CSE 461: Link State Routing

Link State Routing

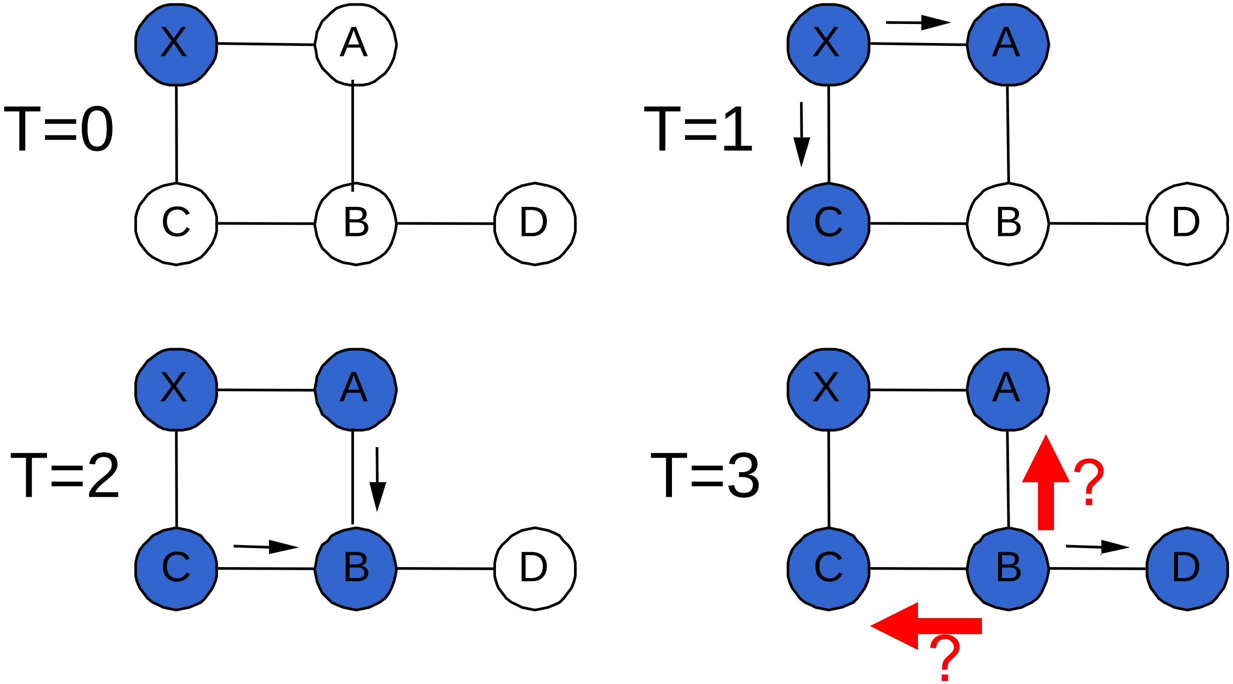
- Same assumptions/goals, but different idea than DV:
  - Make sure all routers have a view of the global topology
  - Have them all independently compute the best routes
    - Note our good old "same input + same algorithm → consistent output" trick
  - Two phases:
    1. Topology dissemination (flooding)
       - New News travels fast.
       - Old News should eventually be forgotten
    2. Shortest-path calculation (Dijkstra's algorithm)
       - N log(n)

Flooding

- Each router monitors state of its directly connected *links*
- Periodically, send this information to your neighbors
  - Generate a *link state packet*
  - Contains router ID, link list, sequence number, time-to-live
- Store and forward LSPs received – if (ID, seqno) is more recent
  - Remember this packet for routing calculations
  - Forward LSP to all ports other than incoming ports
  - This produces a *flood*; each LSP will travel over the same link at most once in each direction
- Flooding is fast, and can be made reliable with acknowledgments

Example

LSP generated by X at T=0



Will B transmit this LSP to C or A? Why or why not?

