# More Routing

EE122 Fall 2012

Scott Shenker

http://inst.eecs.berkeley.edu/~ee122/

Materials with thanks to Jennifer Rexford, Ion Stoica, Vern Paxson and other colleagues at Princeton and UC Berkeley

# Let’s focus on clarifying questions

- I love the degree of interaction in this year’s class
- But there are many people who are confused
- I’d like to give them the chance to ask about basics
- So today, let’s give priority to questions of the form
  - “I don’t understand X” or “how does that work?”
- Ask speculative questions during or after break

# Warning….

- This lecture contains detailed calculations

- Prolonged exposure may induce drowsiness

- To keep you awake I will be tossing beanbags
  - **Do not misplace them**
  - **Do not read the sheet of paper attached**
  - If you've already participated, hand to nbr who hasn't

# Logic Refresher

- A ***if*** B means B → A
  - if B is true, then A is true
- A ***only if*** B means A → B
  - if A is true, then B is true
- A ***if and only if*** B means: A ↔ B
  1. If A is true, then B is true
  2. If B is true, then A is true
- To make the statement that A if and only if B, you must prove statements 1 and 2.

# Short Summary of Course

- Architecture, layering, E2E principle, blah, blah,…
  - How functionality is organized
- There are only two important design challenges:
  - **Reliable Transport** and **Routing**
- Reliable Transport:

*A transport mechanism is "reliable" if and only if it resends all dropped or corrupted packets*

- Routing:

*Global routing state is valid if and only if there are no dead ends (easy) and there are no loops (hard)*

# 10 Years from Now….

- If you remember nothing else from this course except this single slide, I'll be very happy
- *If you don't remember this slide, you have wasted your time…*

# Previous Routing Lecture

- We assume destination-based forwarding
- The key challenge is to compute loop-free routes
- This is easy when the topology is a tree
  - Loops are impossible without reversing a packet
  - Flooding always will find the destination
  - Can use "learning" to reduce need for flooding
- But this approach has serious disadvantages
  - Can't use entire network, must restrict to tree
  - Does not react well to failures or host movement
  - Universally hated by operators….

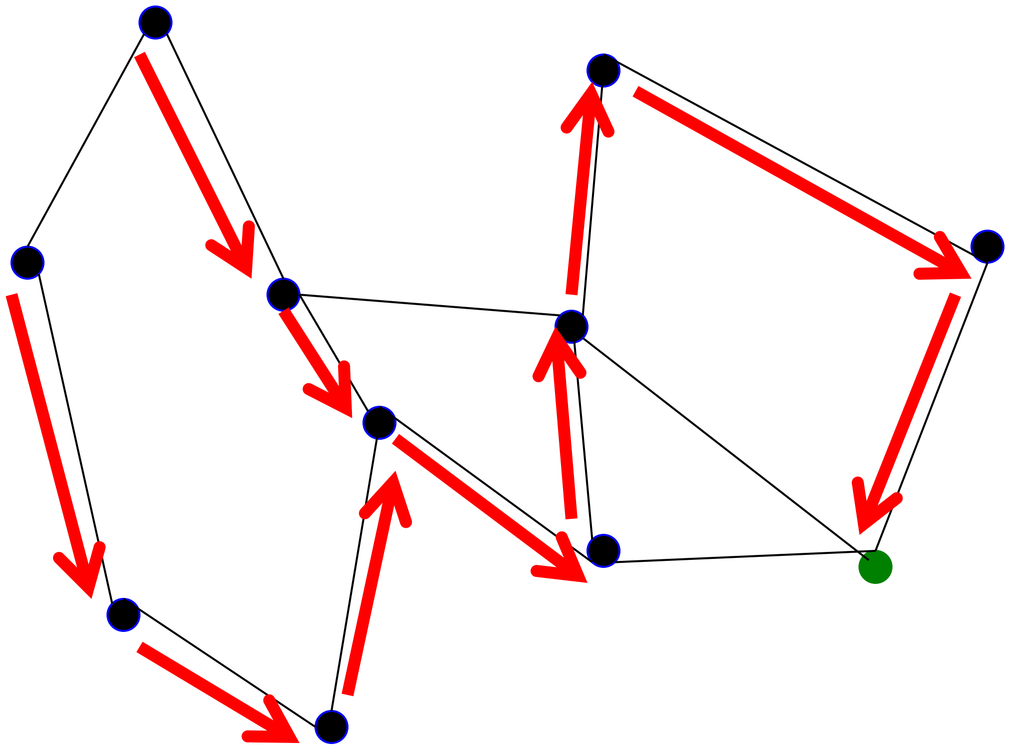
# Other Ways to Avoid Loops?

- If I gave you a network graph, could you define loop-free paths to a given destination?

- Simple algorithm:
  - For given source, pick an arbitrary path that doesn’t loop
  - For any node not on path, draw a path that does not contradict earlier path
  - Continue until all nodes are covered

- Can pick *any* spanning tree rooted at destination

# Example



# Loops are easy to avoid…

- ..if you have the whole graph
- Centralized or pseudo-centralized computation
  - Requirement: ***routes computed knowing global view***
  - One node can do calculation for everyone
  - Or each node can do calculation for themselves
- But question is: how do you construct global view?

