

Minimum Spanning Tree

Given a weighted graph $G = (V, E)$, generate a **spanning tree** $T = (V, E')$ such that the *sum of the weights of all the edges is minimum*.

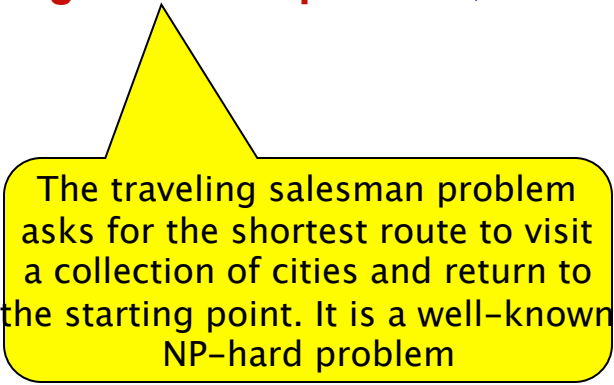
A few applications

Minimum cost vehicle routing.

A cable TV company will use this to lay cables in a new neighborhood.

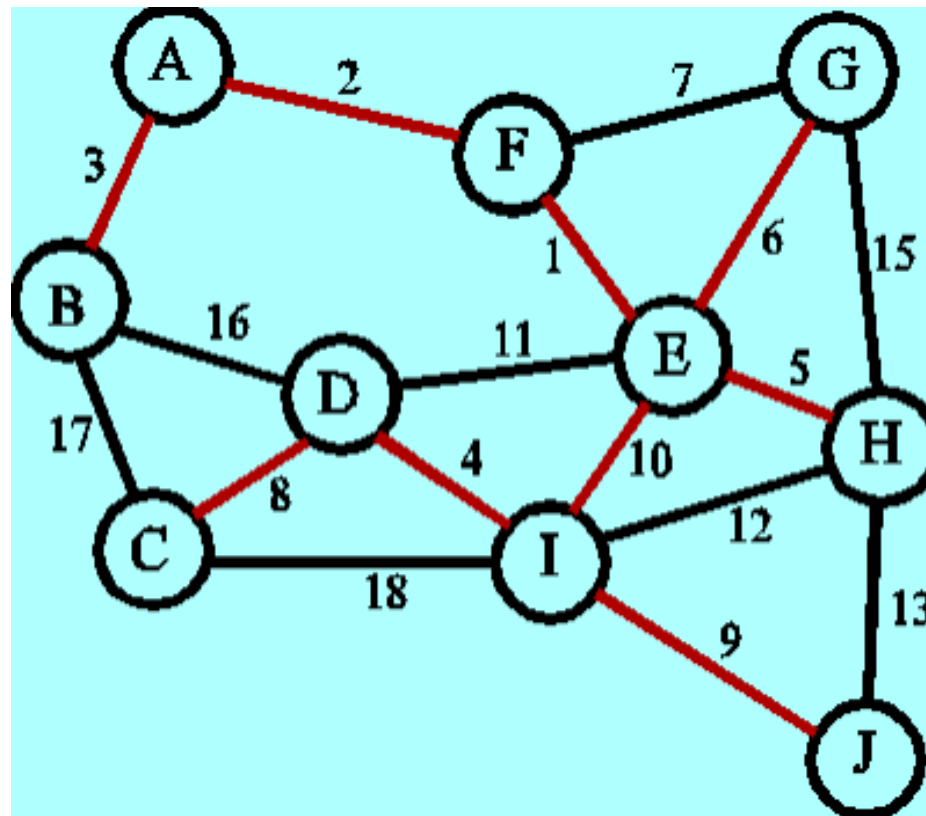
On Euclidean plane, *approximate* solutions to the **traveling salesman problem**,

We are interested in distributed algorithms only



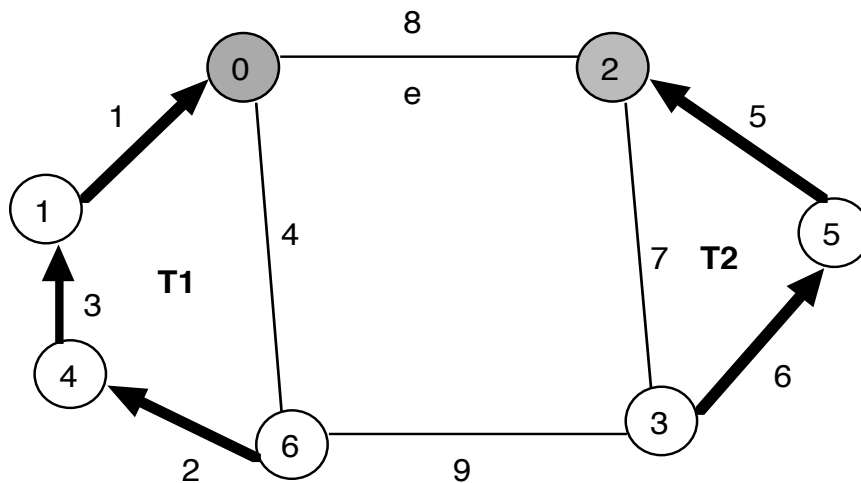
The traveling salesman problem asks for the shortest route to visit a collection of cities and return to the starting point. It is a well-known NP-hard problem

Example



Sequential algorithms for MST

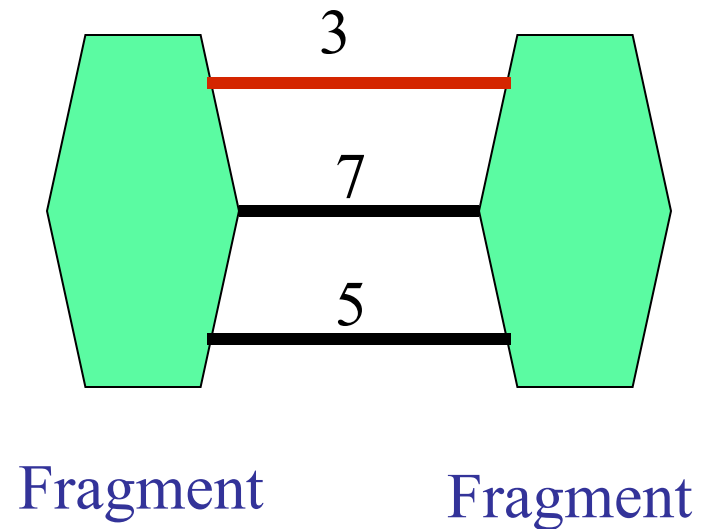
Review (1) Prim's algorithm and (2) Kruskal's algorithm (greedy algorithms)