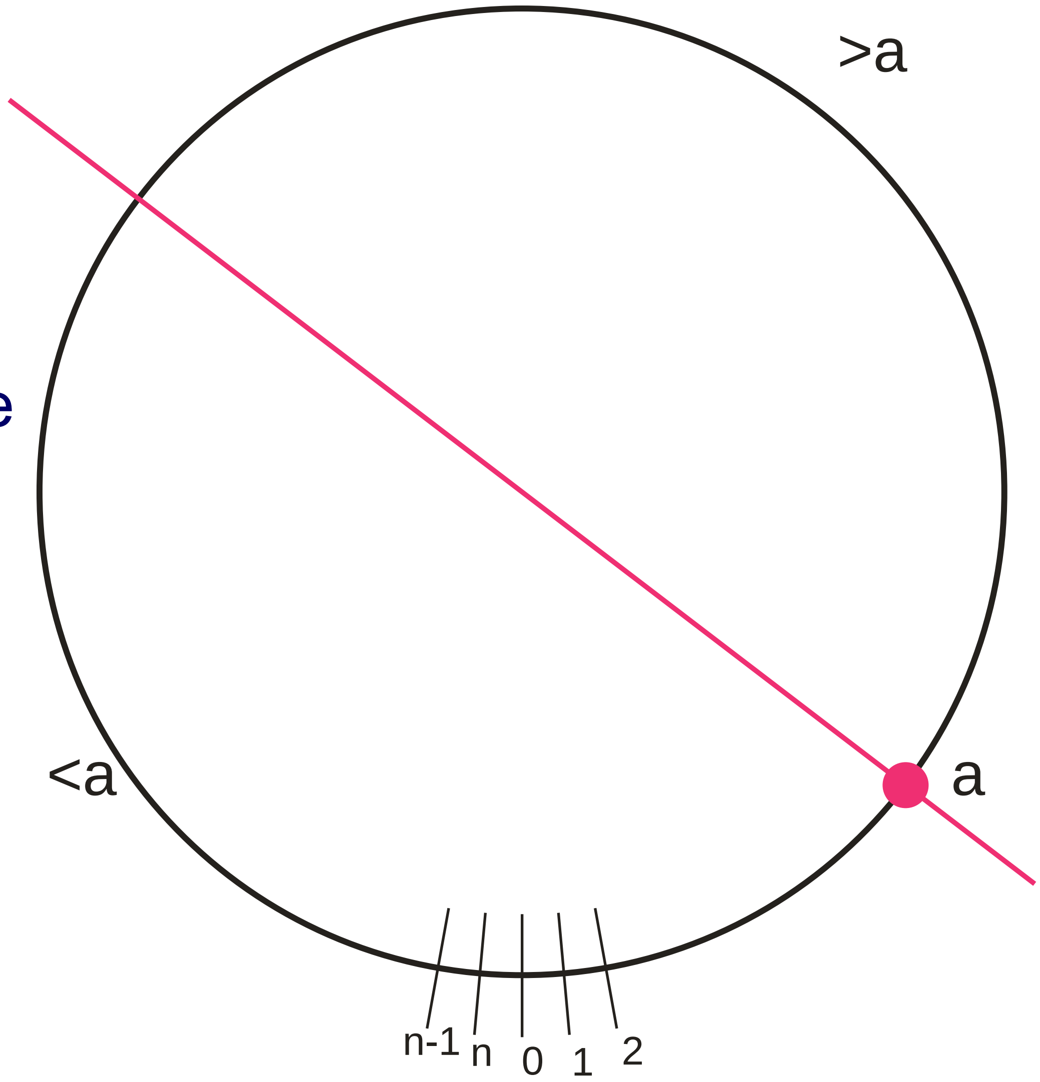
Flooding Sequence Numbers

How do we keep the sequence number space
From being exhausted?

- Use nonces instead of sequence numbers? (i.e., accept any LSP with a nonce not equal to the one stored)
  - Why is this a bad idea?
- Just make the space really big (e.g., 128-bit)?
  - What happens if we accidentally emit an *n-1* seqno?
- Allow the sequence number space to wrap around?

