Routing

Routing means guiding a packet in a network to its destination.

A routing table at node u stores for each $v \neq u$ a neighbor w of u:
Each packet with destination v that arrives at u is passed on to w.

Network as a graph

- Network is a graph : G = (V,E)
- Each vertex/node is a computer/process
- Each edge is communication link between 2 nodes
- Every node has a Unique identifier known to itself.
  - Often used 1, 2, 3, … n
- Every node knows its neighbors – the nodes it can reach directly without needing other nodes to route
  - Edges incident on the vertex
  - For example, in LAN or WLAN, through listening to the broadcast medium
  - Or by explicitly asking: Everyone that receives this message, please report back

Network as a graph

- Distance/cost between nodes p and q in the network
  - Number of edges on the shortest path between p and q (when all edges are same: unweighted)

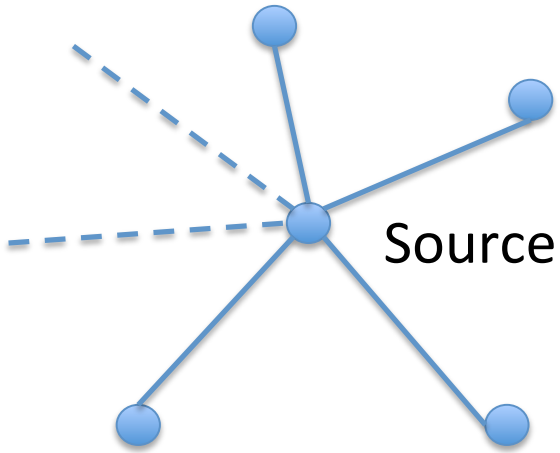
- Sometimes, edges can be weighted
  - Each edge e = (a,b) has a weight *w(e)*
  - w(e) is the cost of using the communication link e (may be length e)
  - Distance/cost between p and q is total weight of edges on the path from p to q with least weight

Network as a graph

- Diameter
  - The maximum distance between 2 nodes in the network
- Radius
  - Half the diameter
- Spanning tree of a graph:
  - A subgraph which is a tree, and reaches all nodes of the graph
  - How many edges does a spanning tree have?

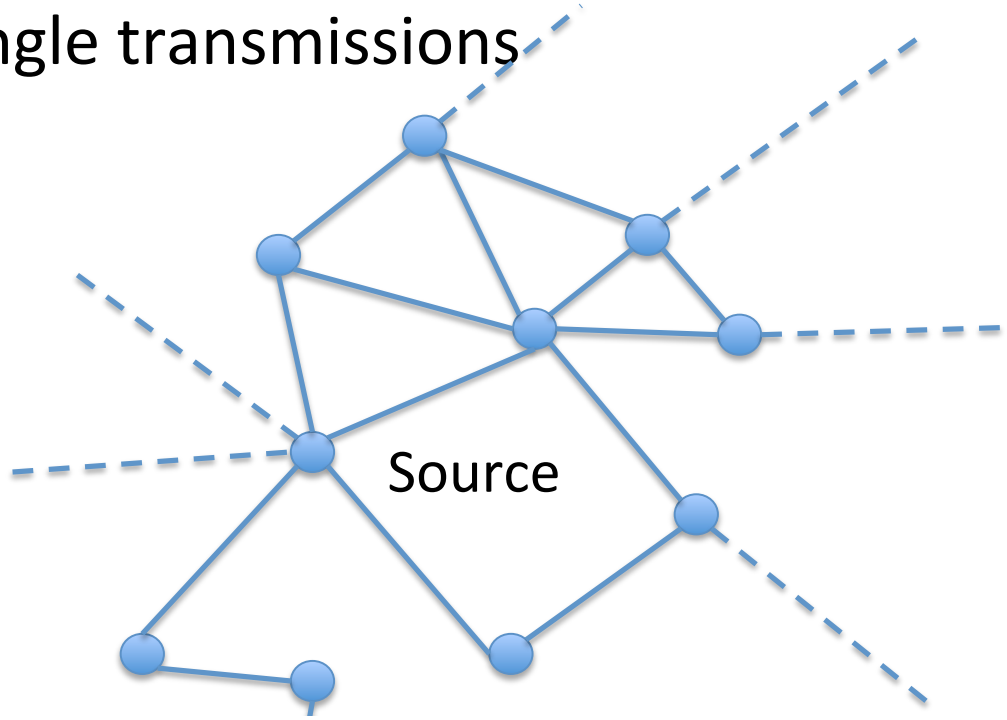
Global Message broadcast

- Message must reach *all nodes in the network*
  - Different from broadcast transmission in LAN
  - All nodes in a large network cannot be reached with single transmissions



Global Message broadcast

- Message must reach *all nodes in the network*
  - Different from broadcast transmission in LAN
  - All nodes in a large network cannot be reached with single transmissions



Flooding for Broadcast

- The source sends a *Flood* message to all neighbors

- The message has
  - *Flood* type
  - *Unique id: (source id, message seq)*
  - *Data*

Flooding for Broadcast

- Every node p that receives a flood message m, does the following:
  - *If m.id was seen before, discard m*
  - *Otherwise, Add m.id to list of previously seen messages and send m to all neighbors of p*

Flooding for broadcast

- Storage
  - Each node needs to store a list of flood ids seen before
  - If a protocol requires x floods, then each node must store x ids

Flooding for broadcast

- Storage
  - Each node needs to store a list of flood ids seen before
  - If a protocol requires x floods, then each node must store x ids
  - Requires $\Omega(x)$ storage
  - (Actual storage depends on size of *m.id*)

Assumptions

- We are assuming:
  - Nodes are working in synchronous *communication rounds*
  - Messages from all neighbors arrive at the same time, and processed together
  - In each round, each node can successfully send 1 message to all its neighbors
  - Any necessary computation can be completed before the next round

Communication complexity

- The the message/communication complexity is:
  - O(|E|)
  - E is set of communication edges in the network.
  - |E| is the number of communication edges

- Since each node sends the message to each neighbor exactly once
  - The actual number of messages is 2|E|

Reducing Communication complexity (slightly)

- Node p need not send message m to any node from which it has already received m
  - Needs to keep track of which nodes have sent the message
  - Saves some messages
  - Does not change asymptotic complexity

Time complexity

- The number of rounds needed to reach all nodes: *diameter of G*

BFS Tree

- Breadth first search tree
  - Every node has a *parent* pointer
  - And zero or more child pointers

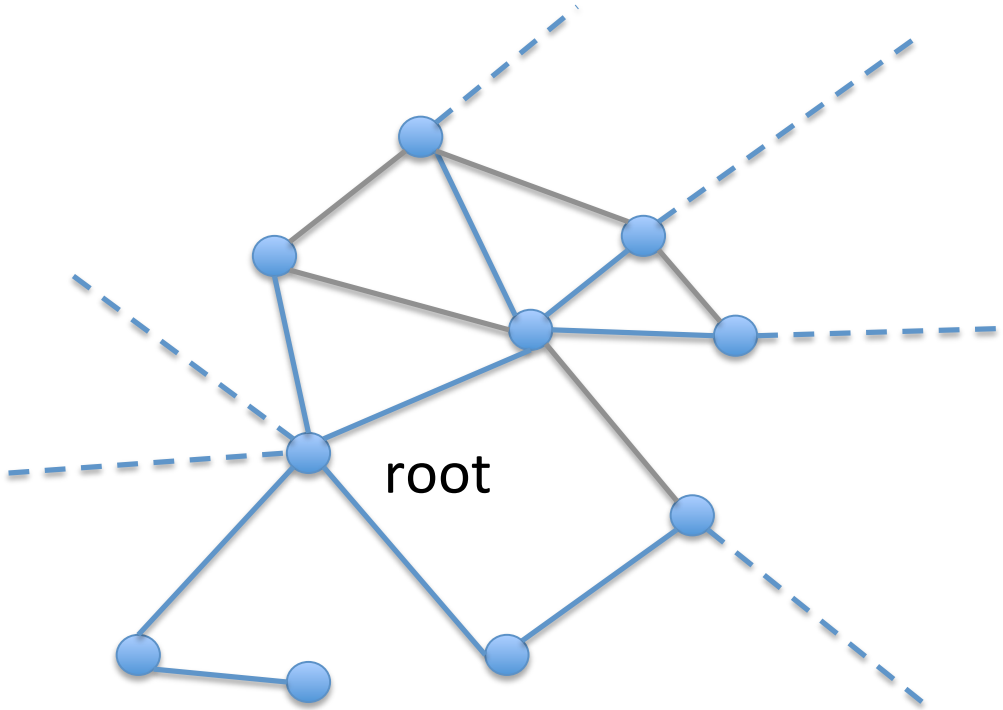
  - BFS Tree construction algorithm sets these pointers

BFS Tree Construction algorithm

- Breadth first search tree
  - The *root(source)* node decides to construct a tree
  - Uses flooding to construct a tree
  - Every node p on getting the message forwards to all neighbors
  - Additionally, every node p stores *parent* pointer: node from which it first received the message
    - If multiple neighbors had first sent p the message in the same round, choose *parent* arbitrarily. E.g. node with smallest id
  - p informs its parent of the selection
    - Parent creates a child pointer to p

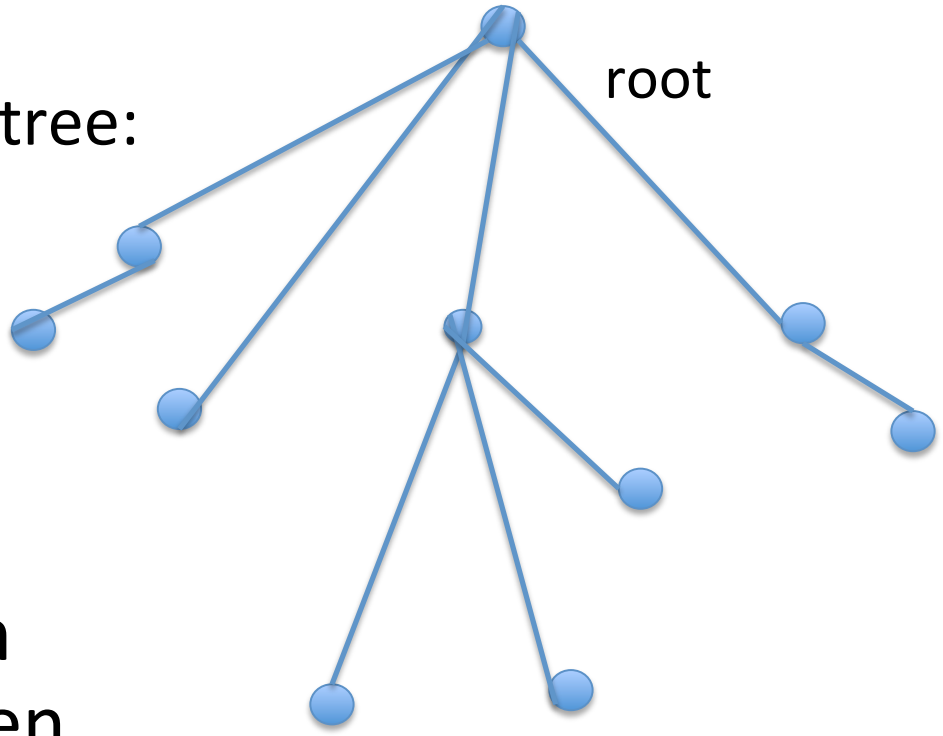
Time & message complexity

- Asymptotically Same as Flooding



Tree based broadcast

- Send message to all nodes using tree
  - BFS tree is a *spanning* tree: connects all nodes
- Flooding on the tree
- Receive message from parent, send to children