# Link-State

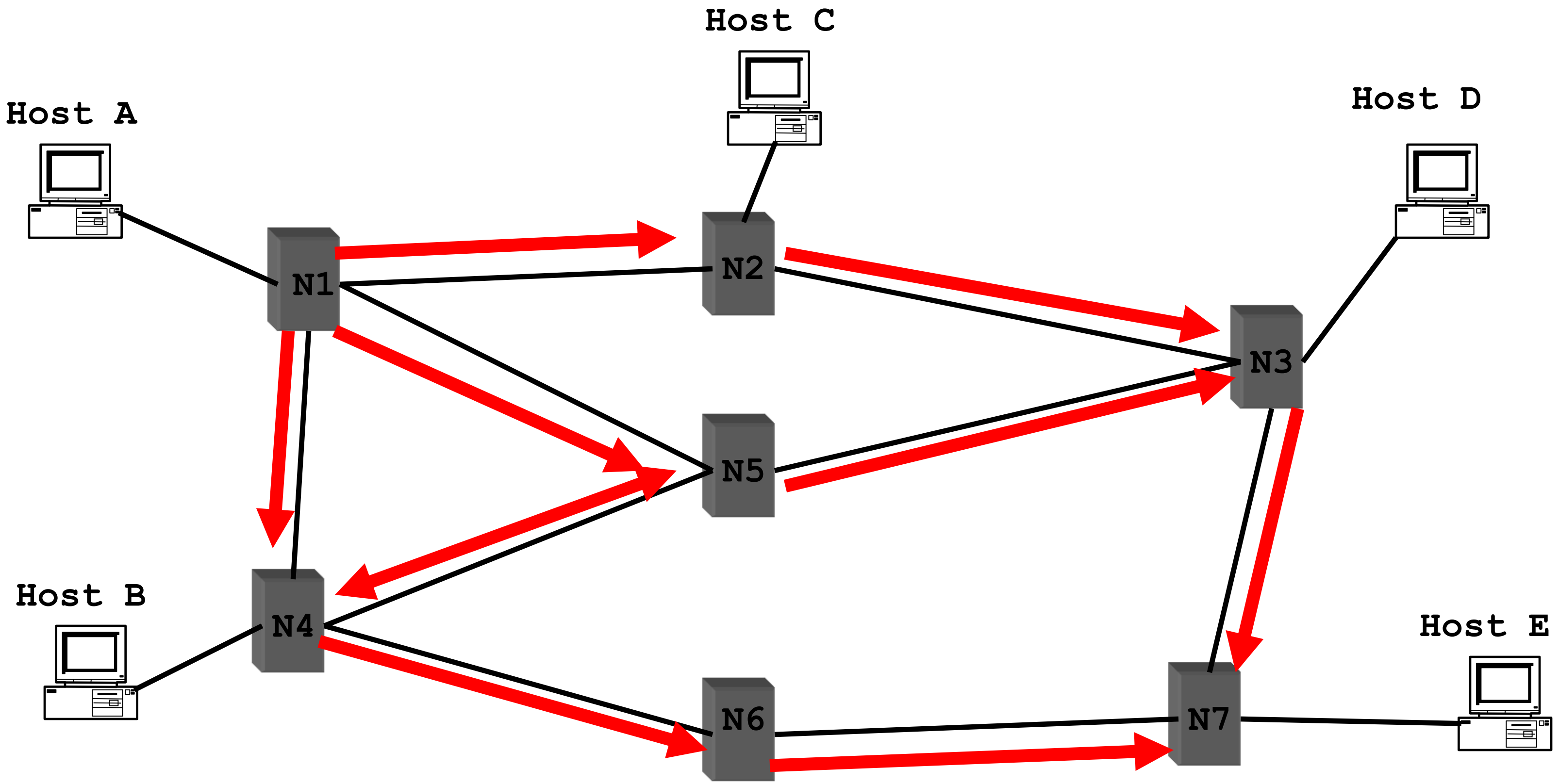
Details in Section

# Link-State Routing Is Conceptually Simple

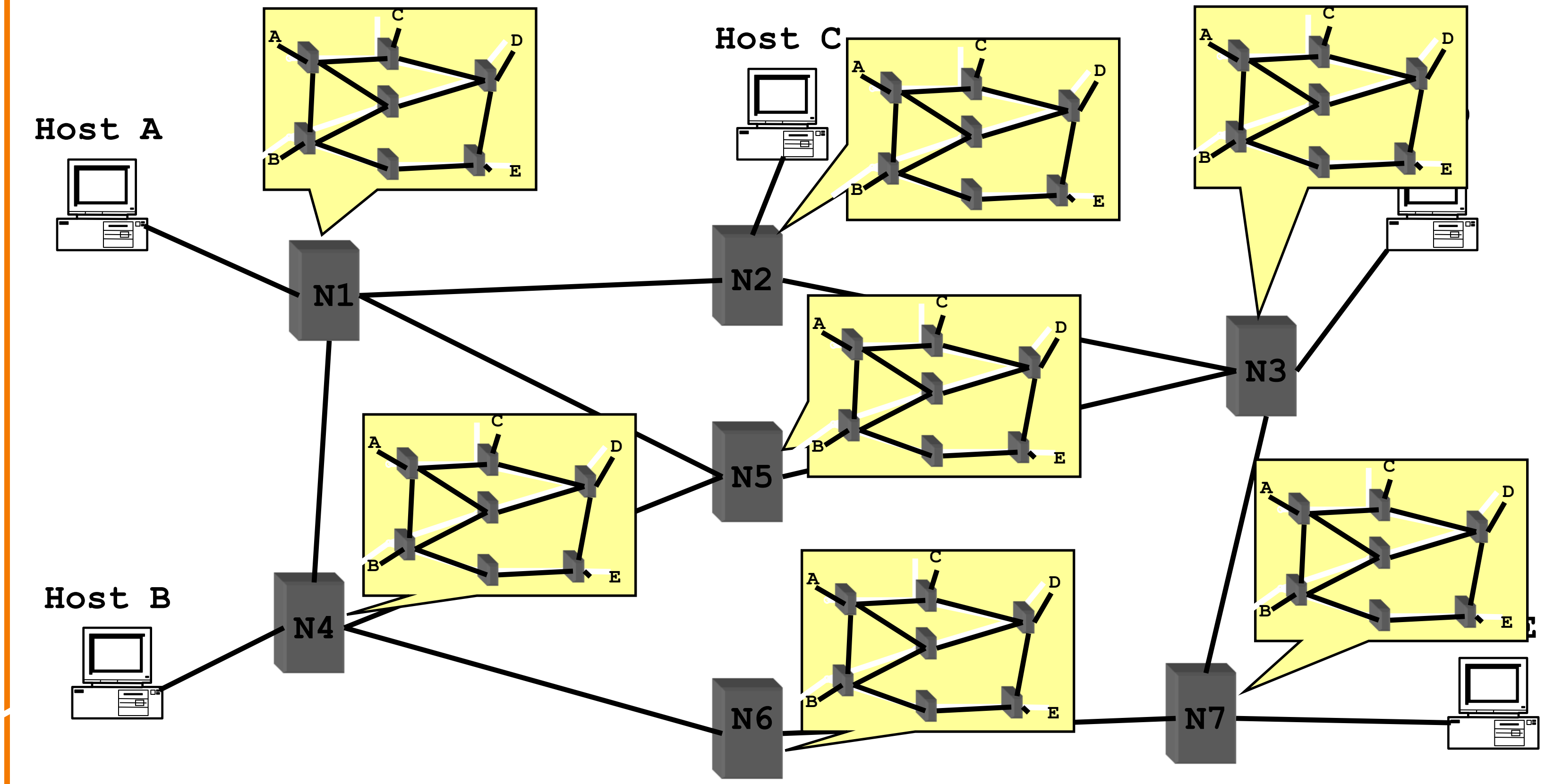
- Each router keeps track of its incident links
- Each router broadcasts the link state
  - To give every router a complete view of the graph
- Each router computes paths using same algorithm
- Example protocols
  - Open Shortest Path First (OSPF)
  - Intermediate System – Intermediate System (IS-IS)
- Challenges: scaling, transient disruptions

# Link State Routing

- Each node floods its local information
- Each node then knows entire network topology



# Link State: Each Node Has Global View



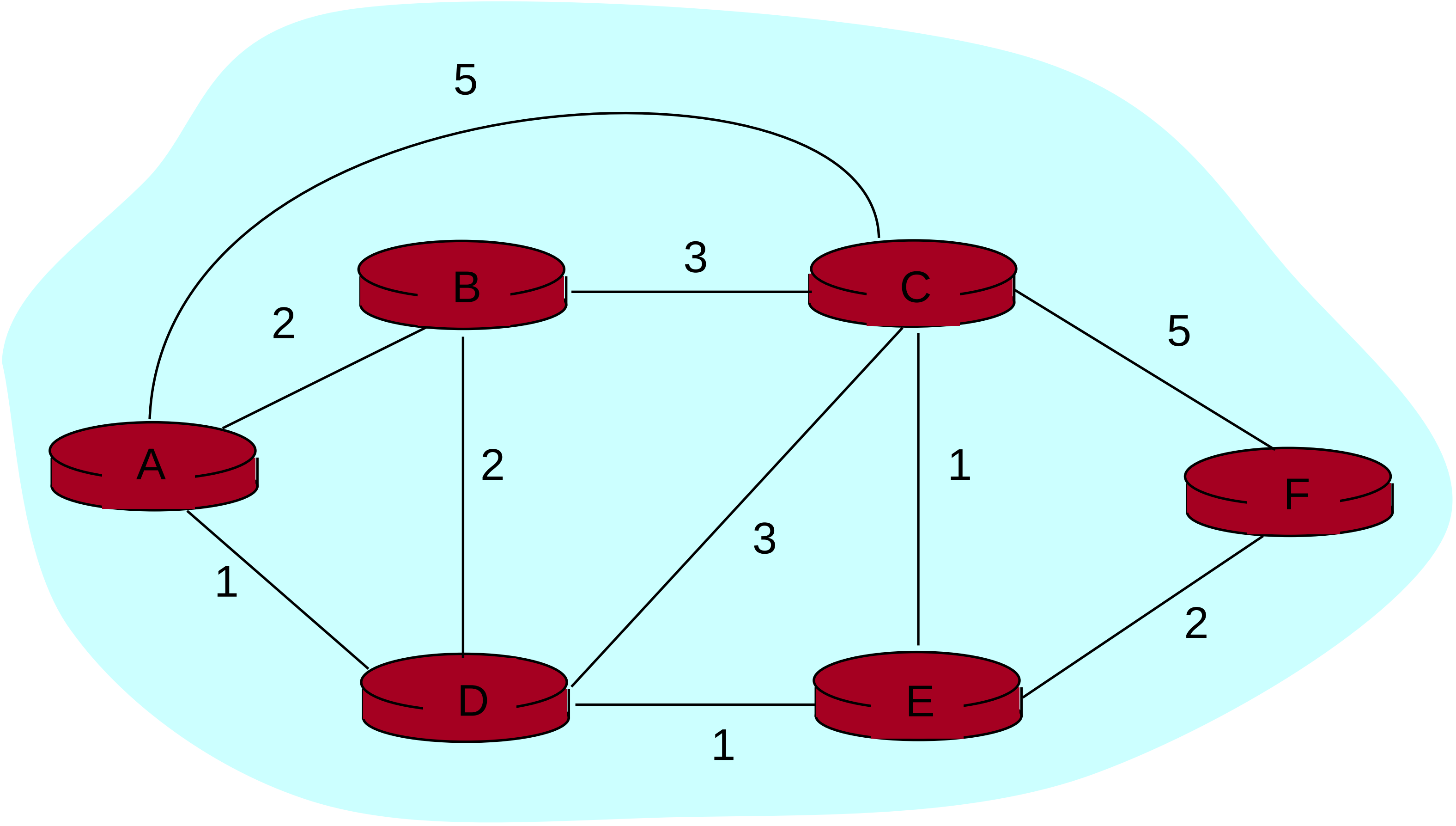
# How to Compute Routes

- Each node should have **same** global view
- They each compute their own routing tables
- Using *exactly* the same algorithm
- Can use *any* algorithm that avoids loops
- Computing shortest paths is one such algorithm
  - Associate "cost" with links, don't worry what it means….
  - Dijkstra's algorithm is one way to compute shortest paths
- We will review Dijkstra's algorithm briefly
  - But that's just because it is expected from such courses
    - Snore….

# "Least Cost" Routes

- No sensible cost metric will be minimized by traversing a loop
- "Least cost" routes an easy way to avoid loops
- Least cost routes are also "destination-based"
  - i.e., do not depend on the source
  - Why is this?
- Therefore, least-cost paths form a spanning tree

