

Link State Routing

Brad Karp

UCL Computer Science



CS 3035/GZ01

1st December 2011

Reminders

- GZ01 CW3 due Friday (tomorrow), 9:05 AM, MPEB 5th floor CS reception desk
- GZ01 CW4 available online Friday (tomorrow)

Outline

- Link State Approach to Routing
- Finding Links: Hello Protocol
- Building a Map: Flooding Protocol
- Healing after Partitions: Bringing up Adjacencies
- Finding Routes: Dijkstra's Shortest-Path-First Algorithm
- Properties of Link State Routing

Link State Approach to Routing

- Finding shortest paths in graph is classic theory problem
- Classic centralized single-source shortest paths algorithm: **Dijkstra's Algorithm**
 - requires map of entire network
- Link State Routing:
 - push a copy of whole network map to every router
 - each router learns link state database
 - each router runs Dijkstra's algorithm locally

Finding Links: Hello Protocol

- Each router configured to know its interfaces
- On each interface, every period P transmit a **hello packet** containing
 - sender's ID
 - list of neighbors from which sender has heard hello during period D
 - $D > P$ (e.g., $D = 3P$)
- Link becomes **up** if have received hello containing own ID on it in last period D
- Link becomes **down** if no such hello received in last period D
- Screens out unidirectional links

Building a Map: Flooding Protocol

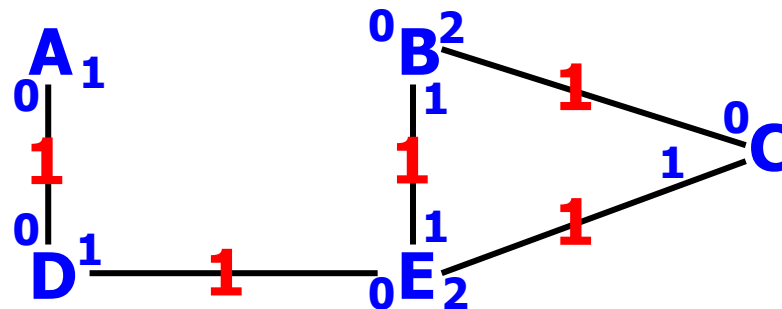
- Whenever node becomes up or becomes down, **flood** announcement to whole network
 - two link endpoint addresses
 - metric for link (configured by administrator)
 - sequence number
- Sequence number stored in **link state database**; incremented on every changed announcement
 - prevents old link states from overwriting new ones
- Upon receiving new link state message on interface *i*:
 - if link not in database, **add it, flood elsewhere**
 - if link in database, and seqno in message higher than one in database, **write into database, flood elsewhere**
 - if link in database and seqno in message lower than one in database, **send link state from database on interface *i***

Outline

- Link State Approach to Routing
- Finding Links: Hello Protocol
- Building a Map: Flooding Protocol
- **Healing after Partitions: Bringing up Adjacencies**
- Finding Routes: Dijkstra's Shortest-Path-First Algorithm
- Properties of Link State Routing

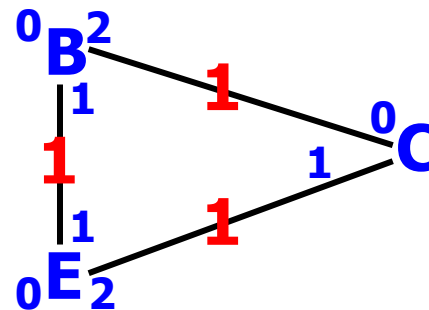
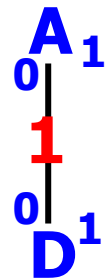
Healing Network Partitions

- Recall example from Distance Vector routing where network **partitions**
- Consider flooding behavior when **partitions heal**



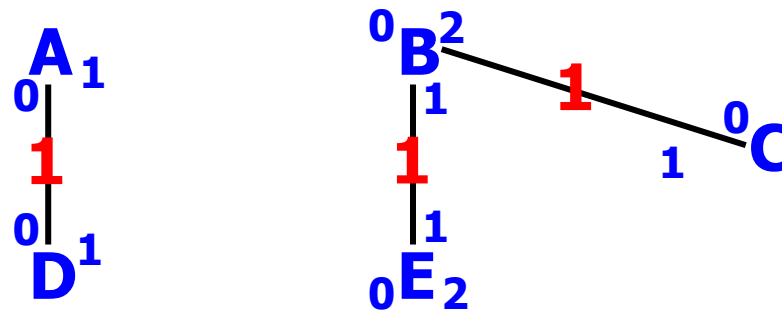
Healing Network Partitions

- Recall example from Distance Vector routing where network **partitions**
- Consider flooding behavior when **partitions heal**



