NAT: Network Address Translation

Advantages:
• can change address of devices in local network without notifying outside world
• devices inside local net not explicitly addressable by or visible to the outside world (a security plus)

Disadvantage:
• devices inside local net not explicitly addressable by or visible to the outside world, making peer-to-peer networking that much harder
• routers should only process up to layer 3 (port#'s are app layer objects)
• address shortage should instead be solved by IPv6, instead NAT hinders the adoption of IPv6 (nothing wrong with that?)

Lesson:
Be careful what you propose as a “temporary” patch, “temporary” solutions have a tendency to stay around beyond expiration date

“The evil that men do lives after them, the good is oft interred with their bones.”
– Shakespeare, Julius Caesar

The Internet Network Layer

Host, router network layer functions:

<table>
<thead>
<tr>
<th>Transport layer: TCP, UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing protocols</td>
</tr>
<tr>
<td>• path selection</td>
</tr>
<tr>
<td>• RIP, OSPF, BGP</td>
</tr>
<tr>
<td>Forwarding protocol (IP)</td>
</tr>
<tr>
<td>• addressing conventions</td>
</tr>
<tr>
<td>• datagram format</td>
</tr>
<tr>
<td>• packet handling conventions</td>
</tr>
<tr>
<td>“Signalling” protocol (ICMP)</td>
</tr>
<tr>
<td>• error reporting</td>
</tr>
<tr>
<td>• router “signaling”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link layer: Ethernet, WiFi, SONET, ATM</th>
</tr>
</thead>
</table>

| Physical layer: copper, fiber, radio, microwave |
Routing on the Internet

Routers on the Internet are store and forward routers:
- Each incoming packet is buffered
- Packet's destination is looked up in the routing table
- Packet is forwarded to the next hop towards the destination

Routing on the Internet: Example

How does a router construct its routing table?

How does a router know which is the next hop towards a destination?

Use a routing protocol to propagate (and update) reachability information
Graph Abstraction

Graph: $G = \{N, E\}$

$N =$ set of nodes = \{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)\}

c(u,v) =$ cost of link (u,v), assume full-duplex (bidirectional)

• e.g., $c(w,z) = 5$
• cost could always manually assigned, e.g., based on price
• could be hop count, or inversely related to bandwidth, reliability
• or could be dynamic, e.g., proportionally related to congestion

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

What is the least-cost path between node $u$ and $z$?

Routing algorithm: algorithm that finds least-cost path

Routing Algorithm Classification

Centralized or decentralized algorithm?

Global or distributed, local information?

Global info:
• all routers have complete topology, link cost info
• “link state” algorithms

Local info:
• routers know of only physically-connected neighbors, link costs to neighbors
• iterative process of computation, exchange of info with neighbors
• “distance vector” algorithms

Static or dynamic routing?

Static routing:
⇒ routes change slowly over time

Dynamic routing:
⇒ routes change more quickly
⇒ periodic update in response to link cost changes
Dynamic Programming

- Used when a problem can be divided into subproblems that overlap
- Solves each subproblem once and store the solution in a table
  - if the same subproblem is encountered again, simply look up its solution in the table
  - reconstruct the solution to the original problem from solutions to the subproblems
  - the more overlap the better, as this reduces the number of subproblems

DP used primarily to solve optimization problem, e.g., find the shortest, longest, “best” way of doing something

Requirement: an optimal solution to the problem must be a composition of optimal solutions to all subproblems

In other words, there must not be an optimal solution that contains suboptimal solution to a subproblem

Distance Vector Algorithm

Origin of the name “dynamic programming”:
- Bellman’s shortest path algorithm (1957)
- dynamic: multi-stage, time-varying process
- programming: planning, decision making by a tabular method

Bellman’s shortest path algorithm:
- centralized distance vector algorithm
- route table $D[\cdot]$ encodes shortest path

Define: $D[x,y] := \text{cost of least-cost path from } x \text{ to } y$
Then: $D[x,y] = \min \{ c(x,v) + D[v,y] \}$,
where $v$ is a neighbor of $x$ and min is taken over all neighbors of $x$

Define two other tables:
- $L[\cdot]$: link table
- $H[\cdot]$: next hop table
Bellman’s Algorithm: Initial Values

