MAD lessons…

Models & **A**lgorithms for **D**istributed algorithms/systems/…

Emmanuelle Anceaume, CNRS, Cidre team

Eric Fabre, INRIA, SuMo team

http://people.rennes.inria.fr/Eric.Fabre/

Overall contents

- Introduction to distributed systems/algos (today)
- Formal methods for distributed systems (4 lessons)
    - asynchrony, runs as partial orders of events
    - models for such systems, with true concurrency semantics
    - simple verification problems
- Distributed algorithms (5 lessons)
    - impossibility proofs : byzantine generals agreement
    - features of the BitCoin and of the BlockChain

What is it about ?

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

[Leslie Lamport]

Leslie Lamport
Turing Award 2013
for his contributions to distributed systems

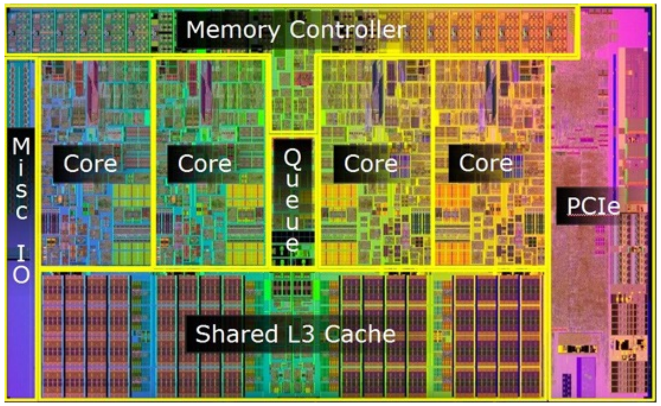
A distributed system is a multiprocessor system in which the time required for interprocess communication is large compared to the time for events within a single processor – in other words, it takes longer for interprocess communication than it does for a process to look at its own memory.

“they are everywhere…”

- Distributed software
  - visio-conference softwares
  - online businesses: comparing, selecting, paying, shipping
  - dynamic web page building: social networks, mashups,…
  - search engines: map-reduce
  - distributed data bases: concurrent access, consistency, atomicity of operations and of transactions
  - cooperative work: shared documents, joint editing, SVNs, GitHub
  - P2P: distributed storage, bitcoin
  - online multi-player games
  - …

"they are everywhere…"

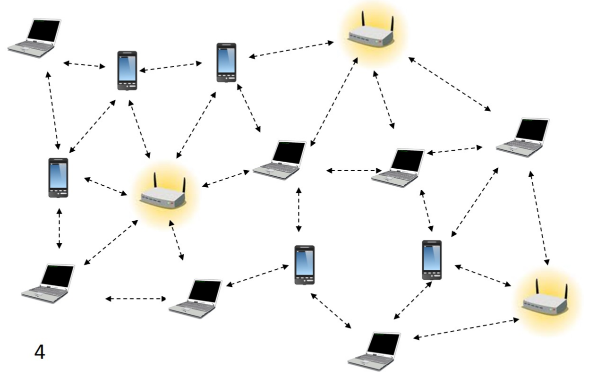
- modern OS
  - multi-core processors
  - multi-processors + asynchronous buses (ex. in cars, planes, …) : GALS
  - byte code produced by some compilers, introducing parallelism
  - multi-thread programming



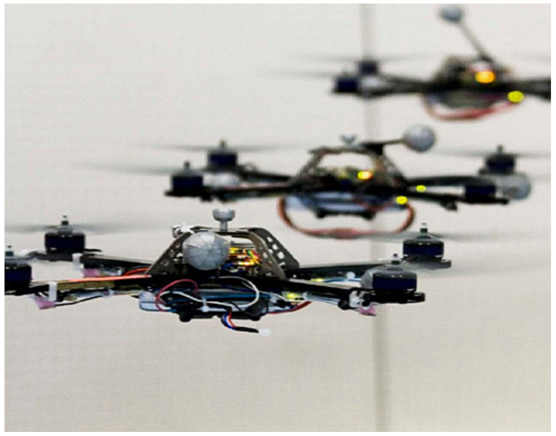
- system
  - computer grids
  - cloud computing, connected virtual machines, connected containers (Docker) : dynamicity
  - power grids

"they are everywhere…"

- telecommunication networks
  - routing algorithms (OSPF): dynamicity, resilience
  - cellular networks: access to shared resources
  - ad hoc networks: fully decentralized, dynamicity
  - delay tolerant networks
  - software defined networks
  - IoT: Internet of Things, connected objects
  - …



- physical systems
  - sensor networks
  - autonomous robot networks: exploration
  - UAV formation flight: consensus problems, distributed control
  - autonomous vehicles: cars, shuttles, trains, subways
  - IoT: massively distributed sensor networks
  - RFID networks…



"they are everywhere…"

- human organizations
  - distributed manufacturing
  - multi-agents organizations: hospital, crisis management ?
  - distributed data collection: Wikipedia, epidemy surveillance and reporting
  - social behaviors: gossip spreading, disease dissemination
  - non-human: ant/bee colonies
  - …

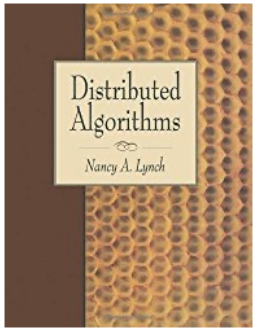
Interest/challenges for Computer Science

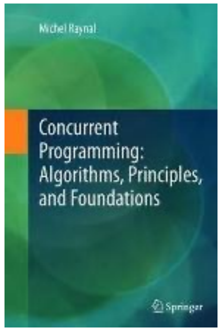
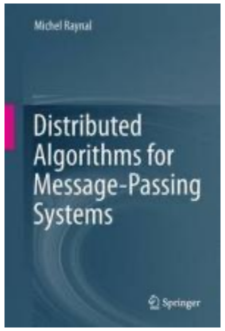
- why distributed systems ?
  - some problems are intrinsically distributed: in space, among agents
  - need to access shared resources (bandwidth, database)
  - need to join forces for a common goal
  - more efficiency/performance: computing power, storage, faster processing (parallelism)
  - better robustness: redundancy of resources, resilience to failures (storage, cloud computing), better performance/cost ratio
  - scalability: on demand performance, easy to up-scale, open to numerous users, distributed management

- challenges
  - fault tolerance, resilience, resistance to dynamicity
  - management (security, control, diagnosis, performance, upgrade…)
  - design (correctness/debug)
  - dedicated algorithms (impossibility results)

Who works on the topic ?

- distributed computing
  - architecture, cloud computing, networks, grids & HPC
- **distributed algorithms**
  - algorithmics focused on DS (= Distributed Systems) : feasibility, complexity
  - basic services : mutual exclusion, election, consensus…
  - higher level algorithms : gossip, routing, control, planning, learning…




Michel Raynal
Concurrent Programming: Algorithms, Principles, and Foundations
Springer


- distributed programming
  - languages and programming principles for DS
- **distributed systems**
  - how to build them : grids, clouds, SDN, multi-cores,…
  - **formal methods** : game theory, verification, concurrency models, test, diagnosis, control…

First models

Definition : a distributed system is a network of autonomous machines/softwares/processes/entities, with a coordination middleware, designed to behave as a single machine

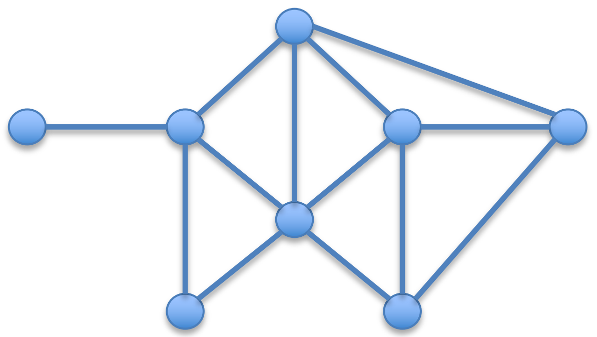
Features :

- no global time, only local time in components
- communication time >> local computing time
- often represented as an undirected graph G=(V,E)
- nodes/machines : generally infinite state machines, can crash, join, leave, lie (malicious)
- comm. by messages, via channels (FIFO/LIFO, (un)bounded in time and size, lossless or not, reliable or not…)
- nodes generally have a local knowledge of the system topology (i.e. ports to neighbors)

Simple model: a finite undirected graph, nodes/vertices are processes, edges are bidirectional communication channels

The topology is fixed, but arbitrary, locally known (each node knows how many neighbors it has).

Messages are reliably transmitted through FIFO channels, with delay.



centralized setting	**distributed setting**
1 component	several components
1 control point	several control points
1 process	several processes, messages
simple failures	complex/multiple failures
synchronous = global clock	asynchronous = no global time
closed/fixed architecture	open/changing architecture

Some basic services to achieve

- election
  - nodes must agree on one of them as their leader
- consensus
  - each node proposes a value ; nodes must agree on one of these values
- mutual exclusion
  - no two nodes can perform some dangerous action at the same time
  - all of them get equal chance to perform this action
- recovering event causality
  - identify causal past of an event ; identify possible reorderings
- definition of snapshots
  - build recovery points for distributed computations (failure protection)
- atomic broadcast
  - guarantee that broadcasts are received in the correct order, despite crashes
- autostabilizing algorithms
  - converge to a desired property (ex. a single token in a ring)
- detection of stable properties
  - ex. detect that a distributed execution has terminated (no pending message)

Synchronous *vs* asynchronous

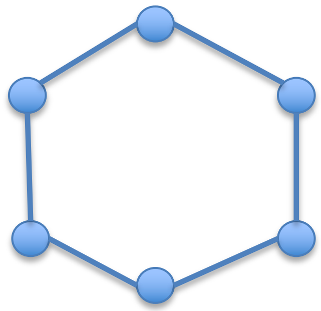
- **Asynchronous**
  - local computation time << communication time
  - no global clock
- **Synchronous**
  - communication time << computation time
    or communication time is bounded
  - global time can be simulated (but it is not always relevant)

In the synchronous case

- processes can wait for all messages to be received
- they can detect msg losses and process crashes
- computations are generally organized in **rounds**
  1. all processes compute
  2. then all processes exchange messages, and back to 1
- time complexity of an algo. is expressed in number of rounds

Synchronous algorithms

1. Leader election on a ring



assumptions

- processes run the same code
- local knowledge : nodes only see ports to their neighbors
- port numbers are symmetric by translation on the ring
- only one node must output “leader”, the other must output “non-leader” (in general, they should also know who is the leader)

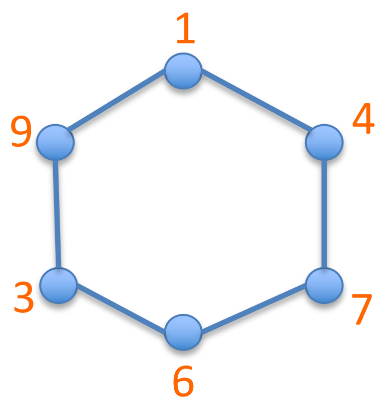
Thm : there exists no synchronous algorithm on a ring with n deterministic indistinguishable processes that solves the leader election problem.

Proof : by contradiction using symmetry

- the execution on each node is the same (same state reached, same messages sent, same reaction to messages)
- by symmetry, if node i outputs “leader”, then all nodes output “leader”

Solution : break the symmetry !

- add unique identifiers (UID) to nodes
- allow for randomness (with different seeds at each node)

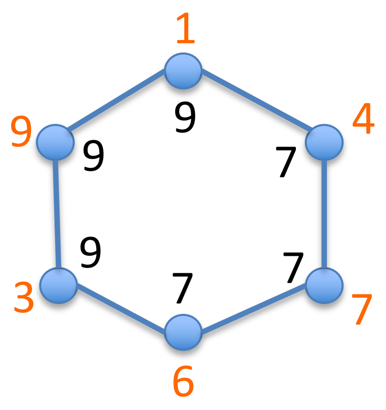


With unique identifiers

- each process sends its id. to neighbours
- each process receives ids from its neighbors, computes the maximal UID received so far, and propagates it to neighbors
- diffusion process
- Q : what is a good stopping criterion ?

Solution : break the symmetry !

- add unique identifiers (UID) to nodes
- allow for randomness (with different seeds at each node)

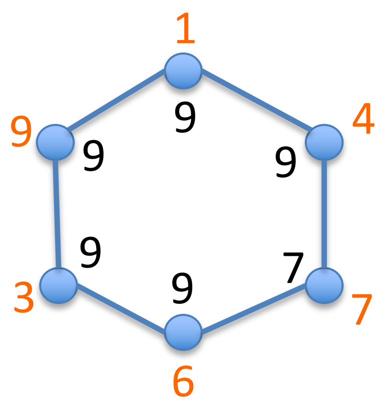


With unique identifiers

- each process sends its id. to neighbours
- each process receives ids from its neighbors, computes the maximal UID received so far, and propagates it to neighbors
- diffusion process
- Q : what is a good stopping criterion ?

Solution : break the symmetry !

- add unique identifiers (UID) to nodes
- allow for randomness (with different seeds at each node)

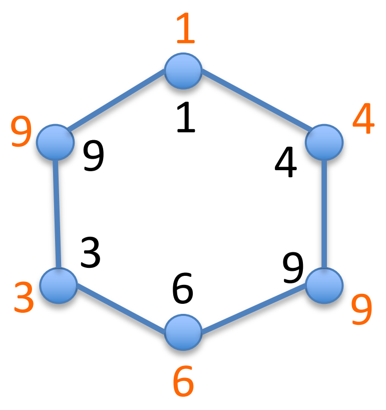


With unique identifiers

- each process sends its id. to neighbours
- each process receives ids from its neighbors, computes the maximal UID received so far, and propagates it to neighbors
- diffusion process
- Q : what is a good stopping criterion ?

Solution : break the symmetry !

- add unique identifiers (UID) to nodes
- allow for randomness (with different seeds at each node)



With unique identifiers

- each process sends its id. to neighbours
- each process receives ids from its neighbors, computes the maximal UID received so far, and propagates it to neighbors
- diffusion process
- Q : what is a good stopping criterion ?

With randomness

- each process draws an id. at random in set {1,2,…,r}
- then runs the previous algorithm
- Rem : seeds of the random number generators are also identifiers…

How many UIDs do we need ?

how large should be {1,2,…,r} compared to the number of nodes n ?