# Example

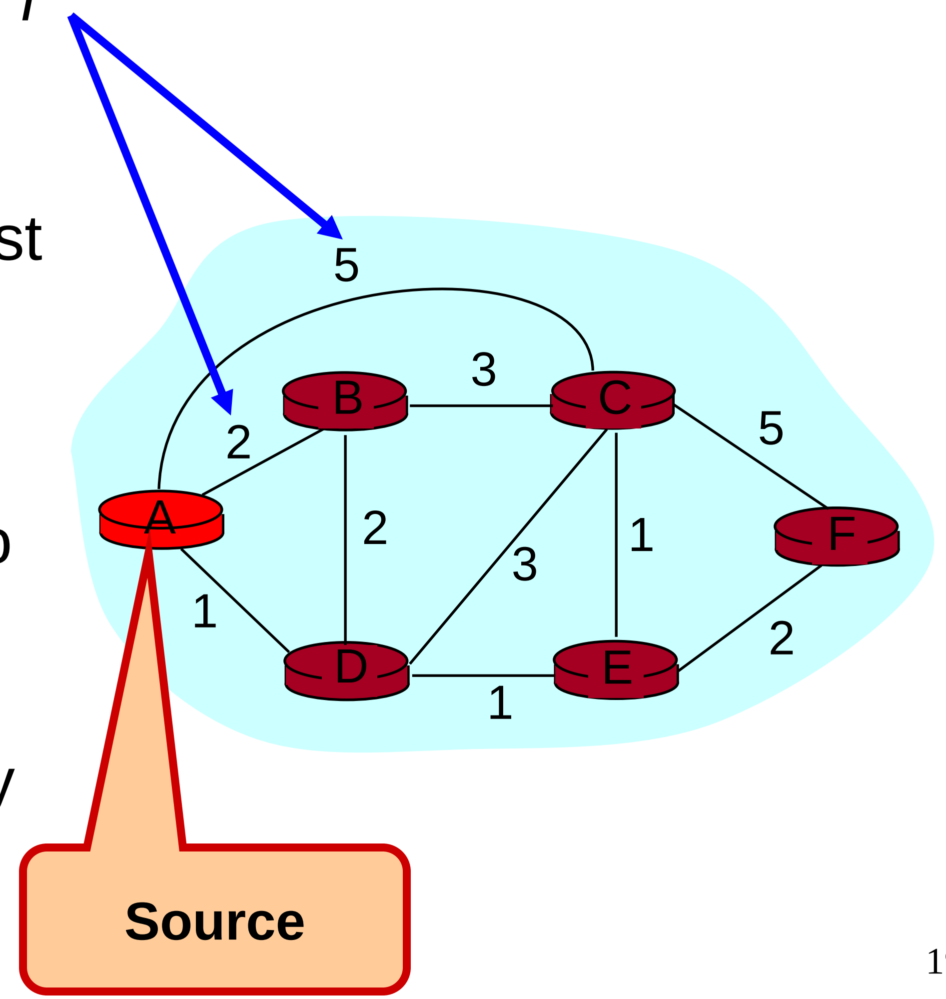


# Dijkstra's Shortest Path Algorithm

- INPUT:
  - Network topology (graph), with link costs
- OUTPUT:
  - Least cost paths **from** one node to all other nodes
  - Produces "tree" of routes
    - Different from what we talked about before
    - Previous tree was rooted at destination
    - This is rooted at source
    - But shortest paths are reversible!
- Warnings:
  - **There is a typo, but I don't remember where (prize!)**
  - Most claim to know Dijkstra, but in practice they don't

# Notation

- c(i,j): link cost from node *i* to *j*; cost infinite if not direct neighbors; **≥ 0**
- D(v): current value of cost of path from source to destination *v*
- p(v): predecessor node along path from source to *v*, that is next to *v*
- S: set of nodes whose least cost path definitively known



# Dijkstra’s Algorithm

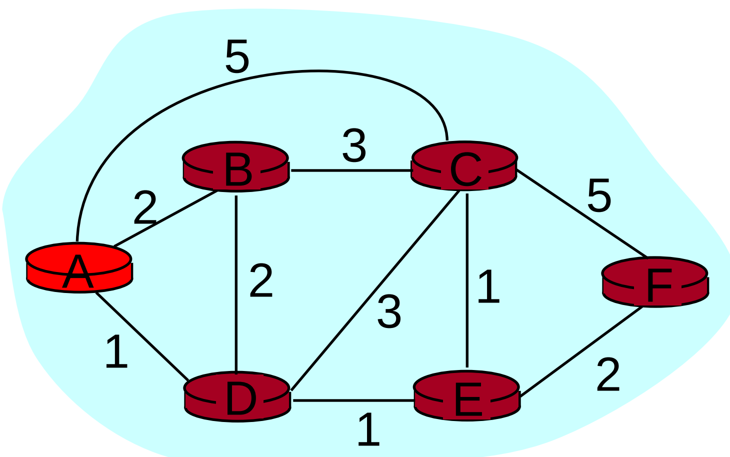
- c(i,j): link cost from node *i* to *j*
- D(v): current cost source → *v*
- p(v): predecessor node along path from source to *v*, that is next to *v*
- S: set of nodes whose least cost path definitively known

***Initialization:***
  **S** = {**A**};
  for all nodes ***v***
    if ***v*** adjacent to ***A***
      then D(v) = c(A,v);
      else D(v) = ¥;

***Loop***
  find **w** not in **S** such that D(w) is a minimum;
  add **w** to **S**;
  update D(v) for all **v** adjacent to **w** and not in **S**:
    if D(w) + c(w,v) < D(v) then
      *// **w** gives us a shorter path to **v** than we’ve found so far*
      D(v) = D(w) + c(w,v); p(v) = w;
***until all nodes in S;***

# Example: Dijkstra’s Algorithm

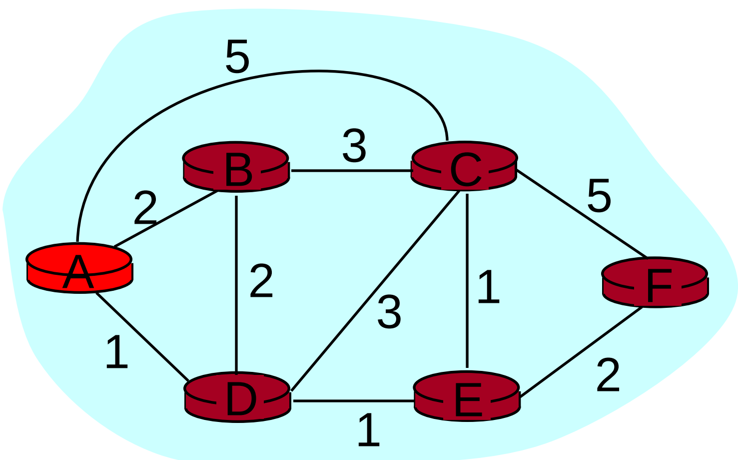
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |




```
Initialization:
  S = {A};
  for all nodes v
    if v adjacent to A
      then D(v) = c(A,v);
      else D(v) = ¥;
…
```

# Example: Dijkstra's Algorithm

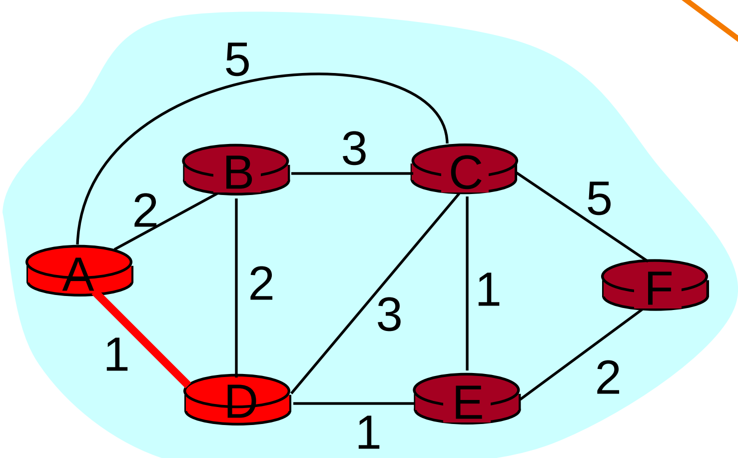
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |




```
…
Loop
  find w not in S s.t. D(w) is a minimum;
  add w to S;
  update D(v) for all v adjacent
     to w and not in S:
  If D(w) + c(w,v) < D(v) then
       D(v) = D(w) + c(w,v); p(v) = w;
until all nodes in S;
```

# Example: Dijkstra's Algorithm

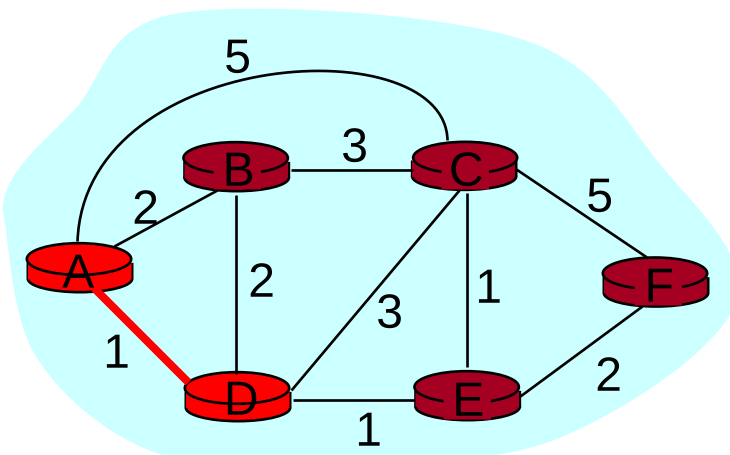
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |




```
…
Loop
    find w not in S s.t. D(w) is a minimum;
    add w to S;
    update D(v) for all v adjacent
      to w and not in S:
    If D(w) + c(w,v) < D(v) then
      D(v) = D(w) + c(w,v); p(v) = w;
until all nodes in S;
```

# Example: Dijkstra’s Algorithm

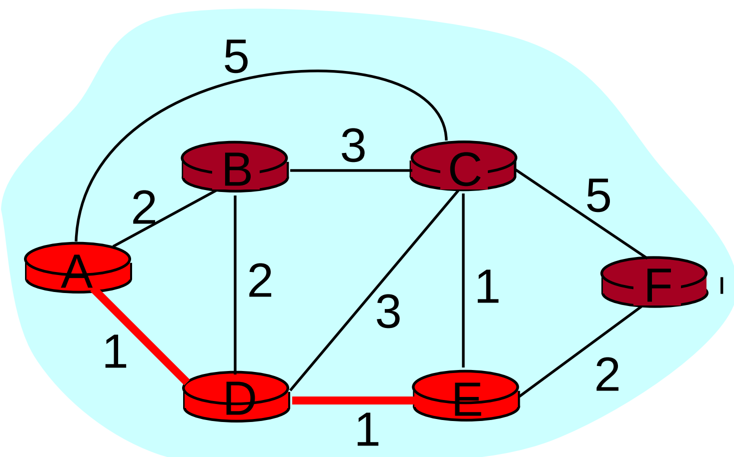
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | 4,D | | 2,D | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |




```
…
Loop
    find w not in S s.t. D(w) is a minimum;
    add w to S;
   update D(v) for all v adjacent
      to w and not in S:
   If D(w) + c(w,v) < D(v) then
       D(v) = D(w) + c(w,v); p(v) = w;
    until all nodes in S;
```