<table>
<thead>
<tr>
<th>$L$</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
<th>$f_5$</th>
<th>$f_6$</th>
<th>$f_7$</th>
<th>$f_8$</th>
<th>$f_9$</th>
<th>$f_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$u$</td>
<td>$v$</td>
<td>$x$</td>
<td>$w$</td>
<td>$y$</td>
<td>$z$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m$</td>
<td>$v$</td>
<td>$x$</td>
<td>$w$</td>
<td>$y$</td>
<td>$z$</td>
<td>$w$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c(n,m)$</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Initial values:

<table>
<thead>
<tr>
<th>$H$</th>
<th>$u$</th>
<th>$v$</th>
<th>$w$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_{10}$</td>
<td>$l_2$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$v$</td>
<td>$l_1$</td>
<td>$l_0$</td>
<td>$l_4$</td>
<td>$l_3$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$w$</td>
<td>$l_{10}$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_1$</td>
<td>$l_7$</td>
<td>$l_6$</td>
</tr>
<tr>
<td>$x$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_0$</td>
<td>$l_6$</td>
<td>-</td>
</tr>
<tr>
<td>$y$</td>
<td>-</td>
<td>-</td>
<td>$l_7$</td>
<td>$l_6$</td>
<td>$l_0$</td>
<td>$l_9$</td>
</tr>
<tr>
<td>$z$</td>
<td>-</td>
<td>-</td>
<td>$l_8$</td>
<td>-</td>
<td>$l_9$</td>
<td>$l_0$</td>
</tr>
</tbody>
</table>

$H_0$: loopback

Bellman’s Algorithm: Example

```java
do {
    for each node $i$ in graph do {
        for each node $k$ not $i$ in graph do {
            for each $j$ neighbor of $i$ {
                $m = c(i,j) + D[j][k]$;
                if ($m < D[i][k]$) {
                    $D[i][k] = m$;
                    $H[i][k] = index$ of $L[]$ where $n=i,m=j$;
                }
            }
        }
    }
} while there has been a change in $D[]$;
```

Initial values:

<table>
<thead>
<tr>
<th>$D$</th>
<th>$u$</th>
<th>$v$</th>
<th>$w$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$v$</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$w$</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$x$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$y$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$1$</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$z$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>5</td>
<td>$\infty$</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Distributed Distance Vector Algorithm

Ford-Fulkerson (1962): modified Bellman’s algorithm to a distributed version (a.k.a. Bellman-Ford algorithm)

Basic idea:
- Each node periodically sends its own distance estimates to neighbors
- When node i receives new distance estimates from a neighbor, it updates its own distance estimates using the Bellman-Ford equation:

\[
D[x,y] = \min \{c(x,v) + D[v,y]\}, \text{ for each node } y \in N
\]

- Under stable conditions, the estimate \(D[x,y]\) converges to the actual least cost
Distributed DVA Implementation

Each node $i$:
- knows the cost to each neighbor
- keeps entries of $i$'s table for local links
- Node $i$ maintains $D(i,*) = \{D[i, k]: k \in N\}$
- $i$'s routing table consists of the $i$-th row of tables $D$ and $H$
- sends $i$-th row of table $D$ as route update from $i$
- upon receiving a route update from another node, $i$ recomputes its routing table (row $i$ of $D$ and $H$)

Example:
- $u$'s link table: $[l_1, l_2]$
- $u$'s routing table:

<table>
<thead>
<tr>
<th>dest</th>
<th>$u$</th>
<th>$v$</th>
<th>$w$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$H$</td>
<td>$l_1$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_2$</td>
<td>$l_2$</td>
<td>$l_2$</td>
</tr>
</tbody>
</table>

- $u$'s route update/distance vector:

<table>
<thead>
<tr>
<th>dest</th>
<th>$u$</th>
<th>$v$</th>
<th>$w$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Distributed DVA

Scenario:
• at time $t_0$: all nodes “wake up” and send route updates containing their reachability information
• at time $t_1$: all nodes have heard from their neighbors and update their routing tables and send out route updates
• at time $t_2$: no further changes

Route Updates

Even statically assigned link cost can change over time, e.g., when a link goes down (breaks)

Each node:
- waits for change in local link cost or msg from neighbor
- recomputes distance estimates
- if distance to any dest has increased, notify neighbors or send periodic update