Tree based broadcast

- Simpler than flooding: send message to all children

- Communication: Number of edges in spanning tree: n-1

Aggregation: Find the sum of values at all nodes

- With BFS tree aka **Convergecast**

- Start from *leaf* nodes
  - Nodes without children
  - Send the value to parent
- Every other node:
  - Wait for all children to report
  - Sum values from children + own value
  - Send to parent

Aggregation

- With Tree aka **Convergecast**

- Once tree is built, any node can use for broadcast
  - Just flood on the tree

- Any node can use for convergecast
  - First flood a message on the tree requesting data
  - Nodes store parent pointer
  - Then receive data
- Fault tolerance not very good
  - If a node fails, the messages in the subtree will be lost
  - Will need to rebuild the tree for future operations

Shortest paths

- BFS tree rooted at node p contains shortest paths to p from all nodes in the network

- From any node q, follow *parent* pointers to p
  - Gives shortest path

BFS trees can be used for routing

- From each node, create a separate BFS tree
- Each node stores a parent pointer corresponding to each BFS tree
- Acts as routing table
- O(n*|E|) message complexity

Exercise:

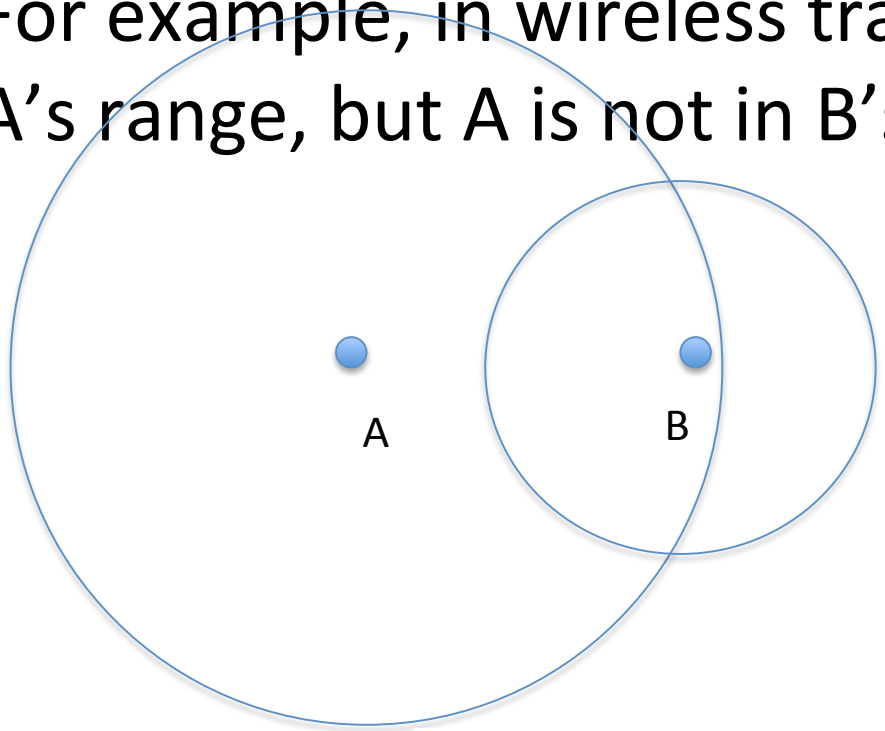
find the diameter of a network

Directed graphs

- We have considered only undirected graphs
- Communication may be directed
- When A can send message to B, but B cannot send message to A

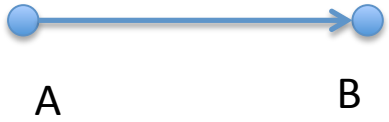
Directed graphs

- When A can send message to B, but B cannot send message to A
- For example, in wireless transmission, if B is in A's range, but A is not in B's range



Directed graphs

- When A can send message to B, but B cannot send message to A
- Or if protocol or technology limitations prevent B from communicating with A



Directed graphs

- Protocols more complex
- Needs more messages

Chandy-Misra algorithm (= Bellman-Ford in a distributed setting)

Consider a weighted network, with weights $\omega_{vw} > 0$.

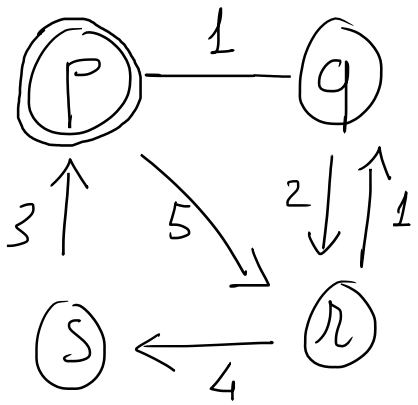
A *centralized* algorithm to compute all shortest paths to initiator u_0.

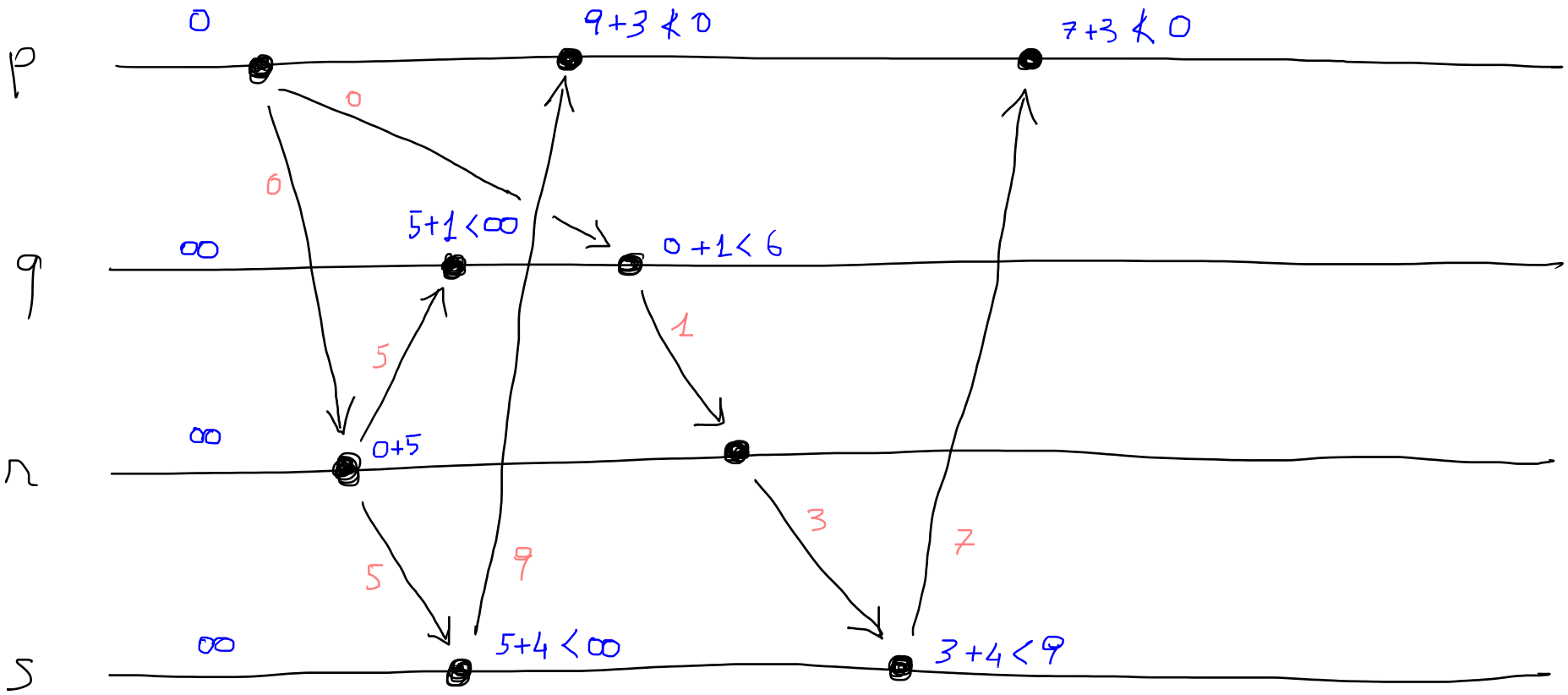
Initially, $dist_{u_0}(u_0) = 0$, $dist_v(u_0) = \infty$ if $v \neq u_0$, and $parent_v(u_0) = \perp$.

u_0 sends the message $\langle 0 \rangle$ to its neighbors.

Let node v receives $\langle d \rangle$ from neighbor w. If $d + \omega_{vw} < dist_v(u_0)$, then:

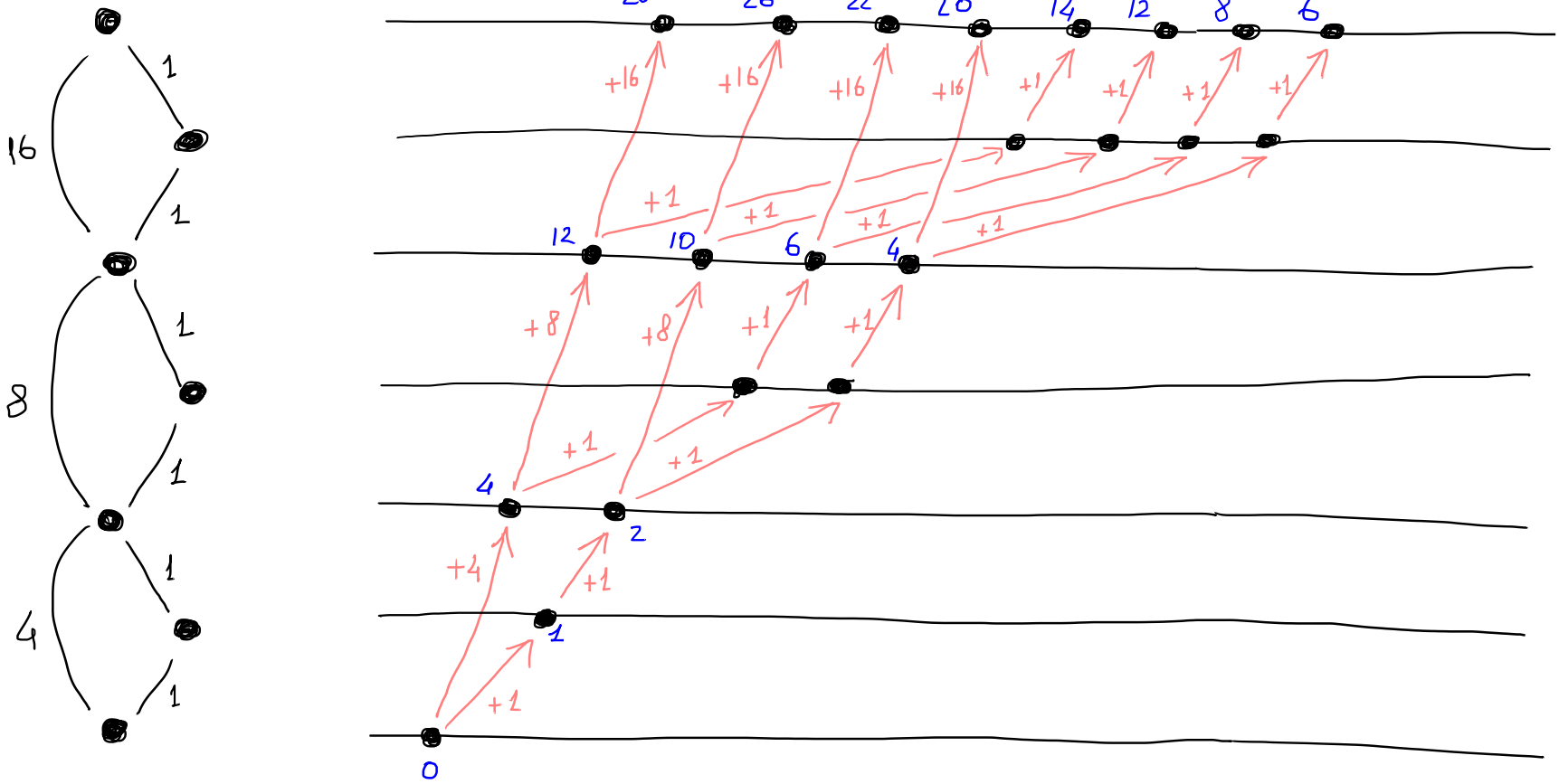
- $dist_v(u_0) \leftarrow d + \omega_{vw}$ and $parent_v(u_0) \leftarrow w$
- v sends $\langle dist_v(u_0) \rangle$ to its neighbors (except w)


p
q
r
s
1
3
5
2
1
4


p
q
r
s
0
9+3 ≮ 0
7+3 ≮ 0
0
6
∞
5+1 < ∞
0+1 < 6
1
5
∞
0+5
5
9
3
7
∞
5+4 < ∞
3+4 < 9