Lemma : with $r = n^2/\epsilon$, the probability that two processes out of n draw the same identifier is less than ϵ

Proof 1

nb of favorable cases = $r(r-1)...(r-n+1) = \frac{r!}{n!}$

total nb of cases = r^n

probability to win = $p = \frac{r}{r}\frac{(r-1)}{r}\dots\frac{(r-n+1)}{r} > \left(\frac{r-n}{r}\right)^n$

with $r = n^2/\epsilon$ one has $p > (1-\epsilon/n)^n = 1 - n\cdot\epsilon/n + ... > 1-\epsilon$

Proof 2

probability that two processes pick the same UID = 1/r

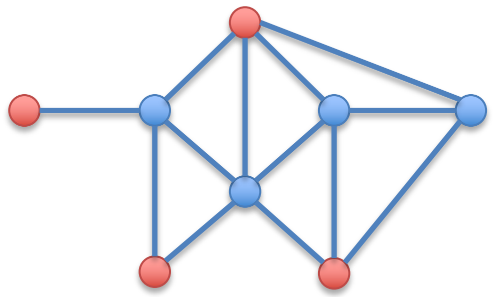
union bound : probability to lose < sum 1/r over all possible pairs

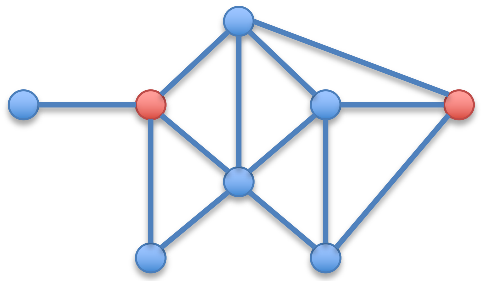
this is $\frac{n(n-1)}{2r} = \epsilon\frac{n-1}{2n} < \epsilon/2$

2. Maximal independence set

assumptions

- graph of n processes, not necessarily connected
- MIS = maximal subset of nodes that are pairwise disconnected
  [*maximal* = one can not add any extra node to the set,
  does not mean that the cardinality of the set is maximal.]
- processes don't have unique ids, so the problem is unsolvable in some graphs (see election thm) : we use randomness to break symmetry
- each process should output "**in**" or "**out**"





Lubi’s algorithm : alternation of 2 rounds, among active nodes

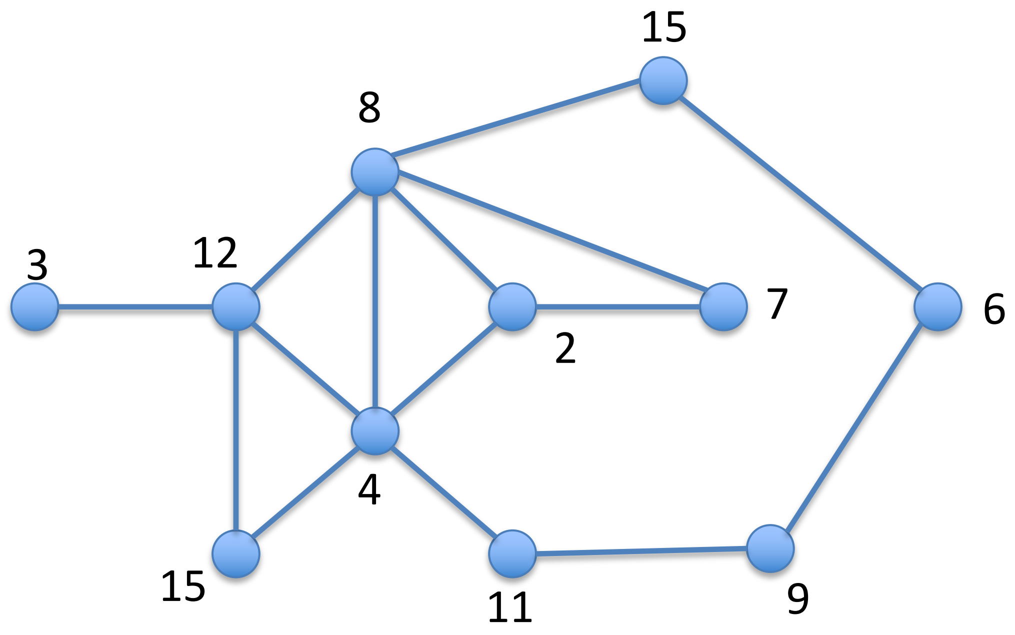
Round 1

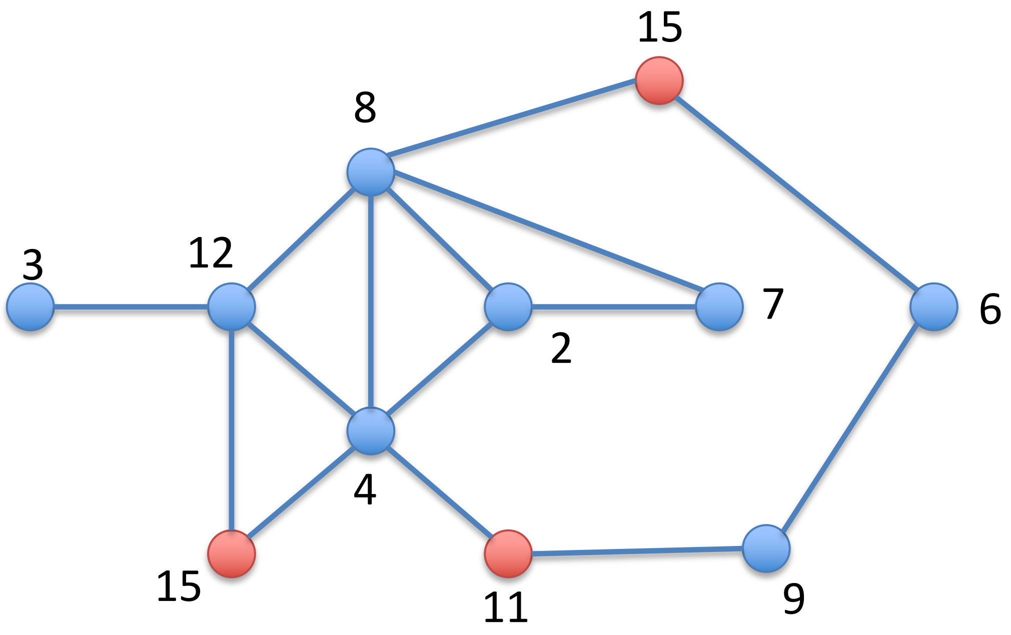
- each node u picks a number x(u) at random,
  then sends it to all its neighbors in N(u) [denotes the neighbor set]
- each node receives messages from all its neighbors
- if node u has the largest value : x(u) > x(v) for v in N(u)
  then it joins the MIS and outputs “in”

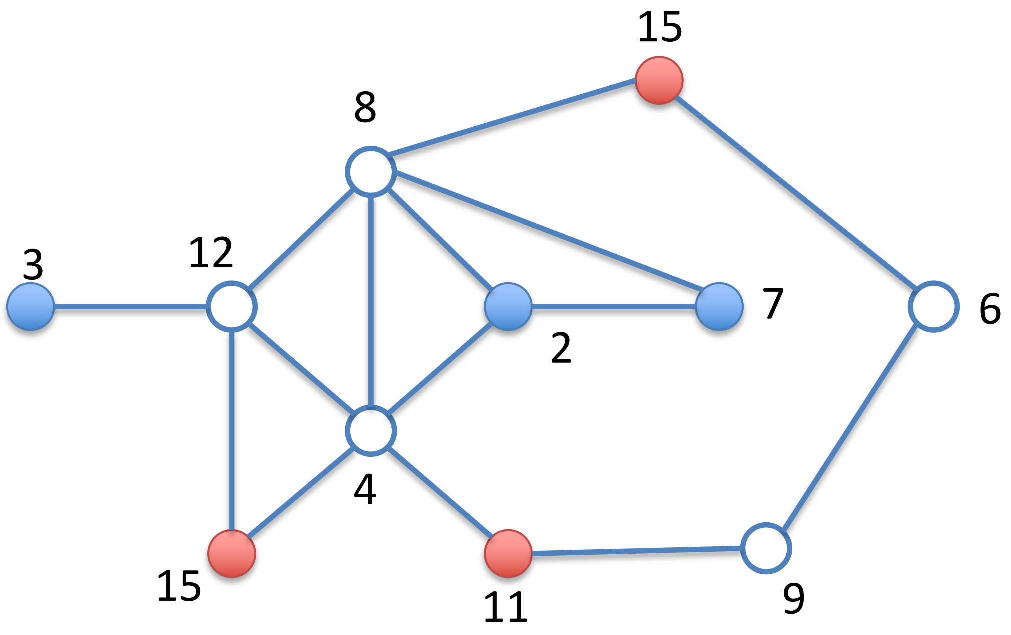
Round 2

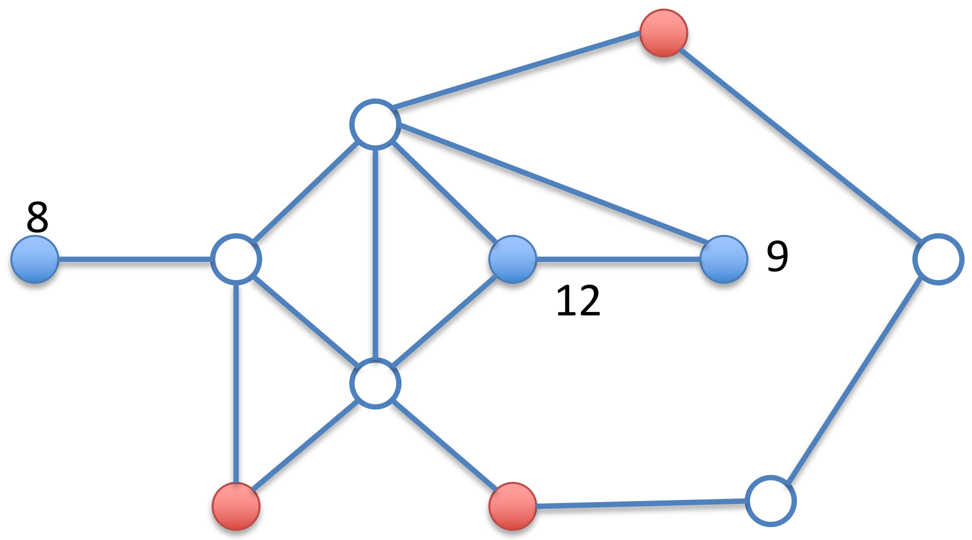
- each node u joining the MIS sends message “in” to its neighbors
- each node v receiving message “in” from a neighbor must not join the MIS, and outputs “out”
- each node having decided either “in” or “out” becomes inactive

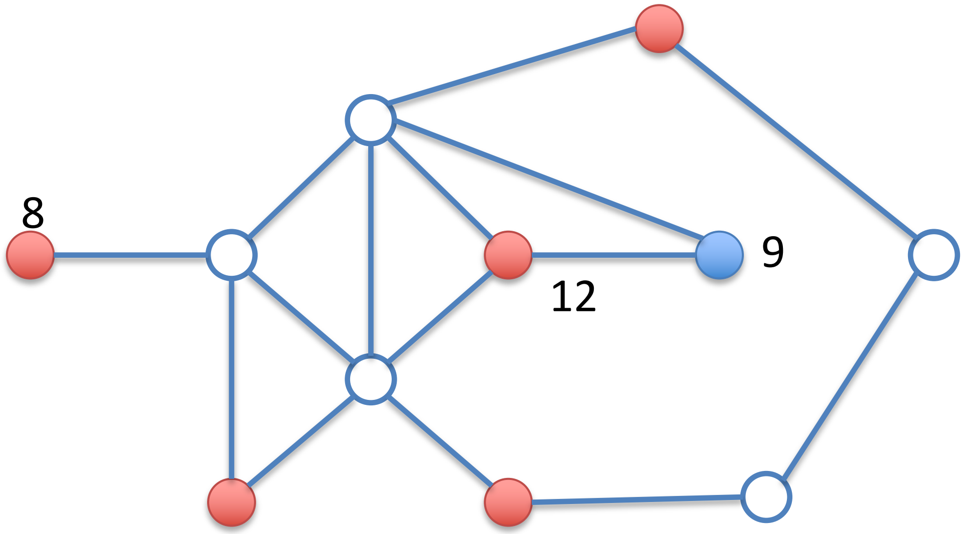
Rounds 1 and 2 proceed until all nodes are inactive (i.e. have decided whether they are in or out)

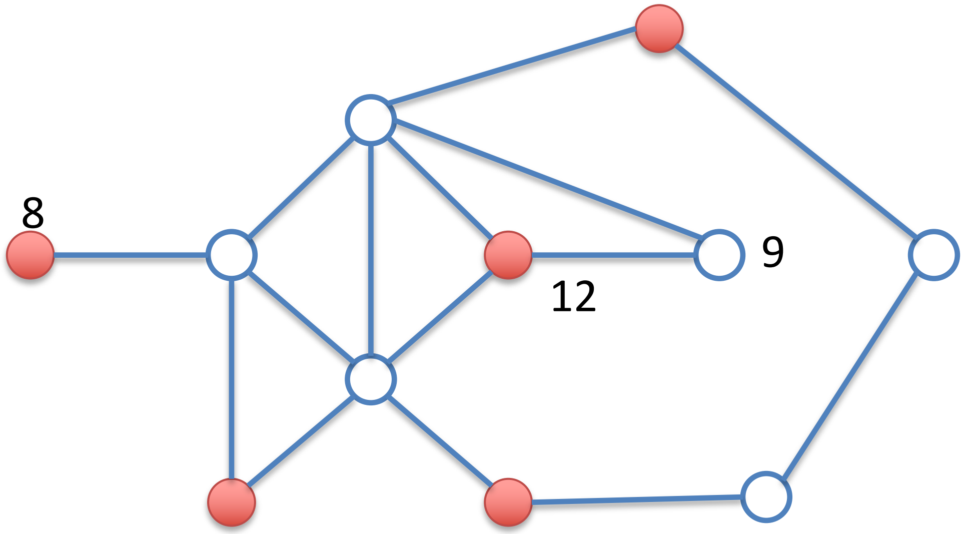

15
8
3
12
7
6
2
4
15
11
9


15
8
3
12
7
6
2
4
15
11
9


15
8
3
12
7
6
2
4
15
11
9


8
9
12


8
9
12


8
9
12

Proof of Lubi’s algorithm

ideas:

- find and prove invariants that are preserved by each round
- prove a monotony property (for convergence)

Invariant : the nodes that joined the MIS can’t be connected

Monotony : the number of active nodes decreases
Q : decreases strictly ?

A : yes, unless if two (or more) nodes pick the same maximal value, which is unlikely (see previous lemma). One can actually show that the average number of active edges is at least divided by 2 at each phase.

Maximality : only inactive nodes could be added to the current MIS, so if all nodes are inactive at termination, the MIS is maximal

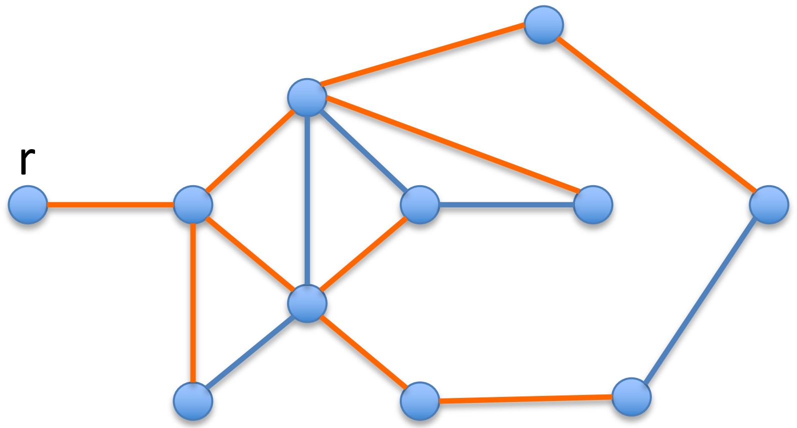
3. Spanning tree construction

assumptions

- finite connected graph G=(V,E), unknown size and topology
- a distinguished vertex r, called the root
- nodes have unique identifiers (UID)
- local knowledge of topology (nodes know their neighbors' UIDs)

goal

- processes must build a spanning tree, rooted at r
- the unique path from node r to each node v must be a shortest path (in number of edges)
- each node v (except r) should output parent(v), possibly children(v)



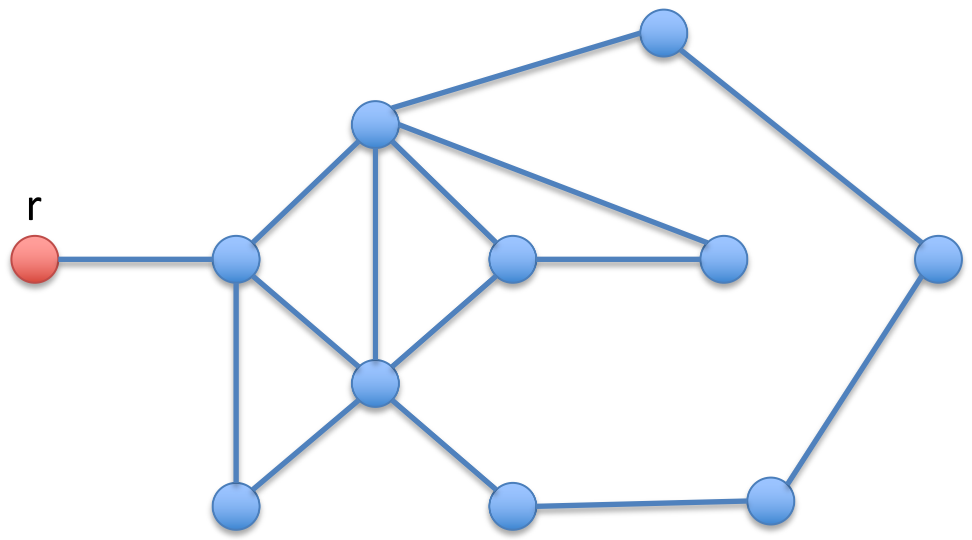
<u>Algorithm</u> : principle = diffusion from root node r