Design question: how does a router communicate changes in link cost to other routers?
- on-demand/triggered updates when cost increases (“bad news”)
- “good news” travels slowly with periodic updates, with random periods

Design principle: soft-state protocol
Propagating Link Breakage

Scenario:
- time $t_5$: link $AB$ breaks; $A$ and $B$ discover failure, update table, and send out route updates
- $t_6$: $C$, $D$, and $E$ receive $A$'s and $B$'s route updates and in turn update their own routing tables and send out their route updates
- $t_7$: $A$, $B$, and $C$ receive route updates sent at time $t_6$ update their routing tables and send out their route updates
- $t_8$: no more changes

Bouncing Effect

One big problem with distributed DV algorithm:
- Bounding Effect causes routing loop

Example scenario (all examples are contrived, to illustrate problems):
- time $t_5$: cost of link $EC$ jumps to 10 (e.g., due to load increase), shortly after, periodic update from $B$ arrives at $E$ and $C$, both end up using $B$ as next hop to each other
- $t_6$: $D$ gets an update from $E$, $A$ and $D$ both send out updates
- $t_7$: link $BC$ breaks and $A$'s update arrives at $B$
- $t_8$: $A$ and $E$ now goes through $B$ to $C$
- Events at $t_7$ and $t_8$ iterate until cost of $B$ to $C$ through $A$ = 10, at which time, $E$ switches to the $EC$ link
- meanwhile, data packets from $A$ or $B$ to $C$ loop until TTL expires
Counting to \( \infty \)

As a means to resolve routing loop

Example scenario:
- time \( t_3 \): link DE breaks, A sends out a periodic route update
- \( t_4 \): D gets an update from A and sends out update, all paths go through A
- \( t_5 \): link AB breaks and D’s update arrives at A, A updates route table: all paths go through D
- \( t_6 \): bouncing effect exists between A and D and routing loops are formed for destinations C and E

Each update brings cost up by one, detect that there’s a loop when cost reaches \( \infty \)

RIP (Routing Information Protocol)

An implementation of distance vector algorithm
- distributed with BSD-UNIX in 1982
- distance metric: # of hops (max 15 hops, \( \infty \) = 16 hops)
- distance vectors sent/advertised once every 30 secs
  - takes 8 minutes to count to \( \infty \) and detect loop if we rely only on periodic updates
  - link failure info quickly propagated with triggered updates
- distance vectors sent using UDP
- each advertisement lists up to 25 destination nets
- if no advertisement heard after 180 sec => neighbor/link declared down
  - all routes via neighbor invalidated
  - new advertisements sent to all other neighbors (triggered updates)
  - neighbors in turn send out new advertisements (if cost increased)

RIP routing tables managed by application-level process called route-d (daemon)
Routing Loop

Problems with distributed DVA:
- bouncing effect
- routing loop
- counting to $\infty$

Cause of routing loop (in 3 variations):
- inconsistent routing tables
- route updates do not reflect reality
- routers do not know when they are in their neighbor’s path to a destination

Heuristics (not solution) to alleviate problem:
- triggered updates to shorten convergence time
- split horizon
- split horizon with poisonous reverse
- path hold-down
- route poisoning

Loop-free routing:
- path vector
- path finding/source tracing
- diffusing computation
- link reversal

Split Horizon

Idea: don’t advertise reachability to next-hop neighbor

Example scenario:
- time $t_2$: link $DE$ breaks, $D$ sends out triggered update, $A$ sends out a periodic route update
- $t_3$: $D$ gets an update from $A$, all paths go through $A$. **Doesn’t send reachability to $B$, $C$, and $E$ to $A$ because it uses $A$ as next hop to get to these nodes**
- $t_4$: link $AB$ breaks and $D$’s update arrives at $A$, $A$ updates route table: only $D$ is now reachable
Why Split Horizon is not a Solution

Example scenario:
• time $t_2$: $A$ and $D$ send periodic updates to each other

$\cdots$

• $t_4$: $D$ and $A$ do not update each other's path to $B$, $C$, and $E$ because they're using each other as next hop