Chandy-Misra algorithm - Complexity

Worst-case message complexity: Exponential


16
8
4
1
1
1
1
1
1
1
28
26
22
20
14
12
8
6
12
10
6
4
4
2
1
0
+16
+16
+16
+16
+1
+1
+1
+1
+1
+1
+1
+1
+8
+8
+1
+1
+1
+1
+4
+1
+1

Chandy-Misra algorithm - Complexity

Worst-case message complexity: Exponential

Worst-case message complexity for minimum-hop: $O(N^2 \cdot E)$

For each root, the algorithm requires at most $O(N \cdot E)$ messages.

Question:

how to detect termination of Chandy-Misra?

Exercise 2:

find minimum-cost bipartite matching

Toueg's algorithm (= Floyd-Warshall in the distributed setting)

Computes for each pair u, v a shortest path from u to v.

$d^S(u, v)$, with S a set of nodes, denotes the length of a shortest path from u to v with all *intermediate* nodes in S.

$$
\begin{aligned}
d^S(u,u) &= 0 \\
d^\emptyset(u,v) &= \omega_{uv} \quad \text{if } u \neq v \text{ and } uv \in E \\
d^\emptyset(u,v) &= \infty \quad \text{if } u \neq v \text{ and } uv \notin E \\
d^{S\cup\{w\}}(u,v) &= \qquad\qquad\qquad\qquad \text{if } w \notin S
\end{aligned}
$$

When S contains all nodes, d^S is the standard distance function.

Toueg's algorithm (= Floyd-Warshall in the distributed setting)

Computes for each pair u, v a shortest path from u to v.

$d^S(u, v)$, with S a set of nodes, denotes the length of a shortest path from u to v with all *intermediate* nodes in S.

$$
\begin{aligned}
d^S(u,u) &= 0 \\
d^\emptyset(u,v) &= \omega_{uv} \quad \text{if } u \neq v \text{ and } uv \in E \\
d^\emptyset(u,v) &= \infty \quad \text{if } u \neq v \text{ and } uv \notin E \\
d^{S\cup\{w\}}(u,v) &= \qquad\qquad\qquad\qquad\qquad\qquad \text{if } w \notin S
\end{aligned}
$$

When S contains all nodes, d^S is the standard distance function.

Toueg's algorithm (= Floyd-Warshall in the distributed setting)

Computes for each pair u, v a shortest path from u to v.

$d^S(u, v)$, with S a set of nodes, denotes the length of a shortest path from u to v with all *intermediate* nodes in S.

$$
\begin{aligned}
d^S(u,u) &= 0 \\
d^\emptyset(u,v) &= \omega_{uv} \quad \text{if } u \neq v \text{ and } uv \in E \\
d^\emptyset(u,v) &= \infty \quad \text{if } u \neq v \text{ and } uv \notin E \\
d^{S\cup\{w\}}(u,v) &= \min\{\, d^S(u,v),\ d^S(u,w) + d^S(w,v) \,\} \quad \text{if } w \notin S
\end{aligned}
$$

When S contains all nodes, d^S is the standard distance function.

Toueg's algorithm (= Floyd-Warshall in the distributed setting)

Computes for each pair u, v a shortest path from u to v.

$d^S(u, v)$, with S a set of nodes, denotes the length of a shortest path from u to v with all *intermediate* nodes in S.

$$
\begin{aligned}
d^S(u,u) &= 0 \\
d^\emptyset(u,v) &= \omega_{uv} \quad \text{if } u \neq v \text{ and } uv \in E \\
d^\emptyset(u,v) &= \infty \quad \text{if } u \neq v \text{ and } uv \notin E \\
d^{S\cup\{w\}}(u,v) &= \min\{\, d^S(u,v),\ d^S(u,w) + d^S(w,v) \,\} \quad \text{if } w \notin S
\end{aligned}
$$

When S contains all nodes, d^S is the standard distance function.

Floyd-Warshall algorithm

We first discuss a *uniprocessor* algorithm.

Initially, $S = \emptyset$; the first three equations define $d^{\emptyset}$.

While S doesn't contain all nodes, a pivot $w \notin S$ is selected:

- $d^{S \cup \{w\}}$ is computed from d^S using the fourth equation.
- w is added to S.

When S contains all nodes, d^S is the standard distance function.

Time complexity: $\Theta(N^3)$

Floyd-Warshall algorithm

We first discuss a *uniprocessor* algorithm.

Initially, $S = \emptyset$; the first three equations define $d^{\emptyset}$.

While S doesn't contain all nodes, a pivot $w \notin S$ is selected:

- $d^{S \cup \{w\}}$ is computed from d^S using the fourth equation.
- w is added to S.

When S contains all nodes, d^S is the standard distance function.

Time complexity: $\Theta(N^3)$

Floyd-Warshall algorithm

We first discuss a *uniprocessor* algorithm.

Initially, $S = \emptyset$; the first three equations define $d^{\emptyset}$.

While S doesn't contain all nodes, a pivot $w \notin S$ is selected:

- $d^{S \cup \{w\}}$ is computed from d^S using the fourth equation.
- w is added to S.

When S contains all nodes, d^S is the standard distance function.

Time complexity: $\Theta(N^3)$

Floyd-Warshall algorithm

We first discuss a *uniprocessor* algorithm.

Initially, $S = \emptyset$; the first three equations define $d^{\emptyset}$.

While S doesn't contain all nodes, a pivot $w \notin S$ is selected:

- $d^{S \cup \{w\}}$ is computed from d^S using the fourth equation.
- w is added to S.

When S contains all nodes, d^S is the standard distance function.

Time complexity: $\Theta(N^3)$

Question

Which complications arise when the Floyd-Warshall algorithm is turned into a distributed algorithm ?

Question

Which complications arise when the Floyd-Warshall algorithm is turned into a distributed algorithm ?

- All nodes must pick the pivots in the same order.

Question

Which complications arise when the Floyd-Warshall algorithm is turned into a distributed algorithm ?

- All nodes must pick the pivots in the same order.
- Each round, nodes need the distance values of the pivot to compute their own routing table.

Toueg's algorithm

Assumption: Each node knows the id's of all nodes.

(Because pivots must be picked uniformly at all nodes.)

Initially, at each node u:

- $S_u = \emptyset$;
- $dist_u(u) = 0$ and $parent_u(u) = \perp$;
- for each $v \neq u$, either
  $dist_u(v) = \omega_{uv}$ and $parent_u(v) = v$ if there is an edge uv, or
  $dist_u(v) = \infty$ and $parent_u(v) = \perp$ otherwise.

Toueg's algorithm

Assumption: Each node knows the id's of all nodes.

(Because pivots must be picked uniformly at all nodes.)

Initially, at each node u:

- $S_u = \emptyset$;
- $dist_u(u) = 0$ and $parent_u(u) = \perp$;
- for each $v \neq u$, either
  $dist_u(v) = \omega_{uv}$ and $parent_u(v) = v$ if there is an edge uv, or
  $dist_u(v) = \infty$ and $parent_u(v) = \perp$ otherwise.

Toueg's algorithm

At the w-pivot round, w broadcasts its values $dist_w(v)$, for all nodes v.

If $parent_u(w) = \perp$ for a node $u \neq w$ at the w-pivot round,
then $dist_u(w) = \infty$, so $dist_u(w) + dist_w(v) \geq dist_u(v)$ for all nodes v.

Hence the sink tree toward w can be used to broadcast $dist_w$.

If u is in the sink tree toward w, it sends $\langle \textbf{request}, w \rangle$ to $parent_u(w)$,
to let it pass on $dist_w$.

If u isn't in the sink tree toward w, it proceeds to the next pivot round,
with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm

At the w-pivot round, w broadcasts its values $dist_w(v)$, for all nodes v.

If $parent_u(w) = \perp$ for a node $u \neq w$ at the w-pivot round,
then $dist_u(w) = \infty$, so $dist_u(w) + dist_w(v) \geq dist_u(v)$ for all nodes v.

Hence the sink tree toward w can be used to broadcast $dist_w$.

If u is in the sink tree toward w, it sends $\langle \textbf{request}, w \rangle$ to $parent_u(w)$,
to let it pass on $dist_w$.

If u isn't in the sink tree toward w, it proceeds to the next pivot round,
with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm

At the w-pivot round, w broadcasts its values $dist_w(v)$, for all nodes v.

If $parent_u(w) = \perp$ for a node $u \neq w$ at the w-pivot round,
then $dist_u(w) = \infty$, so $dist_u(w) + dist_w(v) \geq dist_u(v)$ for all nodes v.

Hence the sink tree toward w can be used to broadcast $dist_w$.

If u is in the sink tree toward w, it sends $\langle \textbf{request}, w \rangle$ to $parent_u(w)$,
to let it pass on $dist_w$.

If u isn't in the sink tree toward w, it proceeds to the next pivot round,
with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm

Consider any node u in the sink tree toward w.

If $u \neq w$, it waits for the values $dist_w(v)$ from $parent_u(w)$.

u forwards the values $dist_w(v)$ to neighbors that send $\langle \textbf{request}, w \rangle$ to u.

If $u \neq w$, it checks for each node v whether

$$dist_u(w) + dist_w(v) < dist_u(v).$$

If so, $dist_u(v) \leftarrow dist_u(w) + dist_w(u)$ and $parent_u(v) \leftarrow parent_u(w)$.

Finally, u proceeds to the next pivot round, with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm

Consider any node u in the sink tree toward w.

If $u \neq w$, it waits for the values $dist_w(v)$ from $parent_u(w)$.

u forwards the values $dist_w(v)$ to neighbors that send $\langle \textbf{request}, w \rangle$ to u.

If $u \neq w$, it checks for each node v whether

$$dist_u(w) + dist_w(v) < dist_u(v).$$

If so, $dist_u(v) \leftarrow dist_u(w) + dist_w(u)$ and $parent_u(v) \leftarrow parent_u(w)$.

Finally, u proceeds to the next pivot round, with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm

Consider any node u in the sink tree toward w.

