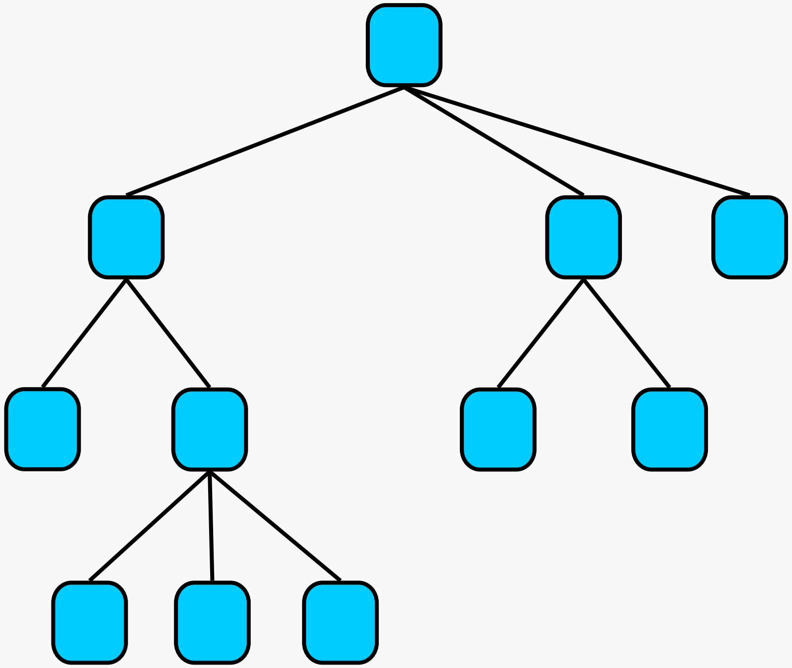
Computations in Trees

Saturation
Minimum Finding
Distributed Functions
Convergecast

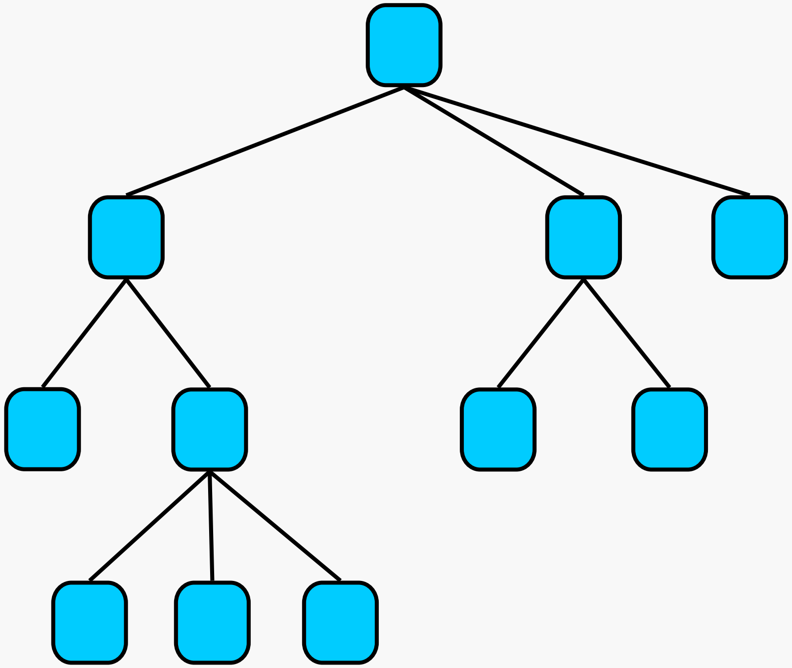


Chapter 2

Paola Flocchini

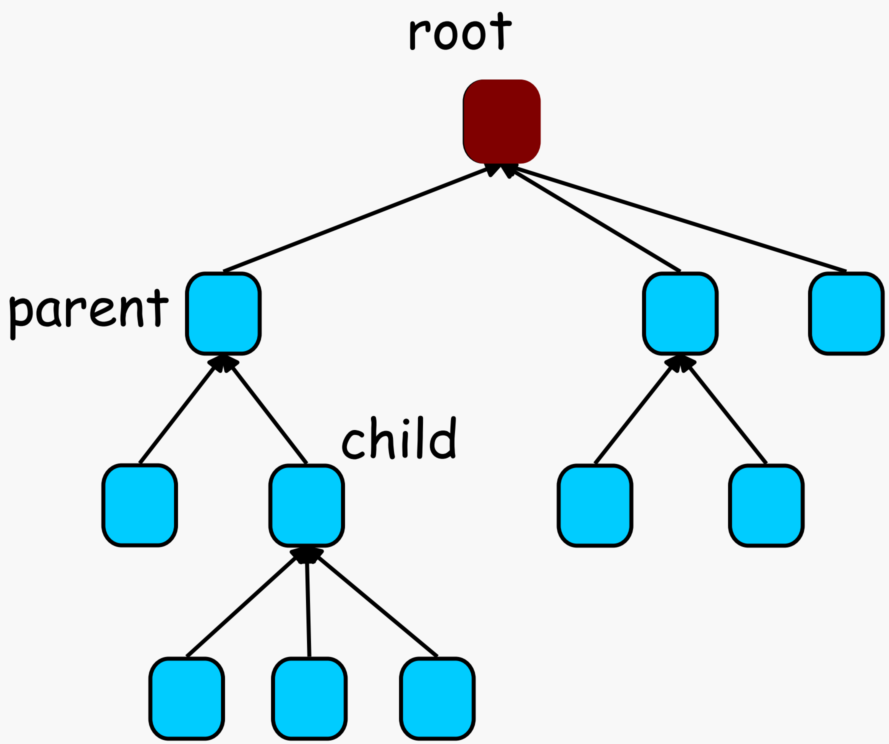
Trees

- Acyclic graph
- n entities
- n - 1 links



Rooted Trees

- Acyclic graph
- n entities
- n - 1 links



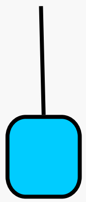
All links are oriented toward the root

Saturation Technique (General Trees)

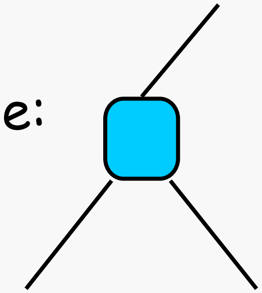
- Bidirectional links