# Example: Dijkstra's Algorithm

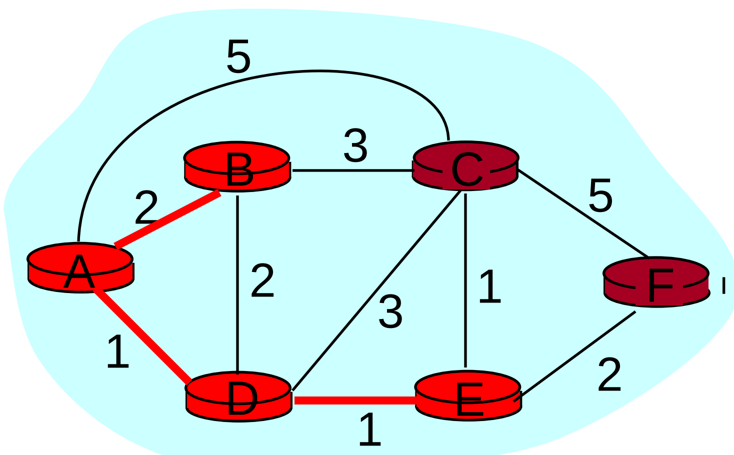
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | 4,D | | 2,D | |
| 2 | ADE | | 3,E | | | 4,E |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |




…
***Loop***
find **w** not in **S** s.t. D(w) is a minimum;
add **w** to **S**;
update D(v) for all **v** adjacent
to **w** and not in **S**:
If D(w) + c(w,v) < D(v) then
D(v) = D(w) + c(w,v); p(v) = w;
***until all nodes in S;***

# Example: Dijkstra's Algorithm

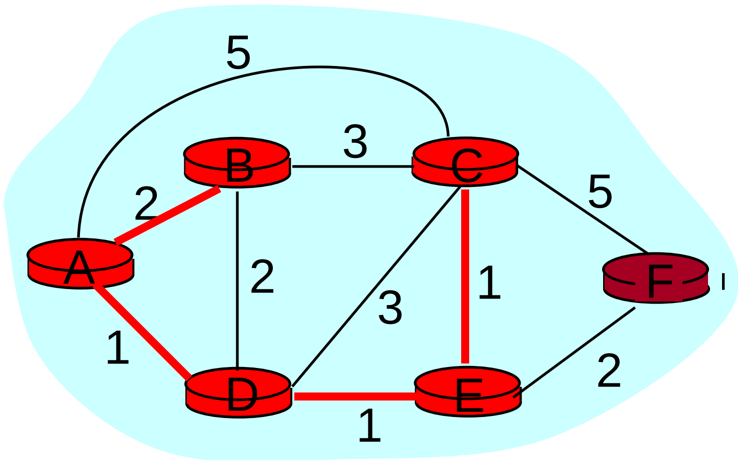
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | 4,D | | 2,D | |
| 2 | ADE | | 3,E | | | 4,E |
| 3 | ADEB | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |




```
…
Loop
    find w not in S s.t. D(w) is a minimum;
    add w to S;
   update D(v) for all v adjacent
      to w and not in S:
   If D(w) + c(w,v) < D(v) then
      D(v) = D(w) + c(w,v); p(v) = w;
 until all nodes in S;
```

# Example: Dijkstra’s Algorithm

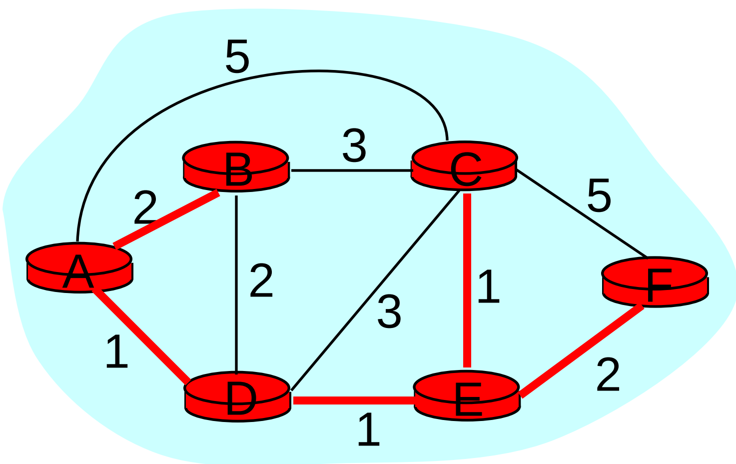
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | 4,D | | 2,D | |
| 2 | ADE | | 3,E | | | 4,E |
| 3 | ADEB | | | | | |
| 4 | ADEBC | | | | | |
| 5 | | | | | | |




```
…
Loop
    find w not in S s.t. D(w) is a minimum;
    add w to S;
   update D(v) for all v adjacent
      to w and not in S:
   If D(w) + c(w,v) < D(v) then
       D(v) = D(w) + c(w,v); p(v) = w;
    until all nodes in S;
```

# Example: Dijkstra's Algorithm

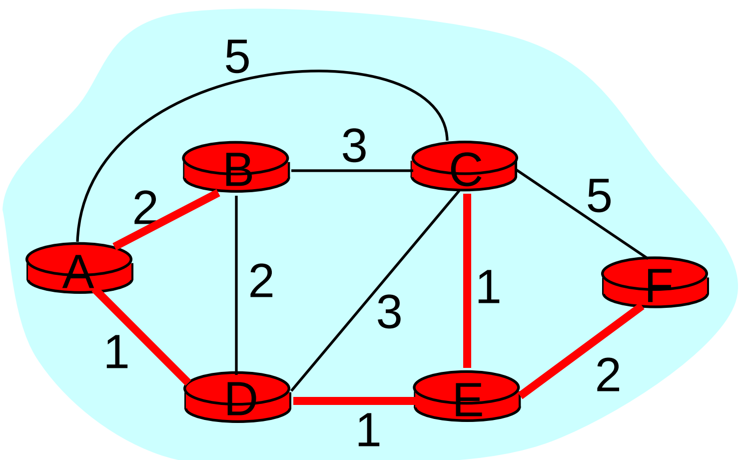
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | 4,D | | 2,D | |
| 2 | ADE | | 3,E | | | 4,E |
| 3 | ADEB | | | | | |
| 4 | ADEBC | | | | | |
| 5 | ADEBCF | | | | | |




```
…
8  Loop
9      find w not in S s.t. D(w) is a minimum;
10     add w to S;
11    update D(v) for all v adjacent
         to w and not in S:
12    If D(w) + c(w,v) < D(v) then
13        D(v) = D(w) + c(w,v); p(v) = w;
14    until all nodes in S;
```

# Example: Dijkstra's Algorithm

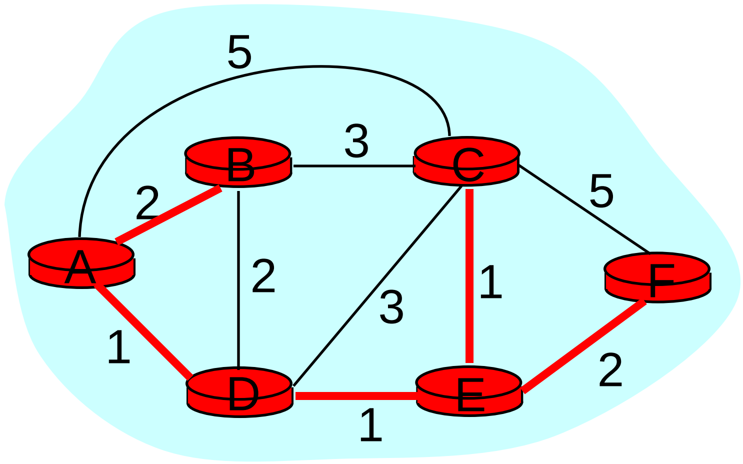
| Step | start S | D(B),p(B) | D(C),p(C) | D(D),p(D) | D(E),p(E) | D(F),p(F) |
|---|---|---|---|---|---|---|
| 0 | A | 2,A | 5,A | 1,A | ¥ | ¥ |
| 1 | AD | | 4,D | | 2,D | |
| 2 | ADE | | 3,E | | | 4,E |
| 3 | ADEB | | | | | |
| 4 | ADEBC | | | | | |
| 5 | ADEBCF | | | | | |




To determine path A → C (say), work backward from C via p(v)

# The Forwarding Table

- Running Dijkstra at node A gives the shortest path from A to all destinations
- We then construct the *forwarding table*



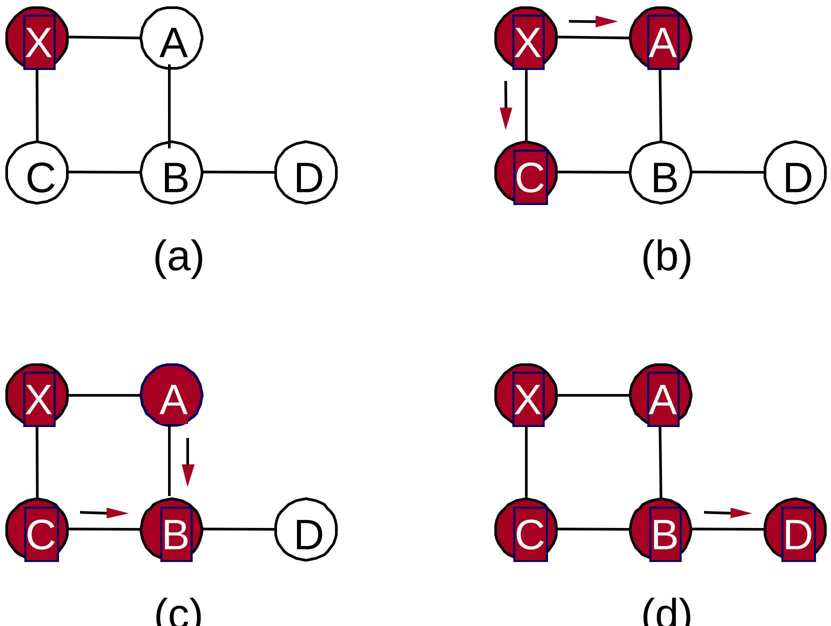

| Destination | Link |
|---|---|
| B | (A,B) |
| C | (A,D) |
| D | (A,D) |
| E | (A,D) |
| F | (A,D) |

# Complexity

- How much processing does running the Dijkstra algorithm take?
- Assume a network consisting of N nodes
  - Each iteration: check all nodes w not in S
  - N(N+1)/2 comparisons: $O(N^2)$
  - More efficient implementations: O(N log(N))

# Flooding the Topology Information

- Each router sends information out its ports
- The next node sends it out through all of its ports
  - Except the one where the information arrived
  - Need to remember previous msgs, suppress duplicates!