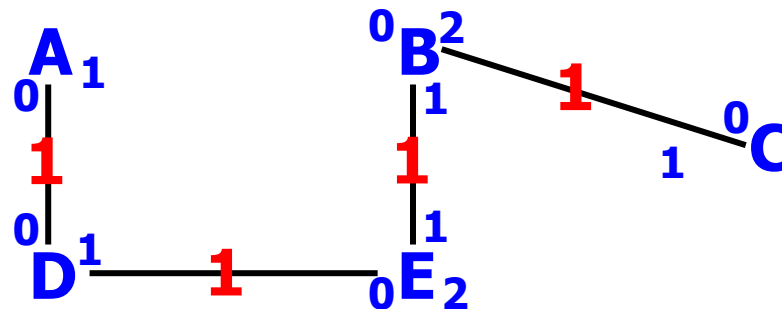
Healing Network Partitions

- Recall example from Distance Vector routing where network **partitions**
- Consider flooding behavior when **partitions heal**



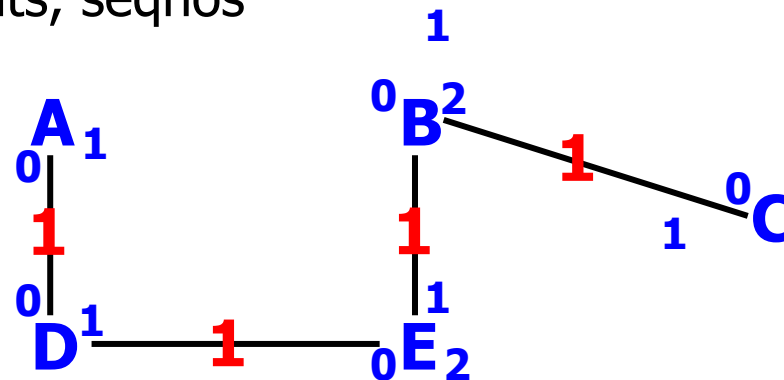
Healing Network Partitions

- Recall example from Distance Vector routing where network **partitions**
- Consider flooding behavior when **partitions heal**



Healing Network Partitions (II)

- D detects link (D, E), floods link state to A
- A and D may still think link (C, E) exists!
- If first time link (D, E) comes up, how will A learn about links (B, E), (B, C)?
- Flooding to report changes only in neighboring links not always sufficient!
- Bringing up adjacencies:
 - when link comes up, routers at ends exchange short summaries (link endpoints, sequence numbers) of their whole databases
 - routers then request missing or newer entries from one another
 - saves bandwidth; real LS database entries contain more than link endpoints, seqnos



Outline

- Link State Approach to Routing
- Finding Links: Hello Protocol
- Building a Map: Flooding Protocol
- Healing after Partitions: Bringing up Adjacencies
- Finding Routes: Dijkstra's Shortest-Path-First Algorithm
- Properties of Link State Routing

Link State Database → Routing Table

- After flooding each router holds **map of entire network graph in memory**
- Need to transform network map into **routing table**
- How: **single-source shortest paths algorithm**
- Router views itself as **source s** , all other routers as **destinations**

Shortest Paths: Definitions

- Each router is a **vertex**, $v \in V$
- Each link is an **edge**, $e \in E$, also written (u, v)
- Each link metric an edge **weight**, $w(u, v)$
- Series of edges is a **path**, whose **cost** is **sum of edges' weights**
- **Single-source shortest paths**: seek path with least cost from s to all other vertices
- **Data structures**:
 - $\pi[v]$ is **predecessor of v** : $\pi[v]$ is vertex **before** v along shortest path from s to v
 - $d[v]$ is **shortest path estimate**: least cost found from s to v so far

Shortest Paths: Definitions

- Each router is a **vertex**, $v \in V$
- Each link is an **edge**, $e \in E$, also written (u, v)
- Each link metric an edge **weight**, $w(u, v)$
- Series of edges is a **path**, whose **cost** is **sum of edges' weights**

Assume all edge weights nonnegative

(Doesn't make sense for a link to have negative cost...)