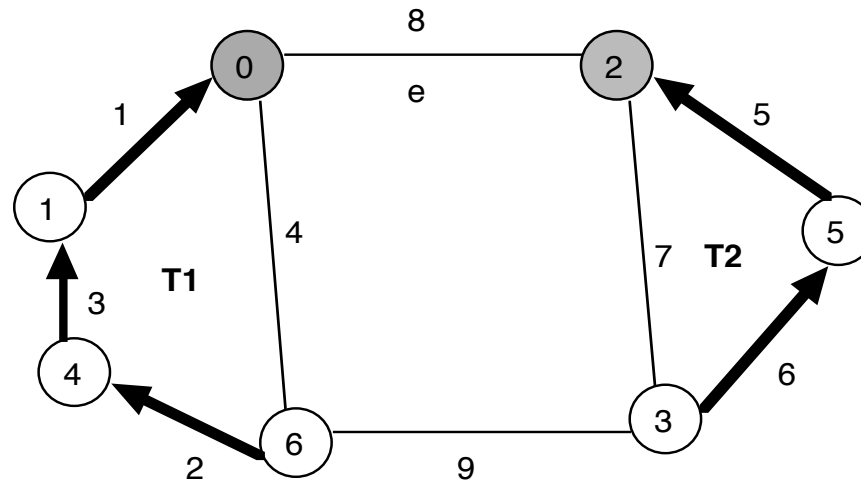
Theorem. If the weight of every edge is distinct, then the MST is unique.

Gallagher-Humblet-Spira (GHS) Algorithm

- GHS is a distributed version of Prim's algorithm.
- Bottom-up approach. MST is recursively constructed by fragments joined by an edge of least cost.



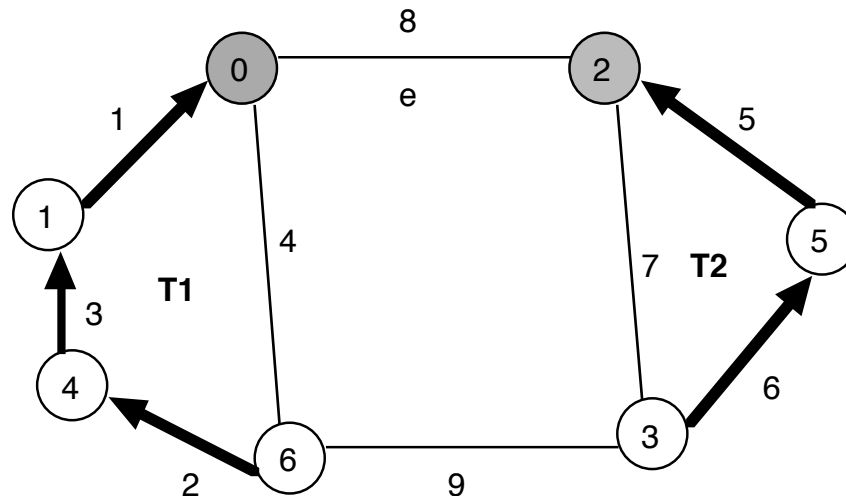
Challenges



Challenge 1. How will the nodes in a given fragment identify the edge to be used to connect with a different fragment?

A root node in each fragment is the root/coordinator

Challenges



Challenge 2. How will a node in **T1** determine if a given edge connects to a node of a different tree **T2** or the same tree **T1**? Why will node 0 choose the edge **e** with weight **8**, and not the edge with weight **4**?

*Nodes in a fragment acquire the **same name** before augmentation.*

Two main steps

- Each fragment has a **level**. Initially each node is a fragment at level 0.
- **(MERGE)** Two fragments at the same level L combine to form a fragment of level $L+1$
- **(ABSORB)** A fragment at level L is absorbed by another fragment at level L' ($L < L'$). The new fragment has a level L' .

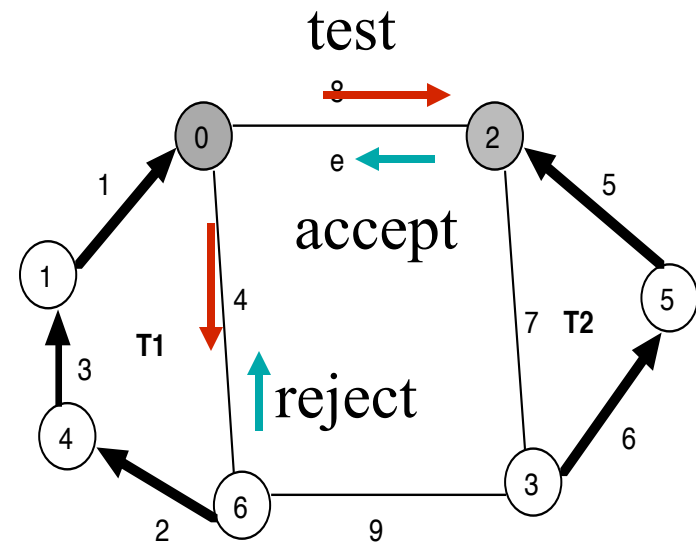
(Each fragment in level L has at least 2^L nodes)

Least weight outgoing edge

To test if an edge is **outgoing**, each node sends a **test** message through a candidate edge. The receiving node may send **accept** or **reject**.

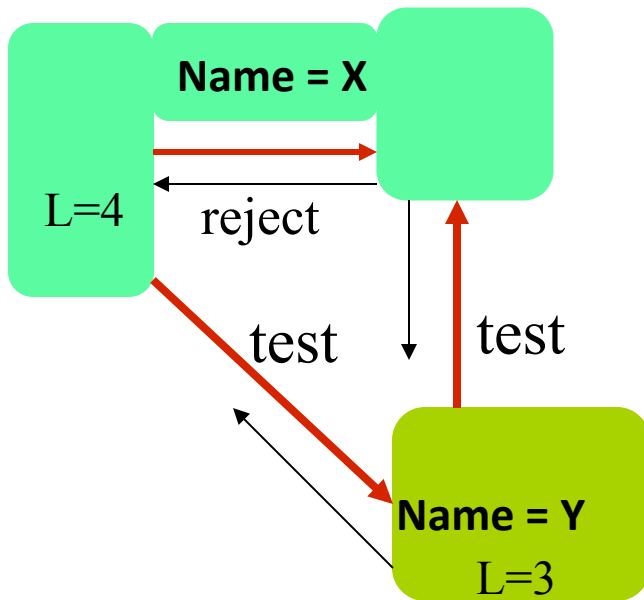
Root broadcasts **initiate** in its own fragment, **collects the report from other nodes** about eligible edges using a **convergecast**, and determines the **least weight outgoing edge**.