# Making Flooding Reliable

- Reliable flooding
  - Ensure all nodes receive link-state information
  - Ensure all nodes use the latest version
- Challenges
  - Packet loss
  - Out-of-order arrival
- Solutions
  - Acknowledgments and retransmissions
  - Sequence numbers
- How can it still fail?

# When to Initiate Flood?

- Topology change
  - Link or node failure
  - Link or node recovery
- Configuration change
  - Link cost change
  - Potential problems with making cost dynamic!
- Periodically
  - Refresh the link-state information
  - Typically (say) 30 minutes
  - Corrects for possible corruption of the data

# Convergence

- Getting consistent routing information to all nodes
  - E.g., all nodes having the same link-state database

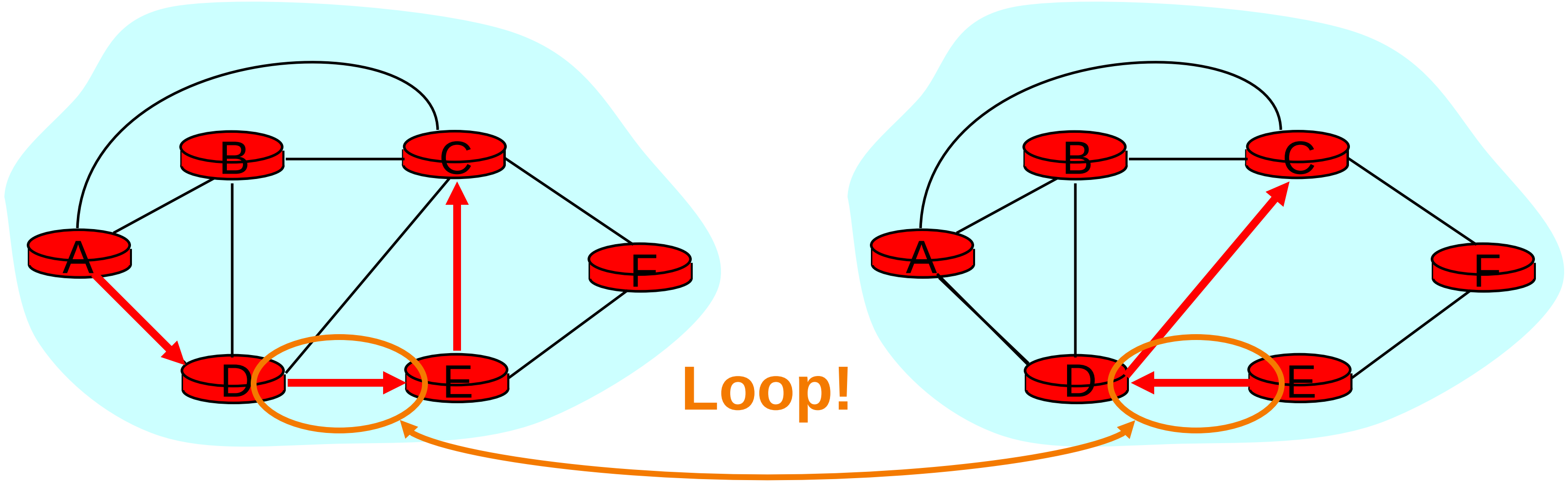
- Forwarding is consistent after convergence
  - All nodes have the same link-state database
  - All nodes forward packets on same paths

# Convergence Delay

- Time elapsed before every router has a consistent picture of the network
- Sources of convergence delay
  - Detection latency
  - Flooding of link-state information
  - Recomputation of forwarding tables
  - Storing forwarding tables
- Performance during convergence period
  - Lost packets due to blackholes and TTL expiry
  - Looping packets consuming resources
  - Out-of-order packets reaching the destination
- Very bad for VoIP, online gaming, and video

# Transient Disruptions

- Inconsistent link-state database
  - Some routers know about failure before others
  - The shortest paths are no longer consistent
  - Can cause transient forwarding loops




**A and D think that this is the path to C**

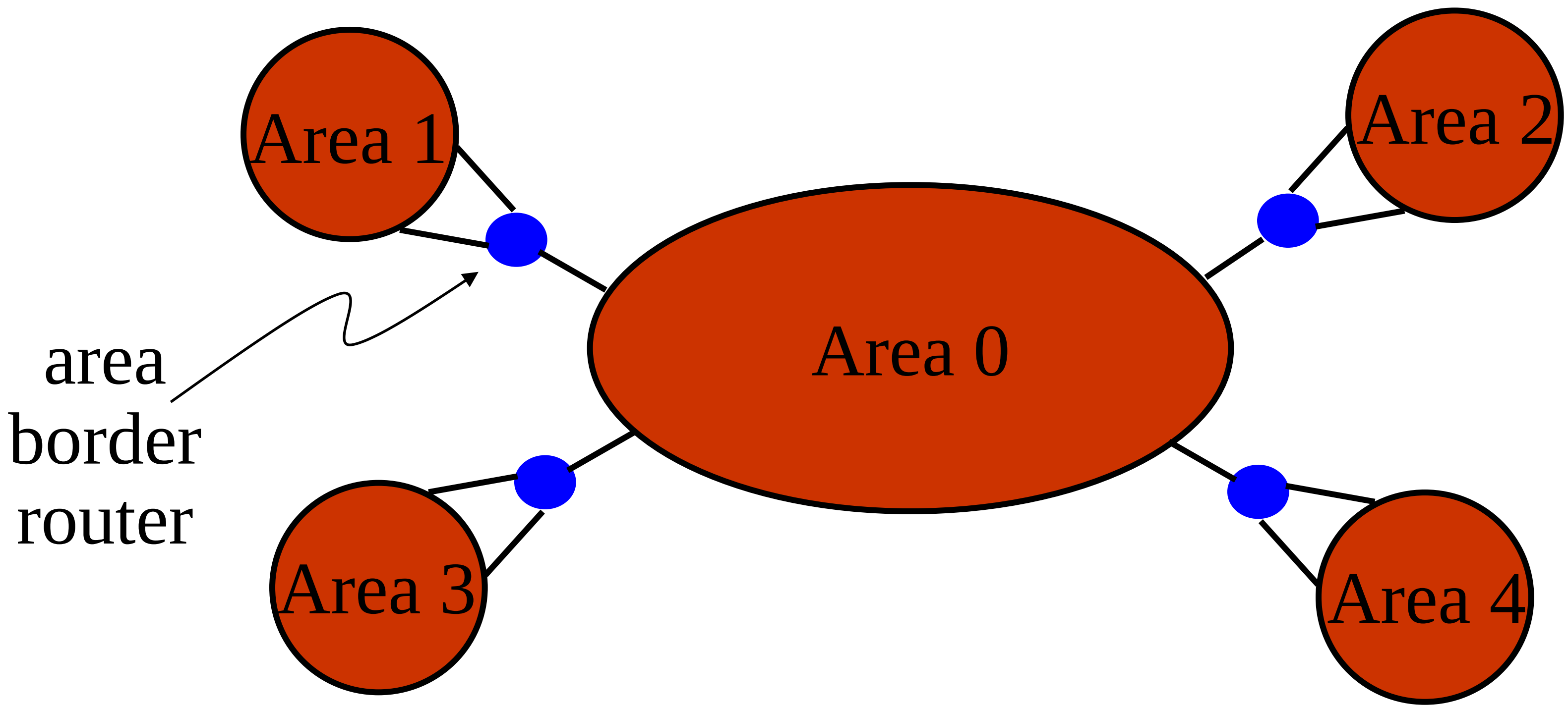
**E thinks that this is the path to C**

# Reducing Convergence Delay

- Faster detection
  - Smaller "hello" timers
  - Link-layer technologies that can detect failures
- Faster flooding
  - Flooding immediately
  - Sending link-state packets with high-priority
- Faster computation
  - Faster processors on the routers
  - Incremental Dijkstra algorithm
- Faster forwarding-table update
  - Data structures supporting incremental updates

# Scaling Link-State Routing

- Overhead of link-state routing
  - Flooding link-state packets throughout the network
  - Running Dijkstra's shortest-path algorithm
  - Becomes unscalable when 100s of routers
- Introducing hierarchy through "areas"



# What about other approaches?

- Link-state is essentially a centralized computation:
  - **Global state, local computation**

- What about a more distributed approach?
  - **Local state, global computation**

## Learn-By-Doing

I need 40 volunteers

***If you haven't participated, this is your chance!***

# The Task

- Remove sheet of paper from beanbag, but do not look at sheet of paper until I say so
- You will have five minutes to complete this task
- Each sheet says:

**You are node X  You are connected to nodes Y,Z**

- **Your job:** find route from source (node 1) to destination (node 40) in five minutes