If $u \neq w$, it waits for the values $dist_w(v)$ from $parent_u(w)$.

u forwards the values $dist_w(v)$ to neighbors that send $\langle \textbf{request}, w \rangle$ to u.

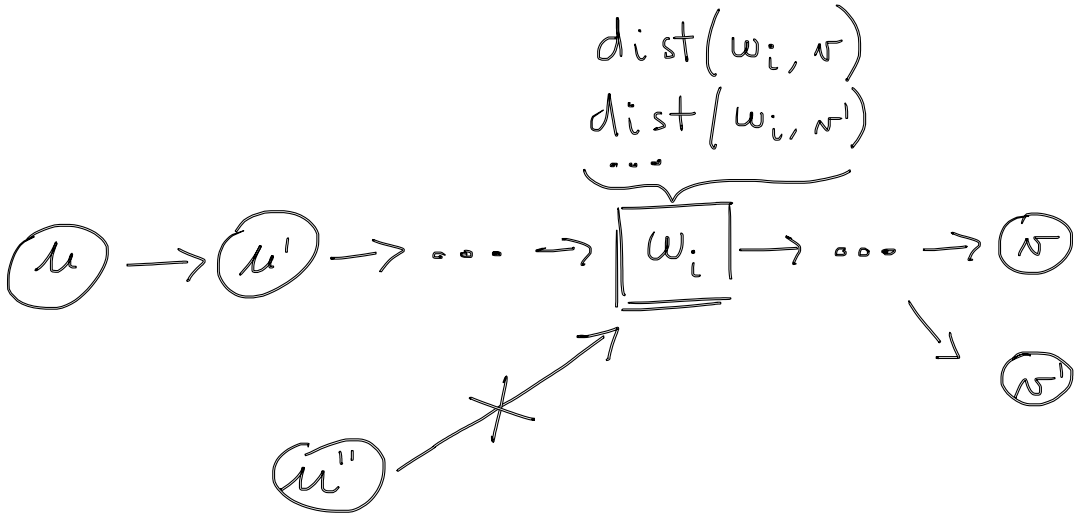
If $u \neq w$, it checks for each node v whether

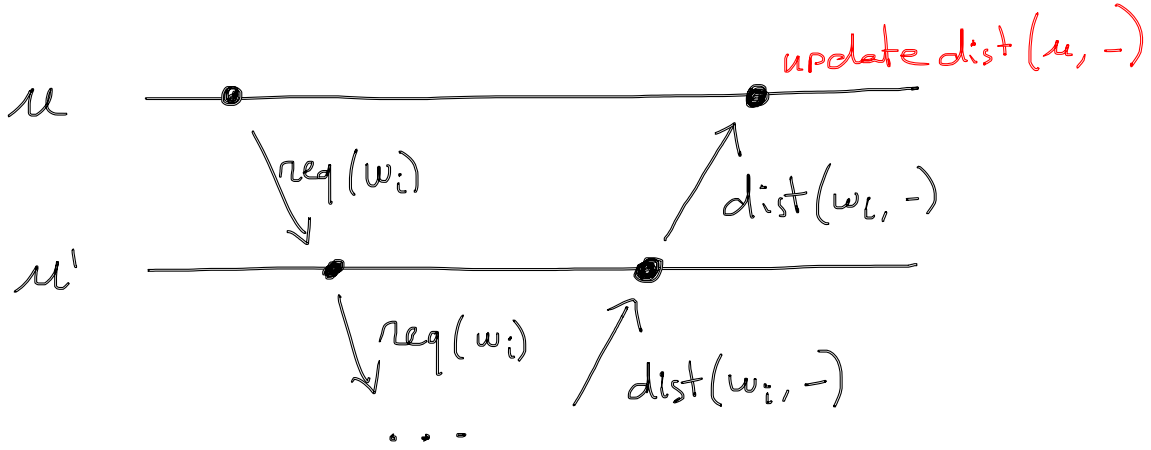
$$dist_u(w) + dist_w(v) < dist_u(v).$$

If so, $dist_u(v) \leftarrow dist_u(w) + dist_w(u)$ and $parent_u(v) \leftarrow parent_u(w)$.

Finally, u proceeds to the next pivot round, with $S_u \leftarrow S_u \cup \{w\}$.

Round with pivot w

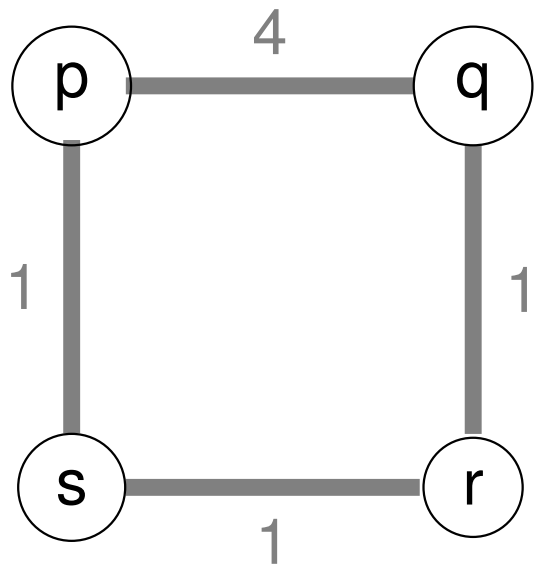




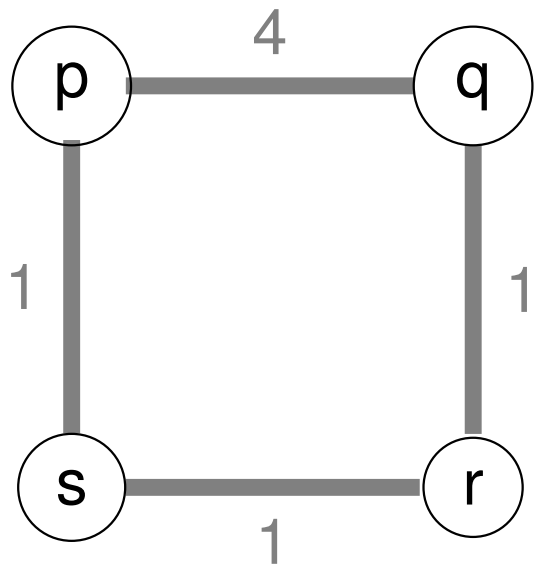
u'' will not send $req(w_i)$ since $dist(u'', w_i) = \infty$

Toueg's Algorithm - Example

$$\text{pivot } p \;\Big|\; \begin{array}{ll} dist_s(q) \leftarrow 5 & dist_q(s) \leftarrow 5 \\ parent_s(q) \leftarrow p & parent_q(s) \leftarrow p \end{array}$$

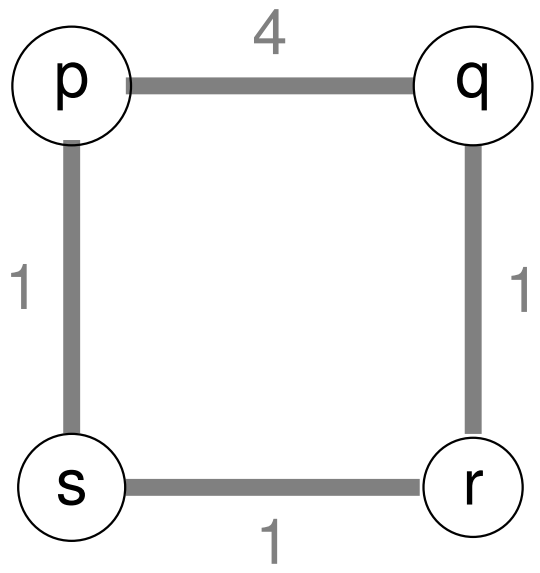


Toueg's Algorithm - Example



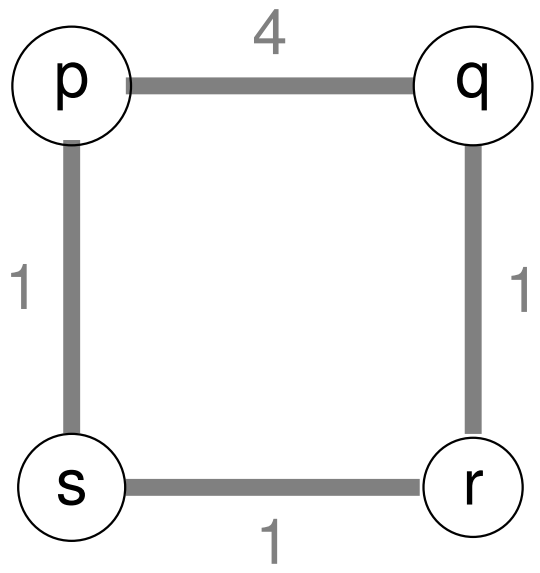
pivot p	$dist_s(q) \leftarrow 5$ $parent_s(q) \leftarrow p$	$dist_q(s) \leftarrow 5$ $parent_q(s) \leftarrow p$
pivot q	$dist_p(r) \leftarrow 5$ $parent_p(r) \leftarrow q$	$dist_r(p) \leftarrow 5$ $parent_r(p) \leftarrow q$

Toueg's Algorithm - Example



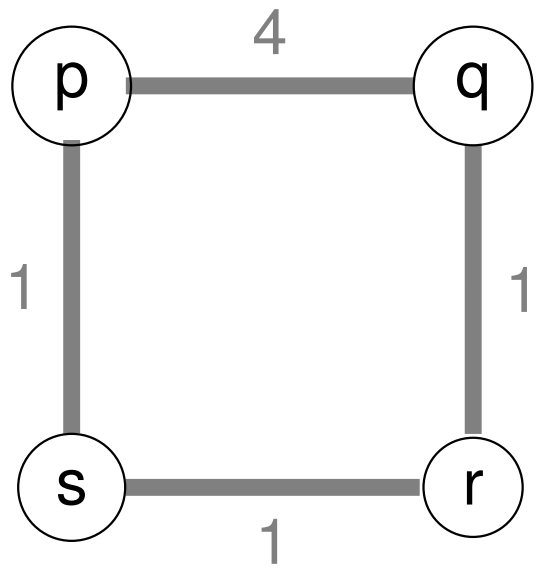
pivot p	$dist_s(q) \leftarrow 5$ $parent_s(q) \leftarrow p$	$dist_q(s) \leftarrow 5$ $parent_q(s) \leftarrow p$
pivot q	$dist_p(r) \leftarrow 5$ $parent_p(r) \leftarrow q$	$dist_r(p) \leftarrow 5$ $parent_r(p) \leftarrow q$
pivot r	$dist_s(q) \leftarrow 2$ $parent_s(q) \leftarrow r$	$dist_q(s) \leftarrow 2$ $parent_q(s) \leftarrow r$