- $\pi[v]$ is **predecessor of v**: $\pi[v]$ is vertex **before** v along shortest path from s to v
- $d[v]$ is **shortest path estimate**: least cost found from s to v so far

Shortest Paths: Initialization

- When we start, we know little:
 - no estimate of cost of any path from s to any other vertex
 - no predecessor of v along shortest path from s to any v

initialize-single-source(V, s)

for each vertex $v \in V$ do

$d[v] \leftarrow \text{infinity}$

$\pi[v] \leftarrow \text{NULL}$

$d[s] = 0$

Shortest Paths Building Block: Relaxation

- Relaxation:
 - Suppose we have current estimates $d[u]$, $d[v]$ of shortest path cost from s to u and v
 - Does it reduce cost of shortest path from s to v to reach v via (u, v) ?

$\text{relax}(u, v, w)$

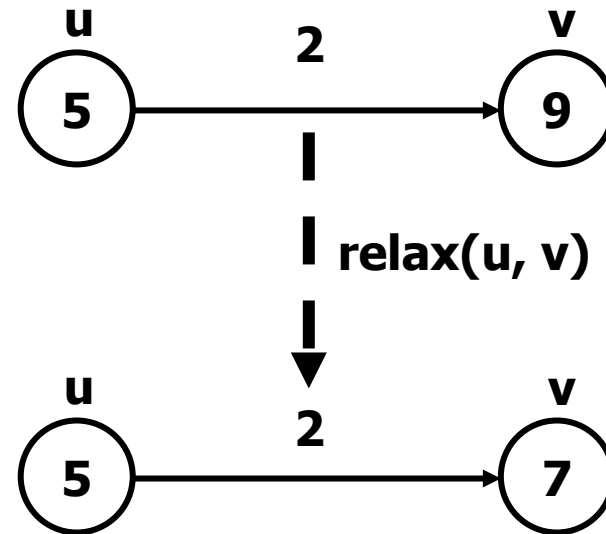
if $d[v] > d[u] + w(u, v)$ then

$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

Relaxation: Example

- Suppose
 - $d[u] = 5$
 - $d[v] = 9$
 - $w(u, v) = 2$
- $\text{relax}(u, v, w)$ computes:
 - $d[v] \text{ ?} > d[u] + w(u, v)$
 - $9 \text{ ?} > 5 + 2$
 - Yes, so reaching v via (u, v) reduces path cost
 - $d[v] = d[u] + w(u, v)$
 - $\pi[v] = u$



Dijkstra's Algorithm: Overall Strategy

- Maintain running estimates of costs of shortest paths to all vertices (initially all infinity)
- Keep a set S of vertices that are “finished”; shortest paths to these vertices already found (initially empty)
- Repeatedly pick the **unfinished** vertex v with **least shortest path cost estimate**
- Add v to set S
- Relax all edges leaving v

Dijkstra's Algorithm: Overall Strategy

- Maintain running estimates of costs of shortest paths to all vertices (initially all infinity)
- Keep a set S of vertices that are

N.B. only correct for graphs where edge weights nonnegative!

with **least shortest path cost estimate**

- Add v to set S
- Relax all edges leaving v

Dijkstra's Algorithm: Pseudocode

Dijkstra(V, E, w, s)

initialize-single-source(V, s)

