Internet Routing Protocols Lecture 02 Intra-domain Routing

Advanced Systems Topics

Lent Term, 2008

Timothy G. Griffin
Computer Lab
Cambridge UK

Shortest Path

- Generalize distance to weighted setting
- Digraph $G = (V,E)$ with weight function w: $E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of the minimum weight

Some slides of this lecture are taken from Alon Efrat's Introduction to algorithms.

Shortest-Path Problems

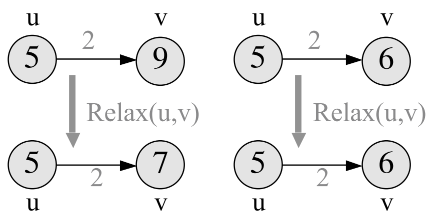
- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.

Negative Weights and Cycles?

- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small "lengths" would be possible)
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles)
  - Any shortest-path in graph G can be no longer than n – 1 edges, where n is the number of vertices

Relaxation

- For each vertex v in the graph, we maintain $v.\mathbf{d}()$, the estimate of the shortest path from s, initialized to ∞ at the start
- Relaxing an edge (u,v) means testing whether we can improve the shortest path to v found so far by going through u




```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
   v.setd(u.d()+G.w(u,v))
   v.setparent(u)
```

Dijkstra's Algorithm

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use Q, a priority queue ADT keyed by $v.\mathbf{d}()$ (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some **d** decreases)
- Basic idea
  - maintain a set S of solved vertices
  - at each step select "closest" vertex u, add it to S, and relax all edges from u

Dijkstra's Pseudo Code

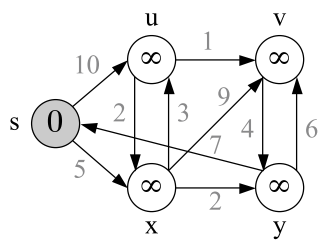
- Input: Graph *G*, start vertex *s*

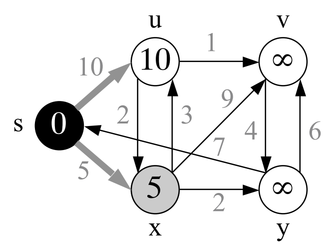
```
Dijkstra(G,s)
for each vertex u ∈ G.V()
    u.setd(∞)
    u.setparent(NIL)
s.setd(0)
S ←∅              // Set S is used to explain the algorithm
Q.init(G.V())     // Q is a priority queue ADT
while not Q.isEmpty()
    u ← Q.extractMin()
    S ←S ∪{u}
    for each v ∈ u.adjacent() do
        Relax(u, v, G)
        Q.modifyKey(v)
```

relaxing edges

Dijkstra's Example

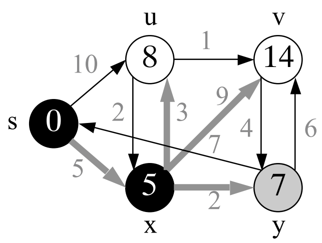
```
Dijkstra(G,s)
for each vertex u ∈ G.V()
    u.setd(∞)
    u.setparent(NIL)
s.setd(0)
S ←∅
Q.init(G.V())
while not Q.isEmpty()
    u ← Q.extractMin()
    S ←S ∪{u}
    for each v ∈ u.adjacent() do
        Relax(u, v, G)
        Q.modifyKey(v)
```

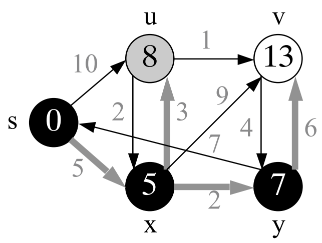




Dijkstra's Example (2)

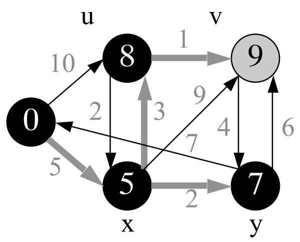
```
Dijkstra(G,s)
for each vertex u ∈ G.V()
    u.setd(∞)
    u.setparent(NIL)
s.setd(0)
S ←∅
Q.init(G.V())
while not Q.isEmpty()
    u ← Q.extractMin()
    S ←S ∪{u}
    for each v ∈ u.adjacent() do
        Relax(u, v, G)
        Q.modifyKey(v)
```