(Broadcast and Convergecast are two handy tools)



Accept or reject?

Let i send **test** to j



Case 1. If name (i) = name (j) then send **reject**

Case 2. If name (i) \neq name (j) AND level (i) \leq level (j) then node j sends **accept**

Case 3. If name (i) \neq name (j) AND level (i) $>$ level (j) then **wait until** level (j) = level (i) and then send **accept/reject**. **WHY?** (See note below)

(Also note that levels can only increase).

Q: Can fragments wait for ever and lead to a deadlock?

Note. It may be the case that the responding node belongs a different fragment when it received the test message, but it is also trying to merge with the sending fragment.

The major steps

repeat

- 1 Test edges as outgoing or not
- 2 Determine **least weight outgoing edge** - it becomes a tree edge
- 3 Send **join** (or respond to **join**)
- 4 Update level & name & identify new coordinator/root

until there are no outgoing edges

Classification of edges

- **Basic** (initially all branches are basic)
- **Branch** (all tree edges)
- **Rejected** (not a tree edge)

Branch and **rejected** are stable attributes

(once tagged as **rejected**, it remains so for ever.

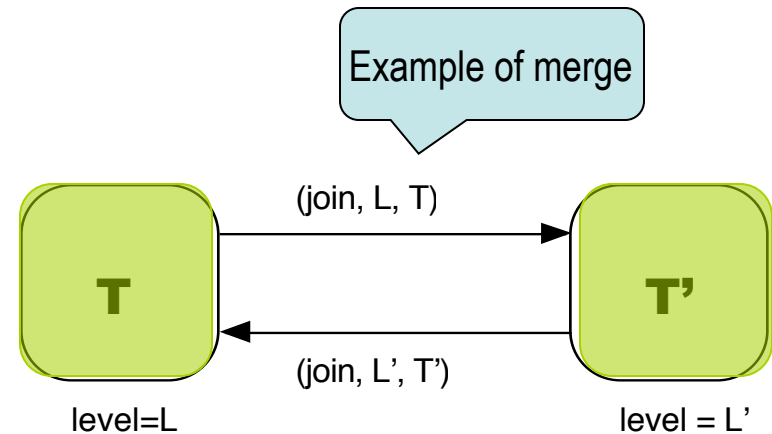
The same thing holds for **tree edges** too.)

Wrapping it up

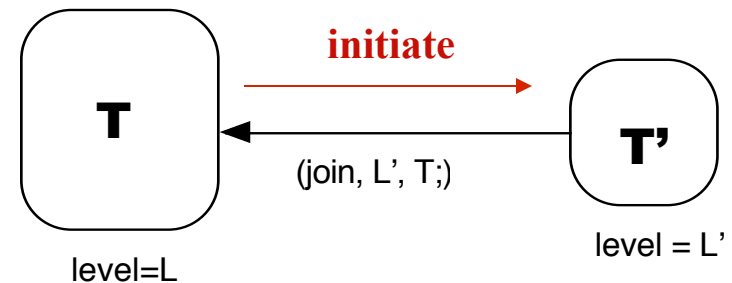
Merge

The edge through which the **join** message is exchanged, changes its status to *branch*, and it becomes a tree edge.

The new root broadcasts an **(initiate, L+1, name)** message to the nodes in its own fragment.



(a) $L = L'$

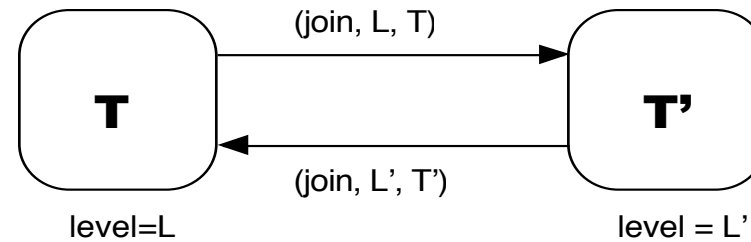


(b) $L > L'$

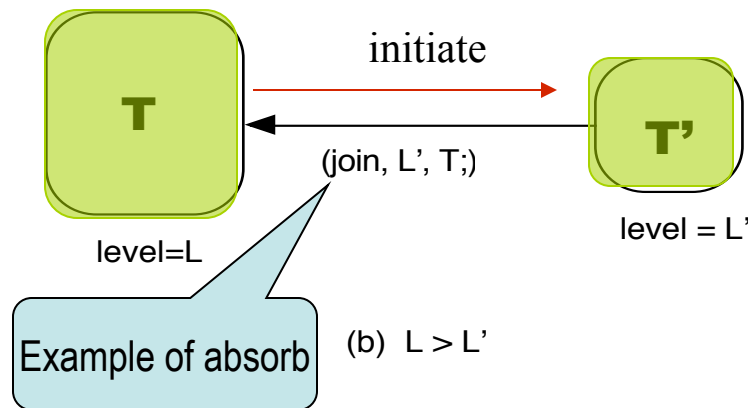
Wrapping it up

Absorb

T' sends a **join** message to T, and receives an **initiate** message. This indicates that the fragment at level **L** has been absorbed by the other fragment at level **L'**. They collectively search for the **lwoe**. The edge through which the **join** message was sent, changes its status to **branch**.