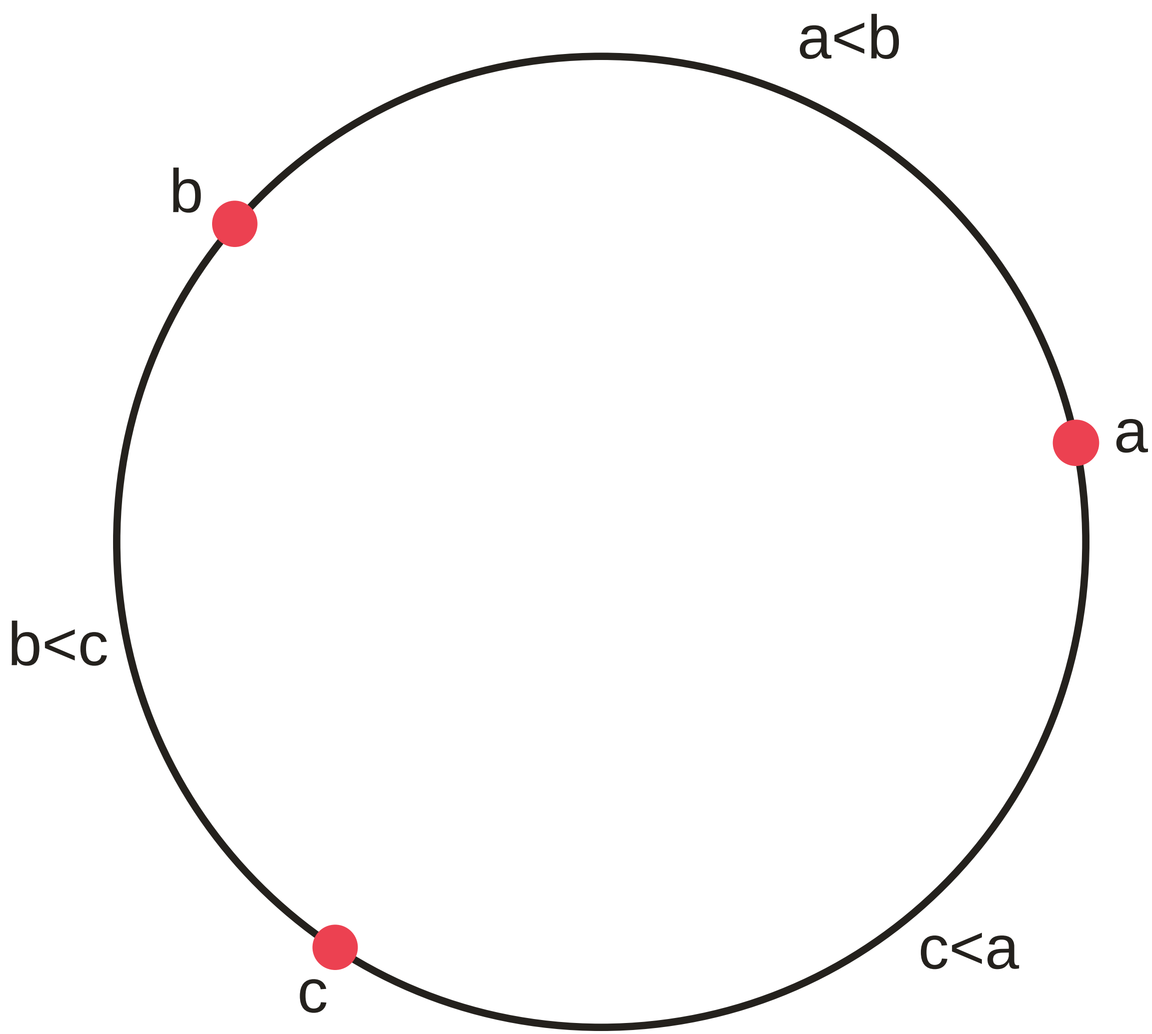
Sequence Number Wraparound

Does this solve sequence number exhaustion?



ARPANet failed in 1981, because…

A dying router emitted 3 LSPs with 3 very unlucky sequence numbers. Soon, the entire network was doing nothing but propagating these same three LSPs everywhere.



Other Complications

- When link/router fails need to remove old data. How?
  - LSPs carry sequence numbers to determine new data
  - Send a new LSP with cost infinity to signal a link down
- What happens if the network is partitioned and heals?
  - Different LS databases must be synchronized
  - Inconsistent data across routers → loops

Shortest Paths: Dijkstra's Algorithm

- *N*: Set of all nodes
- *M*: Set of nodes for which we think we have a shortest path
- *s*: The node executing the algorithm
- *L(i,j):* cost of edge *(i,j)* (inf if no edge connects)
- *C(i):* Cost of the path from *s* to *i*.
- Two phases:
  - Initialize C(n) according to received link states
  - Compute shortest path to all nodes from s
    - Link costs are symmetric

The Algorithm

```
// Initialization
M = {s}   // M is the set of all nodes considered so far.
For each n in N - {s}
      C(n) = L(s,n)

// Find Shortest paths
Forever {
     Unconsidered = N-M
     If Unconsidered == {} break
     M = M + {w} such that C(w) is the smallest in Unconsidered
     For each n in Unconsidered
            C(n) = MIN(C(n), C(w) + L(w,n))
}
```

Open Shortest Path First (OSPF)