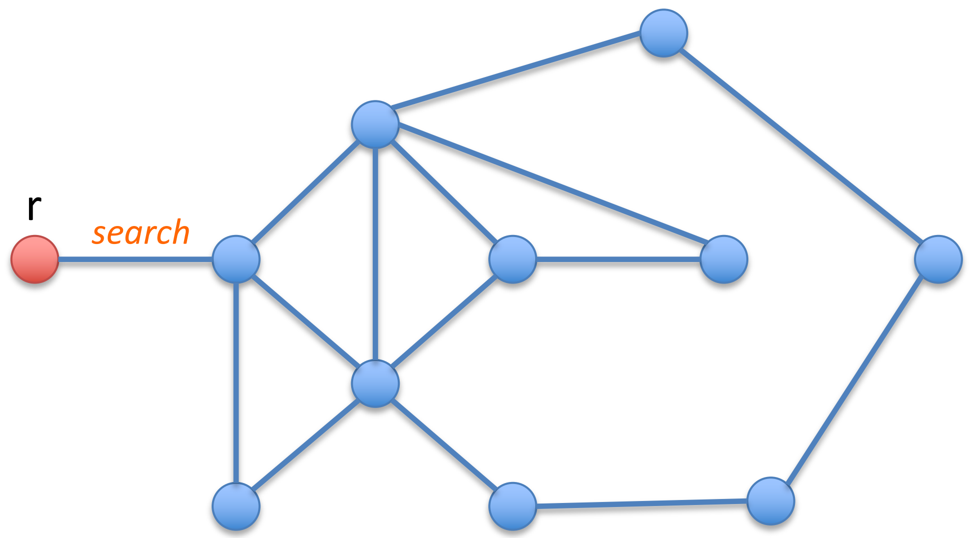
initialization

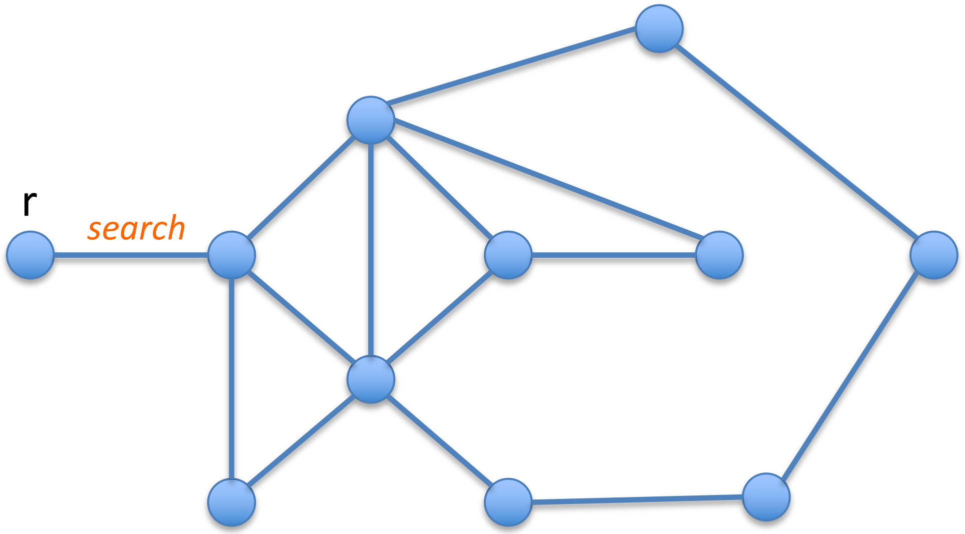
- all nodes inactive, except r
- $\forall v \in V, \quad \text{parent}(v) = \emptyset$

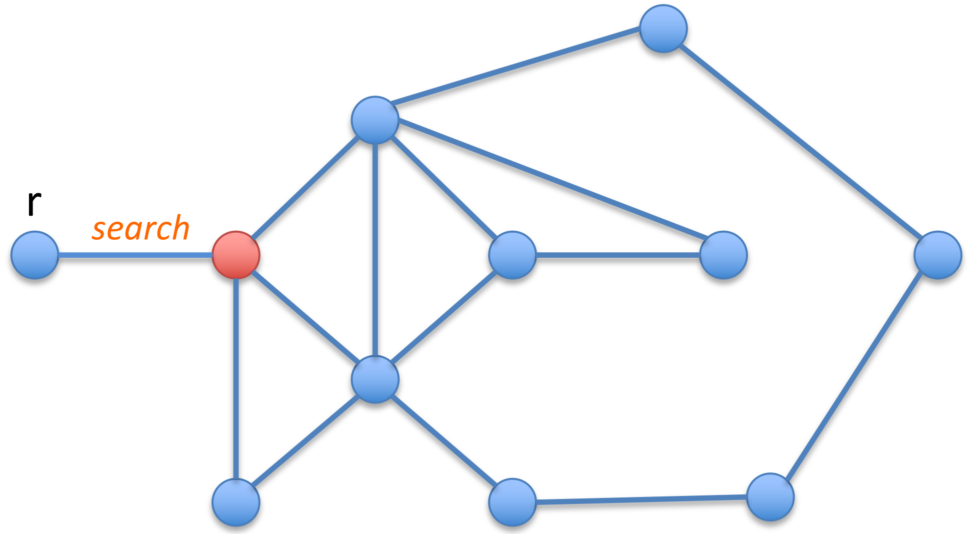
round k

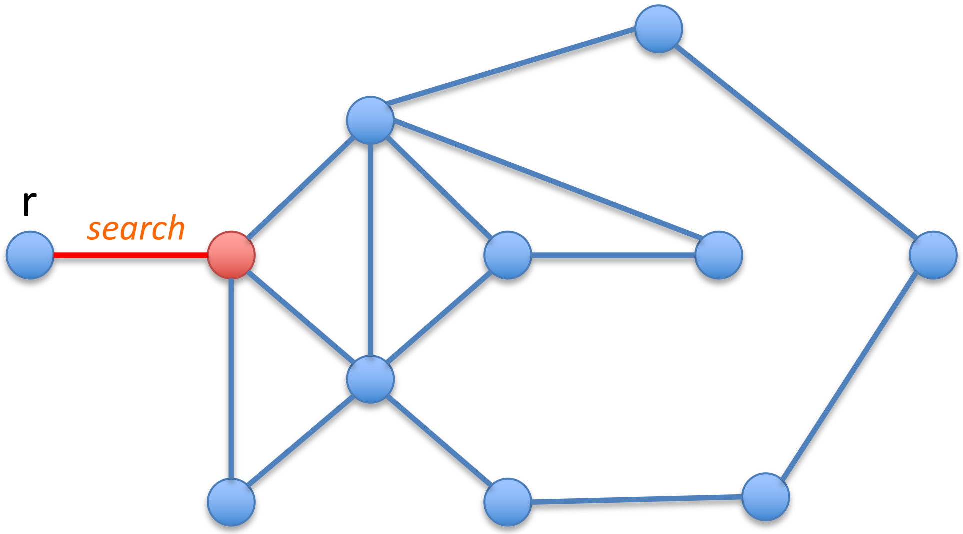
- if node v is active
  - v sends "search" message to all its neighbors, except to its parent
  - then v becomes inactive
- if node v receives one or more "search" messages and does not yet have a parent
  - v selects (non deterministically) one sender u and sets $\text{parent}(v) = u$
  - v may inform u that it is its child
  - v becomes active