Toueg's Algorithm - Example

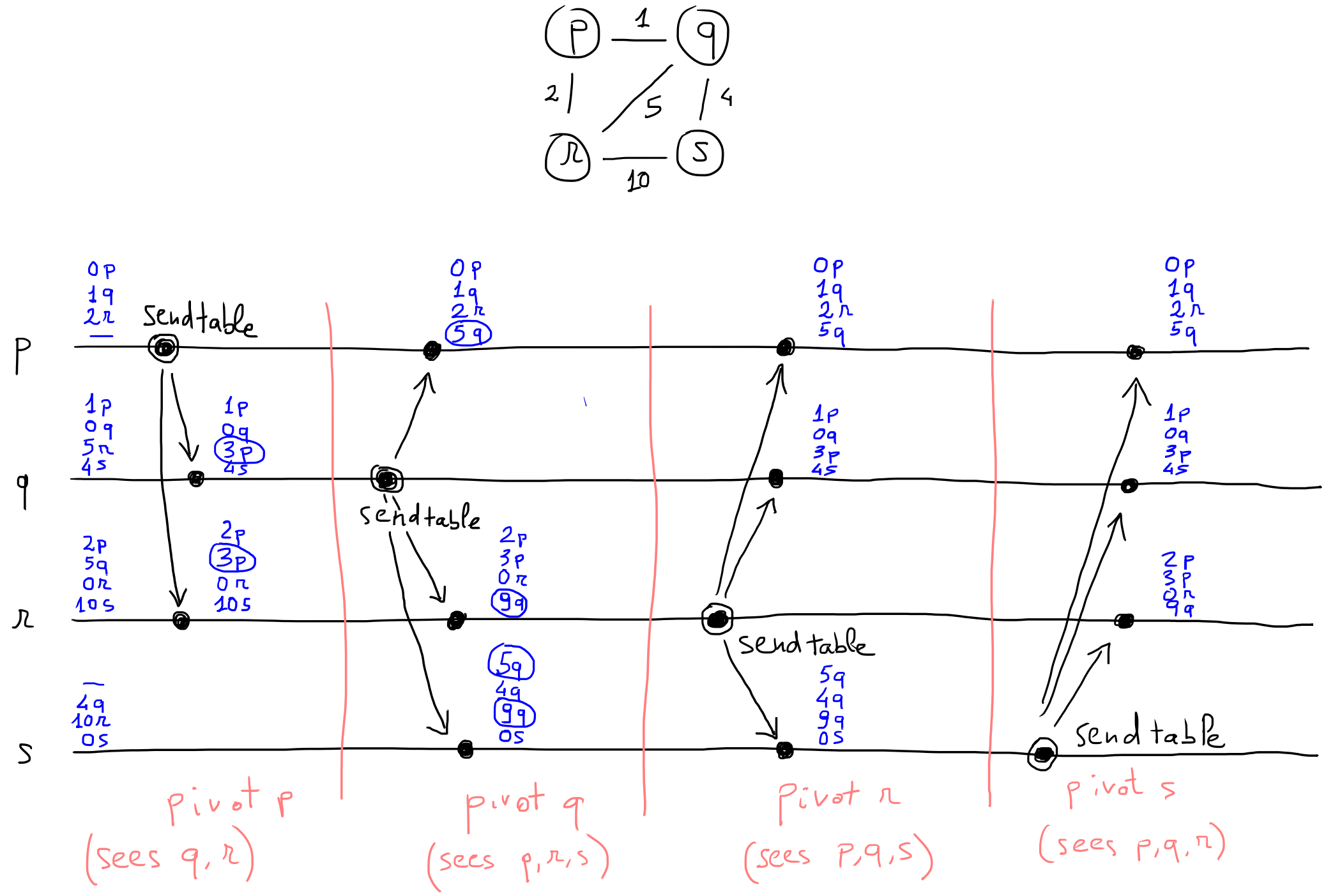


pivot p	$dist_s(q) \leftarrow 5$ $parent_s(q) \leftarrow p$	$dist_q(s) \leftarrow 5$ $parent_q(s) \leftarrow p$
pivot q	$dist_p(r) \leftarrow 5$ $parent_p(r) \leftarrow q$	$dist_r(p) \leftarrow 5$ $parent_r(p) \leftarrow q$
pivot r	$dist_s(q) \leftarrow 2$ $parent_s(q) \leftarrow r$	$dist_q(s) \leftarrow 2$ $parent_q(s) \leftarrow r$
pivot s	$dist_p(r) \leftarrow 2$ $parent_p(r) \leftarrow s$	$dist_r(p) \leftarrow 2$ $parent_r(p) \leftarrow s$

Toueg's Algorithm - Example



pivot p	$dist_s(q) \leftarrow 5$ $parent_s(q) \leftarrow p$	$dist_q(s) \leftarrow 5$ $parent_q(s) \leftarrow p$
pivot q	$dist_p(r) \leftarrow 5$ $parent_p(r) \leftarrow q$	$dist_r(p) \leftarrow 5$ $parent_r(p) \leftarrow q$
pivot r	$dist_s(q) \leftarrow 2$ $parent_s(q) \leftarrow r$	$dist_q(s) \leftarrow 2$ $parent_q(s) \leftarrow r$
pivot s	$dist_p(r) \leftarrow 2$ $parent_p(r) \leftarrow s$ $dist_p(q) \leftarrow 3$ $parent_p(q) \leftarrow s$	$dist_r(p) \leftarrow 2$ $parent_r(p) \leftarrow s$ $dist_q(p) \leftarrow 3$ $parent_q(p) \leftarrow r$


p
q
r
s
1
2
5
4
10
sendtable
sendtable
sendtable
sendtable
pivot p
(sees q, r)
pivot q
(sees p, r, s)
pivot r
(sees p, q, s)
pivot s
(sees p, q, r)

Toueg's algorithm - Complexity + drawbacks

Message complexity: $O(N^2)$

There are N pivot rounds, each taking at most $O(N)$ messages.

(but each message has size O(N)!)

Drawbacks:

- Uniform selection of pivots requires that all nodes know the nodes in the network in advance.
- Global broadcast of $dist_w$ at the w-pivot round causes a high bit complexity.
- Not robust with respect to topology changes.

Toueg's algorithm - Optimization

Let $parent_u(w) = x$ with $x \neq w$ at the start of the w-pivot round.

If $dist_x(v)$ doesn't change in this round, then neither does $dist_u(v)$ (for any v).

Upon reception of $dist_w$, x first updates $dist_x$, and only forwards values $dist_w(v)$ for which $dist_x(v)$ has changed.

Minimum spanning trees

Consider an undirected, weighted network.

We assume that different edges have different weights.

(Or weighted edges can be totally ordered by also taking into account the id's of endpoints of an edge, and using a lexicographical order.)

In a minimum spanning tree, the sum of the weights of the edges in the spanning tree is minimal.

Minimum spanning trees

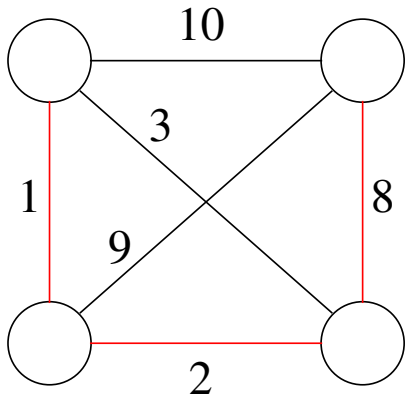
Consider an undirected, weighted network.

We assume that different edges have different weights.

(Or weighted edges can be totally ordered by also taking into account the id's of endpoints of an edge, and using a lexicographical order.)

In a minimum spanning tree, the sum of the weights of the edges in the spanning tree is minimal.

Example:



Fragments

Lemma: Let F be a fragment
(i.e., a connected subgraph of the minimum spanning tree M).

Let e be the *lowest-weight* outgoing edge of F
(i.e., e has exactly one endpoint in F).

Then e is in M.

Proof: Suppose not.

Then $M \cup \{e\}$ has a cycle,
containing e and another outgoing edge f of F.

Replacing f by e in M gives a spanning tree
with a smaller sum of weights of edges.

Fragments

Lemma: Let F be a fragment
(i.e., a connected subgraph of the minimum spanning tree M).

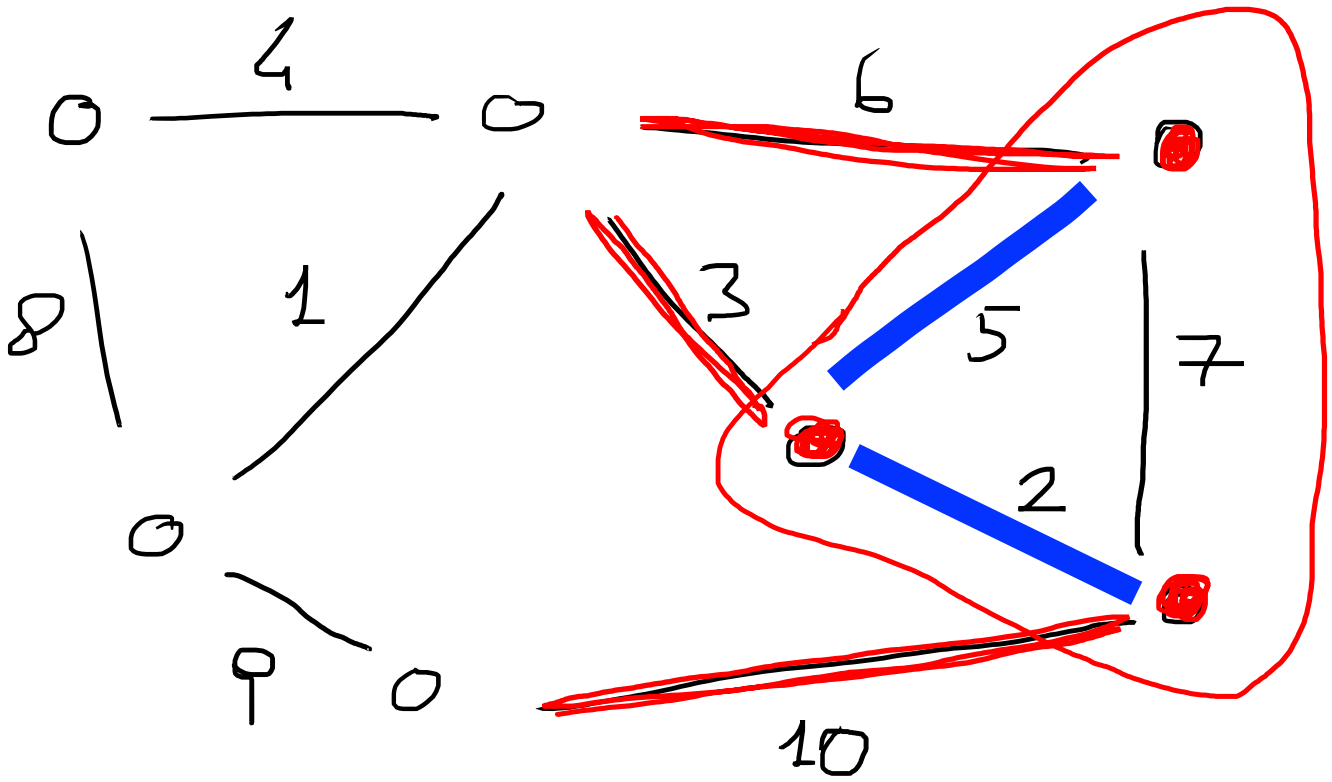
Let e be the *lowest-weight* outgoing edge of F
(i.e., e has exactly one endpoint in F).

Then e is in M.

Proof: Suppose not.

Then $M \cup \{e\}$ has a cycle,
containing e and another outgoing edge f of F.

Replacing f by e in M gives a spanning tree
with a smaller sum of weights of edges.


4
6
8
1
3
5
7
2
9
10

Kruskal's algorithm

A *uniprocessor* algorithm for computing minimum spanning trees.

- Initially, each node forms a separate fragment.
- In each step, a lowest-weight outgoing edge of a fragment is added to the spanning tree, joining two fragments.

This algorithm also works when edges have the same weight.

Then the minimum spanning tree may not be unique.

Kruskal's algorithm

A *uniprocessor* algorithm for computing minimum spanning trees.

