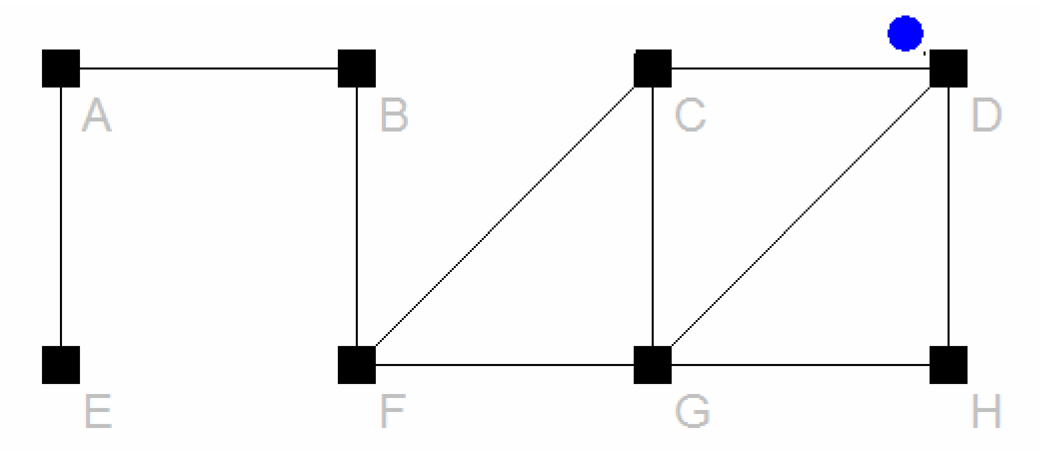
Shortest Path Routing

- Communications networks as graphs
- Graph terminology
- Breadth-first search in a graph
- Properties of breadth-first search

Routing in an arbitrary network



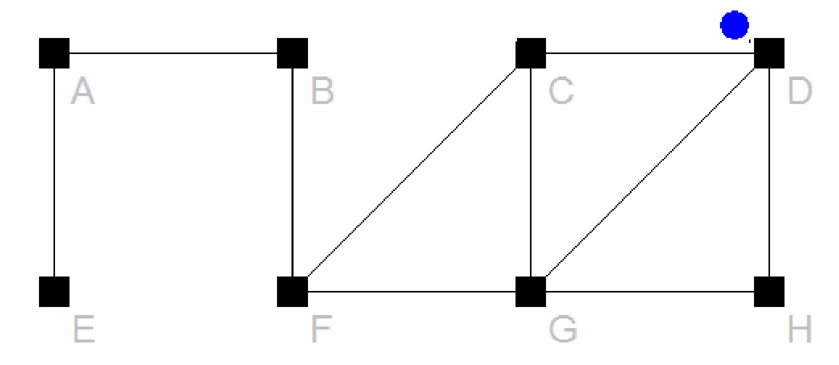
Suppose we'd like to send a packet from node D to node F. There are several possible routes for the packet to take – which should we choose?

One common choice to find a **shortest path**, i.e., a path that traverses the fewest number of communication links, since this will minimize the use of routing resources.

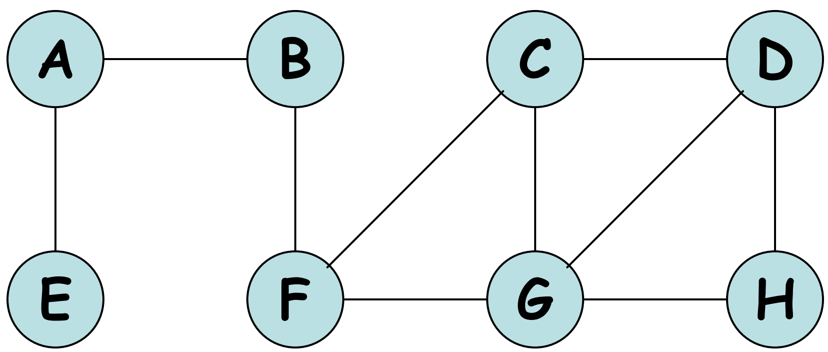
To find a shortest path, we'll turn to an elementary graph algorithm: breadth-first search. Network algorithms are often closely related to graph algorithms for obvious reasons!

