Recap: Finding best paths

No one notion of “best”
Finesse the issue using link costs
Goal becomes finding least cost or shortest path
Distance vector is one way to do it
  • Exchange a vector of known destinations and cost
  • Suffers count to infinity—heuristics help but are not foolproof
Link-State Routing
Link-State Routing

• Second broad class of routing algorithms
  • More computation than DV but better dynamics

• Widely used in practice
  • Used in Internet/ARPANET from 1979
  • Modern networks use OSPF (L3) and IS-IS (L2)
Link-State Setting

Same distributed setting as for distance vector:

1. Nodes know only the cost to their neighbors; not topology
2. Nodes can talk only to their neighbors using messages
3. All nodes run the same algorithm concurrently
4. Nodes/links may fail, messages may be lost
Link-State Algorithm

Proceeds in two phases:

1. Nodes **flood** topology with link state packets
   - Each node learns full topology

2. Each node computes its own forwarding table
   - By running Dijkstra (or equivalent)
Part 1: Flooding
Flooding

• Rule used at each node:
  • Sends an incoming message on to all other neighbors
  • Remember the message so that it is only flood once
Flooding (2)

• Consider a flood from A; first reaches B via AB, E via AE
Flooding (3)

• Next B floods BC, BE, BF, BG, and E floods EB, EC, ED, EF
Flooding (4)

- C floods CD, CH; D floods DC; F floods FG; G floods GF
- F gets another copy
Flooding (5)

• H has no-one to flood ... and we’re done

Each link carries the message, and in at least one direction
Flooding Details

• Remember message (to stop flood) using source and sequence number
  • So next message (with higher sequence) will go through
• To make flooding reliable, use ARQ
  • So receiver acknowledges, and sender resends if needed

Problem?
Flooding Problem

• F receives the same message multiple times

E and B send to each other too
Part 2: Dijkstra’s Algorithm
Edsger W. Dijkstra (1930-2002)

- Famous computer scientist
  - Programming languages
  - Distributed algorithms
  - Program verification

- Dijkstra’s algorithm, 1969
  - Single-source shortest paths, given network with non-negative link costs

By Hamilton Richards, CC-BY-SA-3.0, via Wikimedia Commons
Dijkstra’s Algorithm

Algorithm:

- Mark all nodes tentative, set distances from source to 0 (zero) for source, and \( \infty \) (infinity) for all other nodes
- While tentative nodes remain:
  - Extract N, a node with lowest distance
  - Add link to N to the shortest path tree
  - Relax the distances of neighbors of N by lowering any better distance estimates
Dijkstra’s Algorithm (2)

• Initialization

We’ll compute shortest paths from A
Dijkstra’s Algorithm (3)

- Relax around A
Dijkstra’s Algorithm (4)

• Relax around B

Distance fell!
Dijkstra’s Algorithm (5)

• Relax around C

Distance fell again!
Dijkstra’s Algorithm (6)

- Relax around G (say)

Didn’t fall ...

CSE 461 University of Washington
Dijkstra’s Algorithm (7)

- Relax around F (say)

Relax has no effect
Dijkstra’s Algorithm (8)

- Relax around E
Dijkstra’s Algorithm (9)

- Relax around D
Dijkstra’s Algorithm (10)

• Finally, H ... done
Dijkstra Comments

- Finds shortest paths in order of increasing distance from source
  - Leverages optimality property
- Runtime depends on cost of extracting min-cost node
  - Superlinear in network size (grows fast)
  - Using Fibonacci Heaps the complexity is $O(|E| + |V| \log |V|)$
- Gives complete source/sink tree
  - More than needed for forwarding!
  - But requires complete topology
Bringing it all together...
Phase 1: Topology Dissemination

• Each node floods link state packet (LSP) that describes their portion of the topology

Node E’s LSP flooded to A, B, C, D, and F

<table>
<thead>
<tr>
<th>Seq. #</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
</tbody>
</table>
Phase 2: Route Computation

• Each node has full topology
  • By combining all LSPs

• Each node simply runs Dijkstra
  • Replicated computation, but finds required routes directly
  • Compile forwarding table from sink/source tree
  • That’s it folks!
Forwarding Table

Source Tree for E (from Dijkstra)

E’s Forwarding Table

<table>
<thead>
<tr>
<th>To</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>--</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td>C</td>
</tr>
</tbody>
</table>
## Handling Changes

- On change, flood updated LSPs, re-compute routes
  - E.g., nodes adjacent to failed link or node initiate