Both must rely on soft-state to stale entries

Split Horizon with Poisonous Reverse

Idea: advertise cost $\infty$ to next-hop neighbor

Example scenario:
• time $t_2$: $A$ and $D$ send periodic updates to each other

$\cdots$

• $t_4$: $D$ and $A$ update each other's path to $B$, $C$, and $E$ as cost $\infty$ because they're using each other as next hop
Why Both are not Solution

Example scenario (consider only destination \( D \)):
- \( t_2 \): links \( AB \) and \( DE \) break simultaneously
- \( t_3 \): assumes \( B \) receives \( E \)'s update, but \( C \) doesn't:
  - \( C \) advertises reachability to \( D \) at cost 2 to \( B \)
- \( t_4 \): \( B \) advertises reachability to \( D \) at cost 3 to \( E \)
- \( t_5 \): \( E \) advertises reachability to \( D \) at cost 4 to \( C \)
- \( t_6 \): \( C \) advertises reachability to \( D \) at cost 5 to \( B \):
  - bouncing effect exists between \( B, C, \) and \( E \)

Routing Loop

Problems with distributed DVA:
- bouncing effect
- routing loop
- counting to \( \infty \)

Cause of routing loop (in 3 variations):
- inconsistent routing tables
- route updates do not reflect reality
- routers do not know when they are in their neighbor's path to a destination

Heuristics (not solution) to alleviate problem:
- triggered updates to shorten convergence time
- split horizon
- split horizon with poisonous reverse
- path hold-down
- route poisoning

Loop-free routing:
- path vector
- path finding/source tracing
- diffusing computation
- link reversal
Path Hold-down

Idea: do not switch route for $n$ update periods after cost increase

Each node's cost to $A$,
with hold-down period $n = 2$

<table>
<thead>
<tr>
<th>time</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$1$</td>
<td>$2$</td>
<td>$3$</td>
<td>$2$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$3$</td>
<td>$2$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$3$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$10$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$11$</td>
<td>$11$</td>
<td>$11$</td>
<td>$10$</td>
</tr>
<tr>
<td>$t_6$</td>
<td>$11$</td>
<td>$12$</td>
<td>$11$</td>
<td>$10$</td>
</tr>
<tr>
<td>$t_7$</td>
<td>$11$</td>
<td>$12$</td>
<td>$11$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

Smaller hold-down period $n = 1$

<table>
<thead>
<tr>
<th>time</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$1$</td>
<td>$2$</td>
<td>$3$</td>
<td>$2$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$\infty$</td>
<td>$2$</td>
<td>$3$</td>
<td>$2$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$3$</td>
<td>$\infty$</td>
<td>$3$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$\infty$</td>
<td>$4$</td>
<td>$\infty$</td>
<td>$4$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$5$</td>
<td>$\infty$</td>
<td>$5$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Route Poisoning

Idea: advertise cost $\infty$ if cost from next hop has been increasing for $n$ updates
- does not actually change cost in routing table entry, only what is advertised

Can be used with both path hold-down and split horizon with poisonous reverse

All heuristics rely on counting to $\infty$ to detect loop, they differ only in convergence time
Path Vector

Idea:
- instead of sending only the next hop to a destination in distance vector, send the full path to each destination
- a router adopt a neighbor as the next hop to a destination only if it is not itself in neighbor’s path to the destination
- a router prepends itself to all of its paths before propagating them further

Path vector is used in BGP

A’s path vector:

<table>
<thead>
<tr>
<th>dest</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>path</td>
<td>A</td>
<td>AB</td>
<td>ABC</td>
<td>AD</td>
<td>ADE</td>
</tr>
</tbody>
</table>

Distributed DVA Deployment History

- Early days: GGP, HELLO, Fuzzball (ARPANET, early Internet)
- 1988 (standardized): RIP (routed)
  - v1: 30 secs periodic update with triggered updates and split horizon with poisonous reverse
  - v2 (1993): CIDR
  - v1: split horizon, with path hold-down (n=2)
  - v2: 90 secs periodic update with triggered updates, route poisoning
- 1993: EIGRP (cisco): Enhanced IGRP
  - uses DUAL, supports CIDR
- 1994: BGPv4 for inter-domain routing
  - uses path vector, supports CIDR, runs on TCP
Link State Routing

Observation: loop can be prevented if each node knows the actual network topology