- Most widely-used Link State implementation today
- Basic link state algorithms plus many features:
  - Authentication of routing messages
  - Extra hierarchy: partition into routing areas
    - Only bordering routers send link state information to another area
    - Reduces chatter.
    - Border router "summarizes" network costs within an area by making it appear as though it is directly connected to all interior routers
  - Load balancing

Distance Vector Message Complexity

N: number of nodes in the system

M: number of links

D: diameter of network (longest shortest path)

Da: Average degree of a node (# of outgoing links)

- Size of each update:
- Number of updates sent in one iteration:
- Number of iterations for convergence:
- Total message cost:
- Number of messages:
- Incremental cost per iteration:

Link State Message Complexity

N: number of nodes in the system

M: number of links

D: diameter of network (longest shortest path)

Da: Average degree of a node (# of outgoing links)

- Size of each update:
- Number of updates sent in one iteration:
- Number of iterations for convergence:
- Total message cost:
- Number of messages:
- Incremental cost per iteration:

Distance Vector vs. Link State

- When would you choose one over the other?
  - Be warned when reading about this on the Internet: people rate implementations, not fundamentals
- Bandwidth consumed
- Memory used
- Computation required
- Robustness
- Functionality
  - Global view of network vs. local?
  - Troubleshooting?
- Speed of convergence

Why have two protocols?

- DV: "Tell your neighbors about the world."
  - Easy to get confused
  - Simple but limited, costly and slow
    - Number of hops might be limited
    - Periodic broadcasts of large tables
    - Slow convergence due to ripples and hold down
- LS: "Tell the world about your neighbors."
  - Harder to get confused
  - More expensive sometimes
    - As many hops as you want
    - Faster convergence (instantaneous update of link state changes)
    - Able to impose global policies in a globally consistent way
      - load balancing

Cost Metrics

- How should we choose cost?
  - To get high bandwidth, low delay or low loss?
  - Do they depend on the load?

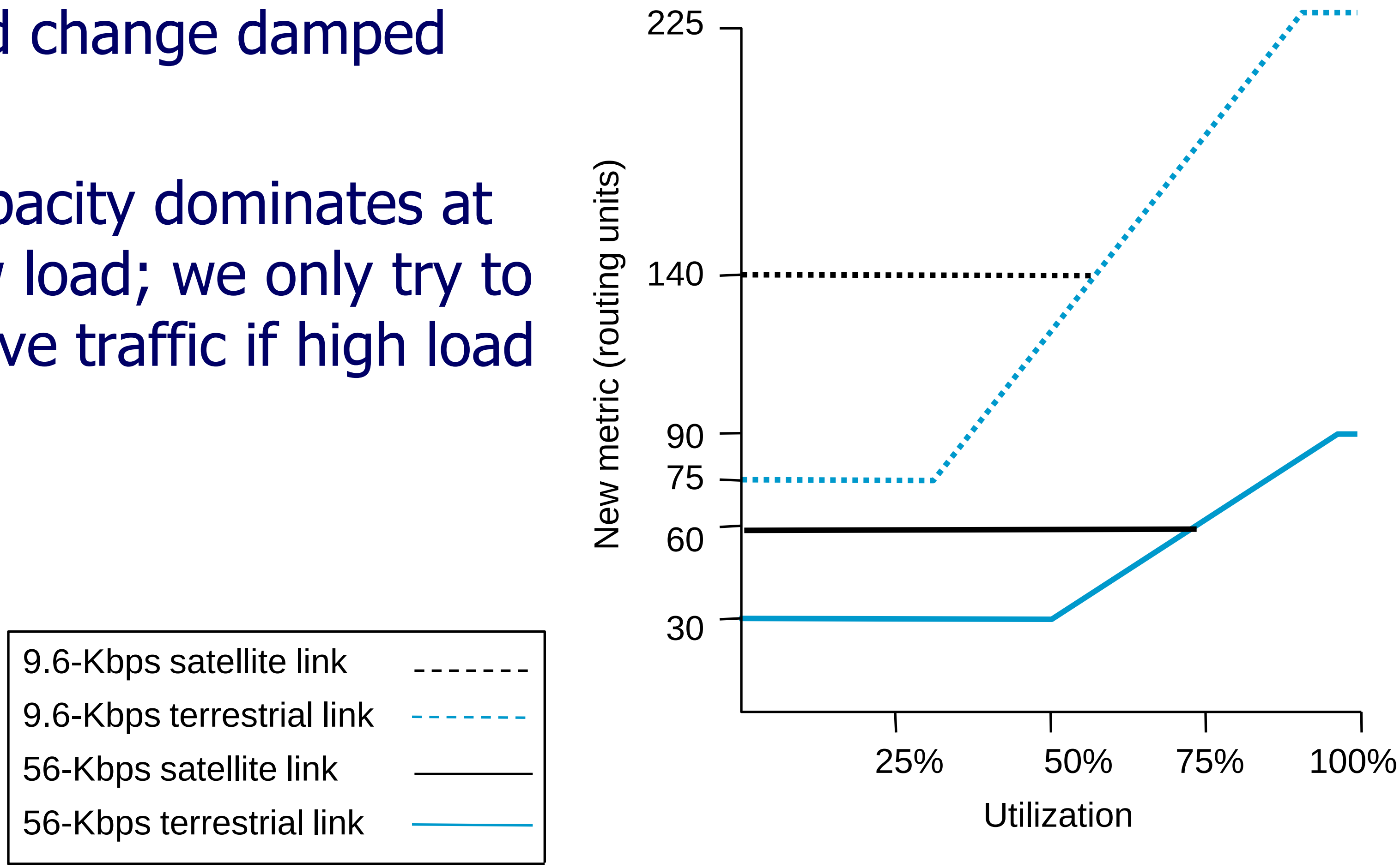
- Static Metrics
  - Hopcount is easy but treats OC3 (155 Mbps) and T1 (1.5 Mbps)
  - Can tweak result with manually assigned costs