$S \leftarrow \emptyset$

$Q \leftarrow V$

while $Q \neq \emptyset$ do

$u \leftarrow \text{extract-min}(Q)$

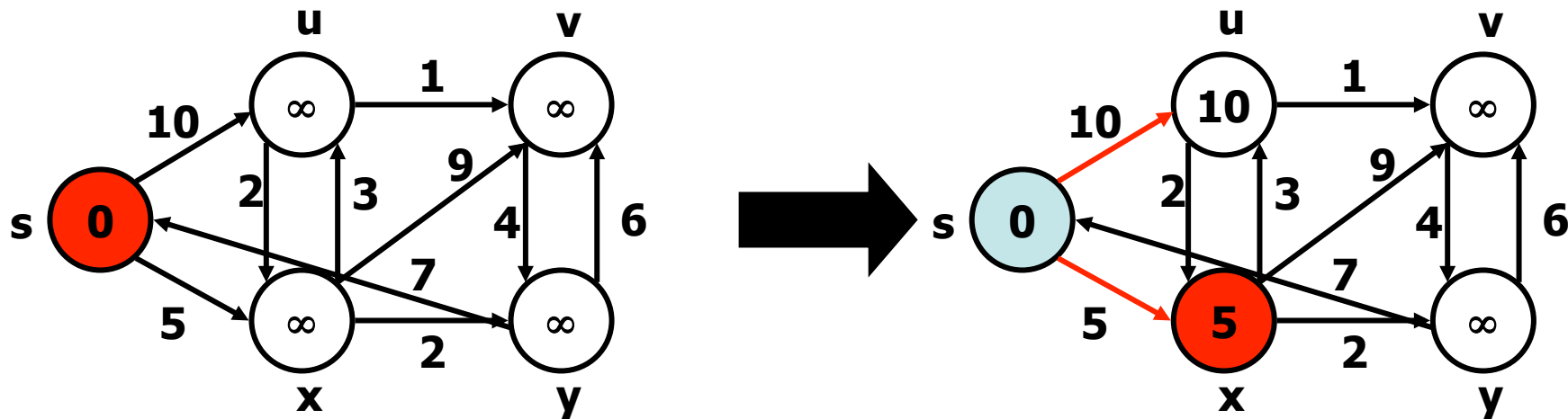
$S \leftarrow S \cup \{u\}$

 for each vertex v that neighbors u do

 relax(u, v, w)

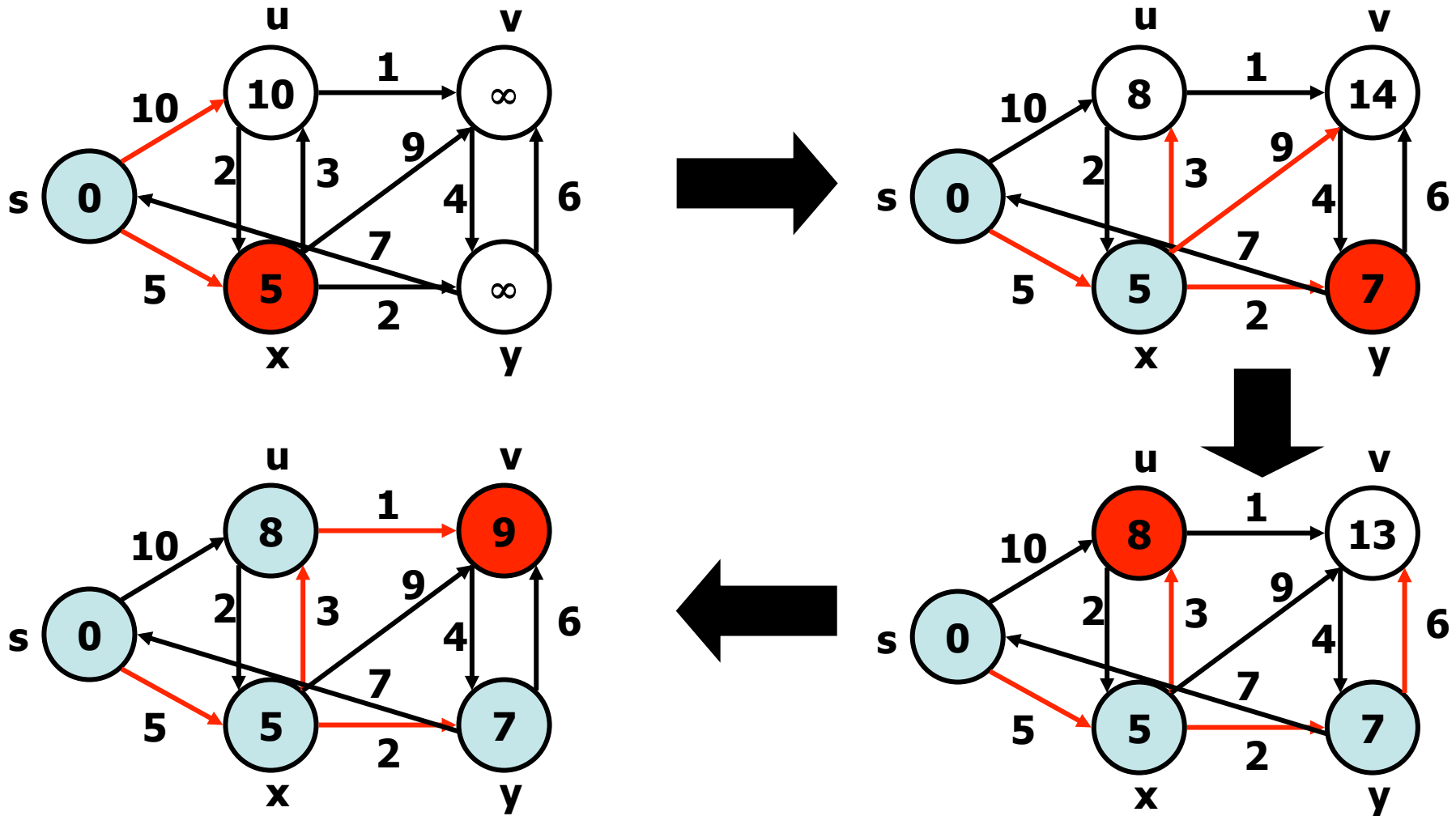
extract-min(Q): return
vertex v in Q with
minimal shortest-path
estimate $d[v]$