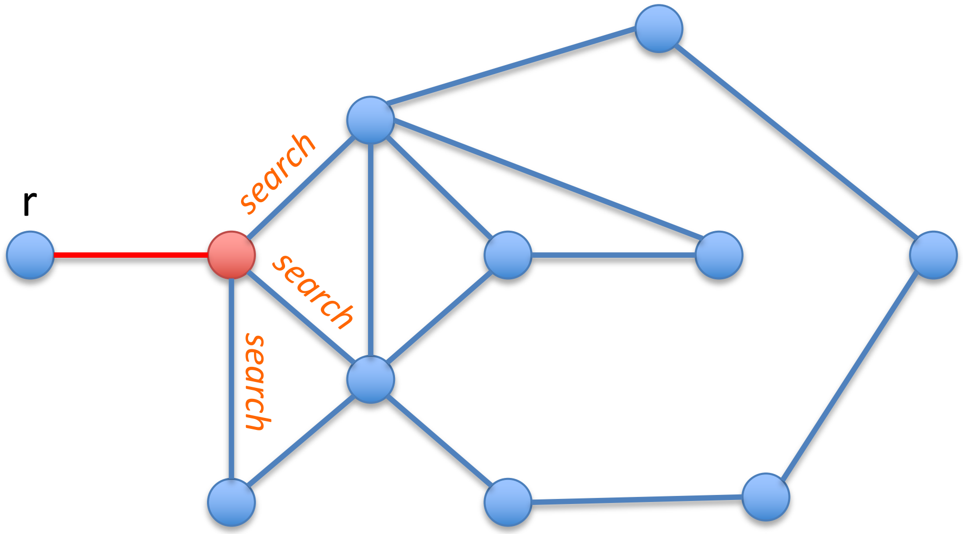

r

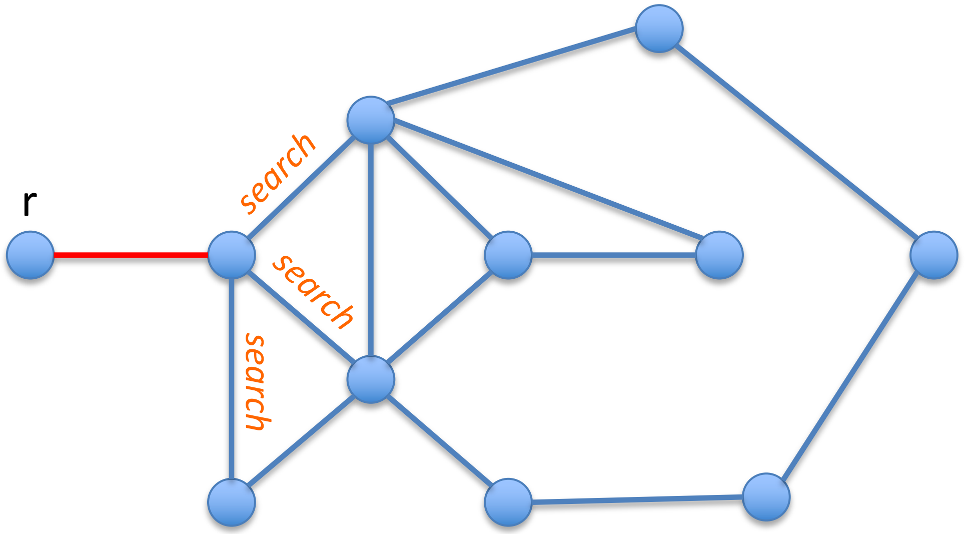

r
search

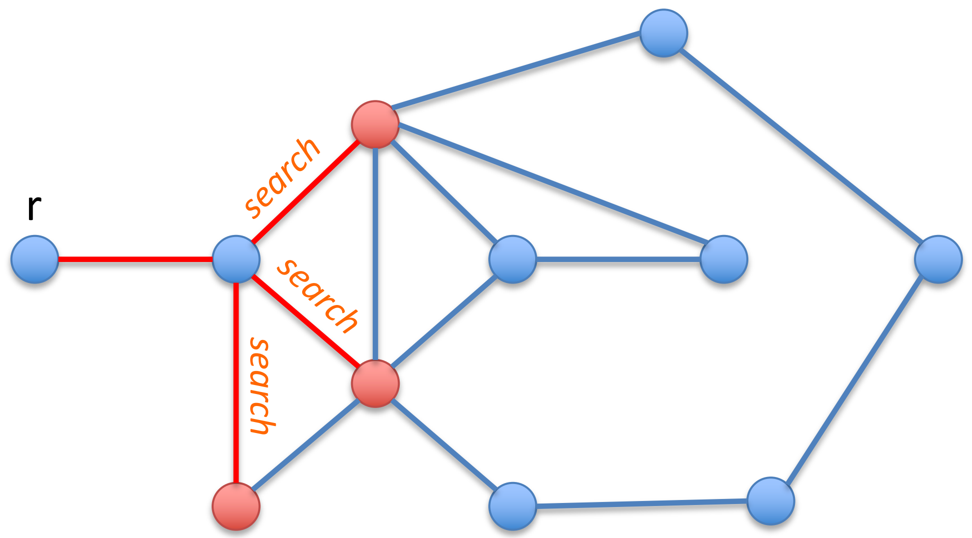

r
search

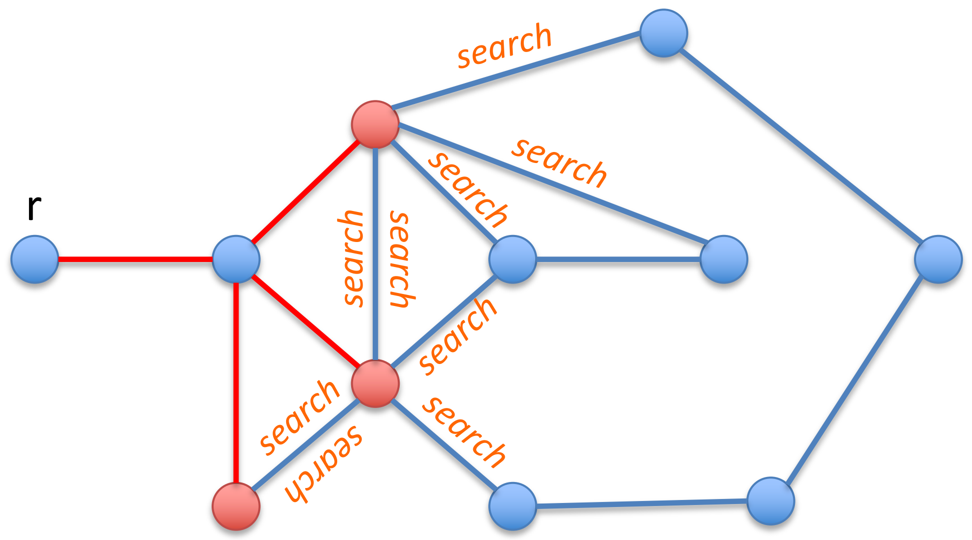

r
search

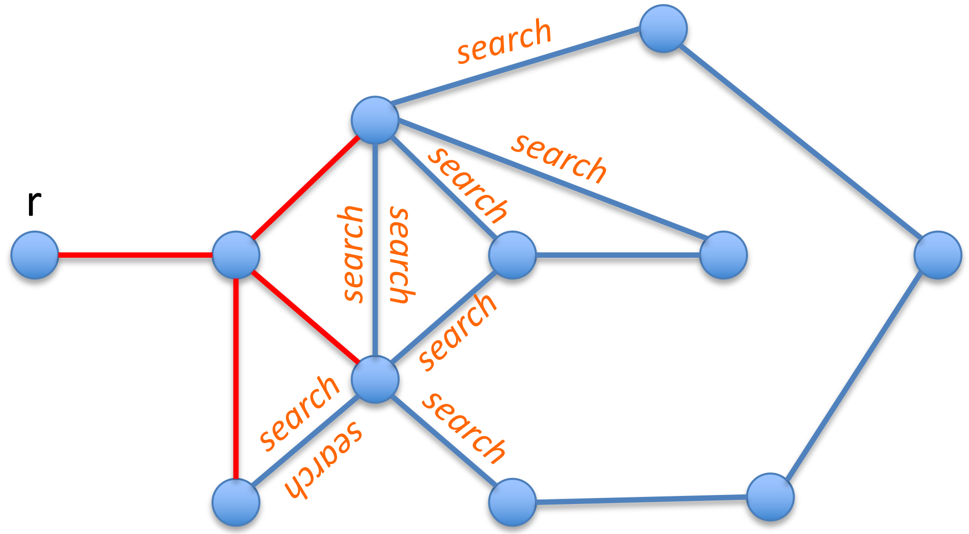

r
search

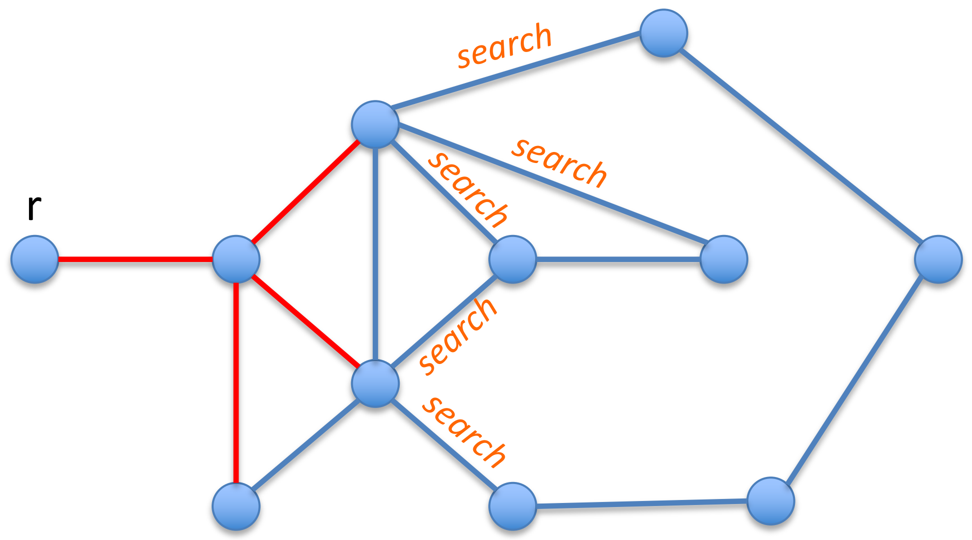

r
search
search
search

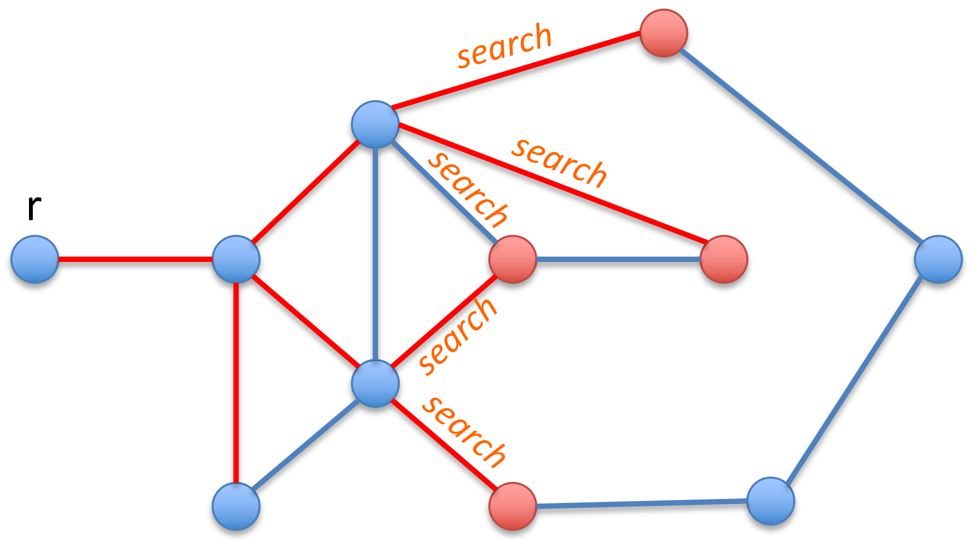

r
search
search
search

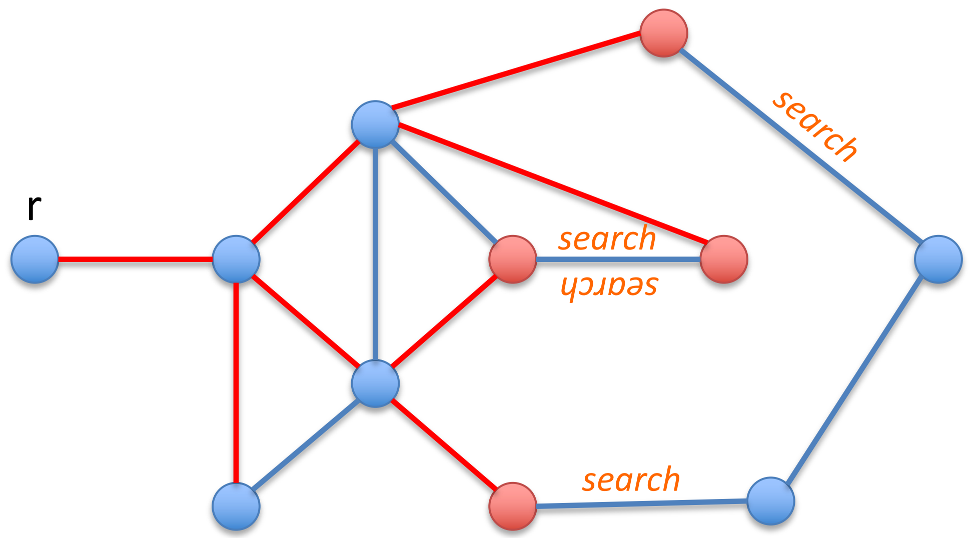

r
search
search
search

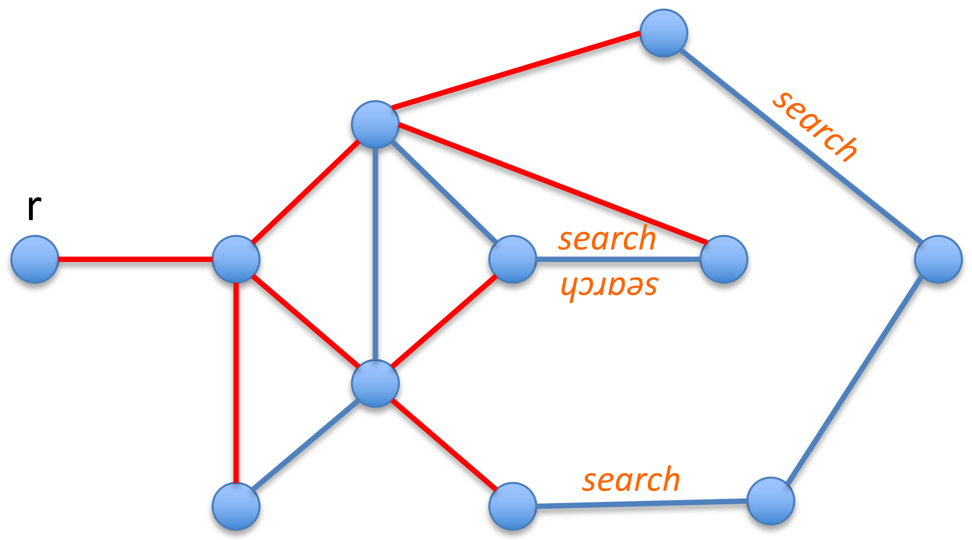

r
search
search
search
search
search
search
search
search
search

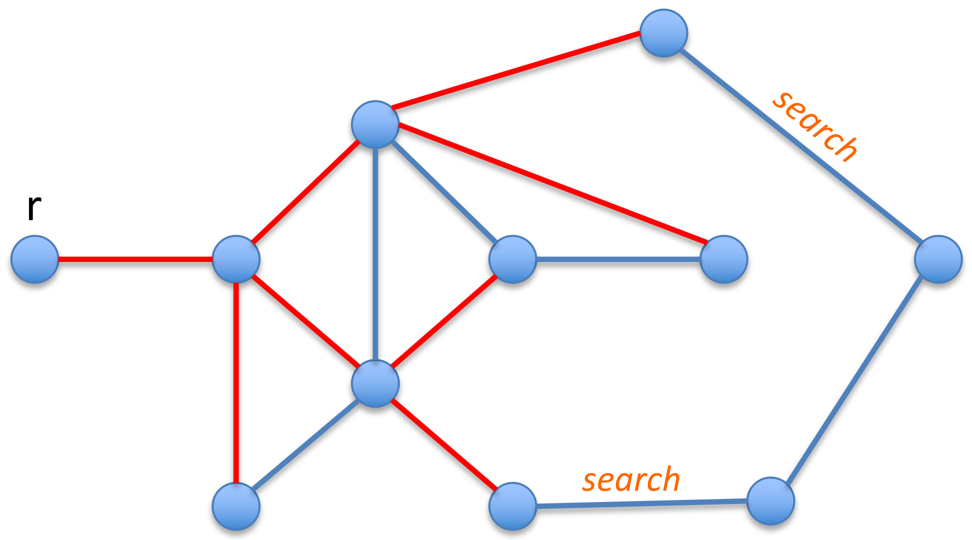

r
search
search
search
search
search
search
search
search
search

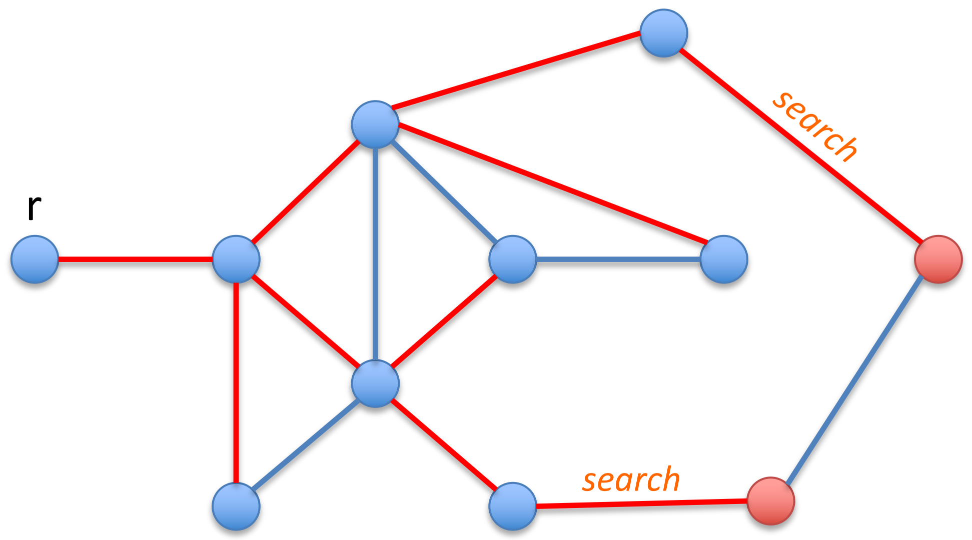

r
search
search
search
search
search

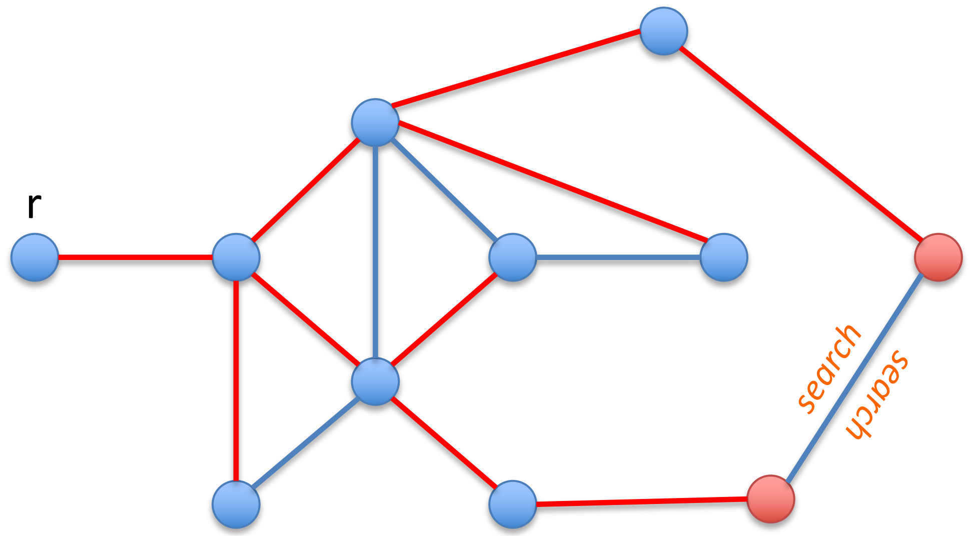

r
search
search
search
search
search

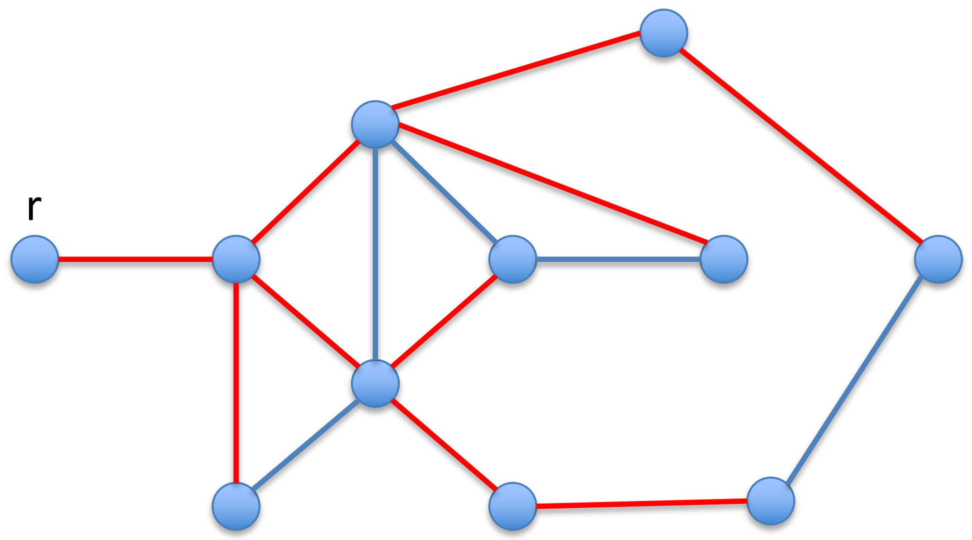

r
search
search
search
search


r
search
search
search
search


r
search
search


r
search
search


r
search
search


r

Algorithm : principle = diffusion from root node r

initialization

– all nodes inactive, except r
– $\forall v \in V, \quad \text{parent}(v) = \emptyset$

round k

- if node v is active
  - v sends “search” message to all its neighbors, except to its parent
  - then v becomes inactive
- if node v receives one or more “search” messages and does not yet have a parent
  - v selects (non deterministically) one sender u and sets $\text{parent}(v) = u$
  - v may inform u that it is its child
  - v becomes active

proof

invariant (+ monotony)

- the selected edges form a tree
- at round k, all nodes at distance k from root r have been reached and connected by a unique path of length k

Algorithm : principle = diffusion from root node r

initialization

– all nodes inactive, except r

– $\forall v \in V, \quad \mathrm{parent}(v) = \emptyset$

round k

- if node v is active
  - v sends "search" message to all its neighbors, except to its parent
  - then v becomes inactive
- if node v receives one or more "search" messages and does not yet have a parent
  - v selects (non deterministically) one sender u and sets $\mathrm{parent}(v) = u$
  - v may inform u that it is its child
  - v becomes active

Remarks

– non-determinism does not prevent correct behavior

– can compute as well the distance of each node to the root

– termination is easy to detect : back propagation from the leaves

- leaf nodes start telling their parent that they are "done"
- an inner node receiving a "done" message from ***all*** its children propagates "done" to its parent
- once the root node receives "done," it broadcasts it down the tree to all nodes

Algorithm : principle = diffusion from root node r

initialization

- all nodes inactive, except r
- $\forall v \in V, \quad \text{parent}(v) = \emptyset$

round k

- if node v is active
  - v sends "search" message to all its neighbors, except to its parent
  - then v becomes inactive
- if node v receives one or more "search" messages and does not yet have a parent
  - v selects (non deterministically) one sender u and sets $\text{parent}(v) = u$
  - v may inform u that it is its child
  - v becomes active

Extension : Bellman-Ford algorithm (dynamic programming)

- computes a shortest path tree rooted at r, when edges have a length
- search messages must carry as well a distance to r
- nodes receiving a shorter distance message get reactivated and change their parent one closer to r

Asynchronous algorithms

changes

- no rounds, much more non-determinism
- processes compute and communicate arbitrarily (atomic actions)
- correctness and convergence must be proved for a much larger and complex set of executions

usual setting

- G = (V,E) undirected graph, V=processes, E=bidirectional channels
- processes need not be distinguishable, know local ports to neighbours, (infinite) state machines
- communication by lossless FIFO channels (unbounded)
- $C_{u,v}$ is the channel (buffer) from u to v
- primitives $send_{u,v}(m)$, $receive_{u,v}(m)$

1. Spanning tree construction

assumptions

- finite connected graph G=(V,E), unknown size and topology
- a distinguished vertex r, called the root
- nodes have unique identifiers (UID)
- local knowledge of topology (nodes know their neighbors' UIDs)

goal

- processes must build a spanning tree, rooted at r
- the unique path from node r to each node v
  <u>is not requested anymore to be a shortest path </u>(in number of edges)
- each node v (except r) should output parent(v), possibly children(v)

1. Spanning tree construction

idea : take the previous synchronous algo, make it asynchronous !

