $u$ to $y$,$ v$ so $u \rightarrow x \rightarrow y$.

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 $(y \rightarrow z \rightarrow w)$ 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.

Oscillations with congestion-based routing

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

March 21, 2016
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

Distance-Vector Routing Algorithm

- At each node \( x \) we store:
  - \( d(x, y) \) = cost for the direct link from \( x \) to \( y \) for each neighbor \( y \)
  - \( \Delta(x) \) = estimate of the cost of the least-cost path from \( x \) to \( y \)
  - Distance Vector is the set of \( \Delta(x) \) for all nodes \( y \) in \( N \)
    \[ \Delta_x = \{ \Delta(x) \mid y \in N \} \]
    - Least-cost estimates from \( x \) to all other nodes \( y \)
  - \( \delta_x \) = distances vectors received from its neighbors
    \[ \delta_x = \{ \Delta(y) \mid y \in N \} \]
    - Set of least-cost estimates from each neighbor \( y \) to each node \( y \)

- Each node \( v \) periodically sends its distance vector, \( \Delta(v) \) to its neighbors
  - When a node receives a distance vector, it saves it and updates its own distance vector using the Bellman-Ford equation:
    \[ \Delta_y = \min \{ \Delta(v) + d(y) \} \] 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 \( \Delta_y \) converges to the actual least-cost \( d_y \)

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 \( \Delta(x) \) the cost of the first hop to \( v \) = (the lowest cost from \( v \) to \( y \))
  - \( d(y) = \min \{ \Delta(x) + d(y) \} \)
  - The value of \( v \) that satisfies the equation is the forwarding table entry in \( x's \) router for destination \( y \)

Distance-Vector Example
The DV algorithm remains quiet once it converges. A will advertise to B that its distance is infinity. B will then never attempt to route through A. If the value changed from its previous value, it sends its DV to its neighbors. Limit size of network by setting a hop (cost) limit. 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. 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

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!

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

Distance-Vector Example

Node x DV table
- Node y DV table
- Node z DV table

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

Node x DV table
- Node y DV table
- Node z DV table

DV table:

- Node x: Distance to y = 2
- Node y: Distance to x = 2

Distance to C = 3

Suppose we lose the link to C: c(B,C) = ∞
A will send an update to B but A thinks its cost to C is 3.
Node y’s vector did not change — it stays quiet.
Node z sends its DV {2, 0, 1} to nodes x and y.
Node x sends its DV {2, 0, 7} to nodes y and z.
We created a Routing Loop.

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

Distance-Vector Example

Node x DV table
- Node y DV table
- Node z DV table

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

Distance to C = 4

From y: c(y, z) = 2 + 1 = 3
From z, y, and x:
c(y, z) = c(z, y) + c(y, z) = 1 + 2 = 3

Distance to C = 2

From y: c(y, z) = 2 + 1 = 3
From z, y, and x:
c(y, z) = c(z, y) + c(y, z) = 1 + 2 = 3

Distance to C = 1

From y: c(y, z) = 2 + 1 = 3
From z, y, and x:
c(y, z) = c(z, y) + c(y, z) = 1 + 2 = 3

Distance to C = 0

From y: c(y, z) = 2 + 1 = 3
From z, y, and x:
c(y, z) = c(z, y) + c(y, z) = 1 + 2 = 3

DV table:

- Node x: Distance to y = 2
- Node y: Distance to x = 2

Distance to C = 3

Suppose we lose the link to C: c(B,C) = ∞
A will send an update to B 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!
The end