- Initially, each node forms a separate fragment.
- In each step, a lowest-weight outgoing edge of a fragment is added to the spanning tree, joining two fragments.

This algorithm also works when edges have the same weight.

Then the minimum spanning tree may not be unique.

Gallager-Humblet-Spira algorithm

Consider an undirected, weighted network,
in which different edges have different weights.

Distributed computation of a minimum spanning tree:

- Initially, each process is a fragment.
- The processes in a fragment F together search for the lowest-weight outgoing edge e_F.
- When e_F has been found, the fragment at the other end is asked to collaborate in a merge.

Gallager-Humblet-Spira algorithm

Consider an undirected, weighted network,
in which different edges have different weights.

Distributed computation of a minimum spanning tree:

- Initially, each process is a fragment.
- The processes in a fragment F together search for the lowest-weight outgoing edge e_F.
- When e_F has been found, the fragment at the other end is asked to collaborate in a merge.

Complications: Is an edge outgoing? Is it lowest-weight?

Level, name and core edge

Each fragment carries a (unique) name $fn : \mathbb{R}$ and a level $\ell : \mathbb{N}$.

Its level is the maximum number of joins any process in the fragment has experienced.

Neighboring fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ can be joined as follows:

$$\ell < \ell' \ \wedge \ F \xrightarrow{e_F} F' : \quad F \cup F' = (fn', \ell')$$

$$\ell = \ell' \ \wedge \ e_F = e_{F'} : \quad F \cup F' = (\text{weight } e_F, \ell + 1)$$

The core edge of a fragment is the last edge that connected two sub-fragments at the same level. Its end points are the core nodes.

Level, name and core edge

Each fragment carries a (unique) name $fn : \mathbb{R}$ and a level $\ell : \mathbb{N}$.

Its level is the maximum number of joins any process in the fragment has experienced.

Neighboring fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ can be joined as follows:

$\ell < \ell' \ \wedge \ F \xrightarrow{e_F} F'$: $\quad F \cup F' = (fn', \ell')$

$\ell = \ell' \ \wedge \ e_F = e_{F'}$: $\quad F \cup F' = (\text{weight } e_F, \ell + 1)$

The core edge of a fragment is the last edge that connected two sub-fragments at the same level. Its end points are the core nodes.

Level, name and core edge

Each fragment carries a (unique) name $fn : \mathbb{R}$ and a level $\ell : \mathbb{N}$.

Its level is the maximum number of joins any process in the fragment has experienced.

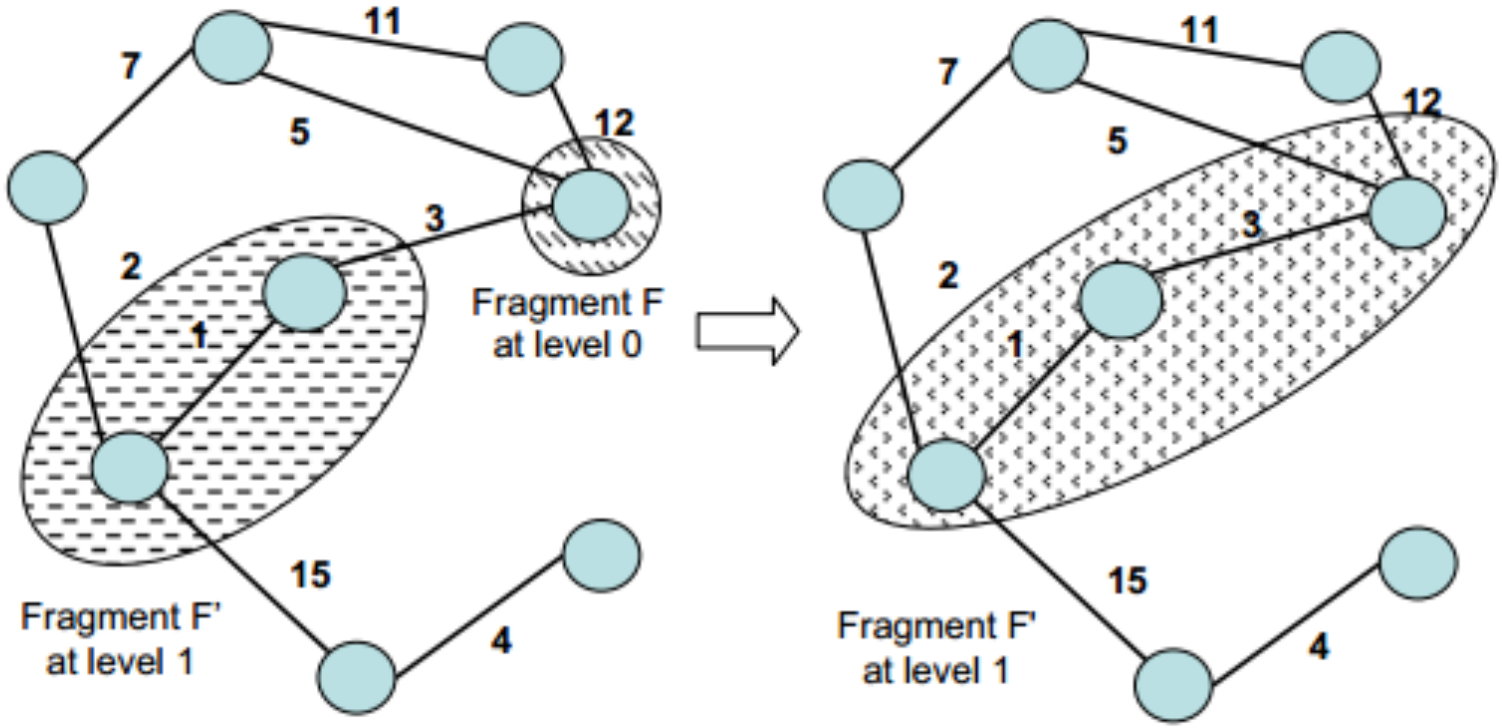
Neighboring fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ can be joined as follows:

$$\ell < \ell' \ \wedge \ F \xrightarrow{e_F} F' : \quad F \cup F' = (fn', \ell')$$

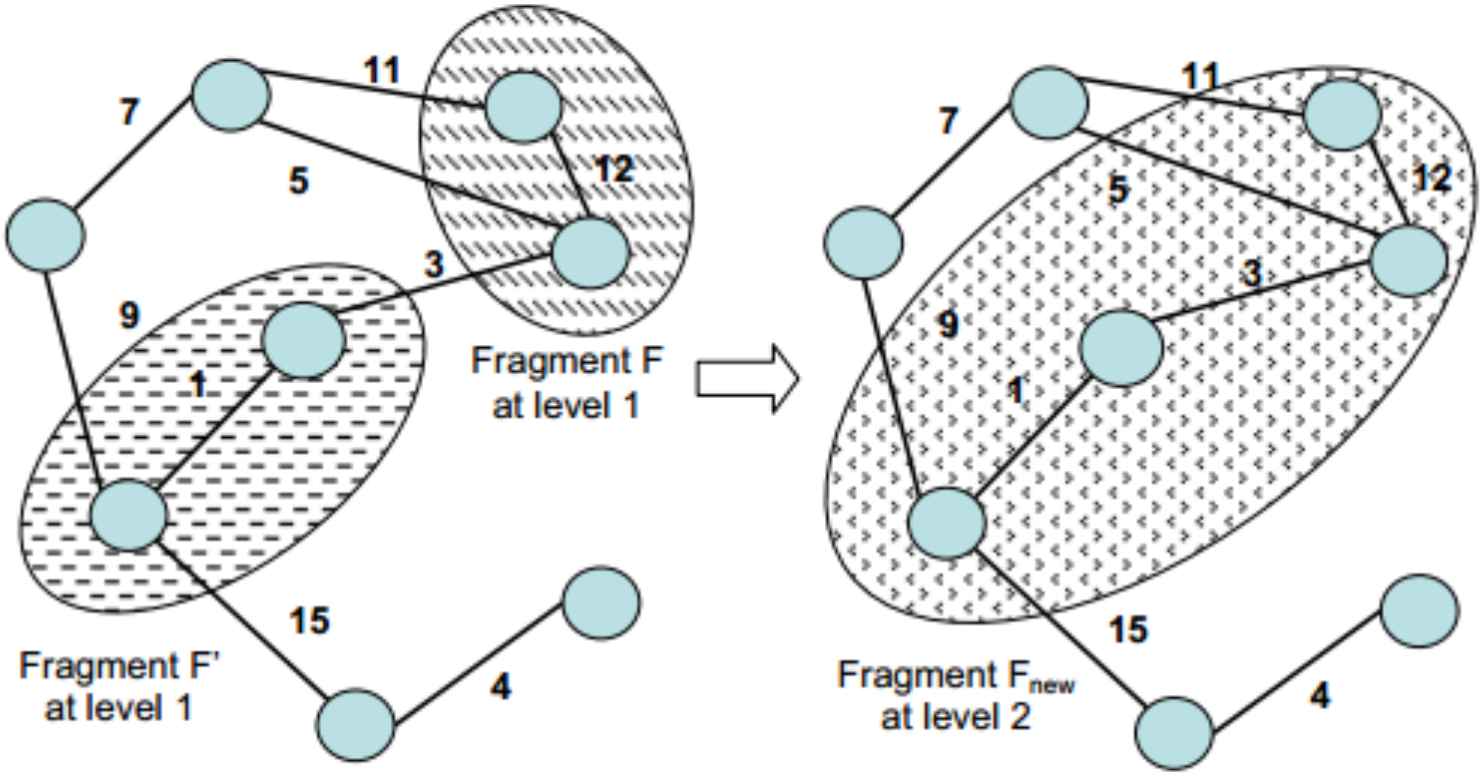
$$\ell = \ell' \ \wedge \ e_F = e_{F'} : \quad F \cup F' = (\text{weight } e_F, \ell + 1)$$

The core edge of a fragment is the last edge that connected two sub-fragments at the same level. Its end points are the core nodes.

Joining two fragments at different levels: lower-level joins into higher-level



Joining two fragments at same level: must agree on branch edge



Parameters of a process

Its *state*:

- find (looking for a lowest-weight outgoing edge)
- found (reported a lowest-weight outgoing edge to the core edge)

The *status* of its *channels*:

- basic edge (undecided)
- branch edge (in the spanning tree)
- rejected (not in the spanning tree)

The *name* and *level* of its fragment.

Its *parent* (toward the core edge).

Parameters of a process

Its *state*:

- find (looking for a lowest-weight outgoing edge)
- found (reported a lowest-weight outgoing edge to the core edge)

The *status* of its *channels*:

- basic edge (undecided)
- branch edge (in the spanning tree)
- rejected (not in the spanning tree)

The *name* and *level* of its fragment.

Its *parent* (toward the core edge).

Parameters of a process

Its *state*:

- find (looking for a lowest-weight outgoing edge)
- found (reported a lowest-weight outgoing edge to the core edge)

The *status* of its *channels*:

- basic edge (undecided)
- branch edge (in the spanning tree)
- rejected (not in the spanning tree)

The *name* and *level* of its fragment.

Its *parent* (toward the core edge).

Initialization

Non-initiators wake up when they receive a (**connect** or **test**) message.

Each initiator, and noninitiator after it has woken up:

- sets its level to 0
- sets its *lowest-weight* edge to branch
- sends $\langle \textbf{connect}, 0 \rangle$ into this channel
- sets its other channels to basic
- sets its state to found

Joining two fragments

Let fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ be joined via channel pq.