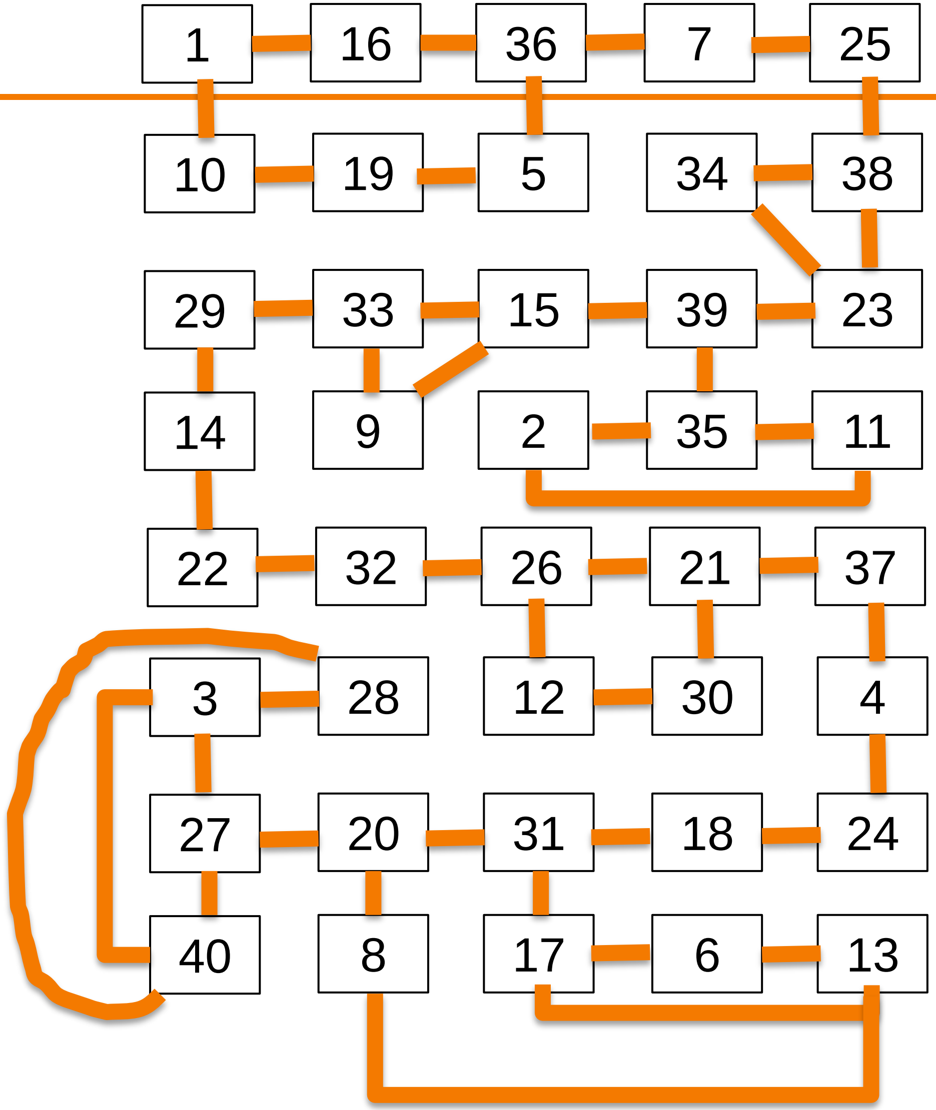
# Ground Rules

- **You may not**:
  - Leave your seat (but you can stand)
  - Pass your sheet of paper
  - Let anyone copy your sheet of paper

- **You may**:
  - Ask nearby friends for advice
  - Shout to other participants (anything you want)
  - Curse your instructor (*sotto voce*)

- **You must**: ***Try***

**Go!**



1
16
36
7
25
10
19
5
34
38
29
33
15
39
23
14
9
2
35
11
22
32
26
21
37
3
28
12
30
4
27
20
31
18
24
40
8
17
6
13

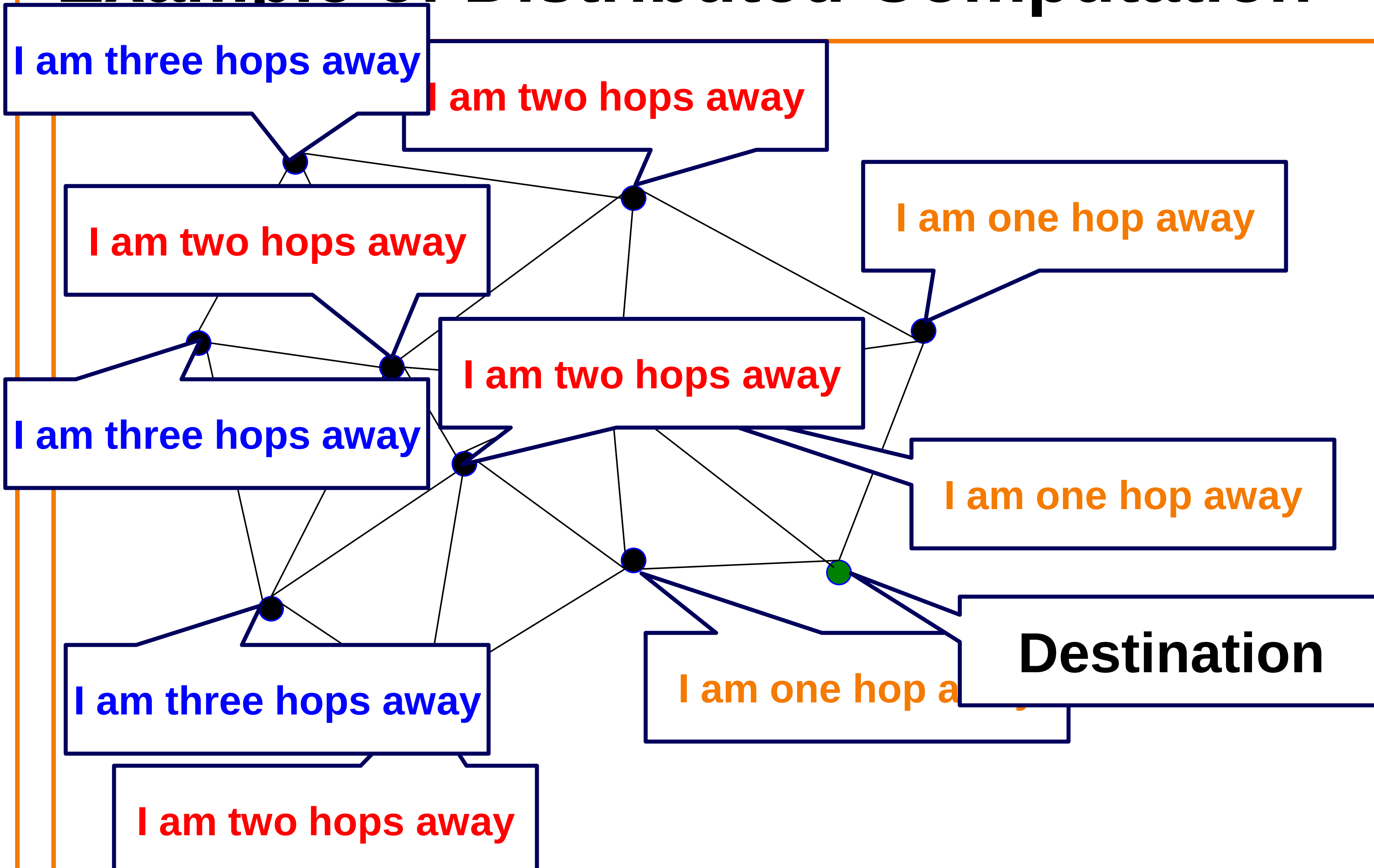
# Distance-Vector

Details in Section

# Distributed Computation of Routes

- More scalable than Link-State
  - No global flooding
- Each node computing the outgoing port based on:
  - Local information (who it is connected to)
  - Paths advertised by neighbors
- Algorithms differ in what these exchanges contain
  - Distance-vector: just the distance to each destination
  - Path-vector: the entire path to each destination
- We will focus on distance-vector for now

# Example of Distributed Computation



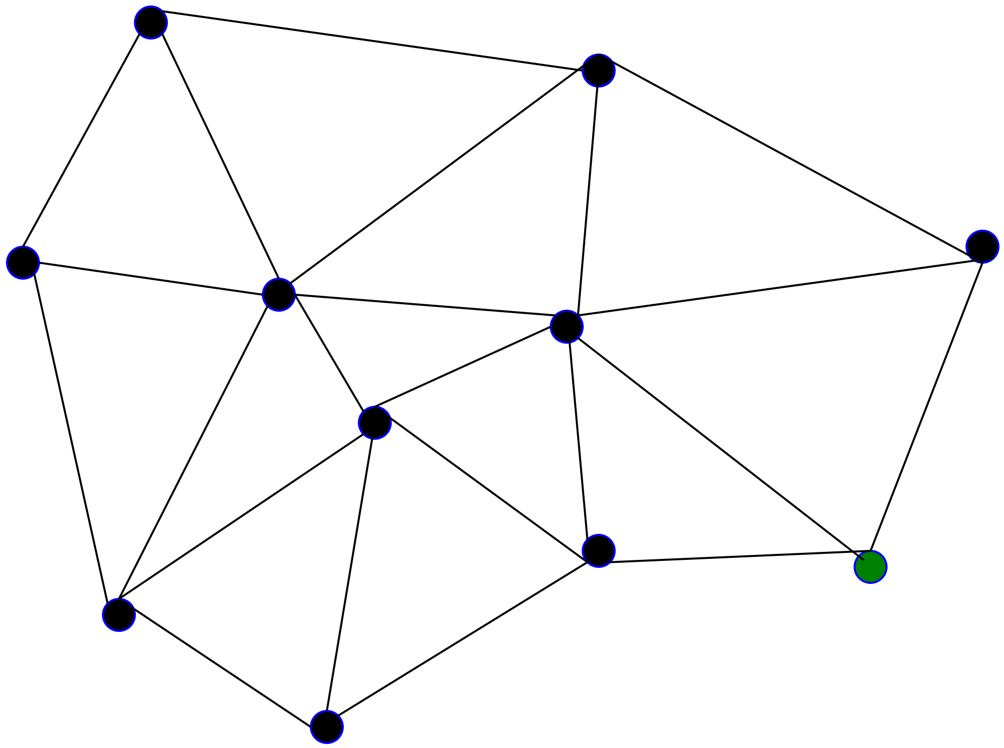
# This is what you could have done

- Destination stands up
- Announces neighbors
  - They stand up
- They announce their neighbors
  - They stand up (if they haven’t already done so)
  - **They remember who called them to stand**
- …..and so on, until source stands
- **Key point: don’t stand up twice!**

