9: Intro to Routing Algorithms

Last Modified: 3/24/2003 2:08:40 PM

Routing Algorithm classification: Static or Dynamic?

Choice 1: Static or dynamic?

Static:
- routes change slowly over time
- Configured by system administrator
- Appropriate in some circumstances, but obvious drawbacks (routes added/removed? sharing load?)
- Not much more to say?

Dynamic:
- routes change more quickly
  - periodic update
  - in response to link cost changes

Routing Algorithm classification: Global or decentralized?

Choice 2, if dynamic: 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 (gossip)
  - "distance vector" algorithms

Roadmap

- Details of Link State
- Details of Distance Vector
- Comparison

Routing

- IP Routing – each router is supposed to send each IP datagram one step closer to its destination
- How do they do that?
  - Static Routing
    - Hierarchical Routing – in ideal world would that be enough? Well its not an ideal world
  - Dynamic Routing
    - Routers communicate amongst themselves to determine good routes (ICMP redirect is a simple example of this)
    - Before we cover specific routing protocols we will cover principles of dynamic routing protocols

Routing protocol:

Goal: determine "good" path (sequence of routers) thru network from source to dest.

Graph abstraction for routing algorithms:
- graph nodes are routers
- graph edges are physical links
  - link cost: delay, $ cost, or congestion level

"good" path:
- typically means minimum cost path
- other definitions possible
Global Dynamic Routing

See the big picture; Find the best Route

What algorithm do you use?

A Link-State Routing Algorithm

Dijkstra's algorithm

- Know complete network topology with link costs for each link is known to all nodes
- accomplished via "link state broadcast"
- In theory, all nodes have same info
- Based on info from all other nodes, each node individually computes least cost paths from one node (‘source”) to all other nodes
- gives routing table for that node
- iterative: after k iterations, know least cost path to k dest.’s

Link State Algorithm: Some Notation

Notation:
- \( c(i,j) \): link cost from node i to j. cost infinite if not direct neighbors
- \( D(v) \): current value of cost of path from source to dest. V
- \( n(v) \): next hop from this source to v along the least cost path
- \( N \): set of nodes whose least cost path definitively known

Dijkstra’s Algorithm

1. Initialization – know \( c(i,j) \) to start:
2. \( N = \{A\} \)
3. for all nodes \( v \)
4. if \( v \) adjacent to \( A \)
5. then \( D(v) = c(A,v) \)
6. else \( D(v) = \text{infy} \)
7. Loop
8. find \( w \) not in \( N \) such that \( D(w) \) is a minimum (optional?)
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 shortest path cost to \( w \) plus cost from \( w \) to \( v \) */
13. until all nodes in \( N \)

Dijkstra's Algorithm: example

<table>
<thead>
<tr>
<th>Step</th>
<th>start ( N )</th>
<th>( D(B),n(B) )</th>
<th>( D(C),n(C) )</th>
<th>( D(D),n(D) )</th>
<th>( D(E),n(E) )</th>
<th>( D(F),n(F) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2,B</td>
<td>5,C</td>
<td>1,A</td>
<td>infinity</td>
<td>infinity</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>2,B</td>
<td>5,C</td>
<td>1,A</td>
<td>infinity</td>
<td>infinity</td>
</tr>
<tr>
<td>2</td>
<td>ADE</td>
<td>2,B</td>
<td>3,D</td>
<td>4,D</td>
<td>2,D</td>
<td>4,D</td>
</tr>
<tr>
<td>3</td>
<td>ADEB</td>
<td>3,D</td>
<td>4,D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADEBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADEBCF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dijkstra's Algorithm gives routing table

<table>
<thead>
<tr>
<th>Destination</th>
<th>Outgoing Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>n(A) = A</td>
</tr>
<tr>
<td>B</td>
<td>n(B) = B</td>
</tr>
<tr>
<td>C</td>
<td>n(C) = D</td>
</tr>
<tr>
<td>D</td>
<td>n(D) = D</td>
</tr>
<tr>
<td>E</td>
<td>n(E) = D</td>
</tr>
<tr>
<td>F</td>
<td>n(F) = D</td>
</tr>
</tbody>
</table>
Complexity of Link State

**Algorithm complexity:** n nodes

- Each iteration
  - Find next w not in N such that D(w) is a minimum
  - Then for that w, check its best path to other destinations

\[ O(n(n+1)/2) \] comparisons: \( O(n^2) \)

- More efficient implementations possible using a heap: \( O(n\log n) \)

Oscillations

- Assume:
  - Link cost = amount of carried traffic
  - Link cost is not symmetric
  - B and D sending 1 unit of traffic; C send e units of traffic

- Initially start with slightly unbalanced routes
- Everyone goes with least loaded, making them most loaded
- For next time, so everyone switches
- Herding effect!

\[
\begin{array}{c|c|c|c}
 & A & D & C \\
\hline
B & 1 & 1 & e \\
C & 0 & 0 & 0 \\
D & 0 & 0 & e \\
\end{array}
\]