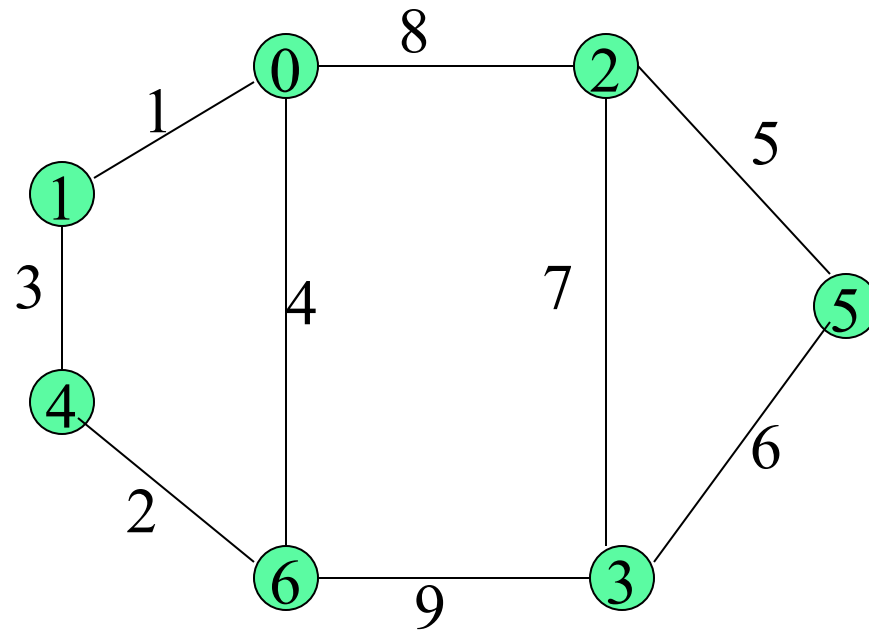


(a) $L = L'$

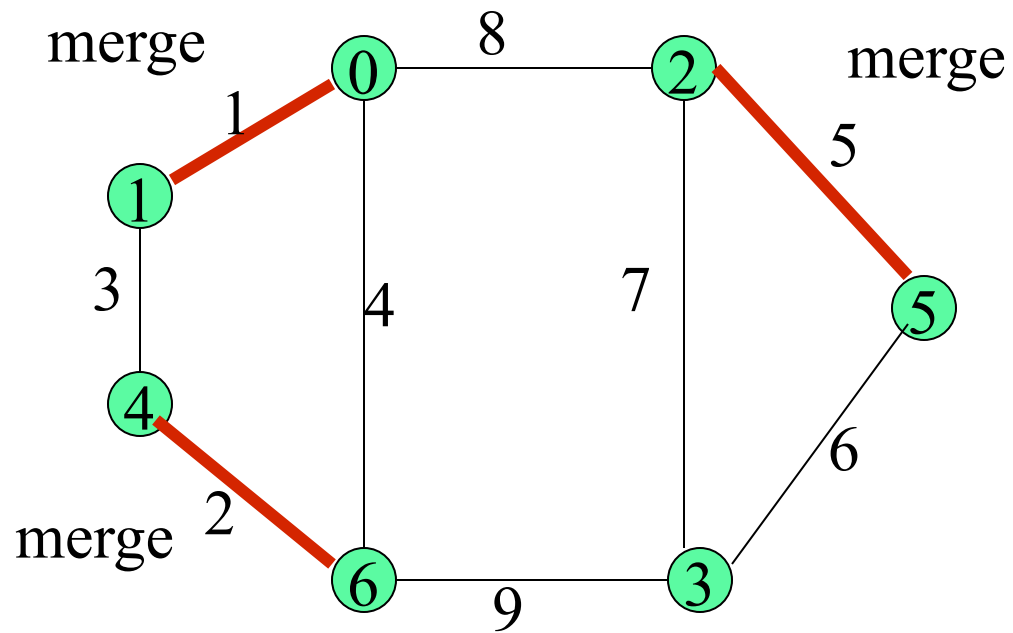


(b) $L > L'$

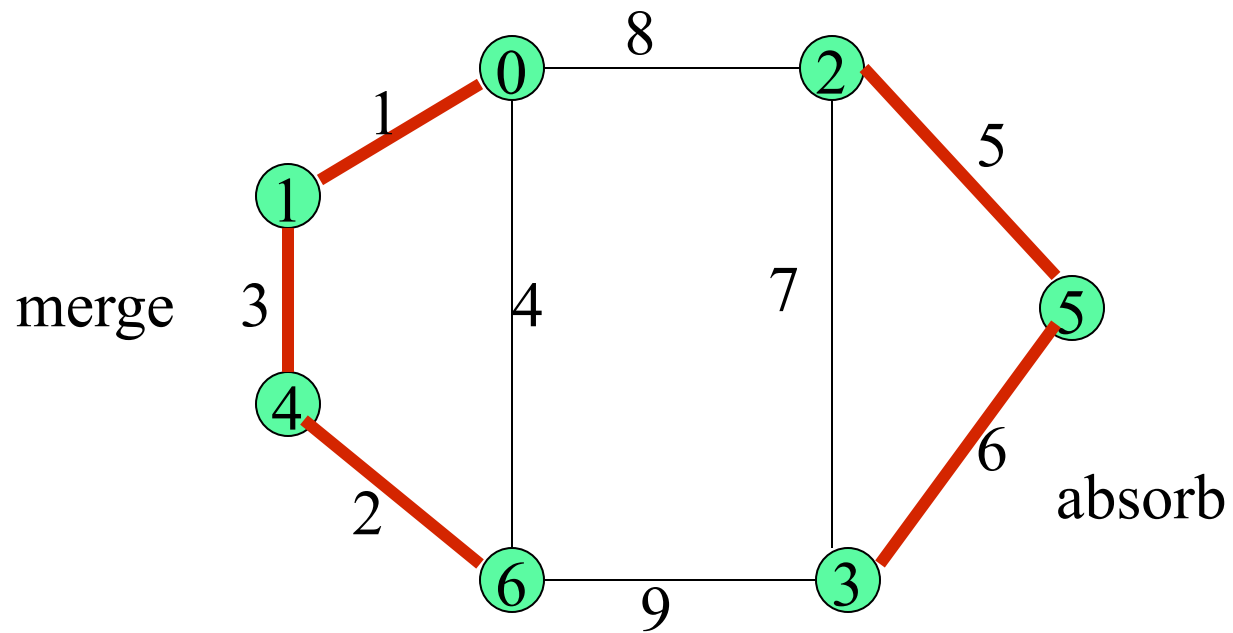
Example



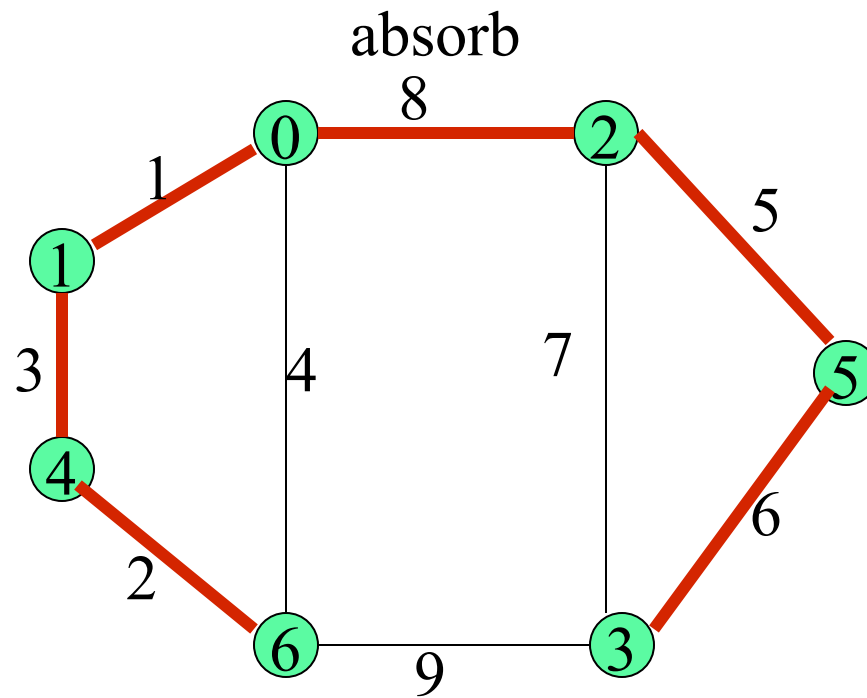
Example



Example



Example



Message complexity

Each edge may be rejected at most once. It requires two messages (*test + reject*). The upper bound is $2|E|$ messages.

At each of the (max) $\log N$ levels, a node RECEIVES at most (1) one *initiate* message and (2) one *accept* message and SENDS (3) one *report* message (4) one *test* message *not* leading to a rejection, and (5) one *changeroot* or *join* message.

So, the total number of messages has an upper bound of $2|E| + 5N \log N$

Coordination Algorithms:

Leader Election

Leader Election

Let $G = (V, E)$ define the network topology. Each process i has a variable $L(i)$ that defines the *leader*. The goal is to reach a configuration, where

$\forall i, j \in V : i, j \text{ are non-faulty} ::$

- (1) $L(i) \in V$ **and**
- (2) $L(i) = L(j)$ **and**
- (3) $L(i)$ is non-faulty

Often reduces to *maxima (or minima) finding problem*.
(if we ignore the failure detection part)

Leader Election

Difference between mutual exclusion & leader election

The similarity is in the phrase “at most one process.” But,

Failure is not an issue in mutual exclusion, a new leader is elected only after the current leader fails.