- Ordered messages
- Full Reliability
- Knowledge of the topology

Note:

Each entity knows whether it is a leaf:



or an internal node:



A schema on which we can implement other algorithms

SATURATION: A Basic Technique

S = {available, awake, processing }

At the beginning, all entities are available

Arbitrary entities can start the computation (**multiple initiators**)

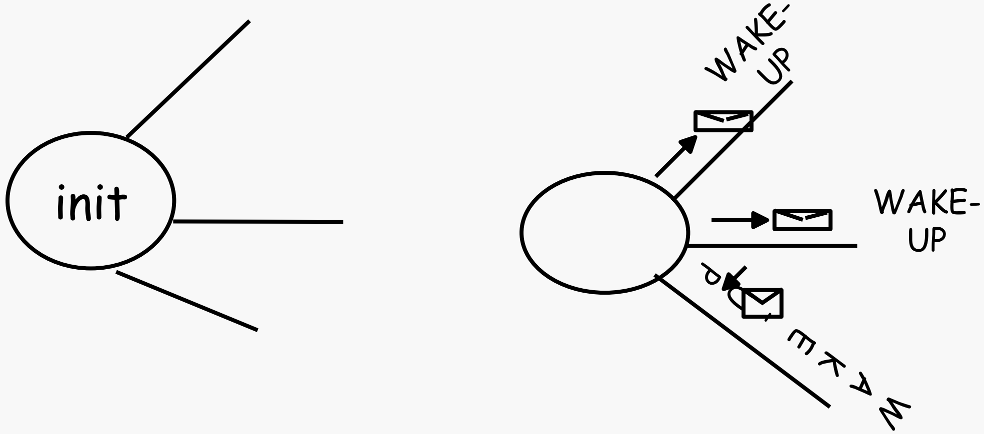
SATURATION: A General Technique

- **Activation phase:**
  started by the initiators: all nodes are activated   wake-up