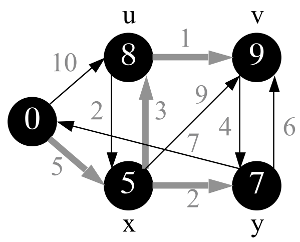




Dijkstra's Example (3)

```
Dijkstra(G,s)
for each vertex u ∈ G.V()
    u.setd(∞)
    u.setparent(NIL)
s.setd(0)
S ←∅
Q.init(G.V())
while not Q.isEmpty()
    u ← Q.extractMin()
    S ←S ∪{u}
    for each v ∈ u.adjacent() do
        Relax(u, v, G)
        Q.modifyKey(v)
```





Dijkstra's Running Time

- Extract-Min executed $|V|$ time
- Decrease-Key executed $|E|$ time
- Time = $|V|\ T_{\text{Extract-Min}} + |E|\ T_{\text{Decrease-Key}}$
- T depends on different Q implementations

Q	T(Extract-Min)	T(Decrease-Key)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$ (amort.)	$O(V \lg V + E)$

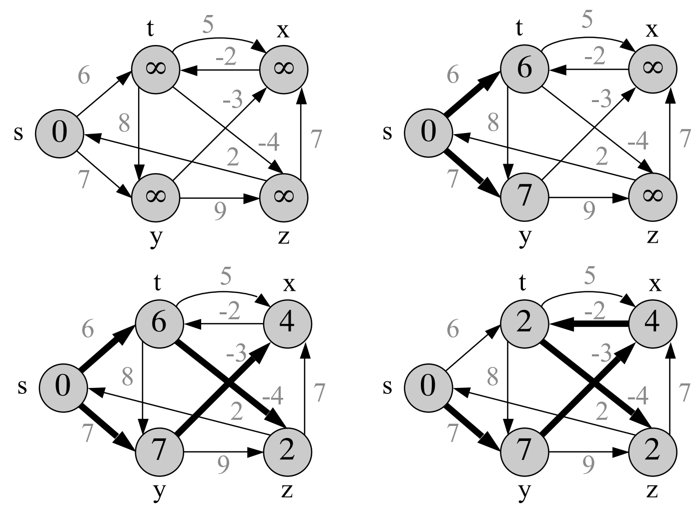
Bellman-Ford Algorithm

- Dijkstra's doesn't work when there are negative edges:
  - Intuition – we can not be greedy any more on the assumption that the lengths of paths will only increase in the future
- Bellman-Ford algorithm detects negative cycles (returns *false*) or returns the shortest path-tree

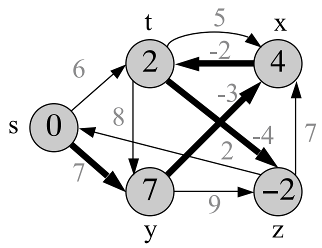
Bellman-Ford Algorithm

```
Bellman-Ford(G,s)
for each vertex u ∈ G.V()
   u.setd(∞)
   u.setparent(NIL)
s.setd(0)
for i ← 1 to |G.V()|-1 do
    for each edge (u,v) ∈ G.E() do
       Relax (u,v,G)
for each edge (u,v) ∈ G.E() do
    if v.d() > u.d() + G.w(u,v) then
       return false
return true
```

Bellman-Ford Example



Bellman-Ford Example



- Bellman-Ford running time:
  - $(|V|-1)|E| + |E| = \Theta(VE)$

RIP

- RIP = Routing Information Protocol
- Does not scale well, designed for small LANs
- Is a “distance vector protocol”
- Very simple, easy to configure, easy to implement
- Is most widely used routing protocol

Read the code! **http://www.quagga.net/**

RIP History

- Developed at Xerox PARC in early 1980s
- Reimplemented in Berkeley UNIX
- 1988 : Standardized in RFC 1058
- 1994 : RIP-2, RFC 1723
  - Support CIDR addressing
  - Authentication
- 1997 : RIPng for IPv6, RFC 2080

RIP Routing Table

Destination	Next Hop	Metric
Net A	Router 1	3
Net B	Direct	0
Net C, Host 3	Router 2	5
Default	Router 1	0

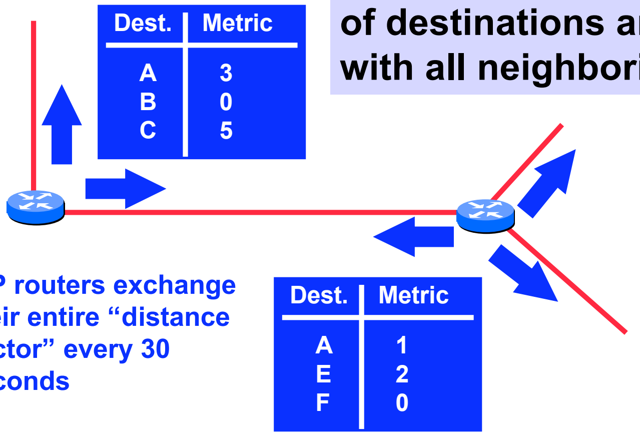
A destination is either a network, a host, or a "gateway of last resort"