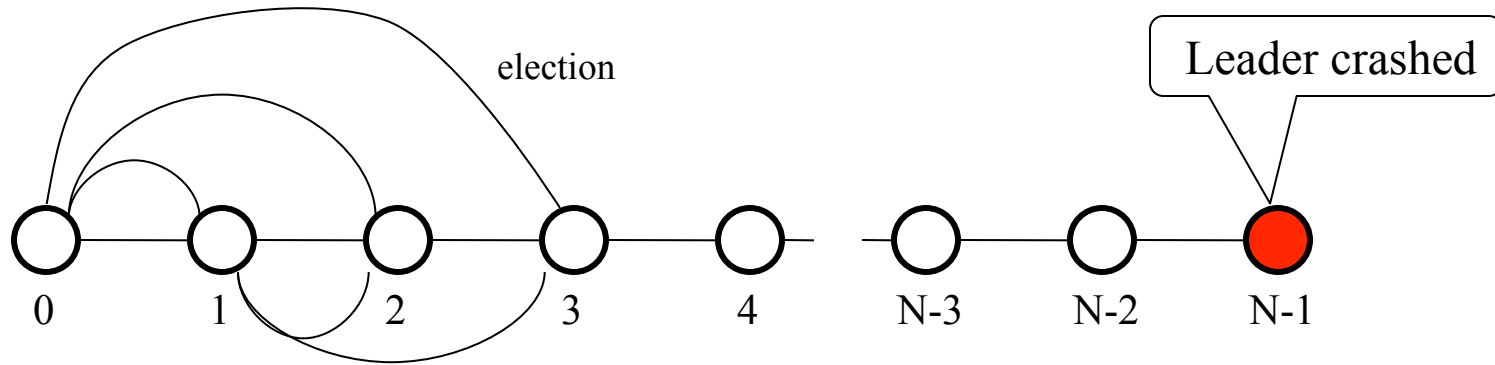
No fairness is necessary - it is not necessary that every aspiring process has to become a leader.

Bully algorithm

(Assumes that the topology is completely connected)

1. Send ***election*** message (*I want to be the leader*) to processes with ***larger id***
2. Give up your bid if a process with ***larger id*** sends a ***reply*** message (*means no, you cannot be the leader*). In that case, wait for the ***leader*** message (*I am the leader*). Otherwise elect yourself the leader and send a ***leader*** message
3. If ***no reply is received***, then elect yourself the leader, and broadcast a ***leader*** message.
4. If you receive a reply, but later don't receive a ***leader*** message from a process of larger id (i.e the leader-elect has crashed), then re-initiate election by sending ***election*** message.

Bully algorithm



Node 0 sends N-1 **election** messages
Node 1 sends N-2 **election** messages
Node N-2 sends 1 **election** messages etc

So, 0 starts all over again

Finally, node N-2 will be elected leader, but
before it sent the **leader** message, it crashed.

The worst-case message complexity = **$O(n^3)$** (This is bad)

Maxima finding on a unidirectional ring

Chang-Roberts algorithm.

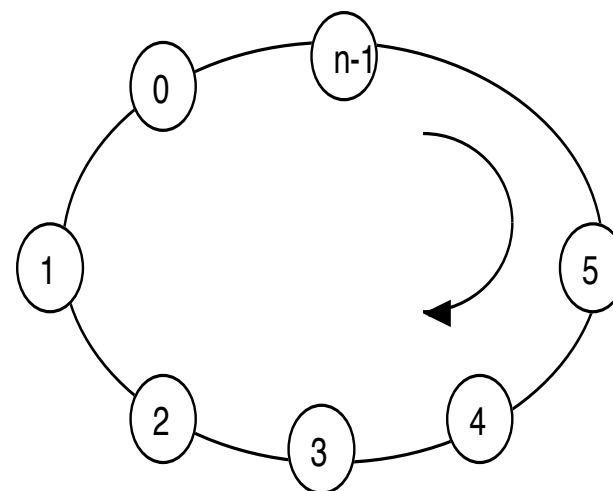
Initially all initiator processes are **red**.