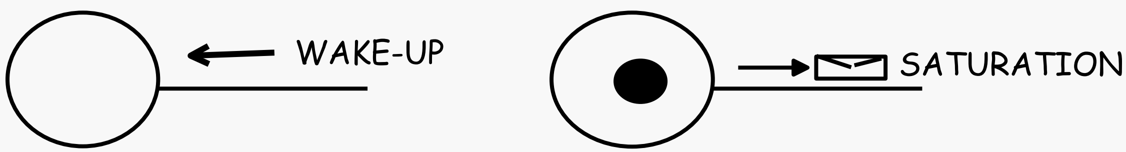
- **Saturation Phase:**
  started by the leaves: a unique pair of neighbours is identified (**saturated nodes**)

- **Resolution Phase:**
  started by the saturated nodes

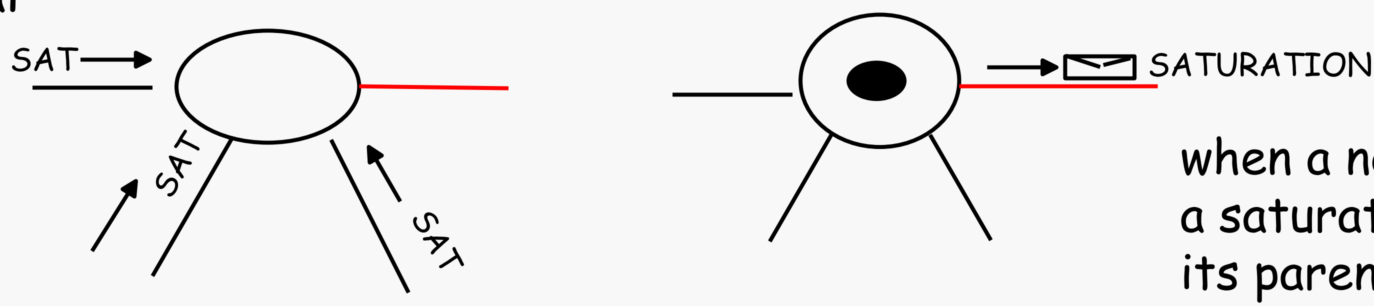
1)



2) leaf

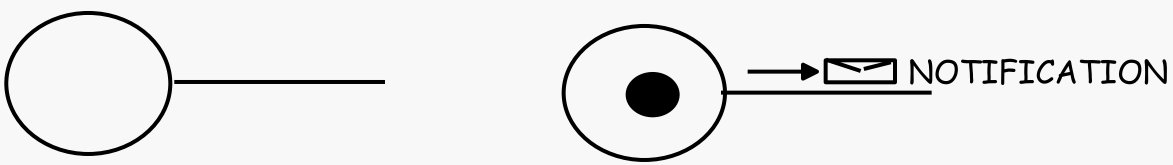


internal



when a node receive a saturation from its parent becomes saturated

3) Saturated node



S = {AVAILABLE, ACTIVE, PROCESSING, SATURATED}
Sinit = AVAILABLE
Restrictions: **R**;T

```
AVAILABLE    I haven't been activated yet
Spontaneously
                    send(Activate) to N(x);
                    Initialize;
                    Neighbours:= N(x)
                    if |Neighbours|= 1 then
                            Prepare_Message; // M := "Saturation"
/* special case if          parent « Neighbours;
I am a leaf */              send(M) to parent;
                            become PROCESSING;
                    else
                            become ACTIVE;
```

```
AVAILABLE
Receiving(Activate)
                send(Activate) to N(x) - {sender};
                Initialize;
                Neighbours:=  N(x);
                if |Neighbours|= 1   then
                        Prepare_Message;   //M := "Saturation"
 /* special case if     parent « Neighbours;
 I am a leaf  */        send(M)   to  parent;
                        become PROCESSING;
                else
                        become ACTIVE;
```

ACTIVE
Receiving(M)

I haven't started the saturation phase yet

```
Process_Message;
Neighbours:= Neighbours - {sender};
if |Neighbours| = 1 then
        Prepare_Message;  //M := "Saturation"
        parent « Neighbours;
        send(M) to parent;
        become PROCESSING;
```

PROCESSING

I have already started the saturation phase

receiving(M)

