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 algorithm:

Routing graphs, neighbors, and cost

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

- \( N \) = set of nodes (routers)
- \( E \) = set of edges (links)

Each edge = pair of 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 \((x_1, x_2, ..., x_p)\)

such that each of the pairs \((x_1, x_2), (x_2, x_3), ..., (x_{p-1}, x_p)\) are edges in \( E \).

The cost of a path is the sum of edge costs:

\[ c(x_1, x_2), c(x_2, x_3), ..., c(x_{p-1}, x_p) \]

There could be multiple paths between two nodes, each with a different cost.

If all edges have the same cost, then least-cost path = shortest path

Algorithm classifications

- Global routing algorithms
  - Computes 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

\[ N' = \{ u, x, y, v \} \]

\[ \rightarrow \text{Node } v \text{ has minimum cost path} \]

\[ \text{Loop until \( N' \) is a minimum} \]

- **Dijkstra's Algorithm**
  - Cost of least-cost path from source to \( v \)
  - Previous node (neighbor of \( v \)) along the least-cost path to \( v \)
  - Subset of nodes for which we found the least-cost path

**Initialize:**
- \( N' = \text{current node} \)

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

\[ p(v) = \text{previous node (neighbor of } v \text{)} \]

\[ D(v) = \text{cost path to } v \]

\[ D(v) = \text{cost path from source to } v \]

\[ \text{path to } v \text{ is even better through } x \]

\[ \text{we have a path to } v \text{ now} \]

\[ \text{node } v \text{ has minimum cost path} \]

\[ \text{Dijkstra's Algorithm} \]

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

\[ p(v) = \text{previous node (neighbor of } v \text{)} \]

\[ N' = \text{subset of nodes for which we found the least-cost path} \]

\[ \text{Loop until } N' \text{ is a minimum} \]

- **Dijkstra's Algorithm**
  - Cost of least-cost path from source to \( v \)
  - Previous node (neighbor of \( v \)) along the least-cost path to \( v \)
  - Subset of nodes for which we found the least-cost path

**Initialize:**
- \( N' = \text{current node} \)

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

\[ p(v) = \text{previous node (neighbor of } v \text{)} \]

\[ D(v) = \text{cost path to } v \]

\[ D(v) = \text{cost path from source to } v \]

\[ \text{path to } v \text{ is even better through } x \]

\[ \text{we have a path to } v \text{ now} \]

\[ \text{node } v \text{ has minimum cost path} \]

\[ \text{Dijkstra's Algorithm} \]

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

\[ p(v) = \text{previous node (neighbor of } v \text{)} \]

\[ N' = \text{subset of nodes for which we found the least-cost path} \]

\[ \text{Loop until } N' \text{ is a minimum} \]

- **Dijkstra's Algorithm**
  - Cost of least-cost path from source to \( v \)
  - Previous node (neighbor of \( v \)) along the least-cost path to \( v \)
  - Subset of nodes for which we found the least-cost path

**Initialize:**
- \( N' = \text{current node} \)

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

\[ p(v) = \text{previous node (neighbor of } v \text{)} \]

\[ D(v) = \text{cost path to } v \]

\[ D(v) = \text{cost path from source to } v \]

\[ \text{path to } v \text{ is even better through } x \]

\[ \text{we have a path to } v \text{ now} \]

\[ \text{node } v \text{ has minimum cost path} \]
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' is N**: find n not in N' such that D(n) is a minimum
- Node n 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 can create a forwarding table** that stores the next hop on the least-cost route

- **Forwarding table for node u**

**Oscillations with congestion-based routing**

- **Link costs** = 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 next run**
  - y determines (y → z → w) cost is 1
  - x determines that x → y → z → w is a lower-cost path

**Computational cost**
- 1st iteration: search n nodes to find the minimum cost node
- 2nd iteration: search n-1 nodes
- 3rd iteration: search n-2 nodes
- nth iteration: search n(n+1)/2
- Complexity = O(n^2)

**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 that 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
  - Let \( d_{x}(y) \) be the cost of the least-cost path from x to y
  - The lowest cost is \( c(x, v) + d_{v}(y) \)
  - 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:
  - \( c(x, y) \) = 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
  - \( D_{x} = \{ D_{x}(y); y \in N \} \)
  - Distance vectors of its neighbors
    \( D_{x} = \{ D_{y}(v); v \in N \} \)
  - Each node periodically sends its distance vector, \( D_{x} \), 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 one to its neighbors
    Each cost estimate \( D_{x}(y) \) converges to the actual least-cost \( d_{x}(y) \)

Distance-Vector Example
The DV algorithm remains quiet once it converges. If the value changed from its previous value, it sends its DV to its neighbors. If there is a change in the cost of any least-cost path to a node, it updates its distance vector. Other approaches include recomputing routes, sending explicit queries for loops, or using poison reverse to mitigate routing loops.

### 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 to a node, its DV informs its neighbors of the new distance vector.
  - Each neighbor computes a new least-cost path to a node.
  - If the value changed from its previous value, it sends its DV to its neighbors.
  - Recompute until values converge.

### 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