- Initially start with almost equal routes
- B and C go clockwise to A
- B, C and D go counterclockwise
- B, C, D go clockwise

Preventing Oscillations

- Avoid link costs based on experienced load
  - But want to be able to route around heavily loaded links...
- Avoid “herding” effect
  - Avoid all routers recomputing at the same time
  - Not enough to start them computing at a different time because will synchronize over time as send updates
  - Deliberately introduce randomization into time between when receive an update and when compute a new route

Distance Vector Routing Algorithm

**Distance Vector Routing Algorithm**

- Distributed: each node communicates only with directly-attached neighbors
- Iterative: continues until no nodes exchange info.
- Self-terminating: no “signal” to stop
- Asynchronous:
  - Nodes need not exchange info/iterate at lock step!

Distance Table data structure

- Each node has its own row for each possible destination
- Column for each directly-attached neighbor to node
- Example: in node X, for dest. Y via neighbor Z:

\[
D^X(Y,Z) = c(X,Z) + \min_w \{D^W(Y,w)\}
\]

- Rows for each possible dest!

Example: Distance Table for E

- Column only for each neighbor
- Cost to destination via

\[
D^E() = Z
\]

- Rows for each possible dest!
**Distance Vector Algorithm:**

At all nodes, X:

1. Initialization (don’t start knowing link costs for all links in graph):  
2. for all adjacent nodes v:  
3. \( D^v(v,v) = c(X,v) \)  
4. \( D^v(v,w) = \infty \)  
5. for all destinations, y  
6. send \( \min D^v(y,w) \) to each neighbor \( w \) over all X’s neighbors \( v \)

Then in steady state...

**Distance Vector Algorithm (cont.):**

8. loop
9. wait (until I see a link cost change to neighbor V  
10. or until I receive update from neighbor V)
11. if \( c(X,V) \) changes by \( d \)  
12. \( \forall \) change cost to all dest's via neighbor v by \( d \) \( \forall \)
13. \( \forall \) note: \( d \) could be positive or negative \( \forall \)
14. for all destinations \( y \): \( D^v(Y,V) = D^v(Y,V) + d \)
15. else if (update received from V wrt destination Y)  
16. \( \forall \) shortest path from V to some Y has changed \( \forall \)
17. \( \forall \) call this received new value is “newval” \( \forall \)
18. for the single destination \( y \): \( D^v(Y,V) = c(X,V) + newval \)
19. if we have a new \( \min_{w} D^v(Y,w) \) for any destination Y  
20. send new value of \( \min_{w} D^v(Y,w) \) to all neighbors
21. forever

**Distance Vector Algorithm: example**

To start just know directly connected links, tell neighbors

In steady state, when have good news tell neighbor
Distance Vector: link cost changes

Link cost changes:
- node detects local link cost change
- updates distance table (line 15)
- if cost change in least cost path, notify neighbors (lines 23, 24)

"good news travels fast"

Algorithm terminates

Distance Vector: poisoned reverse

If Z routes through Y to get to X:
- Originally, Z tells Y its (Z’s) distance to X is infinite (so Y won’t route to X via Z)
- In end, Y tells Z infinity
- will this completely solve count to infinity problem?

Algorithm terminates

Count to Infinity Example with Bigger Loop

B will learn bad news
C will have told B infinity because its route to A is through B, so B won’t reroute through C
E however will have told B about a good route to A through D (cost 6)
B will choose that route instead and advertise it as the new best to C (cost 6 + 8 = 14); it will be worse than the old one it advertised to C (old cost = 1)
C will propagate this updated “best” route to D (cost 15)
D will propagate this new “best” route to E (cost 17)
E will update the “best” route to B (cost 19)
Last time it advertised cost 6 to B
It will loop around adding 13 each time (cost of loop)
Will continue until cost E advertises to B is bigger than 500

Bigger Loops and Poison Reverse

E

D(A,D) = c(E,D) + min (D(A,w))
= 2 + 3 = 5

Loop back through E! Poison reverse will fix this
D tells E infinity because D’s route to A through E

D(B,A) = c(E,B) + min (D(B,A,w))
= 8 + 6 = 14

Loop back through E! Poison reverse will not fix this
B’s route to A is through E but B doesn’t know that so does not tell E infinity
B’s route is through C so no poison reverse
E will try to send through B

Comparison of LS and DV algorithms

Message complexity
- LS: nodes send info on directly connections to all other nodes
- More, smaller messages
-DV: nodes send info on best paths to all destinations to neighbors
- Fewer, larger messages

Speed of Convergence
- LS: O(n^2) algorithm
- may have oscillations
- DV: convergence time varies
- may be routing loops
- count-to-infinity problem

Robustness: what happens if router malfunctions?
- LS:
  - node can advertise incorrect link cost
  - each node computes only its own table
- DV:
  - DV node can advertise incorrect path cost
  - each node’s table used by others
  - error propagate thru network