- If $\ell < \ell'$, then p sent $\langle \textbf{connect}, \ell \rangle$ to q.

  q sends $\langle \textbf{initiate}, fn', \ell', \frac{find}{found} \rangle$ to p.

  $F \cup F'$ inherits the core edge of F'.

- If $\ell = \ell'$, then p and q sent $\langle \textbf{connect}, \ell \rangle$ to each other.

  They send $\langle \textbf{initiate}, \text{weight } pq, \ell + 1, find \rangle$ to each other.

  $F \cup F'$ gets core edge pq.

At reception of $\langle \textbf{initiate}, fn, \ell, \frac{find}{found} \rangle$, a process stores fn and ℓ, sets its state to *find* or *found*, and adopts the sender as its parent.

It passes on the message through its other branch edges.

Joining two fragments

Let fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ be joined via channel pq.

- If $\ell < \ell'$, then p sent $\langle \textbf{connect}, \ell \rangle$ to q.

  q sends $\langle \textbf{initiate}, fn', \ell', \frac{find}{found} \rangle$ to p.

  $F \cup F'$ inherits the core edge of F'.

- If $\ell = \ell'$, then p and q sent $\langle \textbf{connect}, \ell \rangle$ to each other.

  They send $\langle \textbf{initiate}, \text{weight } pq, \ell + 1, find \rangle$ to each other.

  $F \cup F'$ gets core edge pq.

At reception of $\langle \textbf{initiate}, fn, \ell, \frac{find}{found} \rangle$, a process stores fn and ℓ, sets its state to *find* or *found*, and adopts the sender as its parent.

It passes on the message through its other branch edges.

Joining two fragments

Let fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ be joined via channel pq.

- If $\ell < \ell'$, then p sent $\langle \textbf{connect}, \ell \rangle$ to q.

  q sends $\langle \textbf{initiate}, fn', \ell', \frac{find}{found} \rangle$ to p.

  $F \cup F'$ inherits the core edge of F'.

- If $\ell = \ell'$, then p and q sent $\langle \textbf{connect}, \ell \rangle$ to each other.

  They send $\langle \textbf{initiate}, \text{weight } pq, \ell + 1, find \rangle$ to each other.

  $F \cup F'$ gets core edge pq.

At reception of $\langle \textbf{initiate}, fn, \ell, \frac{find}{found} \rangle$, a process stores fn and ℓ, sets its state to *find* or *found*, and adopts the sender as its parent.

It passes on the message through its other branch edges.

Computing the lowest-weight outgoing edge

In case of $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$, p checks in increasing order of weight if one of its *basic* edges pq is *outgoing*, by sending $\langle$**test**, *fn*, $\ell\rangle$ to q.

While $\ell > level_q$, q *postpones* processing the incoming **test** message.

Let $\ell \leq level_q$.

- If q is in fragment *fn*, then q replies **reject**.

  In this case p and q will set pq to rejected.

- Else, q replies **accept**.

When a basic edge is accepted, or there are no basic edges left, p *stops* the search.

Computing the lowest-weight outgoing edge

In case of $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$, p checks in increasing order of weight if one of its *basic* edges pq is *outgoing*, by sending $\langle$**test**, *fn*, $\ell\rangle$ to q.

While $\ell > \mathit{level}_q$, q *postpones* processing the incoming **test** message.

Let $\ell \leq \mathit{level}_q$.

- If q is in fragment *fn*, then q replies **reject**.

  In this case p and q will set pq to rejected.

- Else, q replies **accept**.

When a basic edge is accepted, or there are no basic edges left, p *stops* the search.

Computing the lowest-weight outgoing edge

In case of $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$, p checks in increasing order of weight if one of its *basic* edges pq is *outgoing*, by sending $\langle$**test**, *fn*, $\ell\rangle$ to q.

While $\ell > \mathit{level}_q$, q *postpones* processing the incoming **test** message.

Let $\ell \leq \mathit{level}_q$.

- If q is in fragment *fn*, then q replies **reject**.

  In this case p and q will set pq to rejected.

- Else, q replies **accept**.

When a basic edge is accepted, or there are no basic edges left, p *stops* the search.

Questions

Why does q postpone processing the incoming $\langle \textbf{test}, \_, \ell \rangle$ message from p while $\ell > \mathit{level}_q$?

Why does this postponement not lead to a deadlock ?

Questions

Why does q postpone processing the incoming $\langle \textbf{test}, \_, \ell \rangle$ message from p while $\ell > \mathit{level}_q$?

Answer: p and q might be in the same fragment, in which case $\langle \textbf{initiate}, \mathit{fn}, \ell, \mathit{find} \rangle$ is on its way to q.

Why does this postponement not lead to a deadlock ?

Questions

Why does q postpone processing the incoming $\langle \textbf{test}, \_, \ell \rangle$ message from p while $\ell > \mathit{level}_q$?

Answer: p and q might be in the same fragment, in which case $\langle \textbf{initiate}, \mathit{fn}, \ell, \mathit{find} \rangle$ is on its way to q.

Why does this postponement not lead to a deadlock ?

Questions

Why does q postpone processing the incoming $\langle \textbf{test}, \_, \ell \rangle$ message from p while $\ell > \mathit{level}_q$?

Answer: p and q might be in the same fragment, in which case $\langle \textbf{initiate}, \mathit{fn}, \ell, \mathit{find} \rangle$ is on its way to q.

Why does this postponement not lead to a deadlock ?

Answer: There is always a fragment with a smallest level.

Reporting to the core nodes

- p waits for all branch edges, *except its parent*, to report.
- p sets its state to *found*.
- p computes the minimum λ of (1) these reports, and (2) the weight of its lowest-weight outgoing basic edge (or ∞, if no such channel was found).
- If $\lambda < \infty$, p stores either the branch edge that sent λ, or its basic edge of weight λ.
- p sends $\langle \textbf{report}, \lambda \rangle$ to its parent.

Termination or **changeroot** at the core nodes

A core node receives reports through *all* its branch edges, including the core edge.

- If the minimum reported value $\mu = \infty$, the core nodes terminate.
- If $\mu < \infty$, the core node that received μ first sends **changeroot** toward the lowest-weight outgoing basic edge.

  (The core edge becomes a regular tree edge.)

Ultimately **changeroot** reaches the process p that reported the lowest-weight outgoing basic edge.

p sets this channel to branch, and sends $\langle$**connect**, $level_p\rangle$ into it.

Termination or **changeroot** at the core nodes

A core node receives reports through *all* its branch edges, including the core edge.

- If the minimum reported value $\mu = \infty$, the core nodes terminate.
- If $\mu < \infty$, the core node that received μ first sends **changeroot** toward the lowest-weight outgoing basic edge.

  (The core edge becomes a regular tree edge.)

Ultimately **changeroot** reaches the process p that reported the lowest-weight outgoing basic edge.

p sets this channel to branch, and sends $\langle \textbf{connect}, \mathit{level}_p \rangle$ into it.

Starting the join of two fragments

When q receives $\langle \textbf{connect}, \mathit{level}_p \rangle$ from p, $\mathit{level}_q \geq \mathit{level}_p$.

Namely, either $\mathit{level}_p = 0$, or q earlier sent **accept** to p.

- If $\mathit{level}_q > \mathit{level}_p$, then q sets qp to *branch* and sends $\langle \textbf{initiate}, \mathit{name}_q, \mathit{level}_q, \frac{\mathit{find}}{\mathit{found}} \rangle$ to p.
- As long as $\mathit{level}_q = \mathit{level}_p$ and qp isn't a branch edge, q *postpones* processing the **connect** message.
- If $\mathit{level}_q = \mathit{level}_p$ and qp is a branch edge (meaning that q sent $\langle \textbf{connect}, \mathit{level}_q \rangle$ to p), then q sends $\langle \textbf{initiate}, \text{weight } qp, \mathit{level}_q + 1, \mathit{find} \rangle$ to p (and vice versa).

  In this case pq becomes the core edge.

Starting the join of two fragments

When q receives $\langle \textbf{connect}, \mathit{level}_p \rangle$ from p, $\mathit{level}_q \geq \mathit{level}_p$.

Namely, either $\mathit{level}_p = 0$, or q earlier sent **accept** to p.

- If $\mathit{level}_q > \mathit{level}_p$, then q sets qp to *branch* and sends $\langle \textbf{initiate}, \mathit{name}_q, \mathit{level}_q, \frac{\mathit{find}}{\mathit{found}} \rangle$ to p.
- As long as $\mathit{level}_q = \mathit{level}_p$ and qp isn't a branch edge, q *postpones* processing the **connect** message.
- If $\mathit{level}_q = \mathit{level}_p$ and qp is a branch edge (meaning that q sent $\langle \textbf{connect}, \mathit{level}_q \rangle$ to p), then q sends $\langle \textbf{initiate}, \text{weight } qp, \mathit{level}_q + 1, \mathit{find} \rangle$ to p (and vice versa).

  In this case pq becomes the core edge.

Starting the join of two fragments

When q receives $\langle \textbf{connect}, \mathit{level}_p \rangle$ from p, $\mathit{level}_q \geq \mathit{level}_p$.

Namely, either $\mathit{level}_p = 0$, or q earlier sent **accept** to p.

- If $\mathit{level}_q > \mathit{level}_p$, then q sets qp to *branch* and sends $\langle \textbf{initiate}, \mathit{name}_q, \mathit{level}_q, \frac{\mathit{find}}{\mathit{found}} \rangle$ to p.
- As long as $\mathit{level}_q = \mathit{level}_p$ and qp isn't a branch edge, q *postpones* processing the **connect** message.
- If $\mathit{level}_q = \mathit{level}_p$ and qp is a branch edge (meaning that q sent $\langle \textbf{connect}, \mathit{level}_q \rangle$ to p), then q sends $\langle \textbf{initiate}, \text{weight } qp, \mathit{level}_q + 1, \mathit{find} \rangle$ to p (and vice versa).

  In this case pq becomes the core edge.

Starting the join of two fragments

When q receives $\langle \textbf{connect}, \mathit{level}_p \rangle$ from p, $\mathit{level}_q \geq \mathit{level}_p$.

Namely, either $\mathit{level}_p = 0$, or q earlier sent **accept** to p.

- If $\mathit{level}_q > \mathit{level}_p$, then q sets qp to *branch* and sends $\langle \textbf{initiate}, \mathit{name}_q, \mathit{level}_q, \frac{\mathit{find}}{\mathit{found}} \rangle$ to p.
- As long as $\mathit{level}_q = \mathit{level}_p$ and qp isn't a branch edge, q *postpones* processing the **connect** message.
- If $\mathit{level}_q = \mathit{level}_p$ and qp is a branch edge (meaning that q sent $\langle \textbf{connect}, \mathit{level}_q \rangle$ to p), then q sends $\langle \textbf{initiate}, \text{weight } qp, \mathit{level}_q + 1, \mathit{find} \rangle$ to p (and vice versa).

  In this case pq becomes the core edge.

Questions

If $level_q = level_p$ and qp isn't a branch edge, why does q postpone processing the incoming **connect** message from p ?

Why does this postponement not give rise to a deadlock ?

(I.e., why can't there be a cycle of fragments waiting for a reply to a postponed **connect** message ?)

Questions

If $level_q = level_p$ and qp isn't a branch edge, why does q postpone processing the incoming **connect** message from p?

Answer: The fragment of q might be in the process of joining another fragment at a level $\geq level_q$.

Then the fragment of p should subsume the name and level of that joint fragment, instead of joining q's fragment at an equal level.

Why does this postponement not give rise to a deadlock?