Networks As Graphs

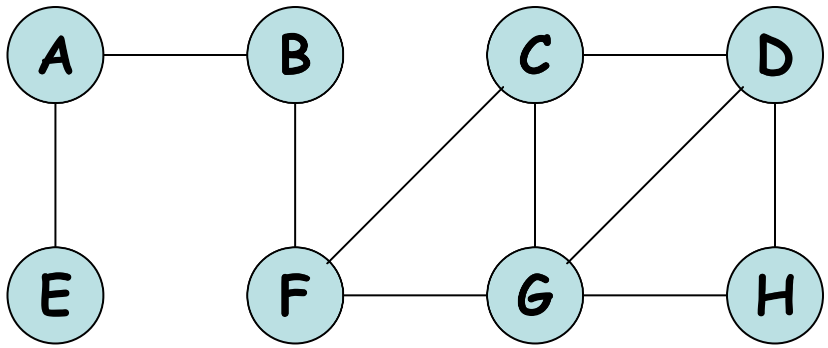
A network composed of nodes and bidirectional communication links



is easily converted into an undirected graph composed of vertices and edges:



Graph Terminology



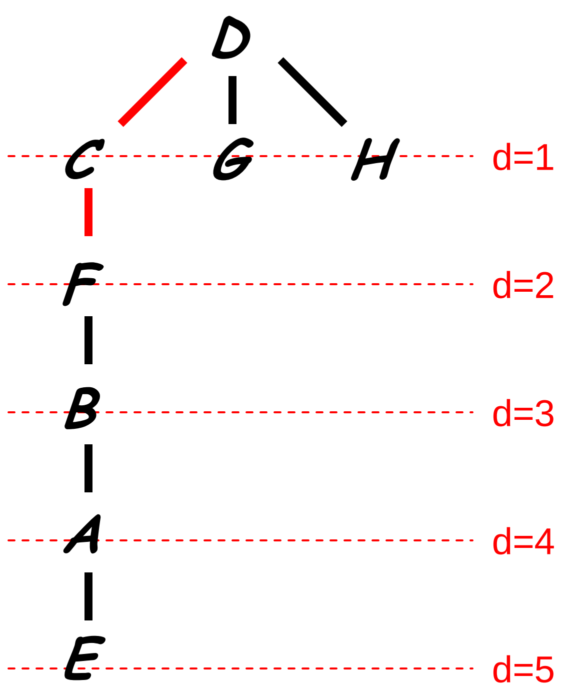
V = { A B C D E F G H }
E = { (A,E) (A,B) (B,F) (C,D) (C,F) (C,G) (D,G) (D,H) (F,G) (G,H) }

A graph is composed of

- a set V of vertices
  - |V| is the number of elements in V
- a set E of directed or undirected edges of the form (u,v) where *u, v* ∈ V
  - (u,v) is an edge connecting vertex *u* and vertex *v*
  - Adj[u] contains all vertices v such that (u,v) ∈ E
  - a graph is called *sparse* if $|E| \ll |V|^2$
- total memory required to store graph = O(|V|+|E|) ≡ O(V+E)

Breadth-first search

- Given a graph G = (V,E) and a distinguished source vertex *s*, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from *s*.
- It produces a "breadth-first tree" with root *s* that contains all reachable vertices.
- Path from root to node *v* in tree will be a shortest path from *s* to *v*.
- It's called "breadth-first" because the algorithm discovers all nodes of distance K from *s* before discovering any nodes of distance K+1.