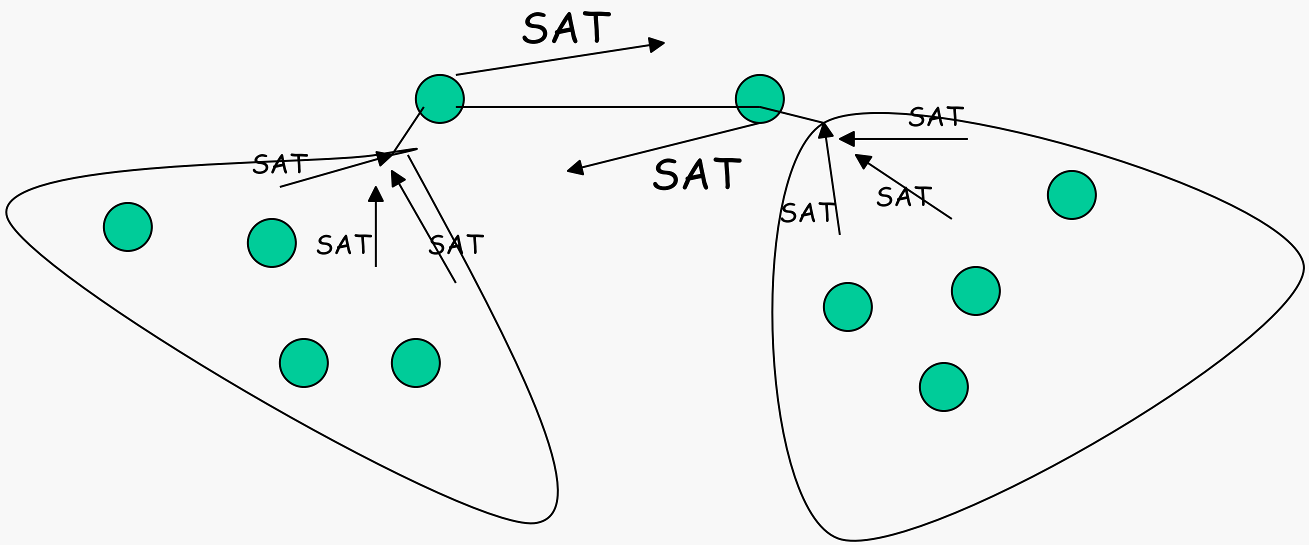
```
        Process_Message;
        become SATURATED; // Resolve
```

We perform different computations by instantiating the procedures

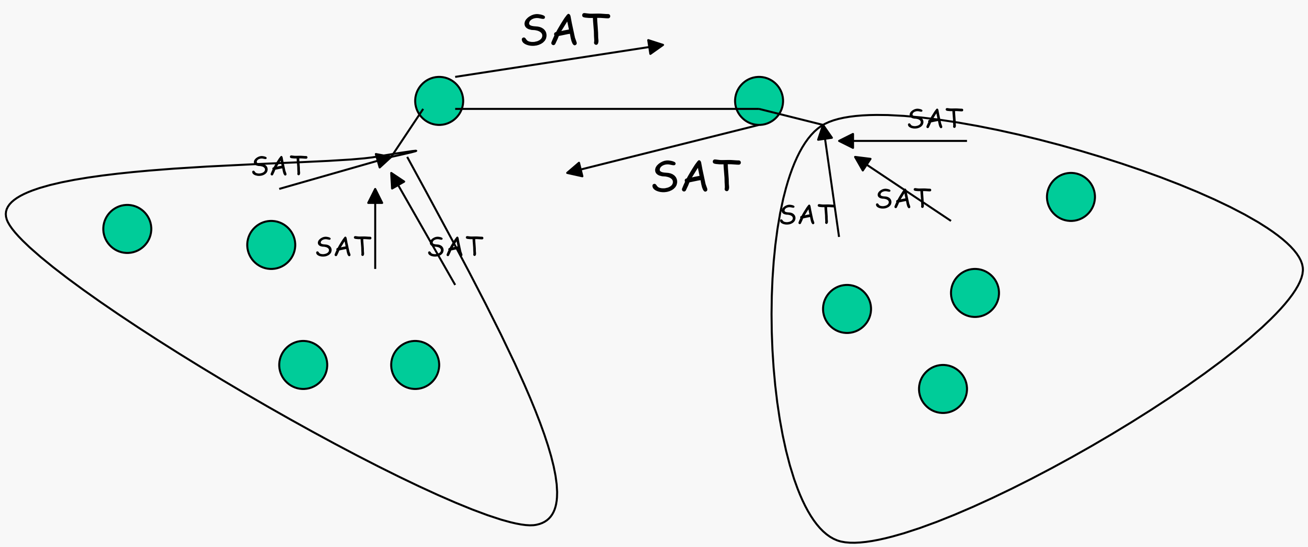
- Initialize
- Prepare_Message
- Process_Message
- Resolve