- Dynamic Metrics
  - Depend on load; try to avoid hotspots (congestion)
  - But can lead to oscillations (damping needed)

Revised ARPANET Cost Metric

- Based on load and link
- Variation limited (3:1) and change damped

- Capacity dominates at low load; we only try to move traffic if high load

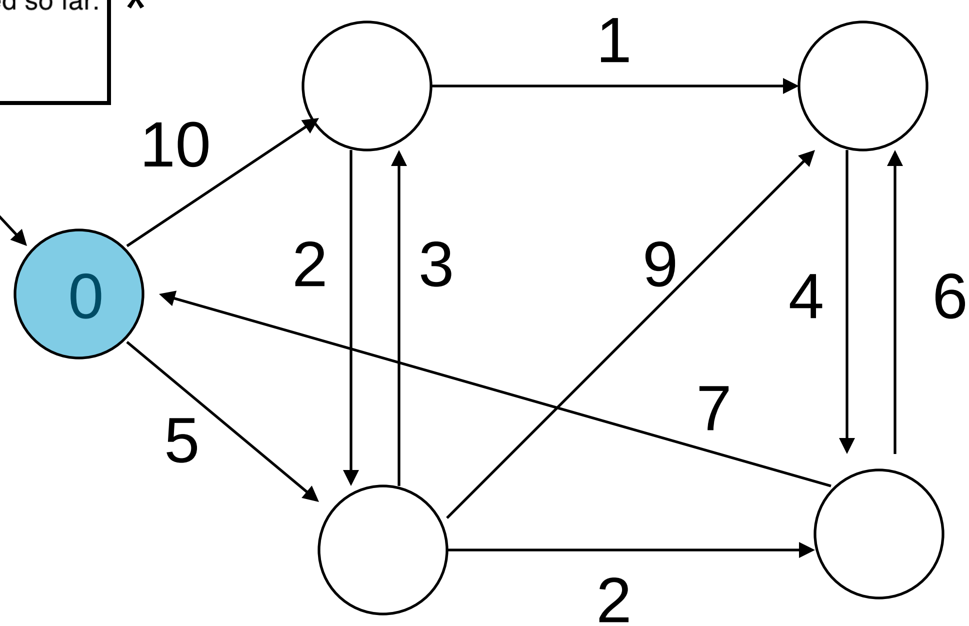


Key Concepts

- Routing uses global knowledge; forwarding is local
- Many different algorithms address the routing problem
  - We have looked at two classes: DV (RIP) and LS (OSPF)
- Challenges:
  - Handling failures/changes
  - Defining "best" paths
  - Scaling to millions of users

Dijkstra Example – After the flood

```
*  // Initialization
   M = {s}   // M is the set of all nodes considered so far.  *
   For each n in N - {s}
       C(n) = L(s,n)
```