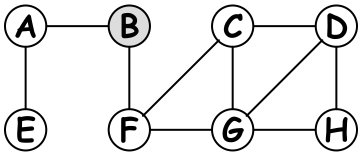
 See section 22.2 of "Introduction to Algorithms" by Cormen, Leiserson, Rivest, Stein

Breadth-first Search Procedure

```
def BFS(G,s):                   # G is a graph with V and Adj
  dist = {}                     # maps vertex to distance from s
  parent = {}                   # maps vertex to parent in BFS tree
  color = {}                    # maps vertex to color: white=undiscovered
                                # gray=discovered, black=processed
  for u in G.V:                 # loop through all nodes
    color[u] = 'white'          #   node hasn't been discovered yet
    dist[u] = infinity          #   no distance from root yet
    parent[u] = None            #   no parent in BFS tree yet
  color[s] = 'gray'             # root has been discovered
  dist[s] = 0                   # root is distance 0 from itself
  parent[s] = None              # root has no parent!
  Q = Queue(0)                  # set up first-in, first-out queue
  Q.put(s)                      # root is the first entry in the queue
  while not Q.empty():          # loop until queue is empty
    u = Q.get()                 #   get next vertex from the queue
    for v in G.Adj[u]:          #   loop through all its neighbors
      if color[v]=='white':     #     if v hasn't been discovered
        color[v] = 'gray'       #       mark v as discovered
        dist[v] = dist[u]+1     #       v is one hop further than parent
        parent[v] = u           #       parent is vertex from Queue
        Q.put(v)                #       process v's neighbors later
    color[u] = 'black'          #   mark vertex from Queue as processed
```

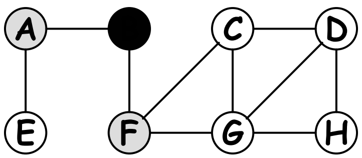
Example (root = B)

1



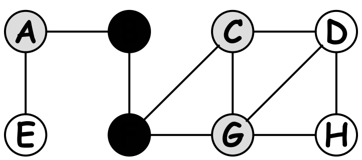
Q: B
0

2



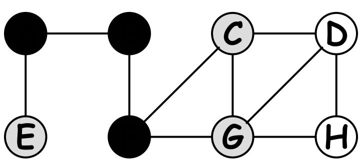
Q: F A
1 1

3



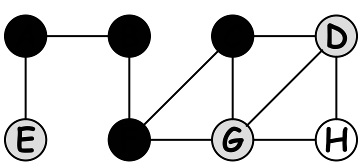
Q: A C G
1 2 2

4



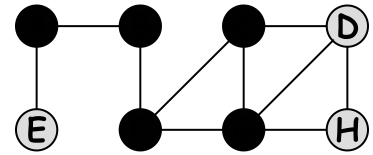
Q: C G E
2 2 2

5



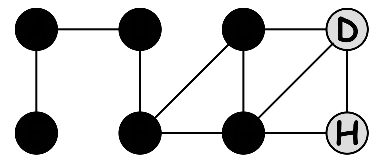
Q: G E D
2 2 3

6



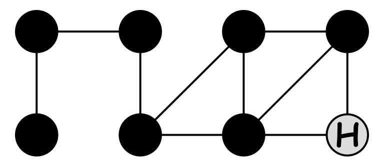
Q: E D H
2 3 3

7



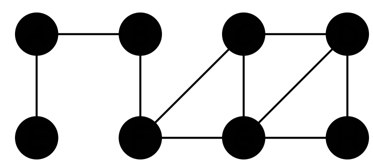
Q: D H
3 3

8



Q: H
3

9



Q:

Node colors and Q shown at start of each iteration of while loop

Questions to ask about algorithms

- Does algorithm terminate on all inputs?
- Does it provably compute the desired result?
  - In BFS, does dist[u] equal the shortest-path distance from s to u?
  - Does it discover all nodes reachable from s?
- How much space and time does the algorithm require?
  - Usually expressed in terms of asymptotic behavior, e.g., O(…) where … is related to the size of the input.
  - In BFS: the size of the graph is given by |V| and |E|.