(I.e., why can't there be a cycle of fragments waiting for a reply to a postponed **connect** message?)

Questions

If $level_q = level_p$ and qp isn't a branch edge, why does q postpone processing the incoming **connect** message from p?

Answer: The fragment of q might be in the process of joining another fragment at a level $\geq level_q$.

Then the fragment of p should subsume the name and level of that joint fragment, instead of joining q's fragment at an equal level.

Why does this postponement not give rise to a deadlock?

(I.e., why can't there be a cycle of fragments waiting for a reply to a postponed **connect** message?)

Questions

If $level_q = level_p$ and qp isn't a branch edge, why does q postpone processing the incoming **connect** message from p?

Answer: The fragment of q might be in the process of joining another fragment at a level $\geq level_q$.

Then the fragment of p should subsume the name and level of that joint fragment, instead of joining q's fragment at an equal level.

Why does this postponement not give rise to a deadlock?

(I.e., why can't there be a cycle of fragments waiting for a reply to a postponed **connect** message?)

Answer: Because different channels have different weights.

Question

Suppose a process reported a lowest-weight outgoing basic edge, and in return receives $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$.

Why must it test again whether this basic edge is outgoing ?

Question

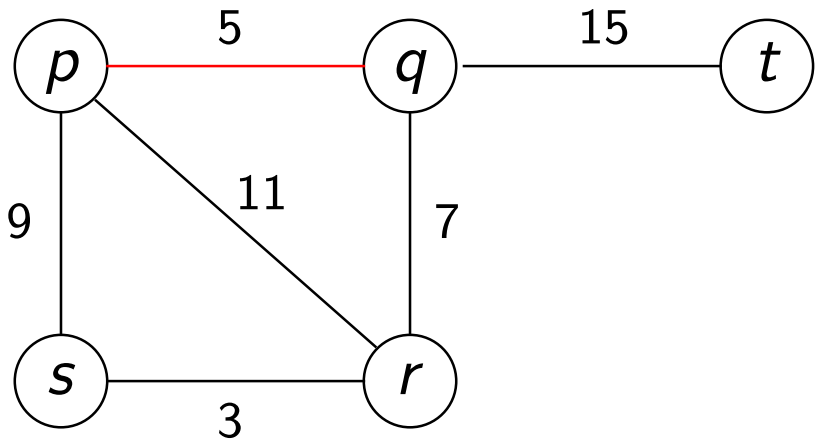
Suppose a process reported a lowest-weight outgoing basic edge, and in return receives $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$.

Why must it test again whether this basic edge is outgoing ?

Answer: Its fragment may in the meantime have joined the fragment at the other side of this basic edge.

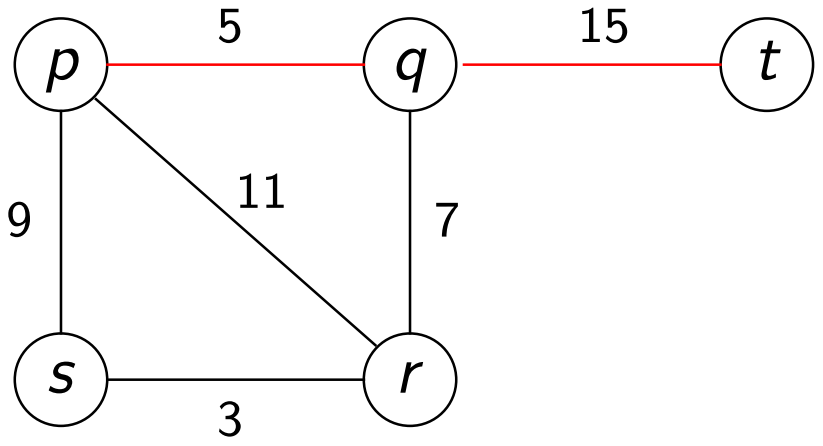
Gallager-Humblet-Spira algorithm - Example

pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**



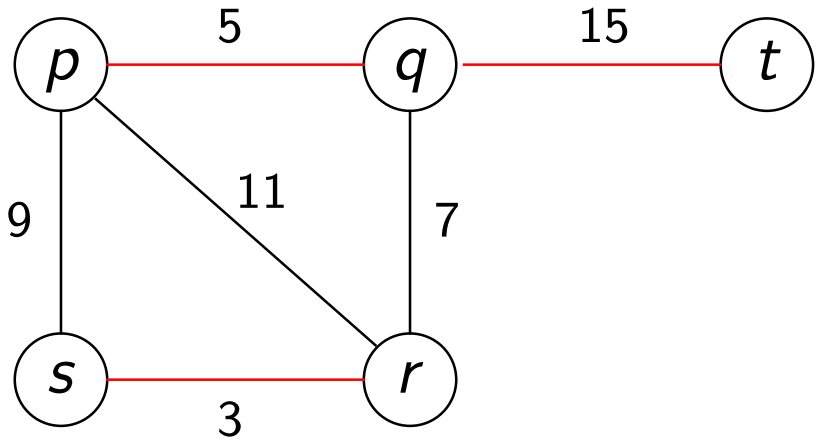
Gallager-Humblet-Spira algorithm - Example

pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**



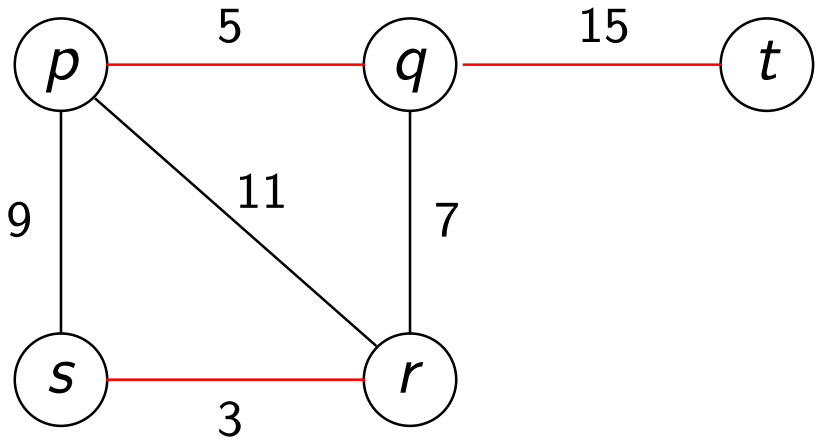
Gallager-Humblet-Spira algorithm - Example

pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**

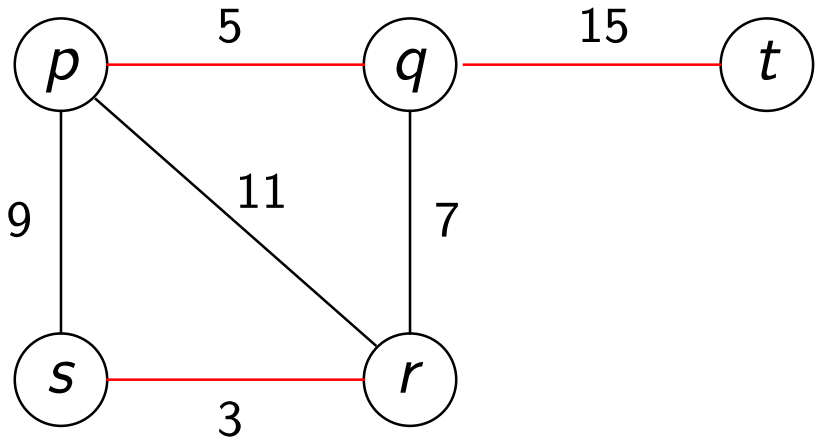


Gallager-Humblet-Spira algorithm - Example

pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**



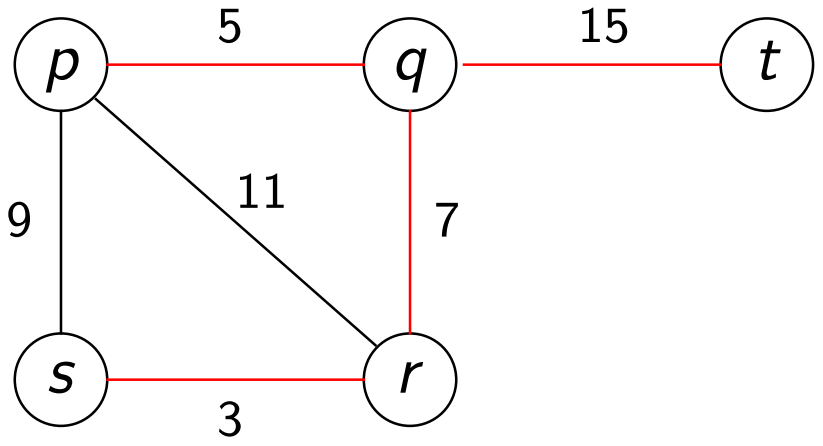
Gallager-Humblet-Spira algorithm - Example



pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**

rs	$\langle$**report**, 7$\rangle$
sr	$\langle$**report**, 9$\rangle$
rq	$\langle$**connect**, 1$\rangle$

Gallager-Humblet-Spira algorithm - Example



pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**

rs	$\langle$**report**, 7$\rangle$
sr	$\langle$**report**, 9$\rangle$
rq	$\langle$**connect**, 1$\rangle$
rq qp qr qt rs	$\langle$**initiate**, 7, 2, *find*$\rangle$
ps sp	$\langle$**test**, 7, 2$\rangle$
pr	$\langle$**test**, 7, 2$\rangle$
rp	**reject**
pq tq qr sr rq	$\langle$**report**, $\infty\rangle$

Gallager-Humblet-Spira algorithm - Complexity

Worst-case message complexity: $O(E + N \log N)$

- A rejected channel requires a **test-reject** or **test-test** pair.

Between two subsequent joins, a process:

- receives one **initiate**
- sends at most one **test** that triggers an **accept**
- sends one **report**
- sends at most one **changeroot** or **connect**

A fragment at level ℓ contains $\geq 2^{\ell}$ processes.

So each process experiences at most $\lfloor \log_2 N \rfloor$ joins.

Gallager-Humblet-Spira algorithm - Complexity

Worst-case message complexity: $O(E + N \log N)$

- A rejected channel requires a **test-reject** or **test-test** pair.

Between two subsequent joins, a process:

- receives one **initiate**
- sends at most one **test** that triggers an **accept**
- sends one **report**
- sends at most one **changeroot** or **connect**

A fragment at level ℓ contains $\geq 2^{\ell}$ processes.

So each process experiences at most $\lfloor \log_2 N \rfloor$ joins.

Gallager-Humblet-Spira algorithm - Complexity

Worst-case message complexity: $O(E + N \log N)$

- A rejected channel requires a **test-reject** or **test-test** pair.

Between two subsequent joins, a process:

- receives one **initiate**
- sends at most one **test** that triggers an **accept**
- sends one **report**
- sends at most one **changeroot** or **connect**

A fragment at level ℓ contains $\geq 2^{\ell}$ processes.

So each process experiences at most $\lfloor \log_2 N \rfloor$ joins.

Gallager-Humblet-Spira algorithm - Complexity

Worst-case message complexity: $O(E + N \log N)$

- A rejected channel requires a **test-reject** or **test-test** pair.

Between two subsequent joins, a process:

- receives one **initiate**
- sends at most one **test** that triggers an **accept**
- sends one **report**
- sends at most one **changeroot** or **connect**

A fragment at level ℓ contains $\geq 2^{\ell}$ processes.

So each process experiences at most $\lfloor \log_2 N \rfloor$ joins.