The next hop is either a directly connected network or a directly connected router

Measures how many "hops away" is the destination

Basic RIP Protocol

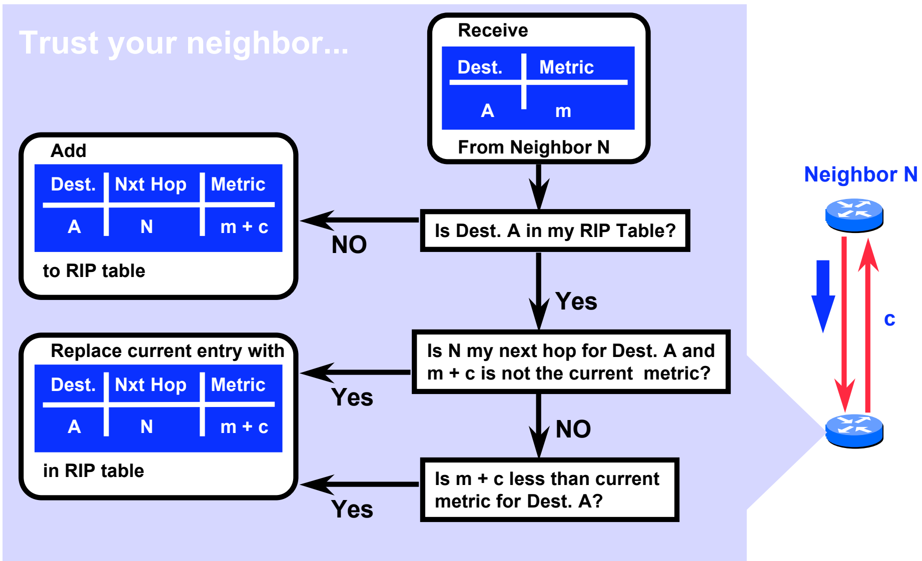
Periodically exchange list of destinations and metrics with all neighboring routers



RIP routers exchange their entire “distance vector” every 30 seconds

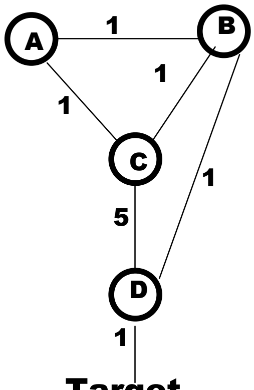
19

Basic RIP Protocol (cont.)



20

Counting to Infinity (and beyond!)



B-D Fails

A	B	C	D
3,B	2,D	3, B	1,_
3,D	inf	3,B	1,_
4,C	4,C	4,A	1,_
5,C	5,C	5,A	1,_
6,C	6,C	6,A	1,_
7,C	7,C	6,D	1,_
7,C	7,C	6,D	1,_

From RFC 1058

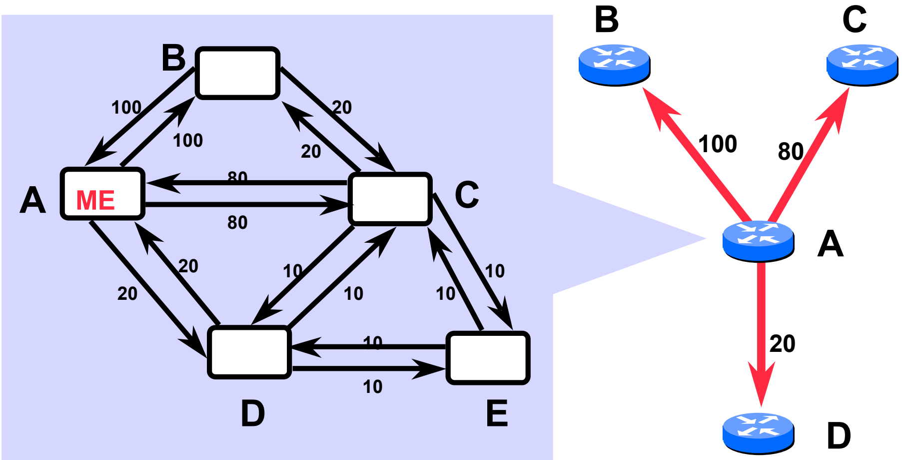
21

OSPF

- OSPF =Open Shortest Path First
- Developed to address shortcomings of RIP
  - has rapid, loop-free convergence
  - does not count to infinity
- Link metrics between 0 and 65,535, no limit on path metric
- Is a "link state protocol"
- Has reputation for being complex
- Scales well
- Defined in RFCs 1247 (1991), 1583 (1994), 2178 (1997), 2328 (1998).