BFS: time & space

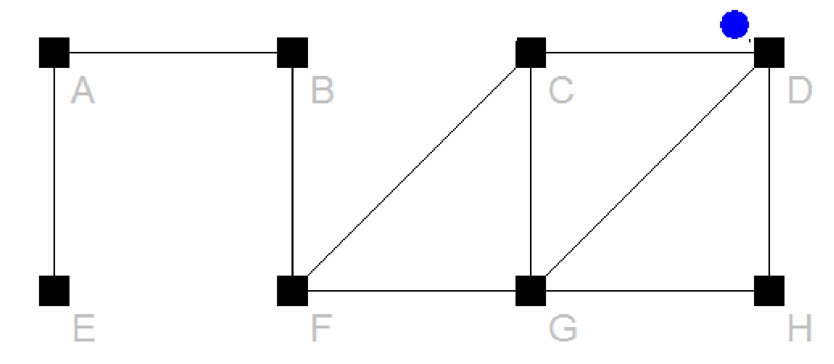
- Initialization of *dist, parent, color* takes $O(V)$ time and they require $O(V)$ space.
- Queue.put and Queue.get take constant time, i.e., time does not depend on size of queue. Maximum queue size is $O(V)$.
- Each vertex is enqueued and dequeued exactly once, so time for all queue operations is $O(V)$.
- The sum of the lengths of the Adj[] lists is $O(E)$ and each adjacency list is scanned once when node is dequeued, so the total amount of time spent scanning adjacency lists is $O(E)$.
- Total processing time: $O(V+E)$
- Total processing space: $O(V+E)$

The "B" in BFS

- **The queue is used to organize the search to be breadth first -- we process all the nodes at distance K before processing their neighbors at distance K+1**
  - Queue ordering from example: `B F A C G E D H`
    distance: `0 1 1 2 2 2 3 3`

- **A vertex's color is used to distinguished nodes that have been processed from nodes that haven't – the search processes each node exactly once. Note that all nodes on the queue are colored gray: they've been discovered but their adjacency lists have not been scanned. Nodes not on the queue are either**
  - Black (the node and its neighbors have been discovered) or
  - White (the node hasn't yet been discovered)

Back to networks…



- How does a node learn about it's neighbors?
  - Periodically send HELLO packets on outgoing links
  - Remember source address of arriving HELLO packets
  - What happens when link goes down?
- How does a node get Adj[] lists from other nodes?
  - Periodically broadcast neighbor list to all nodes (LSA packets)
  - Remember most recent LSA packet from each source
  - What happens when link goes down?
- How does a node build its shortest path routing table?
  - Run modified BFS using LSA info
  - Only want outgoing link to use for first hop (don't need distance or complete route to destination).

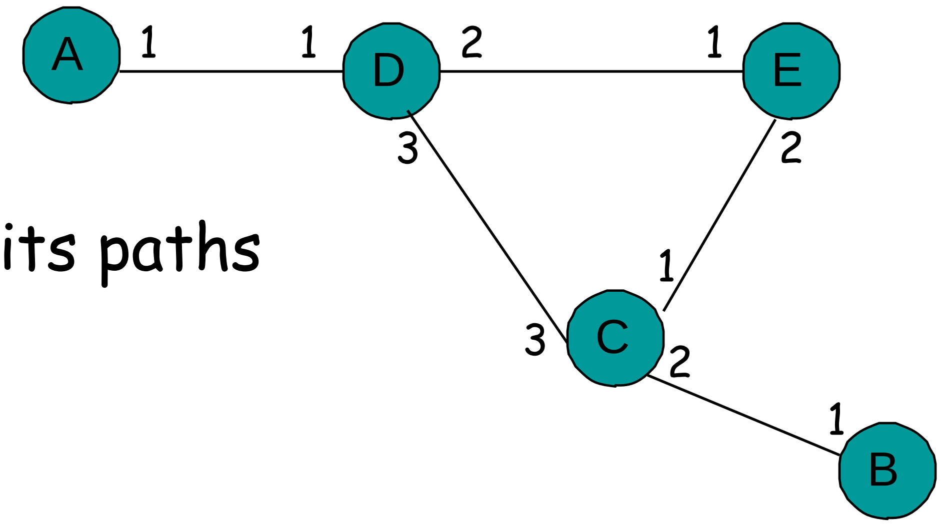
Slides for Friday

Other shortest-path routing algorithms

- **In the link-state routing algorithm of Lab 9**
  - Each node receives neighbor info from every node in the network
  - Each node knows about all the paths through the network
  - Each node selects shortest path using BFS
- **If all we want is the shortest path why learn about all paths?**
  - To choose the right outgoing link, all a node needs to know is which of its neighbors has the shortest path to the destination
  - Idea: Have neighbors only tell us enough info for us to make the right routing decision