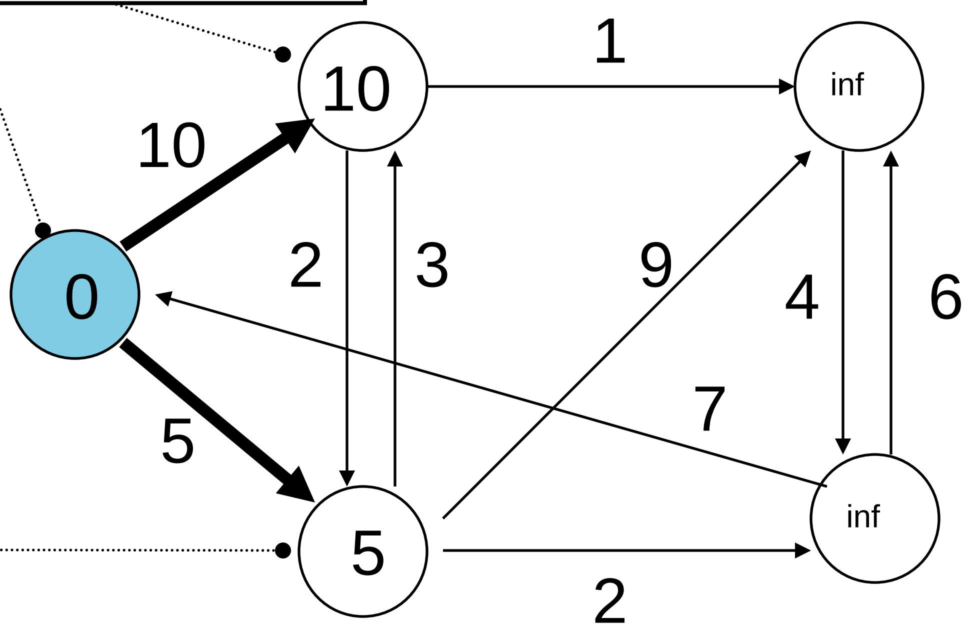


The Considered

The Unconsidered.

Dijkstra Example – Post Initialization

```
// Initialization
M = {s}   // M is the set of all nodes considered so far.
For each n in N - {s}
      C(n) = L(s,n)
```

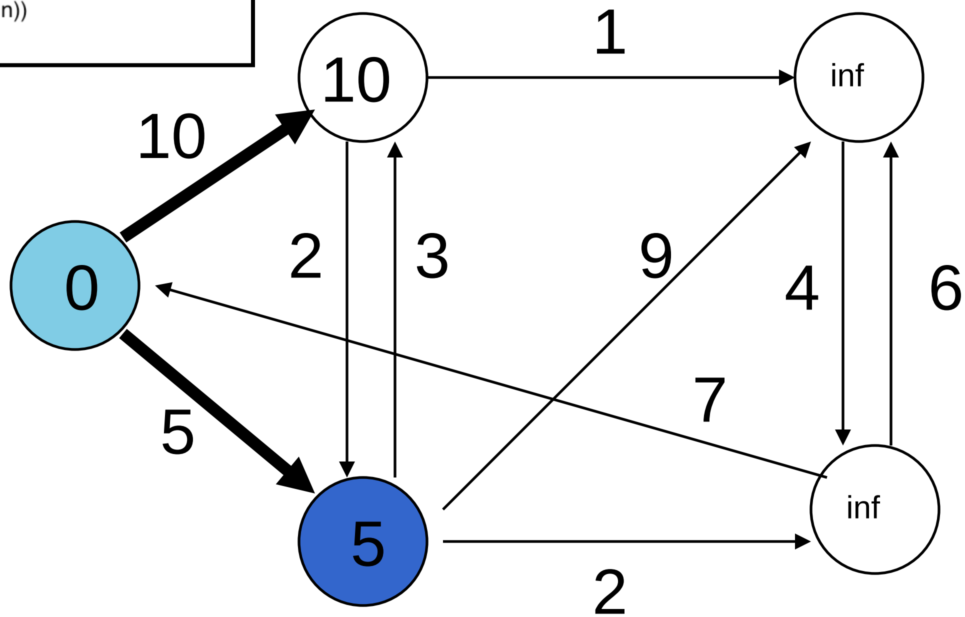


The Considered

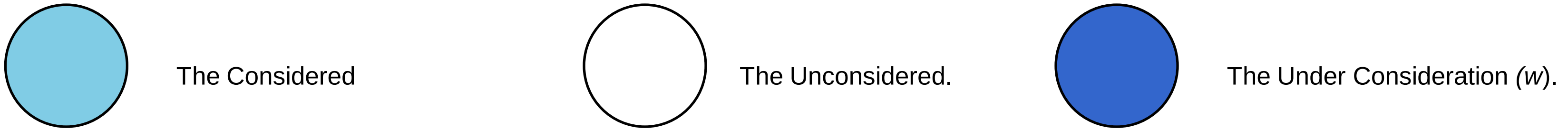
The Unconsidered.