Each initiator process i sends out **token $\langle i \rangle$**

```
{For each initiator  $i$ }  
do token  $\langle j \rangle$  received  $\wedge j < i \rightarrow$  skip (do nothing)  
   token  $\langle j \rangle \wedge j > i \rightarrow$  send token  $\langle j \rangle$ ; color := black  
   token  $\langle j \rangle \wedge j = i \rightarrow L(i) := i$  { $i$  becomes the leader}  
od  
{Non-initiators remain black, and act as routers}  
do token  $\langle j \rangle$  received  $\rightarrow$  send  $\langle j \rangle$  od
```

Message complexity = $O(n^2)$. Why?

What are the best and the worst cases?



The ids may not be nicely ordered like this

Bidirectional ring

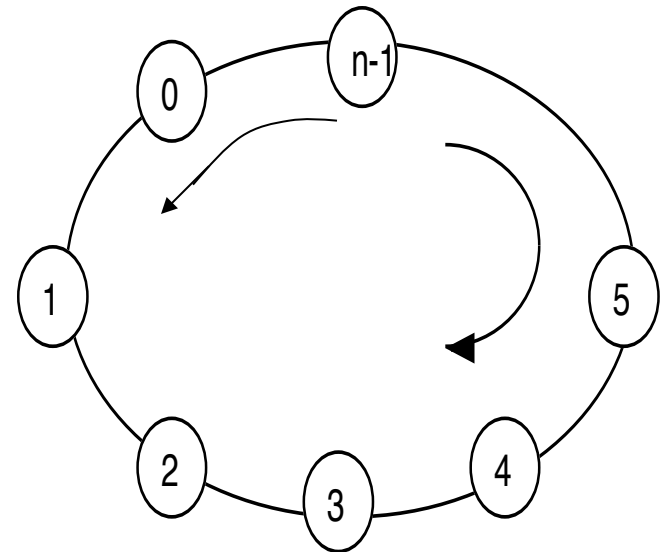
Franklin's algorithm (round based)

In each round, every process sends out *probes (same as tokens)* in **both** directions to its neighbors.

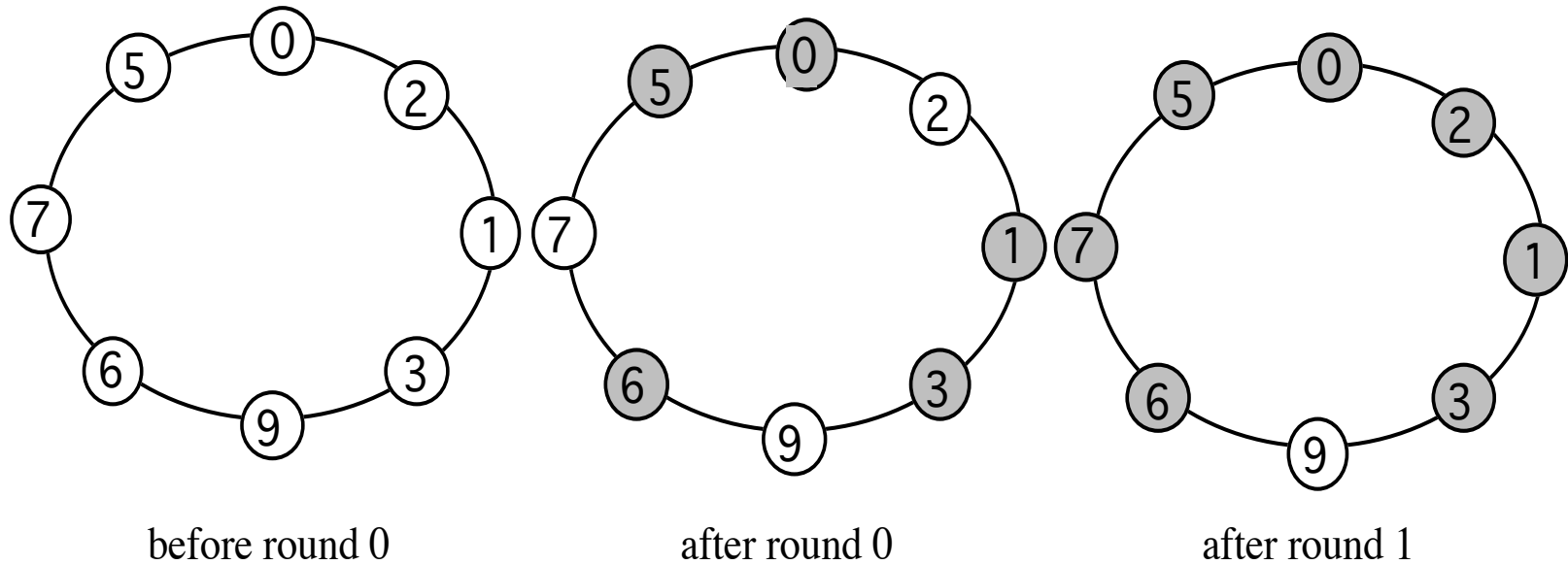
Probes from higher numbered processes will knock the lower numbered processes out of competition.

In each round, out of two neighbors, **at least one must quit**. So at least $1/2$ of the current contenders will quit.

Message complexity = $O(n \log n)$. Why?



Sample execution



Peterson's algorithm

initially $\forall i : \text{color}(i) = \text{red}, \text{alias}(i) = i$

{program for each round and for each red process}

send **alias**; receive **alias (N)**;

if alias = alias (N) \rightarrow **I am the leader**

alias \neq alias (N) \rightarrow send **alias(N)**; receive **alias(NN)**;

if alias(N) > max (alias, alias (NN)) \rightarrow alias := alias (N)