Path Vector Routing Protocol

- **Initialization**
  - **Each node knows the path to itself**



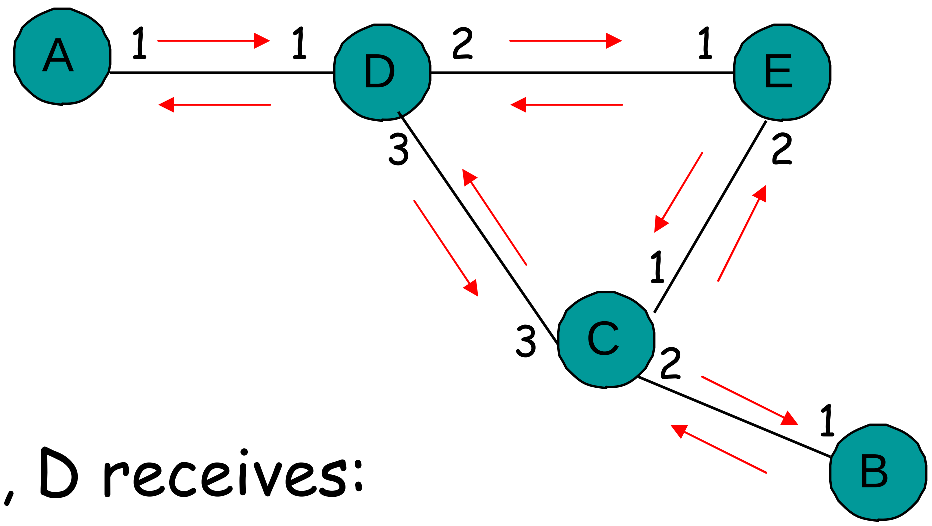
For example, D initializes its paths

DST	Link	Path
D	End layer	null

Slides are from lectures by Nick Mckeown, Ion Stoica, Frans Kaashoek, Hari Balakrishnan, Sam Madden, and Dina Katabi

Path Vector

- **Step 1: Advertisement**
  - **Each node tells its neighbors its path to each node in the graph**



For example, D receives:

From A:

To	Path
A	null

From C:

To	Path
C	null

From E:

To	Path
E	null

Path Vector

- **Step 2: Update Route Info**
  - **Each node use the advertisements to update its paths**

D received:

From A:

To	Path
A	null

From C:

To	Path
C	null

From E:

To	Path
E	null

D updates its paths:

DST	Link	Path
D	End layer	null

→

DST	Link	Path
D	End layer	null
A	1	<A>
C	3	<C>
E	2	<E>

Note: At the end of first round, each node has learned all one-hop paths

Path Vector

- **Periodically repeat Steps 1 & 2**

In round 2, D receives:

From A:

To	Path
A	null
D	<D>

From C:

To	Path
C	null
D	<D>
E	<E>
B	<B>

From E:

To	Path
E	null
D	<D>
C	<C>

D updates its paths:

DST	Link	Path
D	End layer	null
A	1	<A>
C	3	<C>
E	2	<E>

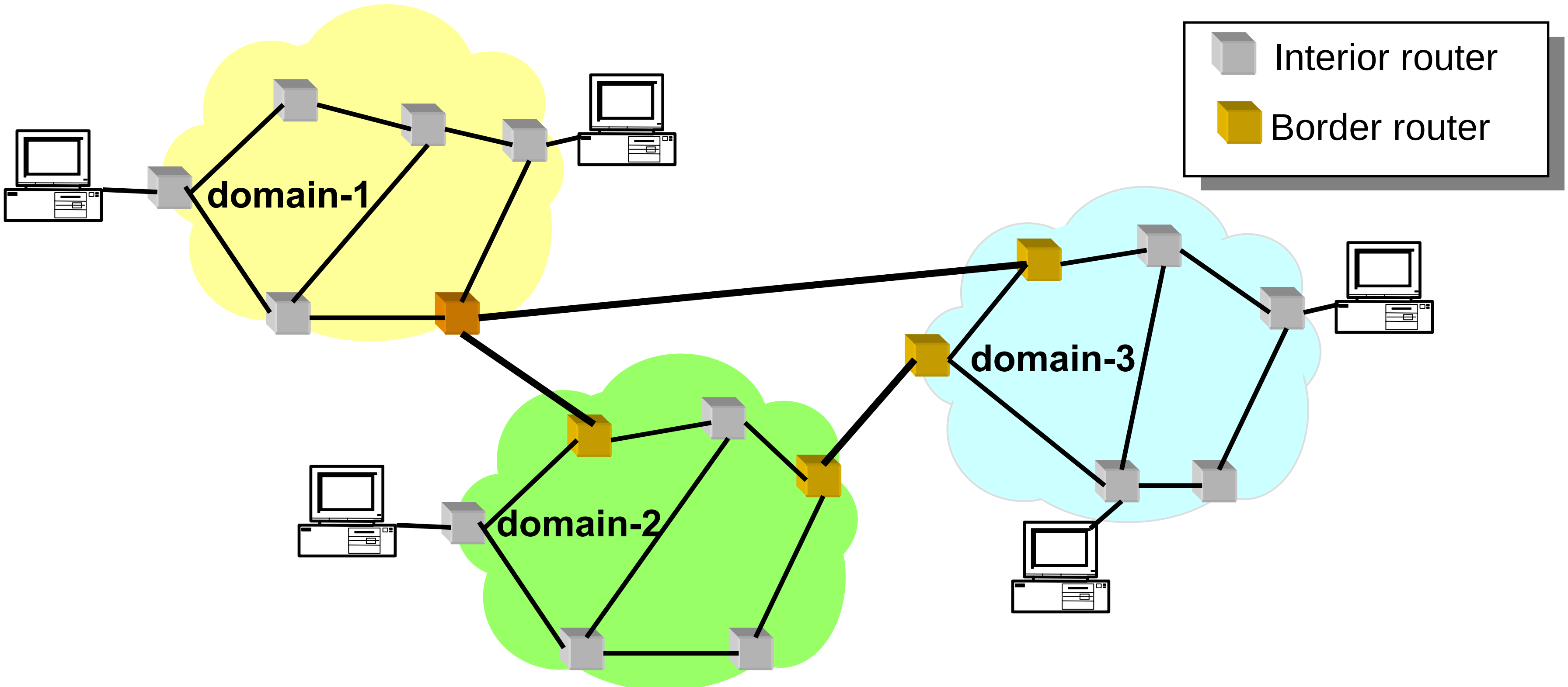
DST	Link	Path
D	End layer	null
A	1	<A>
C	3	<C>
E	2	<E>
B	3	<C, B>