Considering a Node

```
// Find Shortest paths
Forever {
    Unconsidered = N-M
    If Unconsidered == {} break
    M = M + {w} such that C(w) is the smallest in Unconsidered
    For each n in Unconsidered
        C(n) = MIN(C(n), C(w) + L(w,n))
}
```

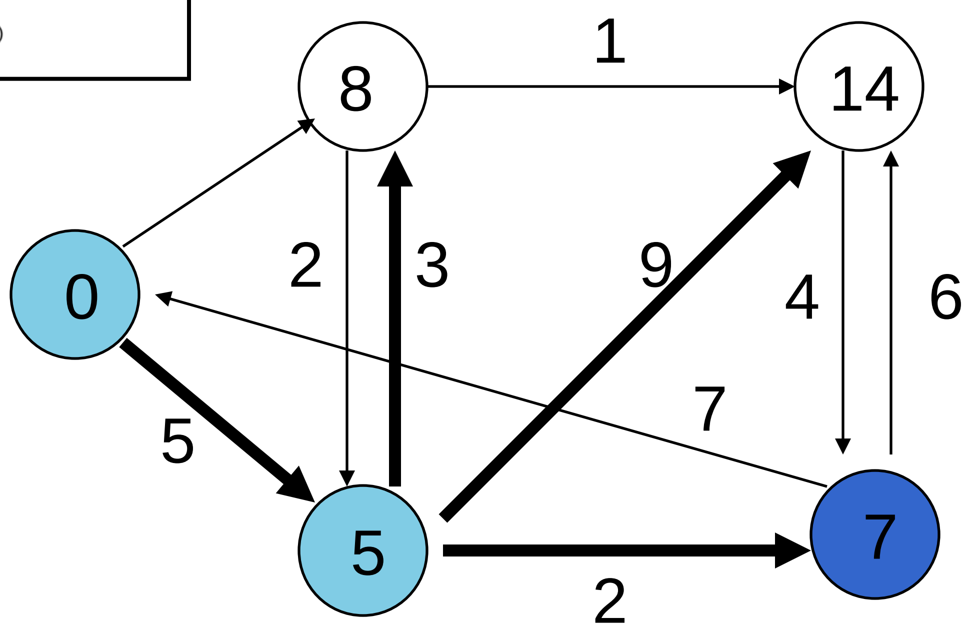


Cost updates of 8,14, and 7

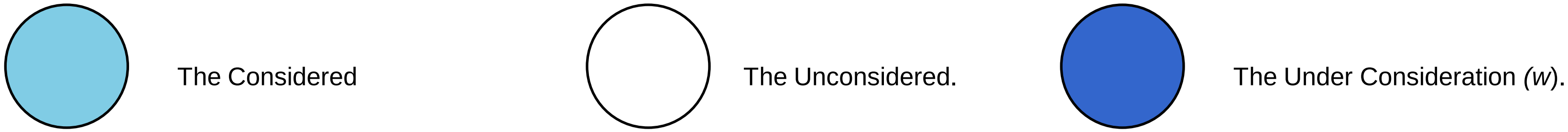


Pushing out the horizon

```
// Find Shortest paths
Forever {
    Unconsidered = N-M
    If Unconsidered == {} break
    M = M + {w} such that C(w) is the smallest in Unconsidered
    For each n in Unconsidered
        C(n) = MIN(C(n), C(w) + L(w,n))
}
```