Algorithm : principle = diffusion from root node r

initialization

– all nodes inactive, except r

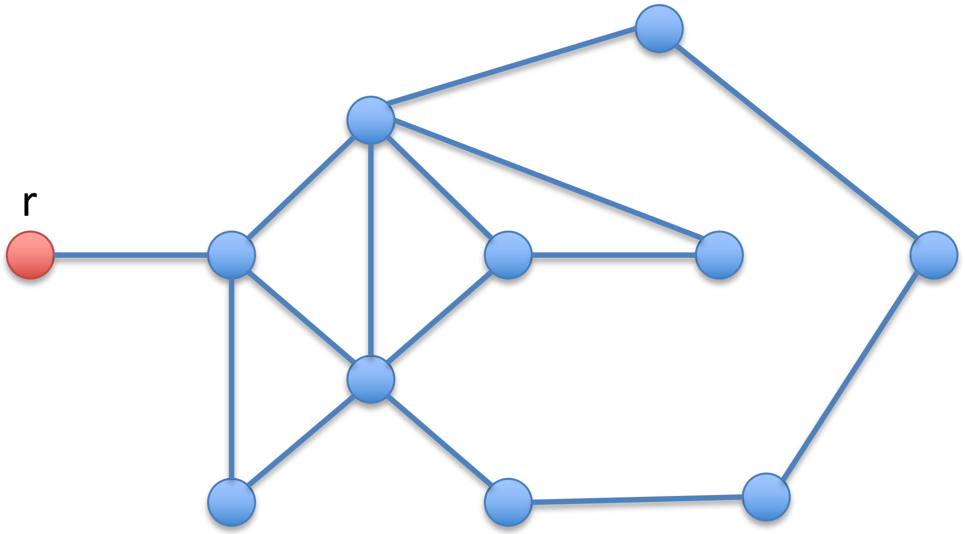
– $\forall v \in V, \quad \text{parent}(v) = \emptyset$

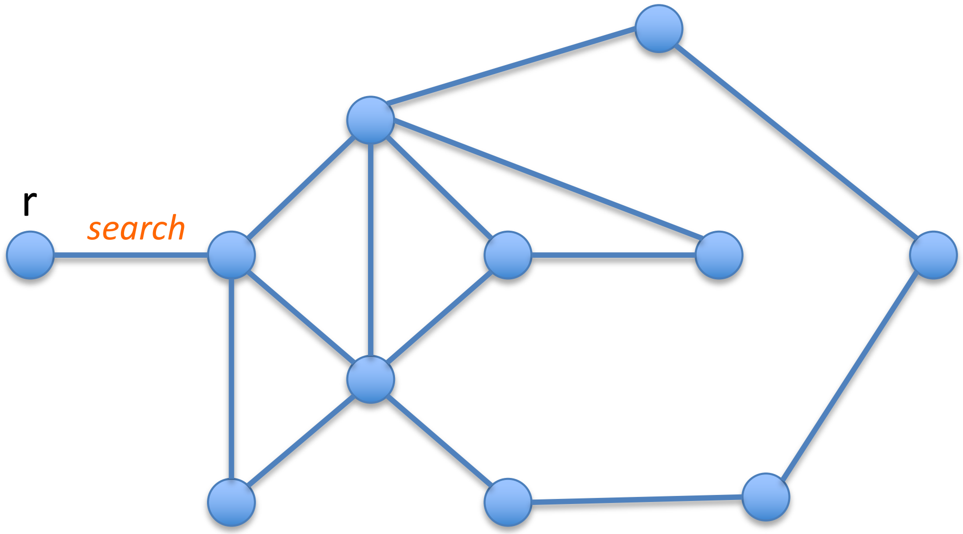
local step : do one of the two actions below

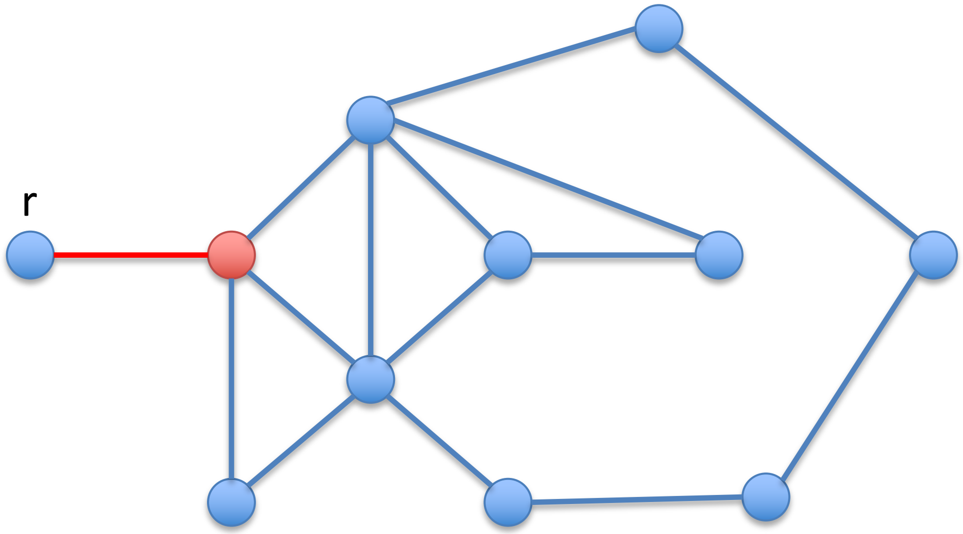
- emission from an active node v
  - v sends "search" message to all its neighbors, except to its parent
  - then v becomes inactive
- reception of a "search" message by node v from node u

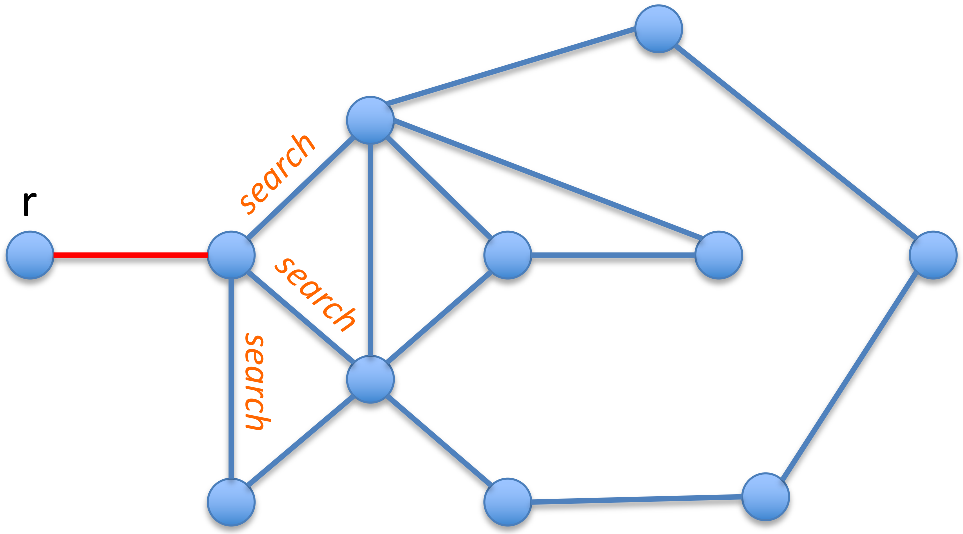
  if node v does not yet have a parent
  - v selects sets u as its parent : $\text{parent}(v) = u$
  - v may inform u that it is its child
  - v becomes active