Note: At the end of round 2, each node has learned all two-hop paths

Questions About Path Vector

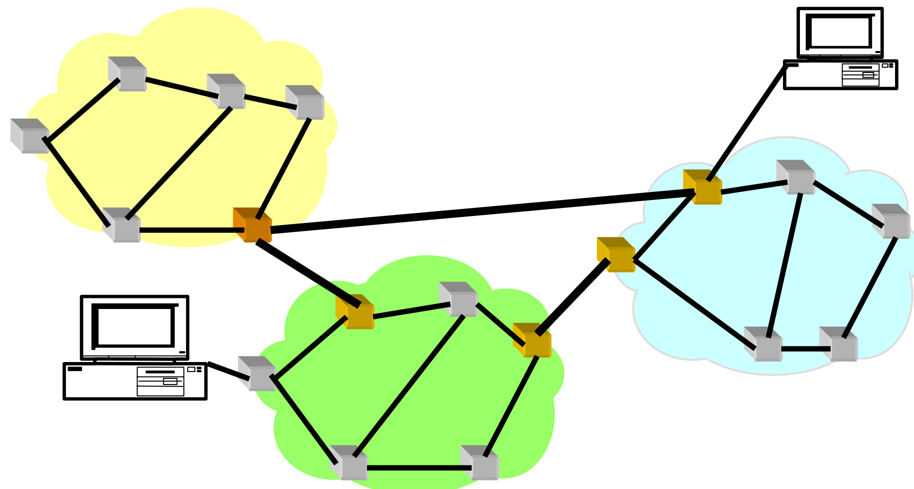
- How do we ensure no loops?
  - When a node updates its paths, it never accepts a path that has itself
- What happens when a node hears multiple paths to the same destination?
  - It picks the better path (e.g., the shorter number of hops)
- What happens if the graph changes?
  - Algorithm deals well with new links
  - To deal with links that go down, each router should discard any path that a neighbor stops advertising

Hierarchical Routing



- Internet: collection of domains/networks
- Inside a domain: Route over a graph of routers
- Between domains: Route over a graph of domains
- Address: concatenation of “Domain Id”, “Node Id”

Hierarchical Routing



Advantage

- scalable
  - Smaller tables
  - Smaller messages
- Delegation
  - Each domain can run its own routing protocol

Disadvantage

- Mobility is difficult
  - Address depends on geographic location
- Sup-optimal paths
  - E.g., in the figure, the shortest path between the two machines should traverse the yellow domain. But hierarchical routing goes directly between the green and blue domains, then finds the local destination → path traverses more routers.

Proof of BFS correctness

Shortest paths

- Define shortest-path distance δ(s,v) from s to v as the minimum number of edges in any path from vertex *s* to vertex *v*. If there is no path from *s* to *v*, δ(s,v) = ∞.
- A path of length δ(s,v) from *s* to *v* is said to be a shortest path.
- Claim: BFS computes shortest-path distances, i.e., after running BFS(G,s), dist[v] = δ(s,v).

  We'll need to prove a couple of properties about shortest paths before we can prove the claim above…

Proof techniques

- **Proof by contradiction**
  – Usually trying to prove "X is true for all Y".
  – Hypothesize that's not true: "There exists a Y for which is X is not true"
  – Reason forward from the hypothesis, arriving at a contradiction
  – Conclude that the hypothesis is false and hence the original statement must be true.
- **Proof by induction on a sequence of steps**
  – Create an induction hypothesis related to the statement you're trying to prove – it should be an "invariant" that will be maintained by each step of the process
  – (Basis) Show hypothesis is true after the first step
  – (Induction step) Assume hypothesis is true after step N, show it's true after step N+1
  – Conclude hypothesis is true for all N

Lemma 1

Lemma 1: Let G = (V,E) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then for any edge $(u,v) \in E$,

$$\delta(s,v) \leq \delta(s,u) + 1$$

Proof

if u is not reachable from s, $\delta(s,u) = \infty$, and the inequality holds.

If u is reachable from s, then so is v. The shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u,v), and thus the inequality holds.

Lemma 2

Lemma 2: Let *G* = (V,E) be a directed or undirected graph, and let *s* $\in$ V be an arbitrary vertex. Then after running BFS(G,s), for each v $\in$ V

$$\text{dist}[v] \geq \delta(s,v)$$

Proof: If *v* is not reachable from *s*, then during initialization dist[v] = ∞ = δ(s,v). Otherwise, we proceed by induction on the number of Q.put operations. Our inductive hypothesis is that dist[v] $\geq \delta$(s,v) for each v $\in$ V.

Basis: Just after Q.put(s), dist[s] = 0 = δ(s,s), and dist[v$\neq$s]=∞.