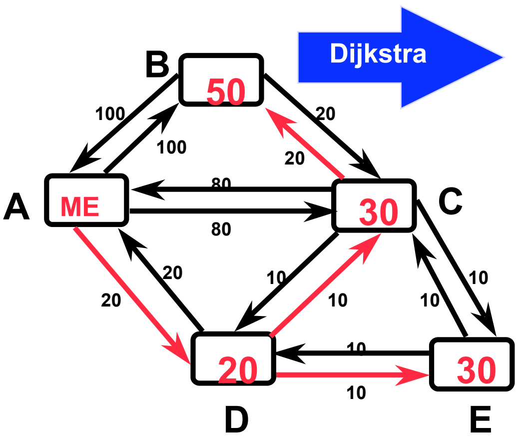
22

Link State Database



Each Router has a database representing the entire network
that is constructed from the local knowledge at each router

Building OSPF Routing Table



Dest.	Nxt Hop	Metric
B	D	50
C	D	30
D	D	20
E	D	30

Compute locally using Link State Database!

That's Easy!

Not so fast!

RIP RFC 1058 : 33 pages

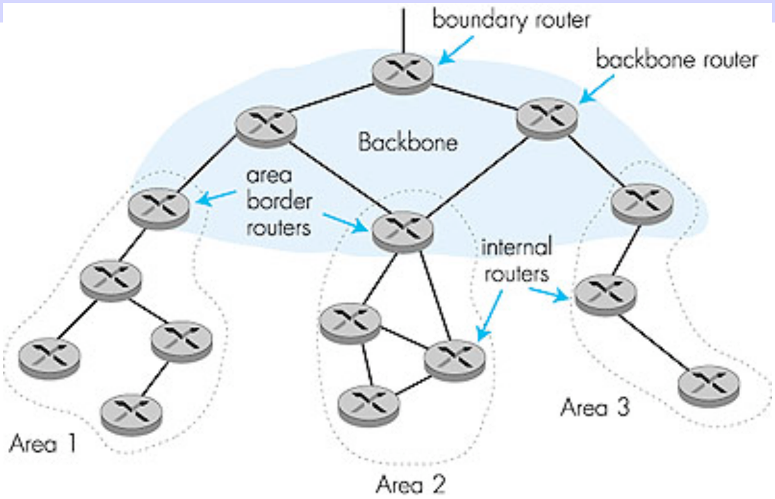
OSPF RFC 2328 : 244 pages

Much of this complexity is related to the synchronization of the distributed, replicated link state database.
Plus network modeling ….

Distance Vector vs. Link State….

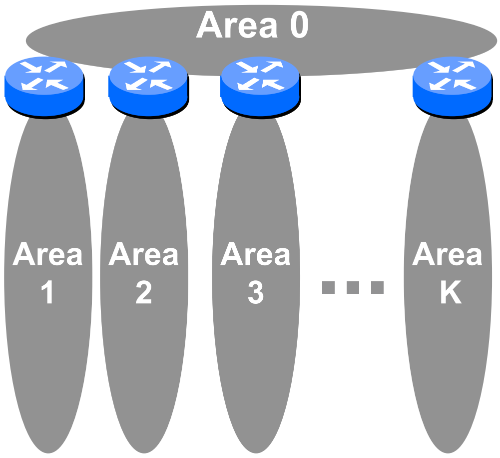
25

Hierarchical OSPF



Scalability: OSPF Areas

LS database unique within an area



- Decentralize administration
- Reduce memory usage per router
- Reduce bandwidth used by flooding

Special OSPF protocol to exchange routes between areas. This is a “distance vector” protocol!

Link-state vs. vectoring

- Link state has faster convergence, but requires more memory, CPU, and message overhead
- Vectoring requires few resources, but convergence can be very slow. Counting to infinity can be a problem.
- Both protocols can induce transient forwarding loops during convergence
  - This is one of the issues addressed by Cisco’s EIGRP.