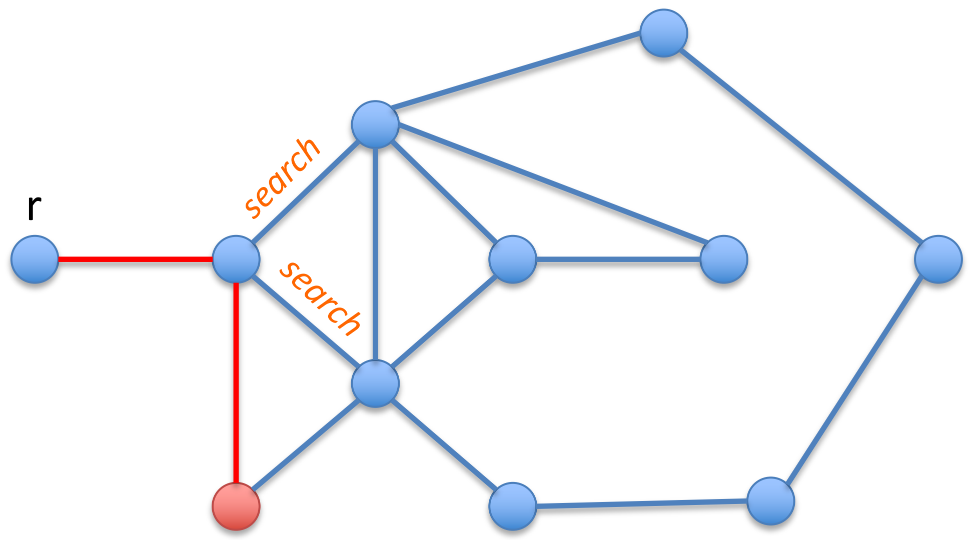
  otherwise do nothing

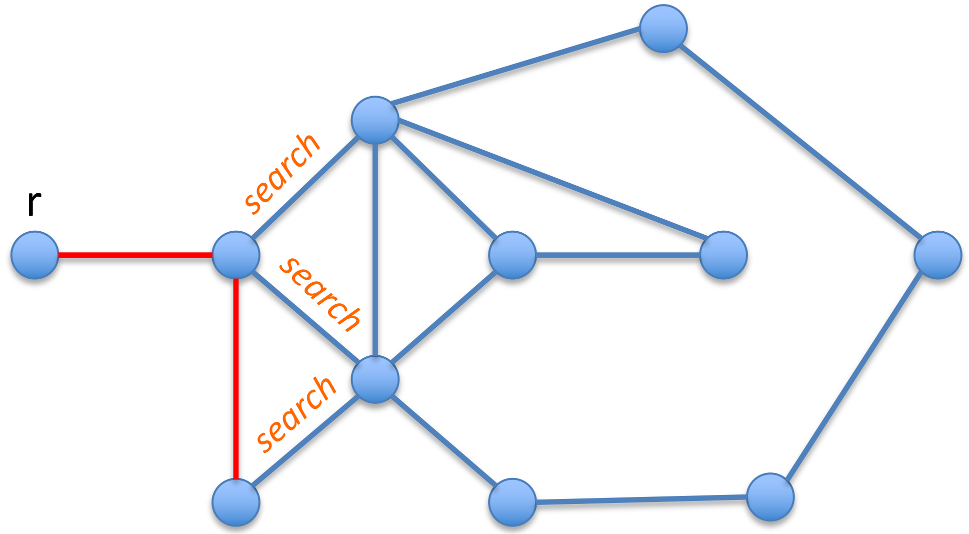

r

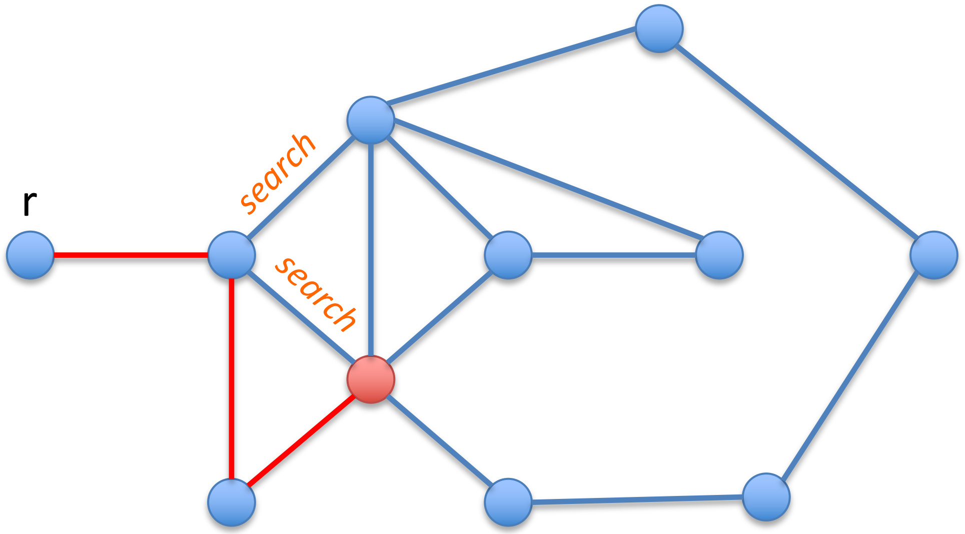

r
search

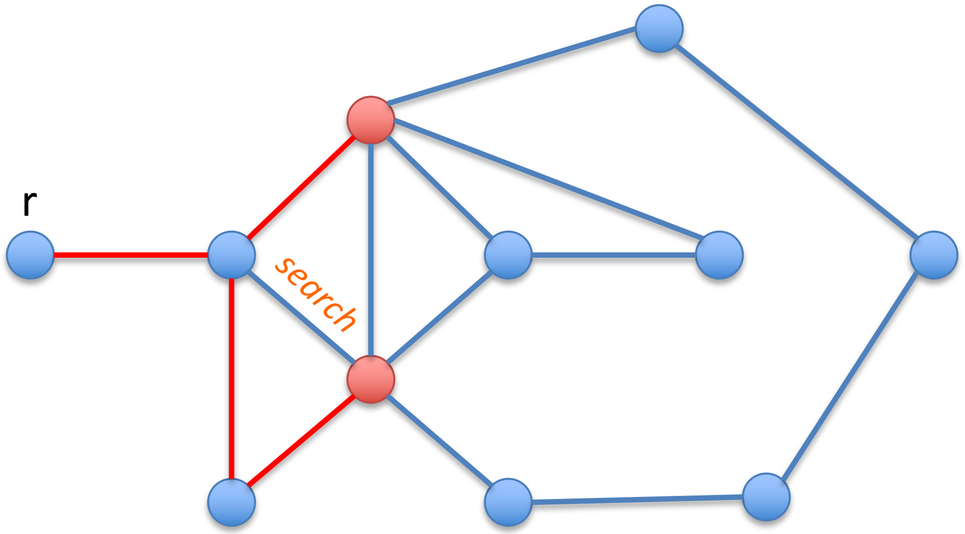

r

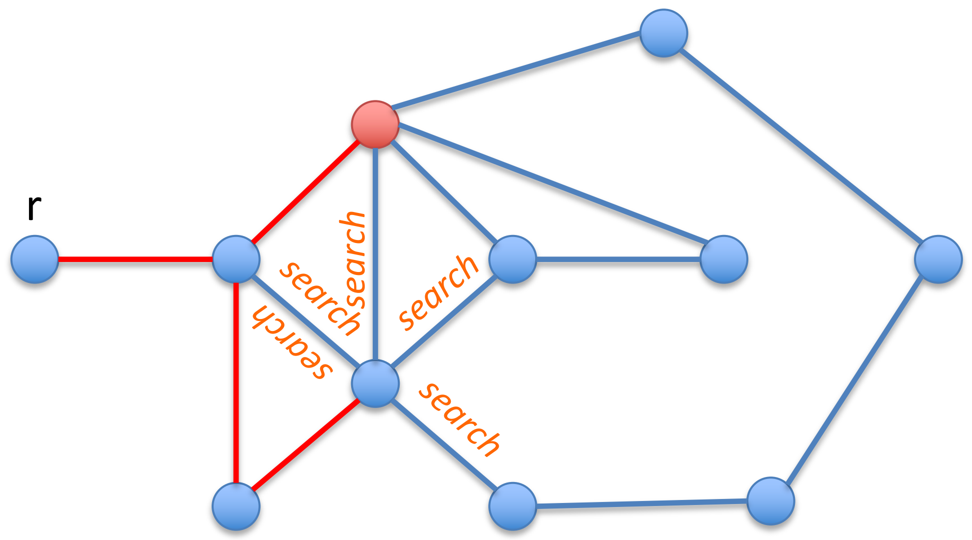

r
search
search
search

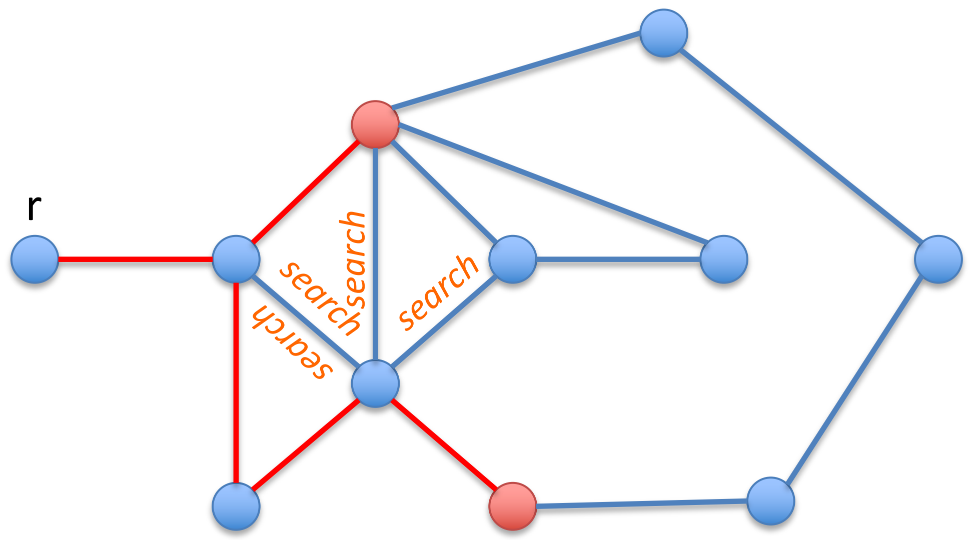

r
search
search

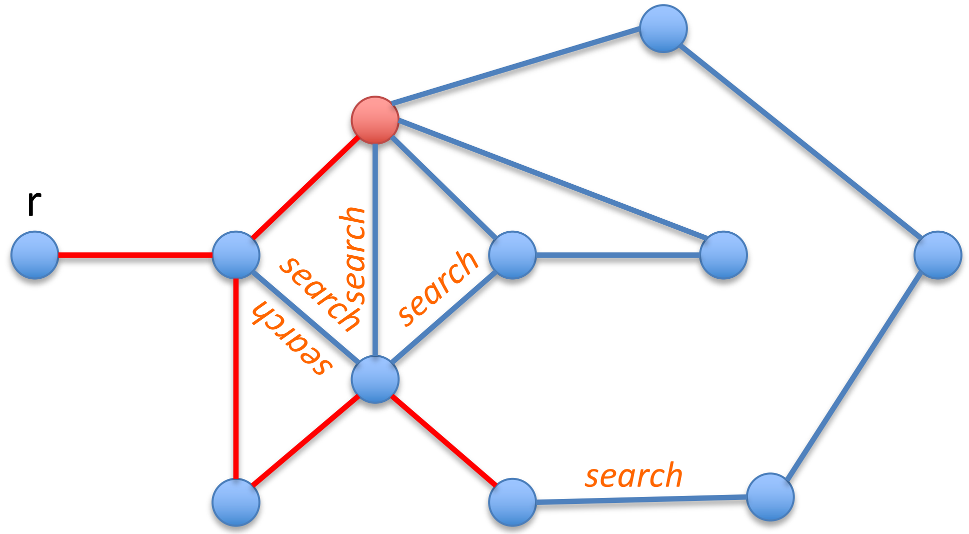

r
search
search
search

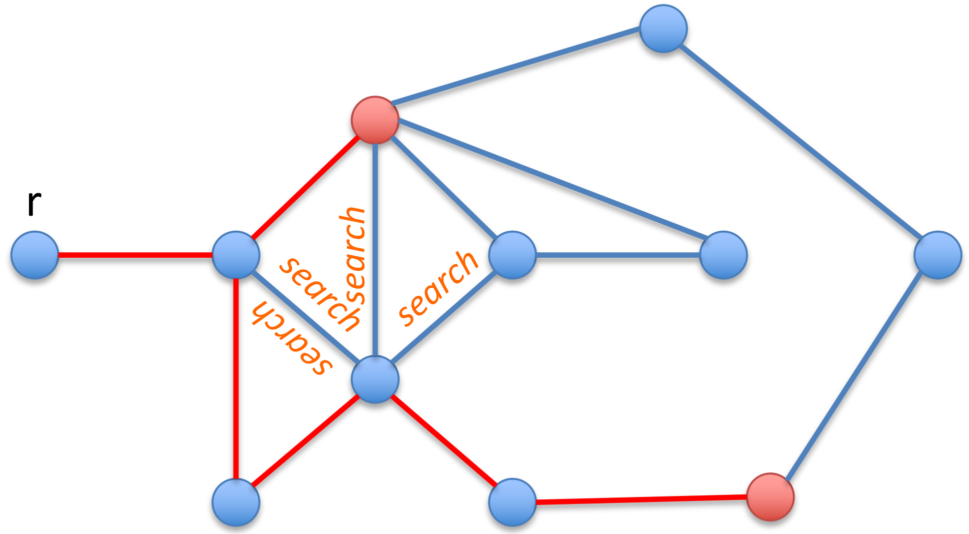

r
search
search

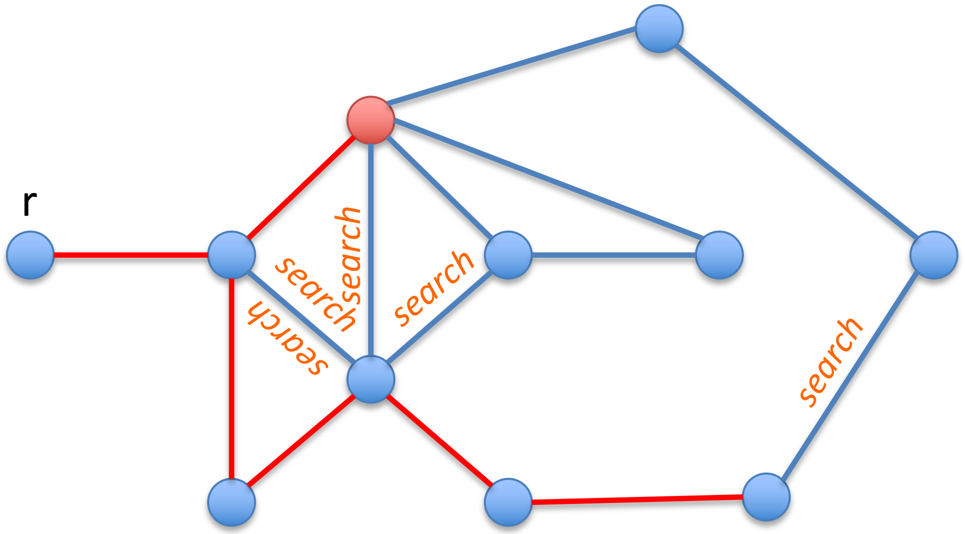

r
search

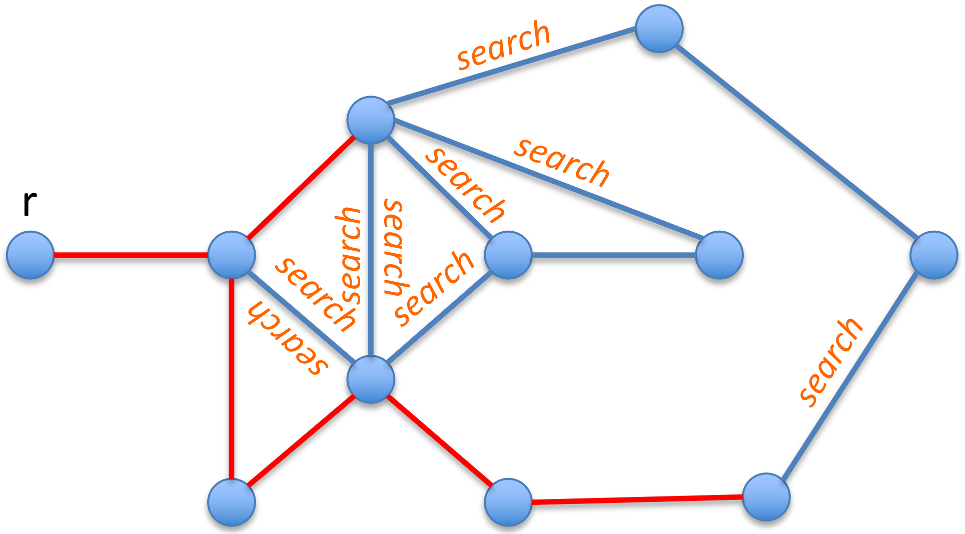

r
search
search
search
search
search

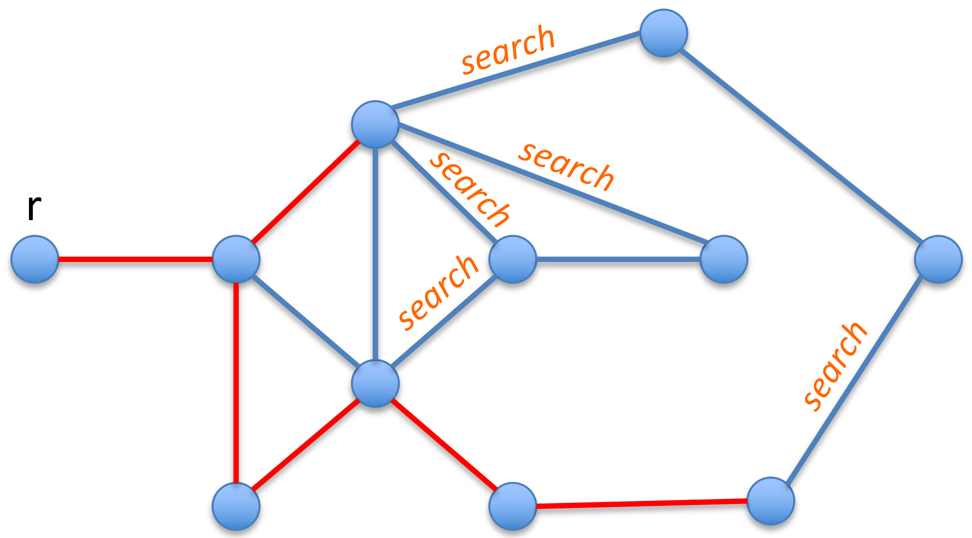

r
search
search
search
search

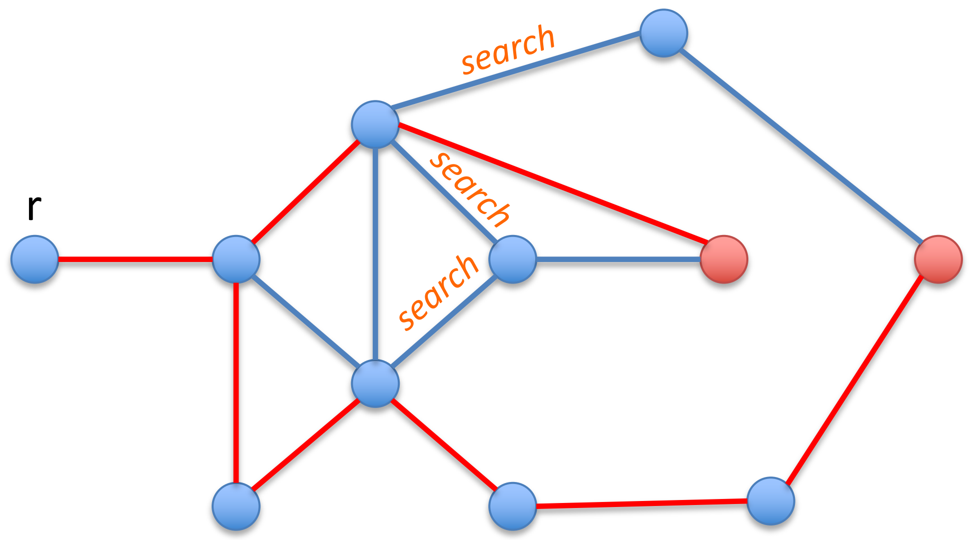

r
search
search
search
search
search

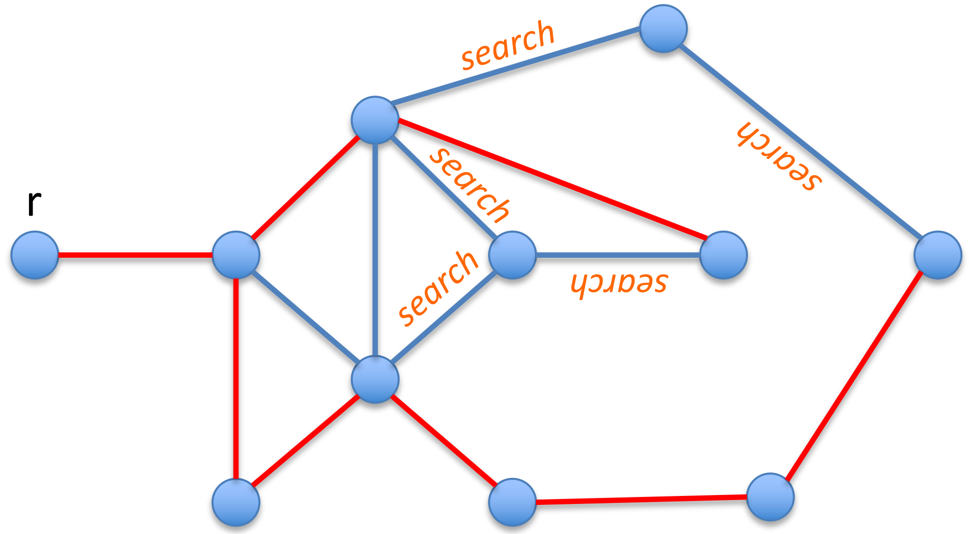

r
search
search
search
search

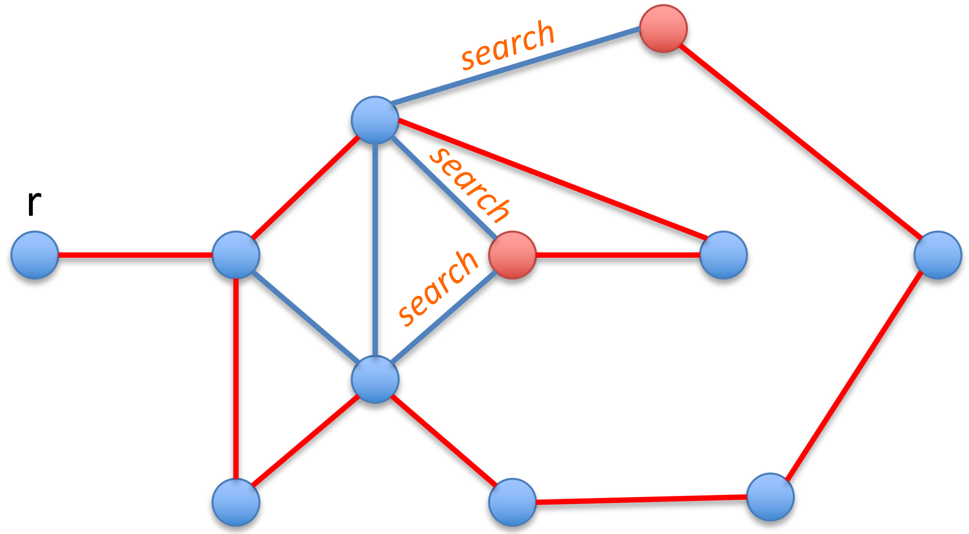

r
search
search
search
search
search

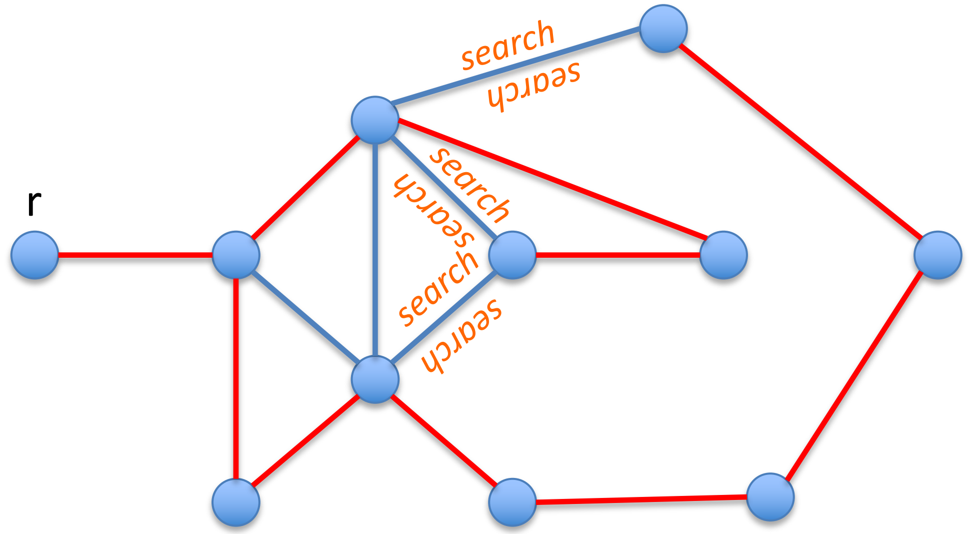

r
search
search
search
search
search
search
search
search
search

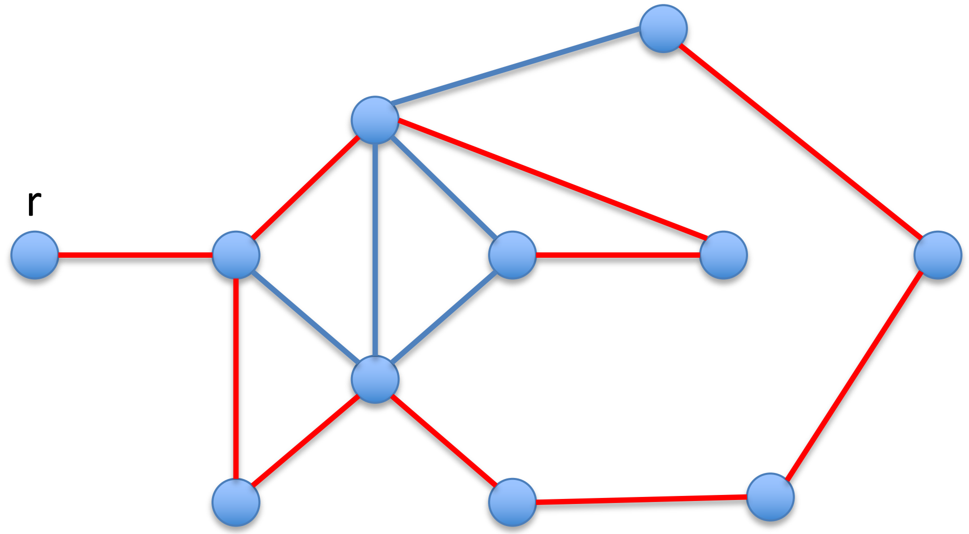

r
search
search
search
search
search


r
search
search
search


r
search
search
search
search
search


r
search
search
search


r
search
search
search
search
search
search


r

2. Asynchronous Bellman-Ford

context and goal

- undirected connected graph G=(V,E)
- positive weights *w(e)* on edges, or *w(u,v)* on edge *(u,v)*
- distinguished root node *r*