Induction step: consider a white vertex *v* discovered during search from vertex *u* (i.e., there's an edge (u,v) $\in$ E). Since *u* was just fetched from Q, the inductive hypothesis implies dist[u] $\geq \delta$(s,u). Thus just after Q.put(v)

dist[v] = dist[u] + 1	- by assignment in BFS
$\geq \delta$(s,u) + 1	- by inductive hypothesis
$\geq \delta$(s,v)	- by Lemma 1

Lemma 3

Lemma 3: Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$ where v_1 is the head of Q and v_r is the tail. Then

$$dist[v_r] \leq dist[v_1] + 1, \text{ and}$$
$$d[v_i] \leq d[v_i+1] \text{ for } i = 1, 2, \ldots, r-1$$

In words: there are at most two distinct distances for nodes in the queue at any given time and that nearer nodes have been enqueued before nodes that are further away.

Proof: by induction on the number of queue operations.

Basis: initially the queue contains only *s* and the lemma is trivially true.

Induction step: prove lemma after both dequeuing and enqueuing a vertex.

Proof of Lemma 3 (cont'd.)

After dequeuing: After the head v_1 has been dequeued, v_2 becomes the new head. By inductive hypothesis $dist[v_1] \le dist[v_2]$ and $dist[v_r] \le dist[v_1]+1$. Putting the two facts together, $dist[v_r] \le dist[v_2] + 1$ and the remaining inequalities are unaffected. Thus the lemma holds with v_2 as the head.

After enqueuing: We perform enqueuing operations while searching the adjacency list of a node *u* that was just dequeued. By the inductive hypothesis, we know the relationships between dist[u] and the current contents of the queue: $dist[u] \le dist[v_1]$ and $dist[v_r] \le dist[u]+1$.

When we enqueue a white neighbor of *u* it becomes v_{r+1}. BFS sets $dist[v_{r+1}] = dist[u]+1$, and so $dist[v_{r+1}] \le dist[v_1]+1$ and $dist[v_r] \le dist[v_{r+1}]$. The remaining inequalities are unaffected and the thus the lemma holds with v_{r+1} as the tail.

Corollary 4: If v_i is enqueued before v_j, then $dist[v_i] \le dist[v_j]$

Theorem: Correctness of BFS

Theorem 5: Let G = (V,E) be a directed or undirected graph, and let *s* ∈ V be an arbitrary vertex. Then after running BFS(G,s)

- BFS discovers every vertex *v* ∈ V that is reachable from *s*
- Upon termination dist[v] = δ(s,v)
- For every reachable *v* ≠ *s,* one of the shortest paths from *s* to *v* is a shortest path from *s* to parent[v] followed by the edge (parent[v],v).

Proof: Assume, for the purpose of contradiction, that there is some vertex *v* with the minimum δ(s,v) that is assigned a distance that is not equal to δ(s,v). By Lemma 2, dist[v] > δ(s,v). Vertex *v* must be reachable from *s*, for if it is not, δ(s,v) = ∞ ≥ dist[v]. Let *u* be the vertex immediately before *v* on a shortest path from *s* to *v*, so that δ(s,v) = δ(s,u) + 1. Because δ(s,u) < δ(s,v), and because of how we chose *v*, we have dist[u] = δ(s,u). Putting this together:

$$\text{dist}[v] > \delta(s,v) = \delta(s,u) + 1 = \text{dist}[u]+1 \qquad [5.1]$$

Proof of Theorem 5 (cont'd.)

Now consider the time when *u* is dequeued during BFS. At that time *v* is either white, gray or black.

v is white: then BFS will set dist[v] = dist[u]+1, contradicting [5.1]

v is black: then *v* has already been removed from the queue and, by Corollary 4, dist[v] ≤ dist[u], contradicting [5.1]

v is gray: then it was painted gray upon dequeuing some other node *w* which dequeued before u. BFS has set dist[v] = dist[w]+1. But by Corollary 4, dist[w] ≤ dist[u], so we have dist[v] ≤ dist[u]+1, again contradicting [5.1].

Thus we conclude that dist[v] = δ(s,v). We must have processed all discoverable nodes, otherwise they'd have infinite dist values. Finally if parent[v] = *u*, then dist[v] = dist[u]+1. So a shortest path to *v* is obtained by a shortest path to *u* followed by the edge (parent[v],v).