### B’s LSP

<table>
<thead>
<tr>
<th>Seq. #</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>∞</td>
</tr>
</tbody>
</table>

### F’s LSP

<table>
<thead>
<tr>
<th>Seq. #</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>∞</td>
</tr>
</tbody>
</table>

![Graph](image-url)

- **G XXXX**
- **Failure!**

---

CSE 461 University of Washington
Handling Changes (2)

• Link failure
  • Both nodes notice, send updated LSPs
  • Link is removed from topology

• Node failure
  • All neighbors notice a link has failed
  • Failed node can’t update its own LSP
  • But it is OK: all links to node removed
Handling Changes (3)

• Addition of a link or node
  • Add LSP of new node to topology
  • Old LSPs are updated with new link

• Additions are the easy case ...
Link-State Complications

• Things that can go wrong:
  • Seq. number reaches max, or is corrupted
  • Node crashes and loses seq. number
  • Network partitions then heals

• Strategy:
  • Include age on LSPs and forget old information that is not refreshed

• Much of the complexity is due to handling corner cases
## DV/LS Comparison

<table>
<thead>
<tr>
<th>Goal</th>
<th>Distance Vector</th>
<th>Link-State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Distributed Bellman-Ford</td>
<td>Replicated Dijkstra</td>
</tr>
<tr>
<td>Efficient paths</td>
<td>Approx. with shortest paths</td>
<td>Approx. with shortest paths</td>
</tr>
<tr>
<td>Fair paths</td>
<td>Approx. with shortest paths</td>
<td>Approx. with shortest paths</td>
</tr>
<tr>
<td>Fast convergence</td>
<td>Slow – many exchanges</td>
<td>Fast – flood and compute</td>
</tr>
<tr>
<td>Scalability</td>
<td>Excellent – storage/compute</td>
<td>Moderate – storage/compute</td>
</tr>
</tbody>
</table>
IS-IS and OSPF Protocols

• Widely used in large enterprise and ISP networks
  • IS-IS = Intermediate System to Intermediate System
  • OSPF = Open Shortest Path First

• Link-state protocol with many added features
  • E.g., “Areas” for scalability
Equal-Cost Multi-Path Routing
Multipath Routing

• Allow multiple routing paths from node to destination be used at once
  • Topology has them for redundancy
  • Using them can improve performance

• Questions:
  • How do we find multiple paths?
  • How do we send traffic along them?
Equal-Cost Multipath Routes

• One form of multipath routing
  • Extends shortest path model by keeping set if there are ties

• Consider $A \to E$
  • $ABE = 4 + 4 = 8$
  • $ABCE = 4 + 2 + 2 = 8$
  • $ABCDE = 4 + 2 + 1 + 1 = 8$
  • Use them all!
Source “Trees”

• With ECMP, source/sink “tree” is a directed acyclic graph (DAG)
  • Each node has set of next hops
  • Still a compact representation
Source “Trees” (2)

• Find the source “tree” for E
  • Procedure is Dijkstra, simply remember set of next hops
  • Compile forwarding table similarly, may have set of next hops

• Straightforward to extend DV too
  • Just remember set of neighbors
Source “Trees” (3)

Source Tree for E

E’s Forwarding Table

<table>
<thead>
<tr>
<th>Node</th>
<th>Next hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C, D</td>
</tr>
<tr>
<td>B</td>
<td>B, C, D</td>
</tr>
<tr>
<td>C</td>
<td>C, D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>--</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td>C, D</td>
</tr>
</tbody>
</table>
Forwarding with ECMP

• Could randomly pick a next hop for each packet based on destination
  • Balances load, but adds jitter

• Try sending packets from a flow on the same path
  • Flow identified using 5-tuple
  • Map flow identifier to single next hop
  • No jitter within flow, but less balanced
Forwarding with ECMP (2)

Multipath routes from F/E to C/H

E’s Forwarding Choices

<table>
<thead>
<tr>
<th>Flow</th>
<th>Possible next hops</th>
<th>Example choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>F → H</td>
<td>C, D</td>
<td>D</td>
</tr>
<tr>
<td>F → C</td>
<td>C, D</td>
<td>D</td>
</tr>
<tr>
<td>E → H</td>
<td>C, D</td>
<td>C</td>
</tr>
<tr>
<td>E → C</td>
<td>C, D</td>
<td>C</td>
</tr>
</tbody>
</table>

Use both paths to get to one destination