- compute a shortest path from *r* to any node

main idea

- extension of spanning tree construction to min distance computation: Bellman-Ford (with reactivation of nodes)
- desynchronization

<u>Algorithm</u> : principle = diffusion+ distance computation from root node *r*,
then from all updated nodes

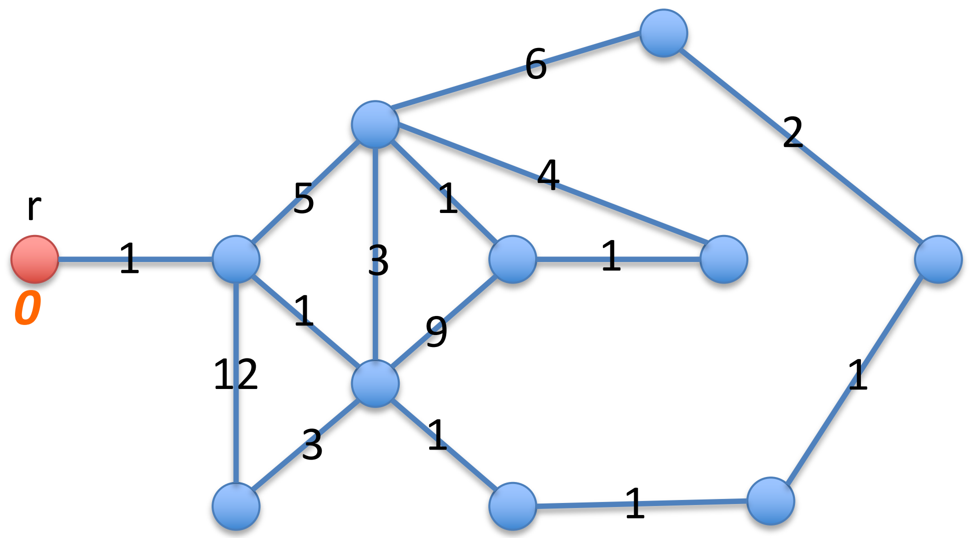
initialization

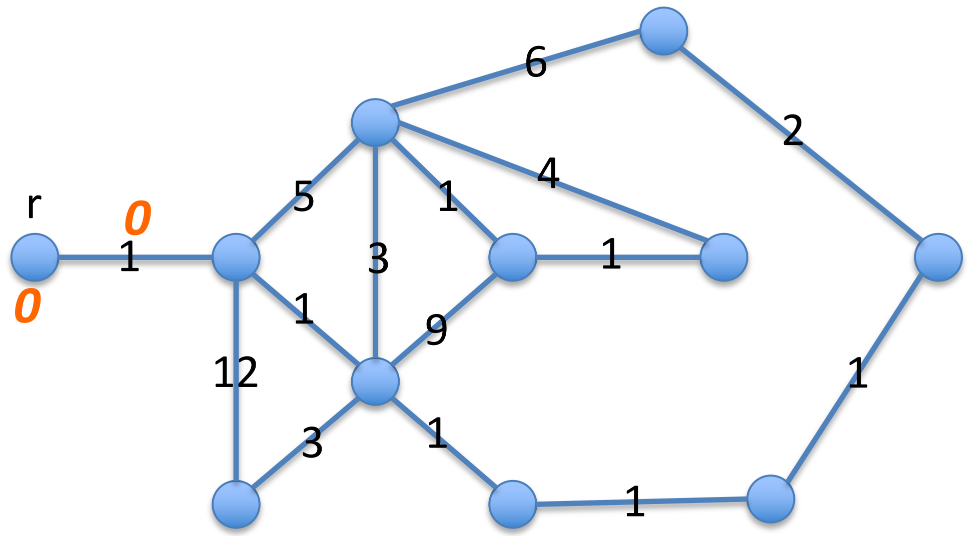
– all nodes inactive, except *r*
– distances to the root node *r*

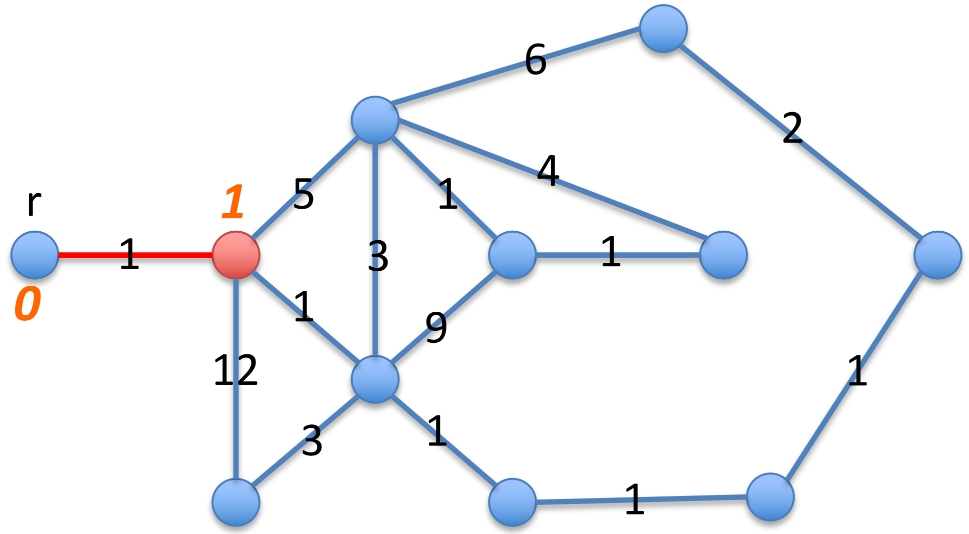
$$\forall v \in V,\ \mathrm{parent}(v) = \emptyset,\ \mathrm{dist}(v) = ?$$
$$\mathrm{dist}(r) = 0$$

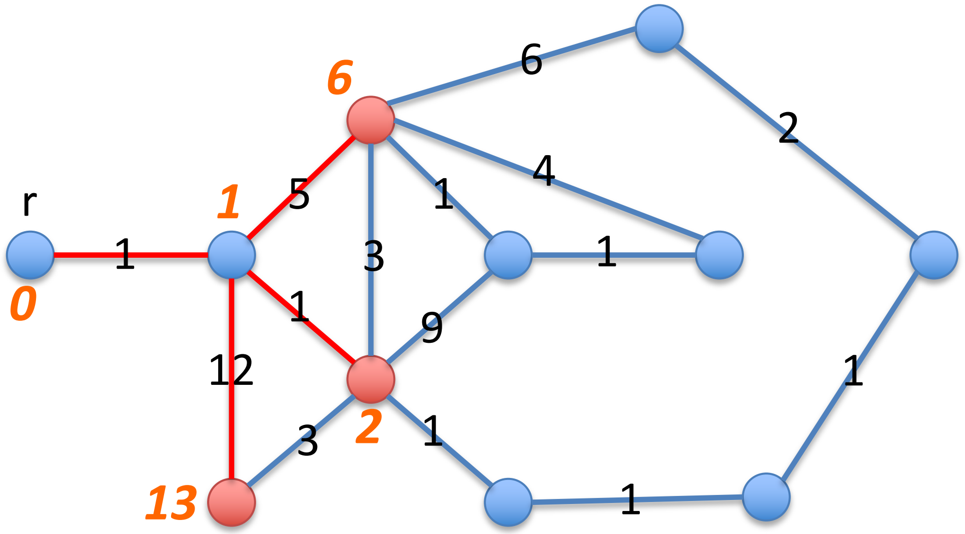
local step : do one of the two actions below

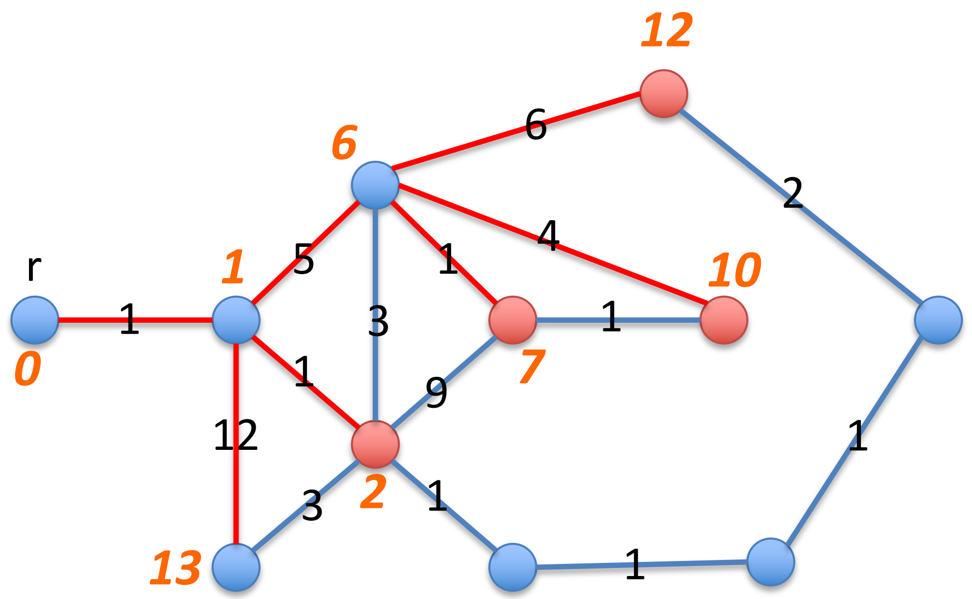
- <u>emission</u> from an active node *u*
  - *u* sends message $m = \mathrm{dist}(u)$ to all its neighbors *v*, except to its parent
  - then *u* becomes inactive
- <u>reception</u> of a message *m* by node *v* from node *u*
  - if node *v* does not yet have a parent
    or if the path through *u* is shorter : $\mathrm{dist}(v) > m + w(u,v)$
    - v updates its distance : $\mathrm{dist}(v) := m + w(u,v)$
    - v selects sets u as its parent : $\mathrm{parent}(v) = u$
    - v may inform u that it is its child (and its former parent that he left…)
    - v becomes active
  - otherwise do nothing (message m consumed with no impact)