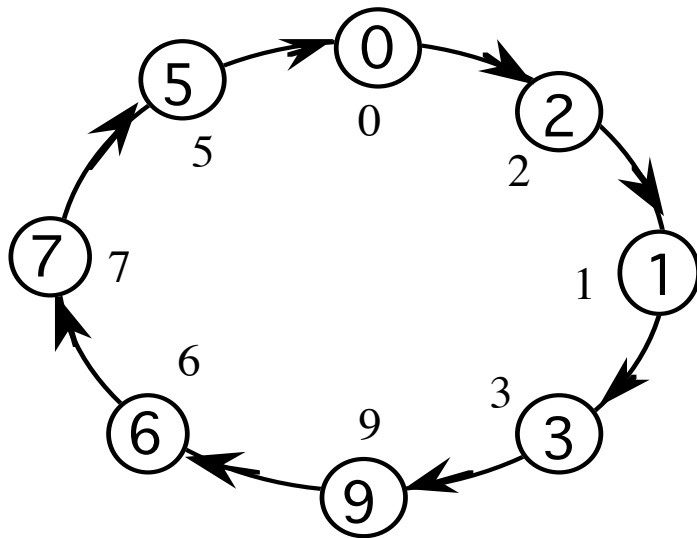
alias(N) < max (alias, alias (NN)) \rightarrow color := **black**

fi

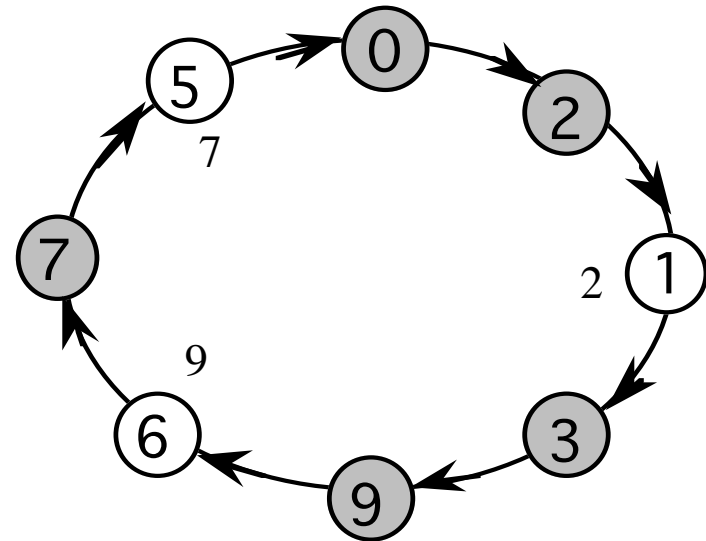
fi

{N(i) and NN(i) denote *neighbor* and *neighbor's neighbor* of i}

Peterson's algorithm



before round 0



after round 0

Round-based. Finds maxima on a **unidirectional ring** using $O(n \log n)$ messages. Uses an ***id*** and an ***alias*** for each process.

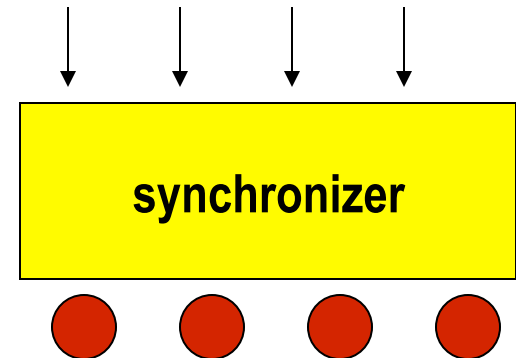
Synchronizers

Synchronous algorithms (round-based, where processes execute actions in lock-step synchrony) are easier to deal with than **asynchronous algorithms**. In each **round** (or **clock tick**), a process

- (1) receives messages from neighbors,
- (2) performs local computation
- (3) sends messages to ≥ 0 neighbors

A **synchronizer** is a protocol that enables synchronous algorithms to run on an asynchronous system.

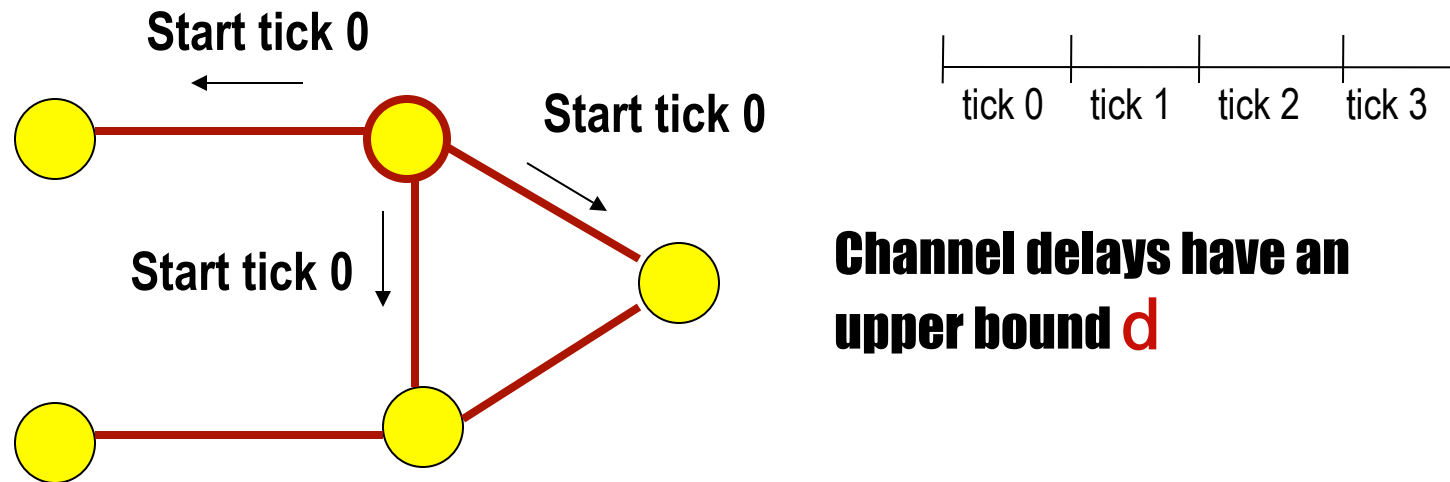
Synchronous algorithm



Asynchronous system

Synchronizers

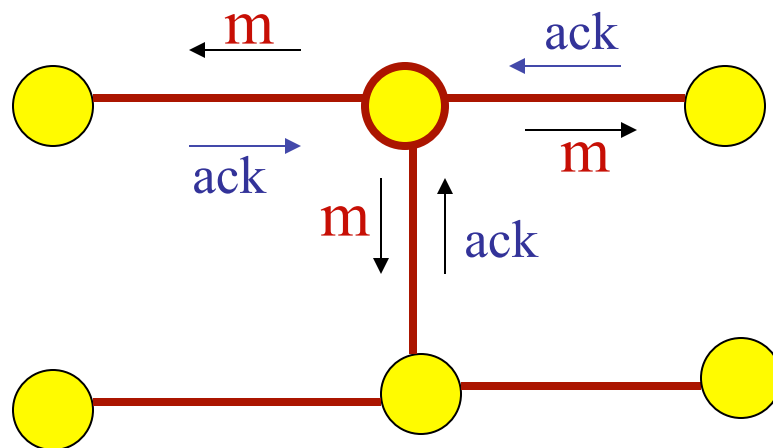
“Every message sent in *clock tick k* must be received by the neighbors in the *clock tick k* .” This is not automatic - some extra effort is needed.
Consider a basic *Asynchronous Bounded Delay (ABD)* synchronizer



Each process will *start the simulation of a new clock tick after $2d$ time units*, where d is the maximum propagation delay of each channel

α -synchronizers

What if the propagation delay is arbitrarily large but finite?
The α -synchronizer can handle this.