# Step 1

- Destination stands up

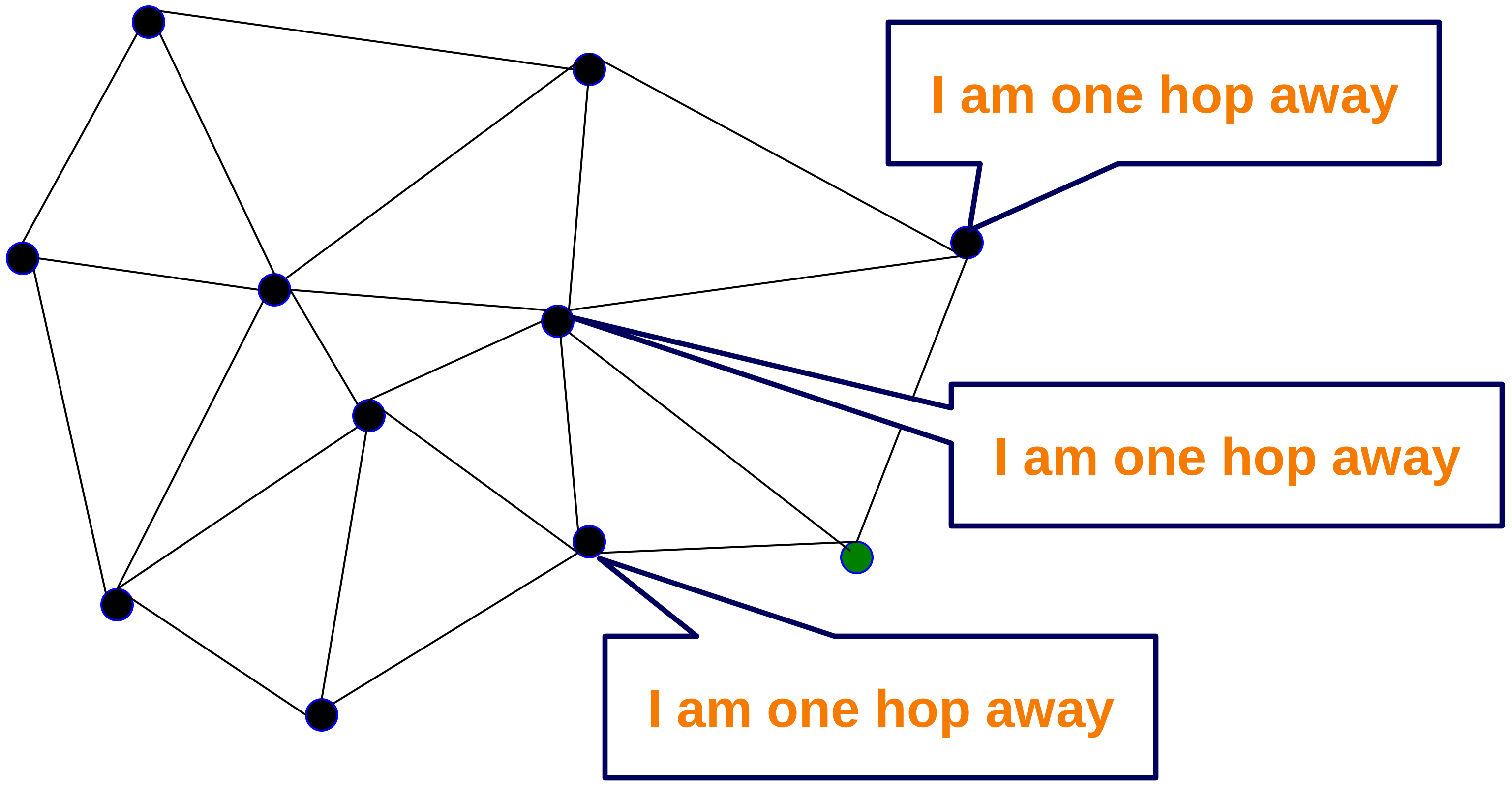
# Step 1



# Step 2

- Destination stands up
- Announces neighbors
  - They stand up

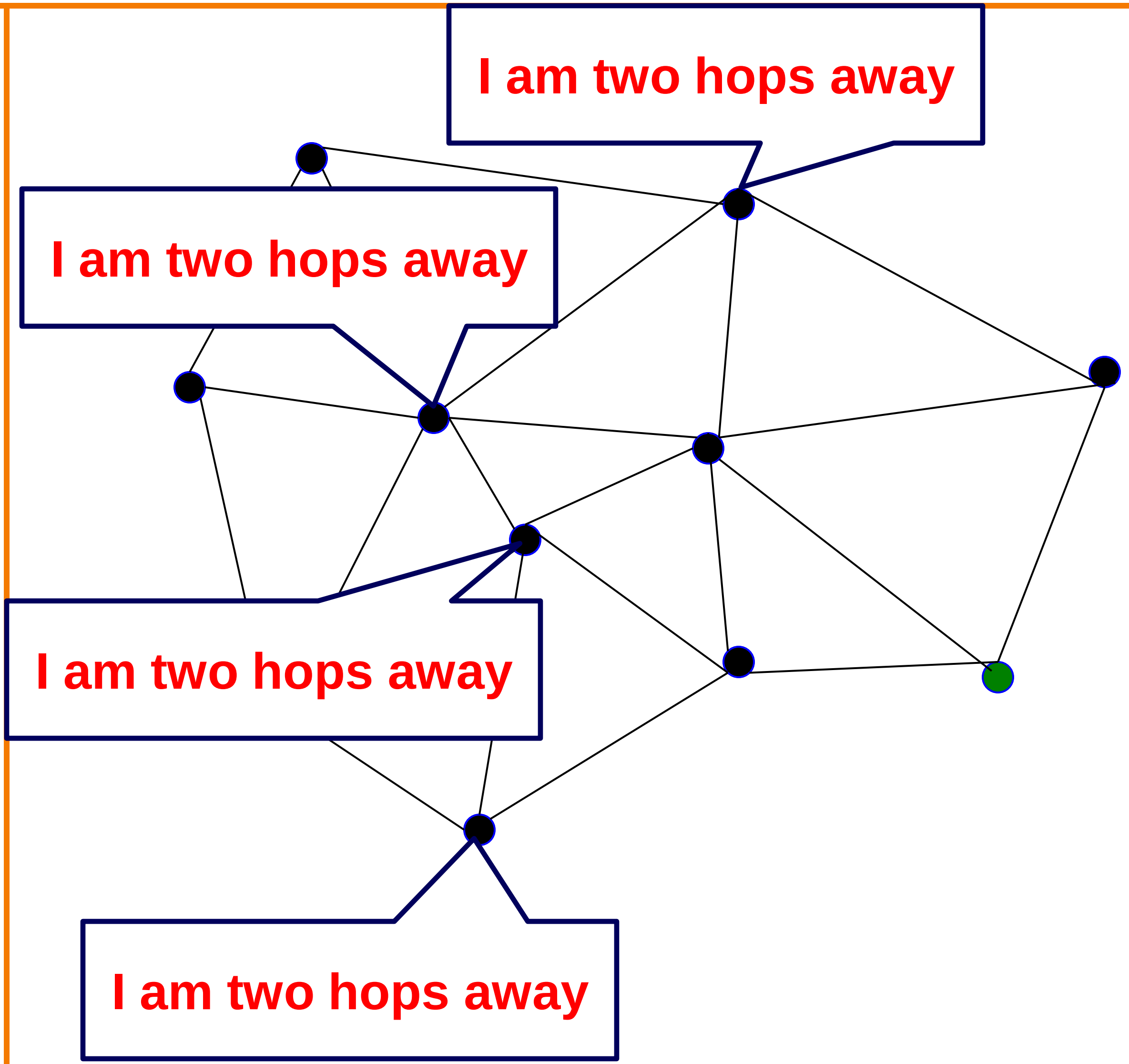
# Step 2



# Step 3

- Destination stands up
- Announces neighbors
  - They stand up
- They announce their neighbors
  - They stand up

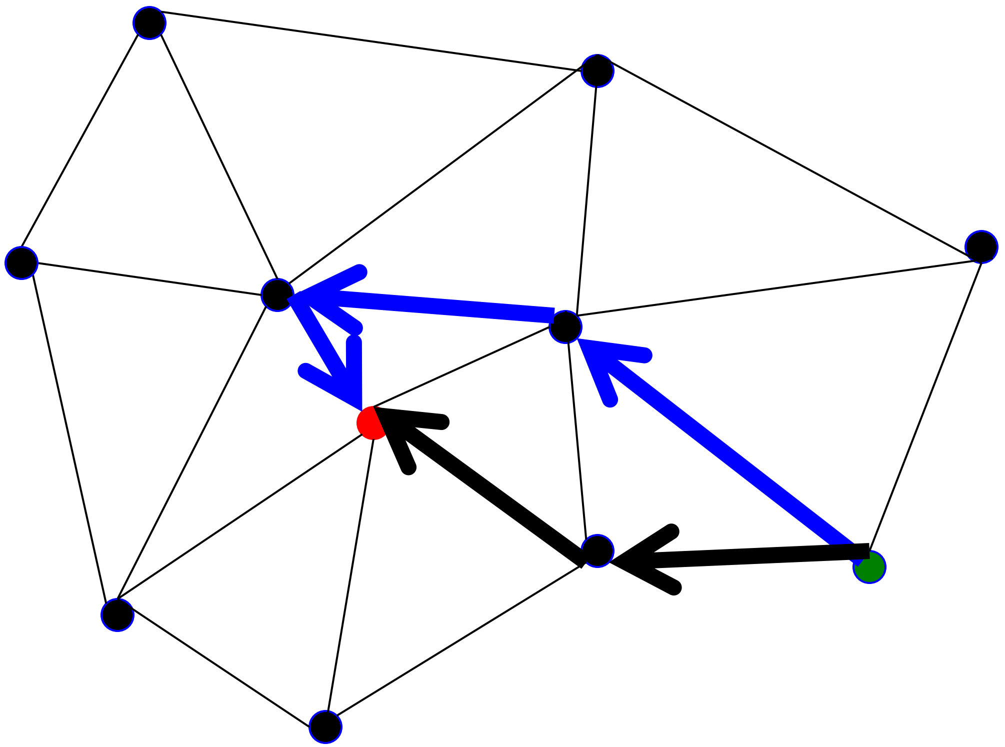
# Step 3



I am two hops away
I am two hops away
I am two hops away
I am two hops away

# Why Not Stand Up Twice?

- Being called a second time means that there is a second (and longer) path to you
  - You already contacted your neighbors the first time
  - Your distance to destination is based on shorter path



# Congratulations!

- You have “implemented” Distance-Vector routing
  - For a single destination
  - With the slowest code possible
- OK, so now let’s consider this more generally….