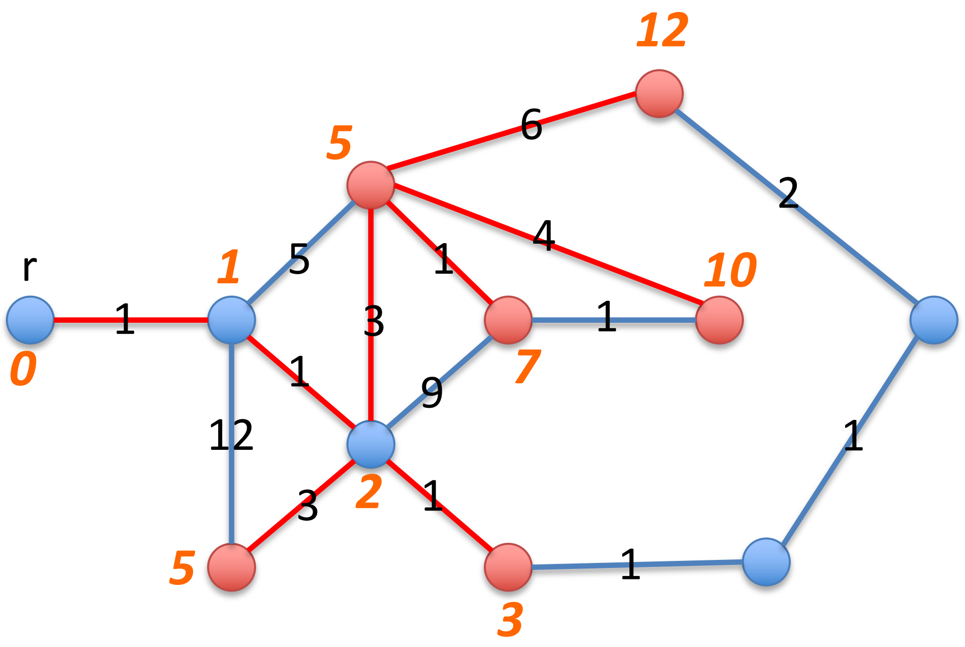

r
0
1
5
12
1
3
3
6
4
1
9
1
1
1
2
1
1

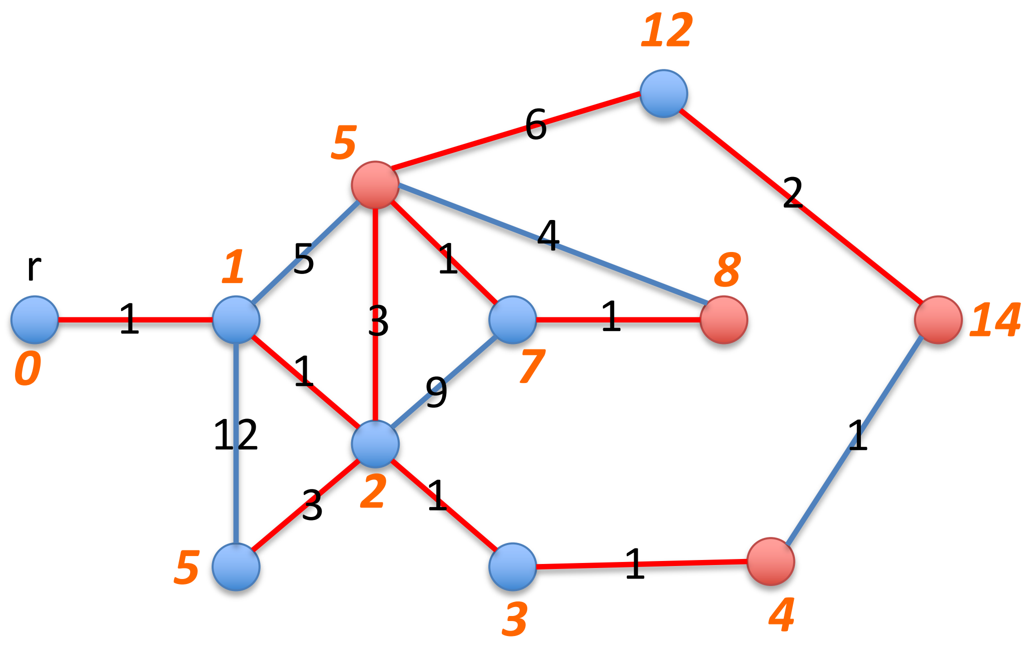

r
0
0
1
5
1
12
3
3
1
9
1
6
4
1
2
1
1
1

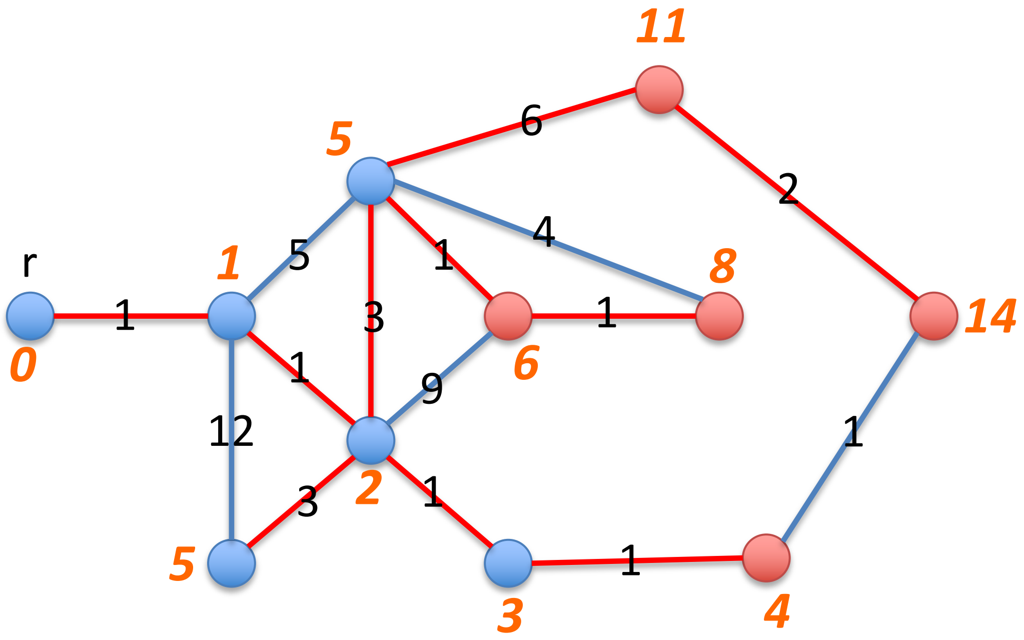

r
0
1
1
5
1
12
3
6
4
1
3
9
1
1
2
1
1

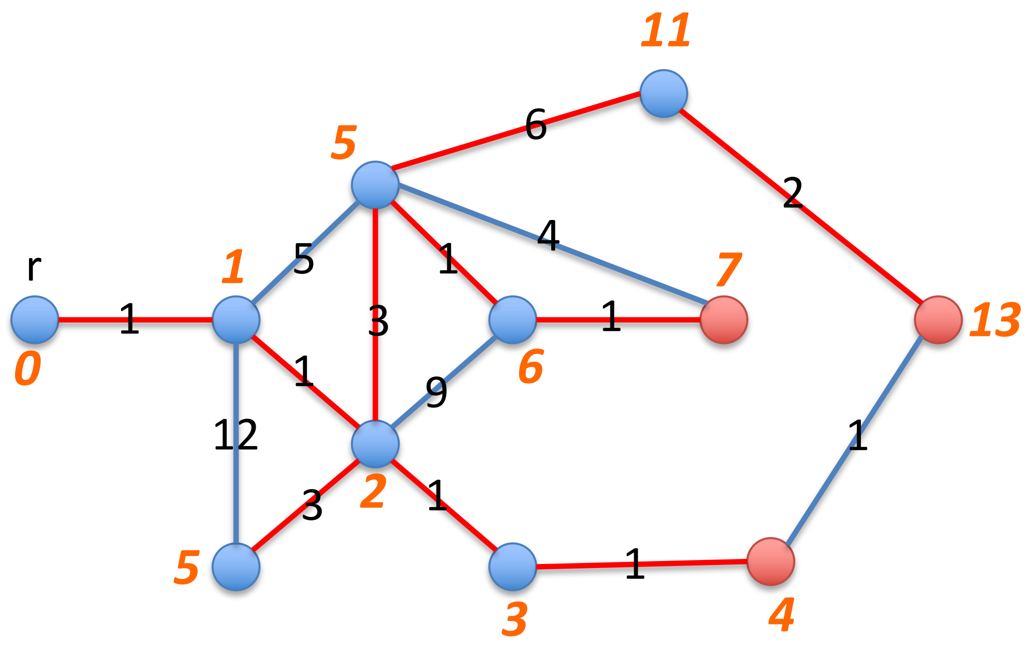

r
0
1
6
2
13
1
5
1
12
3
3
9
1
4
1
6
2
1
1
1


12
6
2
6
4
5
1
r
1
10
1
1
0
3
1
7
9
12
1
2
3
1
13
1

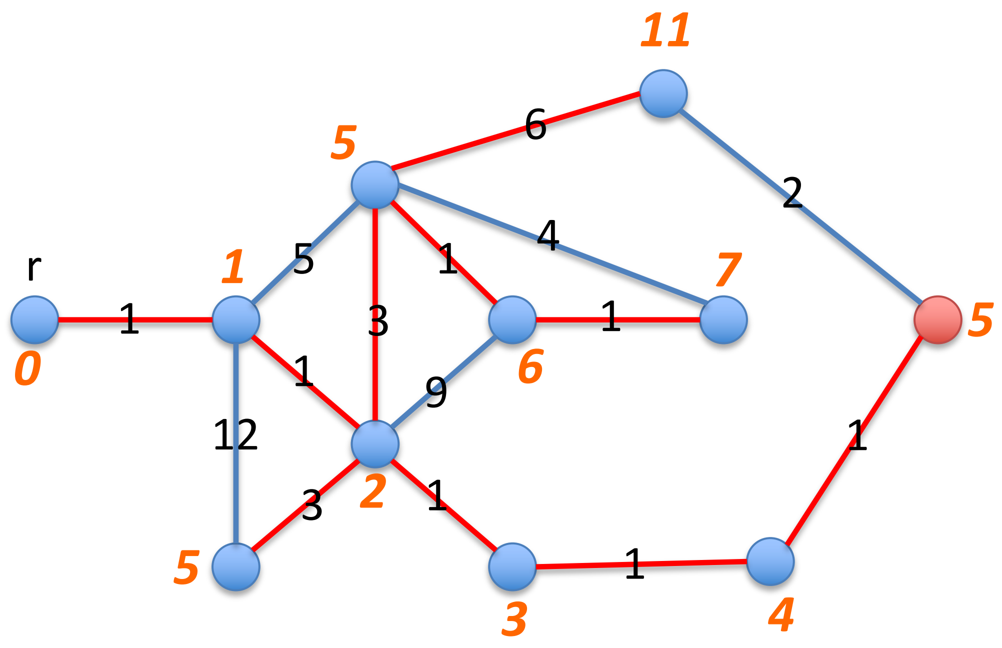

r
0
1
1
5
12
12
5
1
3
3
5
1
4
6
2
1
1
9
2
1
7
10
1
3
1


r
0
1
5
12
8
14
7
2
5
3
4
1
5
3
1
6
4
2
1
1
9
12
3
1
1
1

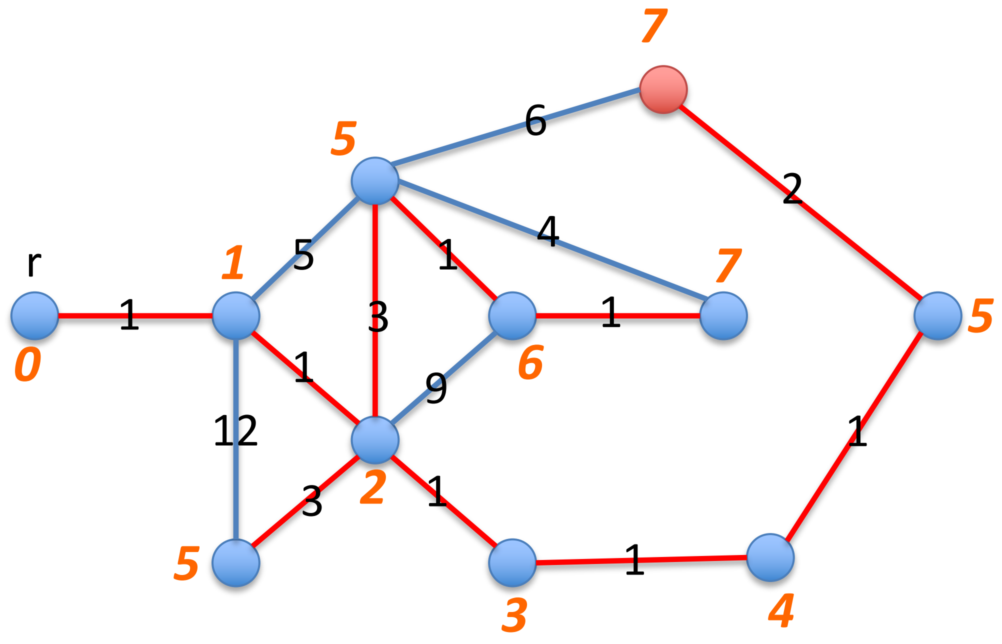

r
0
1
5
5
2
3
4
6
8
11
14
1
5
1
12
3
3
1
1
9
1
4
6
2
1
1

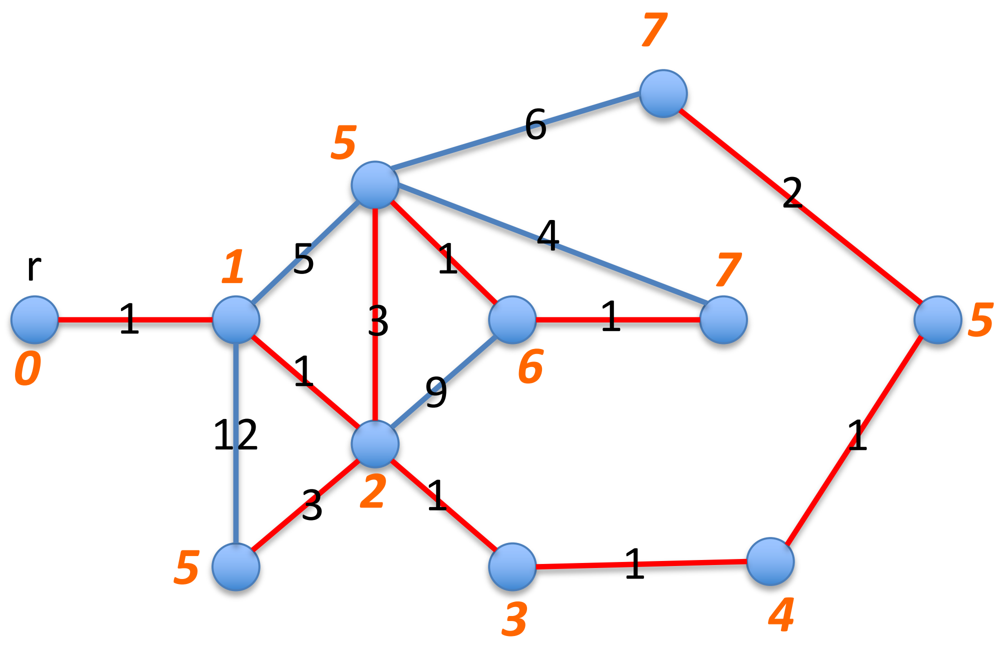


r
0
1
5
11
7
13
6
2
5
3
4
1
5
3
1
6
4
2
1
1
9
12
3
1
1
1



r
0
1
5
11
7
6
5
2
5
3
4
1
5
6
2
4
1
3
1
1
9
12
3
1
1
1


r
0
1
5
2
5
6
7
7
5
3
4
1
5
12
1
3
3
1
6
4
1
9
1
1
2
1


r
0
1
5
7
7
5
6
2
5
3
4
1
5
6
2
4
1
3
1
1
1
9
12
3
1
1
1

remark one can assume that messages pile up into channels,
or that the last message erases the previous one

proof:

invariant(s)

- current structure is always a tree
- distance of node *v* is the shortest on some path from *r* to *v*, and the parent of *v* is the predecessor of *v* on that path
- distance of node *v* to the root node *r* is the shortest among paths explored by the messages that reached *v* so far

monotony

- distances at nodes decrease, bounded from below by true shortest distances
- there is at least one shortest path to a node that will be progressively activated (all nodes are reached)

complexity ? i.e. how many successive elementary steps ?

- homework…
- hint : each (re)activated node may (re)initiate a message flow on the whole graph

Take home messages

distributed systems

- are everywhere in modern computer science
- require different models/assumptions to be correctly described/analyzed

a major phenomenon is asynchrony

- due to the absence of a global clock
- it becomes necessary to model and understand the concurrency of events
- this makes the study of algorithms/systems much more involved

next time

- representation of runs as partial orders of events
- vector clocks
- snapshot algorithms