Simulation of each
clock tick

1. Send and receive **messages** for the current tick.
2. Send **ack** for each incoming message, and receive **ack** for each outgoing message
3. Send a **safe message** to each neighbor after sending and receiving all **ack** messages (then follow steps 1-2-3-1-2-3- ...)

Complexity of α -synchronizer

Message complexity $M(\alpha)$

Defined as the number of messages passed around the *entire network* for the simulation of each clock tick.

$$M(\alpha) = O(|E|)$$

Time complexity $T(\alpha)$

Defined as the number of *asynchronous rounds* needed for the simulation of each clock tick.

$$T(\alpha) = 3$$

(since each process exchanges *m, ack, safe*)

Complexity of α -synchronizer

$$M_A = M_S + T_S \cdot M(\alpha)$$

MESSAGE complexity
of the algorithm
implemented on top of the
asynchronous platform

Message complexity
of the original synchronous
algorithm

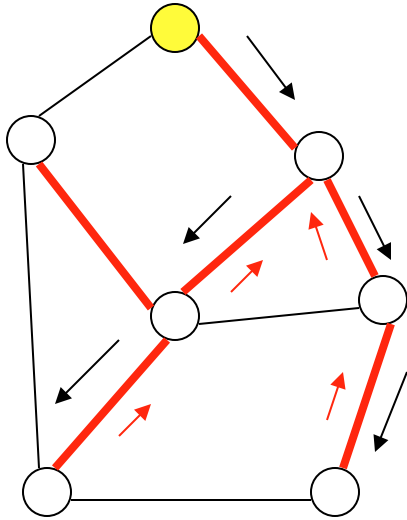
Time complexity
of the original synchronous
algorithm in rounds

$$T_A = T_S \cdot T(\alpha)$$

TIME complexity
of the algorithm
implemented on top of the
asynchronous platform

Time complexity
of the original synchronous
algorithm

The β -synchronizer



Form a *spanning tree* with any node as the root. The **root** initiates the simulation of each tick by sending message $m(j)$ for each clock **tick** j . Each process responds with $ack(j)$ and then with a $safe(j)$ message **along the tree edges** (that represents the fact that the entire subtree under it is safe). When the root receives $safe(j)$ from every child, it initiates the simulation of clock tick $(j+1)$ using a **next** message.

*To compute the message complexity $M(\beta)$, note that in each simulated tick, there are m messages of the original algorithm, m acks, and $(N-1)$ **safe** messages and $(N-1)$ **next** messages along the tree edges.*

*Time complexity $T(\beta)$ = depth of the tree.
For a balanced tree, this is $O(\log N)$*

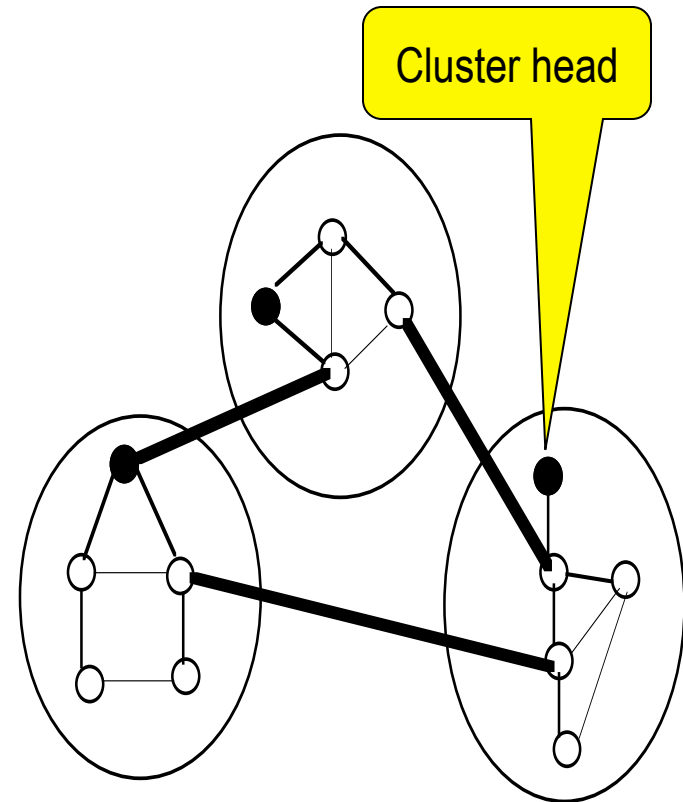
γ -synchronizer

Uses the best features of both α and β synchronizers. (*What are these?*)*

The network is viewed as a tree of clusters. Each cluster has a cluster-head. Within each cluster, β -synchronizers are used, but for inter-cluster synchronization, α -synchronizer is used

Preprocessing overhead for cluster formation.

The number and the size of the clusters is a crucial issue in reducing the message and time complexities



* α -synch has lower time complexity, β -synchronizers have lower message complexity

Example of application: Shortest path

- Consider Synchronous Bellman-Ford:
 - $O(n |E|)$ messages, $O(n)$ rounds
- Asynchronous Bellman-Ford
 - Many corrections possible (exponential), due to message delays.
 - Message complexity exponential in n .
 - Time complexity exponential in n , counting message pileups.
- Using (e.g.) Synchronizer α :
 - Behaves like Synchronous Bellman-Ford.
 - Avoids corrections due to message delays.
 - Still has corrections due to low-cost high-hop-count paths.
 - $O(n |E|)$ messages, $O(n)$ time
 - Big improvement.