# Routing "Metrics"

- Algorithm finds path with smallest hop-count
  - More complicated if you route with a different metric

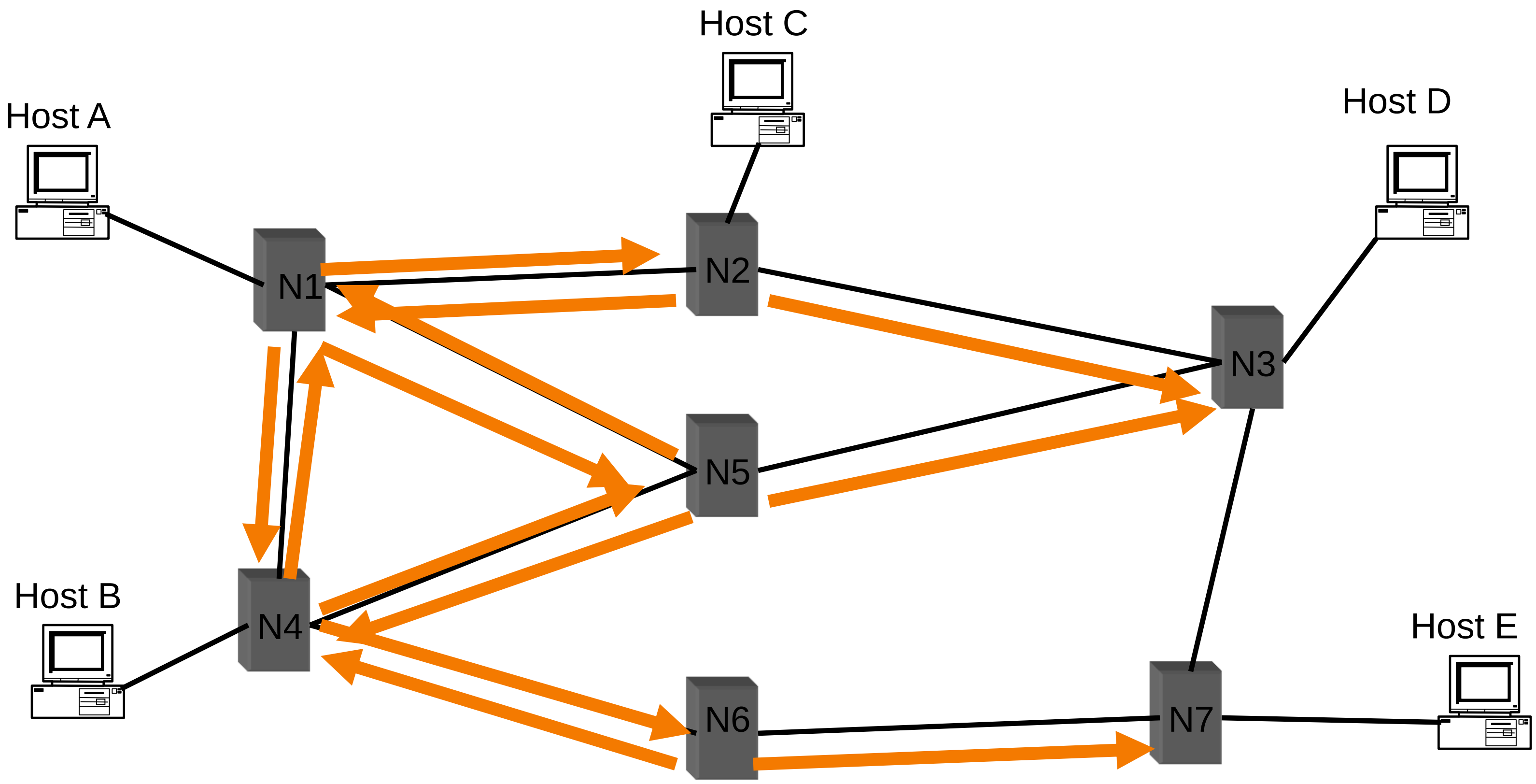
- Other routing goals (besides hop-count)
  - Path with highest capacity
  - Path with lowest latency
  - Path with most reliable links
  - ….

- Generally, assume every link has "cost" or weight associated with it, and you want to minimize cost

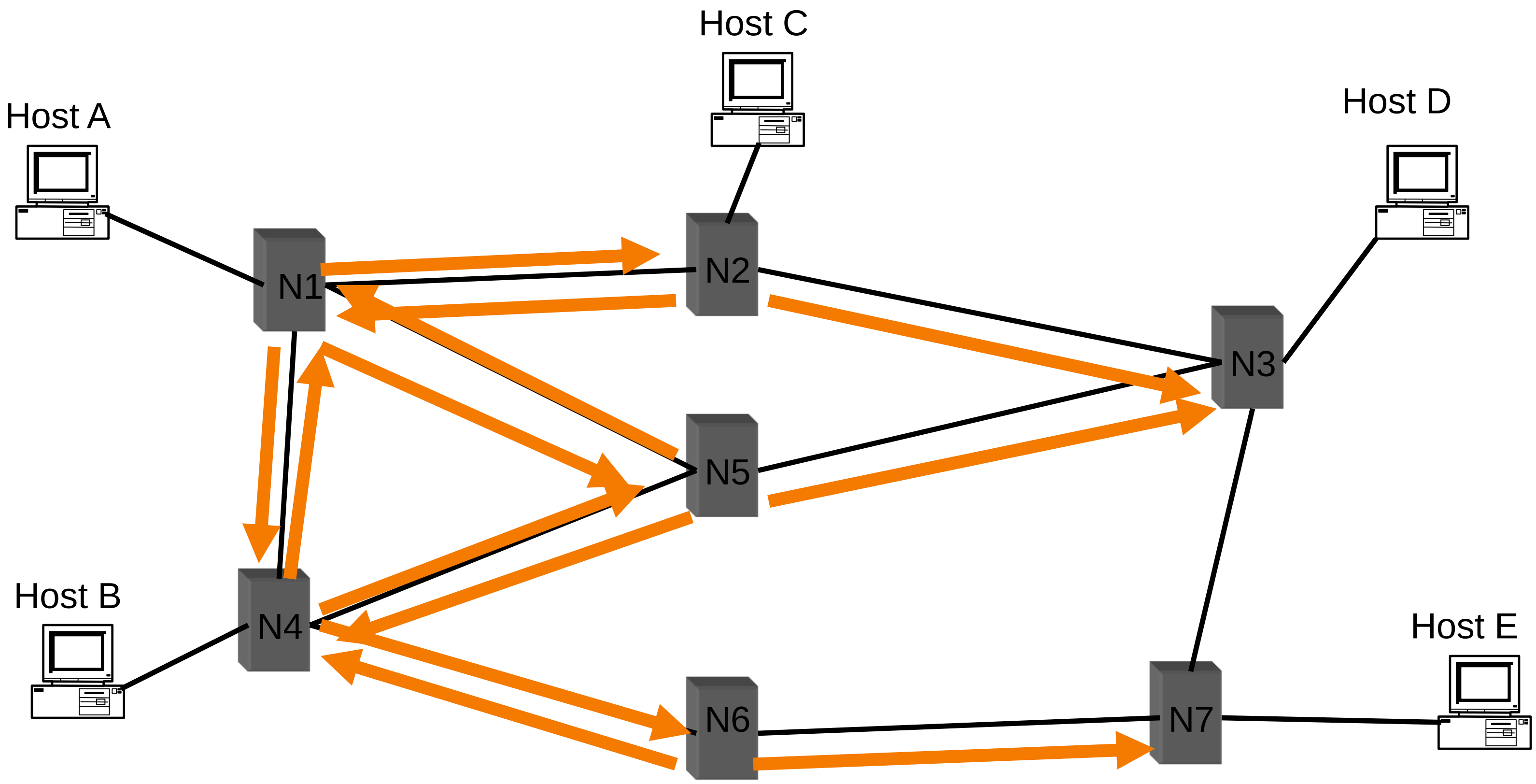
# Distance Vector Routing

- Each router knows the links to its neighbors
  - Does *not* flood this information to the whole network
- Each router has provisional "shortest path"
  - E.g.:  Router A: "I can get to router B with cost 11 via next hop router D"
- Routers exchange this *Distance-Vector* information with their neighboring routers
  - Vector because one entry per destination
- Routers update their idea of the best path using info from neighbors
- Iterative process converges to set of shortest paths

# Information Flow in Distance Vector



# Information Flow in Distance Vector



# Information Flow in Distance Vector