In link-state routing, each node:
• floods the network with the state (up, down) of its links
• uses Dijkstra’s Shortest Path First (SPF) algorithm to compute a shortest-path tree

What is advertised:
• DV: all nodes reachable from me, advertised to all neighbors
• LS: all my immediate neighbors, advertised to all nodes

Dijkstra’s Shortest Path First (SPF) Algorithm

A greedy algorithm for solving single-source shortest path problem
• assume non-negative edge weights
• even if we’re only interested in the path from $s$ to a single destination, $d$, we need to find the shortest path from $s$ to all vertices in $G$ (otherwise, we might have missed a shorter path)
• if the shortest path from $s$ to $d$ passes through an intermediate node $u$, i.e., $P = \{s, \ldots, u, \ldots, d\}$, then $P' = \{s, \ldots, u\}$ must be the shortest path from $s$ to $u$
Dijkstra’s Shortest Path First (SPF) Algorithm

SPF(startnode s)
{ // Initialize
    table = createtable(|V|); // stores spf, cost, predecessor
    table[*].spf = false; table[*].cost = INFINITY;
    pq = createpq(|E|); // empty pq
    table[s].cost = 0;
    pq.insert(0, s); // pq.insert(cost, v)
    while (!pq.isempty()) {
        v = pq.getMin();
        if (!table[v].spf) { // not on sp tree
            table[v].spf = true;
            for each u = v.neighbors() {
                newcost = weight(u, v) + table[v].cost;
                if (table[u].cost > newcost) {
                    table[u].cost = newcost;
                    table[u].pred = v;
                    pq.insert(newcost, u);
                }
            }
        }
    }
    extract SPF from table;
}
Dijkstra's SPF Example (v = s = b)

<table>
<thead>
<tr>
<th>u</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>F</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>c</td>
<td>F</td>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

Dijkstra's SPF Example (v = a)

<table>
<thead>
<tr>
<th>u</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>c</td>
<td>F</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>F</td>
<td>8</td>
<td>a</td>
</tr>
<tr>
<td>e</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s SPF Example ($v = c$)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$c$</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>$d$</td>
<td>F</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>

Dijkstra’s SPF Example ($v = d$)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$c$</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>$d$</td>
<td>T</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>11</td>
<td>d</td>
</tr>
</tbody>
</table>
Dijkstra’s SPF Example \((v = e)\)

<table>
<thead>
<tr>
<th>u</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>c</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>T</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>T</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>f</td>
<td>F</td>
<td>9</td>
<td>e</td>
</tr>
</tbody>
</table>

Dijkstra’s SPF Example \((v = f)\)

<table>
<thead>
<tr>
<th>u</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>c</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>T</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>T</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>f</td>
<td>T</td>
<td>9</td>
<td>e</td>
</tr>
</tbody>
</table>
Dijkstra’s SPF Algorithm

Algorithm complexity: \( N \) nodes
- each iteration: extract minHeap \( O(\log N) \)
- total \( O(N \log N) \)

Each neighbor of each node could also potentially go thru the minHeap once: \( O(|E| \log N) \)
Total: \( O(N \log N + |E| \log N) = O(|E| \log N) \)
- \(|E| \geq |N| - 1 \) for a connected graph

Oscillations possible:
e.g., link cost = amount of carried traffic, asymmetric link cost

OSPF (Open Shortest Path First)

“Open”: publicly available
Uses Link State algorithm
- LS packet dissemination
  - advertisements disseminated to entire network
    (via flooding protocol: forward to all interfaces except the incoming one)
  - advertisement carried in OSPF messages directly over IP (rather than TCP or UDP)
- route computation using Dijkstra’s algorithm
- topology map at each node
  - OSPF is not loop free due to delay in topology propagation
  - maintaining LS database consistency is hard due to router reboot:
    - how to determine which LS is newer?
OSPF (Open Shortest Path First)

Advance features (not in RIP):
• security: all OSPF messages authenticated (to prevent fake advertisement)
• multiple same-cost paths allowed (only one path in RIP)
• for each link, multiple cost metrics for different TOS (e.g., satellite link cost set to “low” for best effort; high for real time)
• integrated uni- and multicast support:
  • Multicast OSPF (MOSPF) uses same topology data base as OSPF
  • Hierarchical OSPF in large domains