Dijkstra's Algorithm: Example

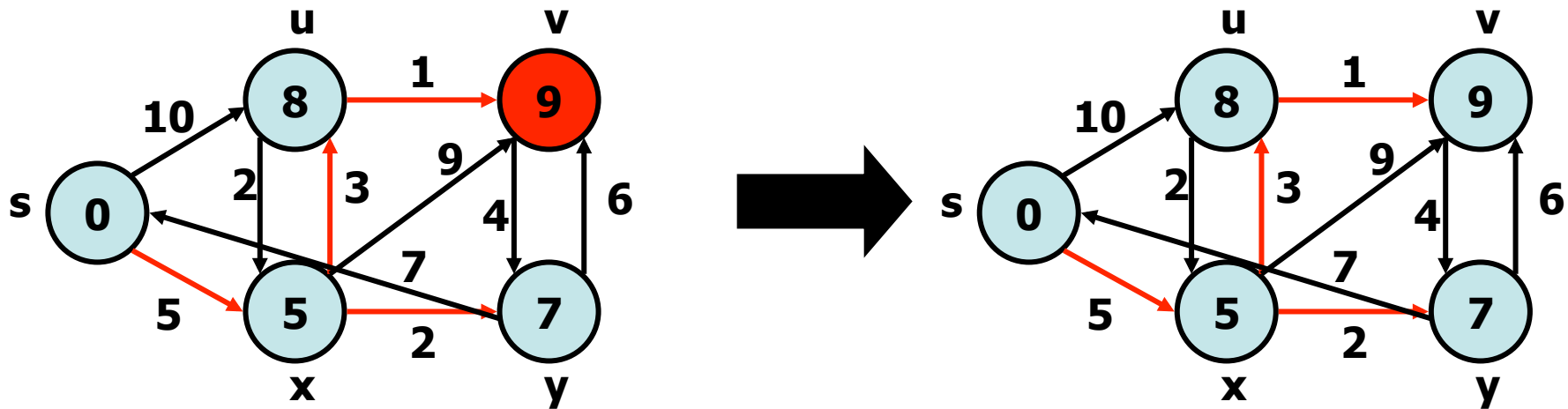


- s : source
- $d[i]$: number inside of vertex i
- $\pi[b]$: if (a, b) red, then $\pi[b] = a$
- members of set S : blue-shaded vertices
- members of priority queue Q : non-shaded vertices

Dijkstra's Algorithm Example (cont'd)



Dijkstra's Algorithm Example (cont'd)



- At termination, know shortest-path routes from s to all other routers
- **Shortest-path tree**, rooted at s

Dijkstra's Algorithm: Efficiency

- Most networks are sparse graphs
 - far fewer edges than $O(N^2)$
- Implement Q with **binary heap**
 - for N items in heap, cost of `extract-min()` is $O(\log_2 N)$
- Begin with $|V|$ entries in Q , call `extract-min()` once for each
 - Cost: $O(V \log_2 V)$
- Total cost to insert $|V|$ entries into Q : $O(V)$
- Each call to `relax()` reduces $d[]$ value for vertex in Q
 - Cost: $O(\log_2 V)$
- At most $|E|$ calls to `relax()`
- Total cost: **$O((V + E) \log_2 V)$** , or **$O(E \log_2 V)$** when all vertices reachable from source

Outline

- Link State Approach to Routing
- Finding Links: Hello Protocol
- Building a Map: Flooding Protocol
- Healing after Partitions: Bringing up Adjacencies
- Finding Routes: Dijkstra's Shortest-Path-First Algorithm
- **Properties of Link State Routing**

Link State Routing: Properties

- At first glance, flooding status of all links seems costly
 - It is! Doesn't scale to thousands of nodes without other tricks, namely hierarchy (more when we discuss BGP)
 - Cost reasonable for networks of hundreds of routers
- In practice, for intra-domain routing, LS has won, and DV no longer used
 - LS: after flooding, no loops in routes, provided all nodes have consistent link state databases
 - LS: flooding offers fast convergence after topology changes
- LS more complex to implement than DV
 - Sequence numbers crucial to protect against stale announcements
 - Bringing up adjacencies
 - Maintains both link state database and routing table