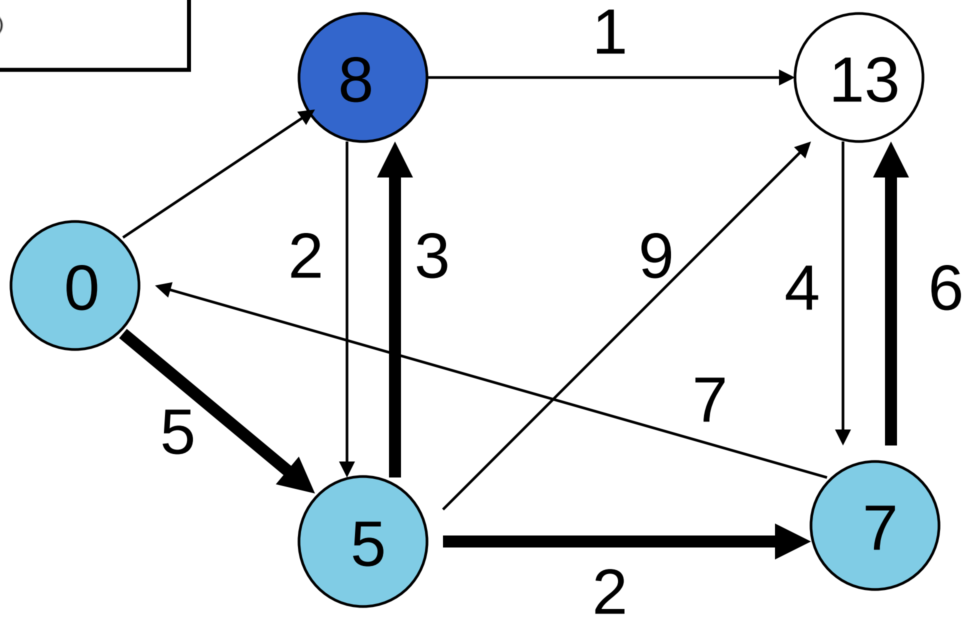


Cost updates of 13

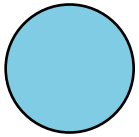


Next Phase

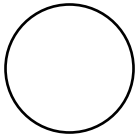
```
// Find Shortest paths
Forever {
    Unconsidered = N-M
    If Unconsidered == {} break
    M = M + {w} such that C(w) is the smallest in Unconsidered
    For each n in Unconsidered
        C(n) = MIN(C(n), C(w) + L(w,n))
}
```



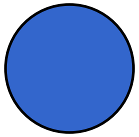
Cost updates of 9



The Considered



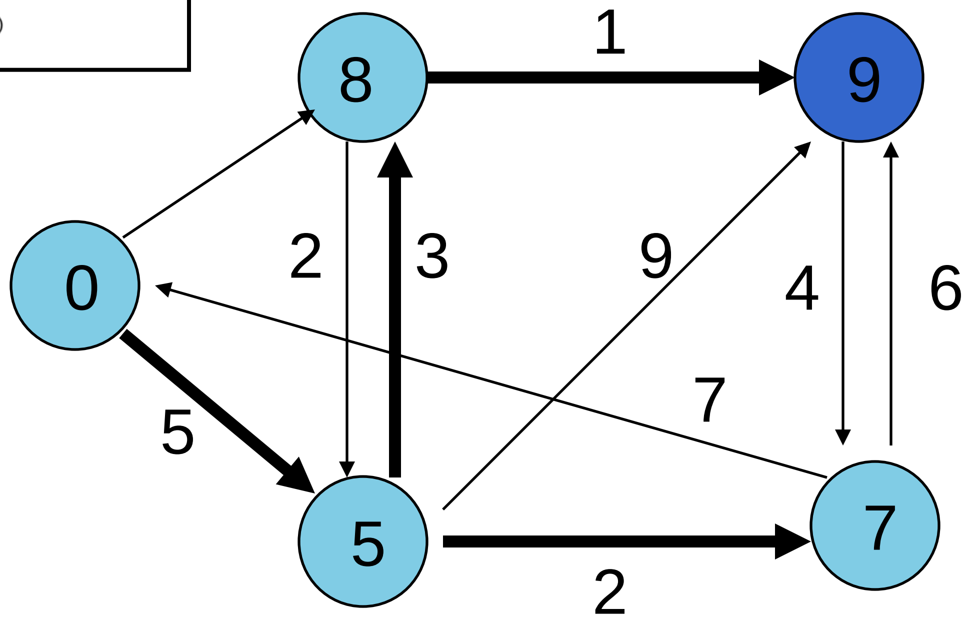
The Unconsidered.



The Under Consideration (*w*).

Considering the last node

```
// Find Shortest paths
Forever {
    Unconsidered = N-M
    If Unconsidered == {} break
    M = M + {w} such that C(w) is the smallest in Unconsidered
    For each n in Unconsidered
          C(n) = MIN(C(n), C(w) + L(w,n))
}
```



The Considered

The Unconsidered.

The Under Consideration *(w)*.

Dijkstra Example – Done