Property:

Exactly two processing nodes become saturated, and they are neighbours.



A node becomes PROCESSING only after sending saturation to its parent



A node become SATURATED only after receiving a message in the state PROCESSING from its parent

TWO neighbouring entities become saturated

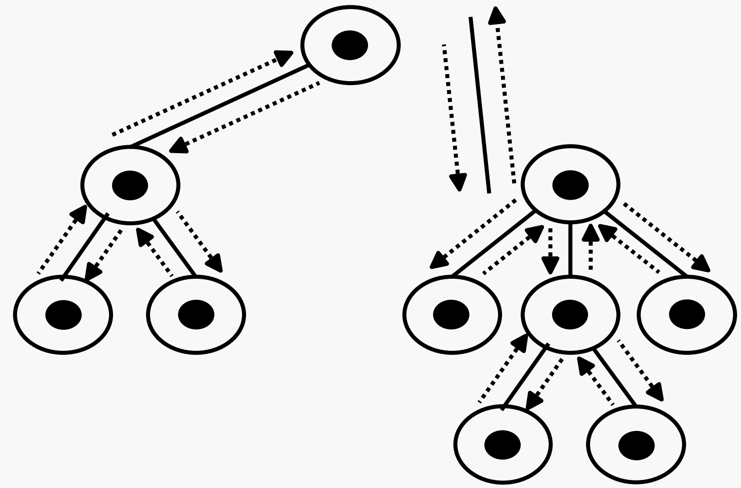
Which entities become saturated
depends on the unpredictable delays

Any pair of neighbors can become saturated

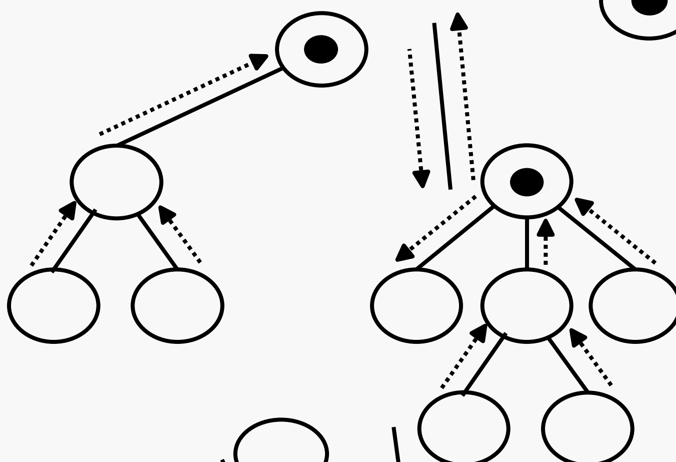
Message Complexity

Activation: Worst case - n initiators

$2(n-1)$

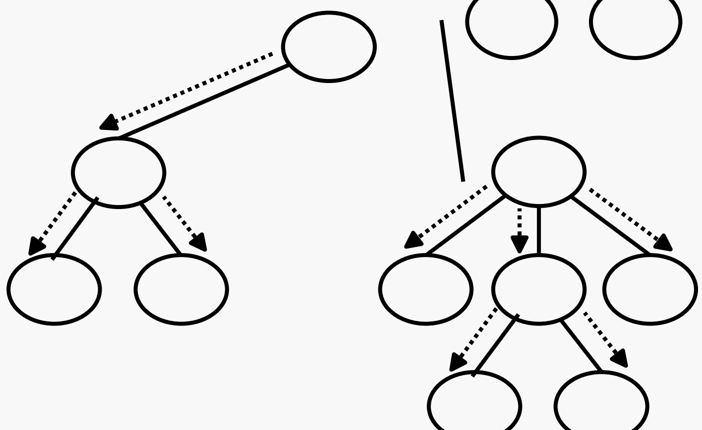


Saturation:



n

Notification:



$n - 2$

Tot: 2n -2+n+n-2=4n-4

Message Complexity

Activation: In general - k^* initiators

$n + k^* - 2$ (wake-up in the tree)

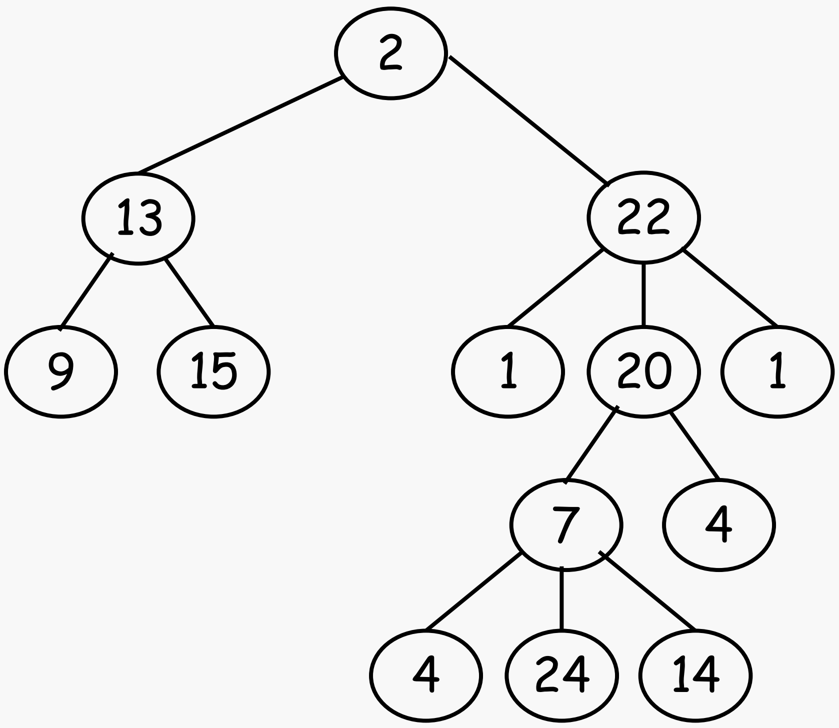
Saturation: n do not depend on number of initiators

Notification: $n - 2$

TOT: $2n+k^*-2$

Put information in the saturation message

Minimum Finding



Entity x has in input value(x)

At the end each entity should know whether it is the minimum or not

States S {AVAILABLE, ACTIVE, PROCESSING, SATURATED}
Sinit = AVAILABLE

```
AVAILABLE

Spontaneously
        send(Activate) to N(x);
        min := v(x);    // Initialize
        Neighbours := N(x)
        if |Neighbours| = 1 then
                M:=("Saturation", min);  // Prepare_Message
                parent « Neighbours;
                send(M) to parent;
                become PROCESSING;
         else become ACTIVE;
```

```
AVAILABLE

Receiving(Activate )
        send(Activate) to N(x) – {sender};
        min:=v(x);       // Initialize
        Neighbours :=  N(x);
        if |Neighbours|=1   then
            M:=("Saturation", min);  // Prepare_Message
            parent « Neighbours;
            send(M)   to  parent;
            become PROCESSING;
        else  become ACTIVE;
```

```
ACTIVE
Receiving(M)
        min:= MIN{min, M}   // Process_Message
        Neighbours:= Neighbours - {sender};
        if |Neighbours|=1 then
                M:=("Saturation", min);  // Prepare_Message
                parent « Neighbours;
                send(M) to parent;
                become PROCESSING;
```

PROCESSING

receiving(M)

```
        min:= MIN{min, M}    // Process_Message
        Notification:= ("Resolution", min)       // Resolve
        send (Notification)  to N(x) -parent
        if v(x) = min then
                become MINIMUM
        else
                become LARGE
```

receiving(Notification)

```
        send(Notification)  to N(x) - parent
        if  v(x) = Received_Value  then
                become  MINIMUM;
        else
                become  LARGE;
```

New rule

Message Complexity

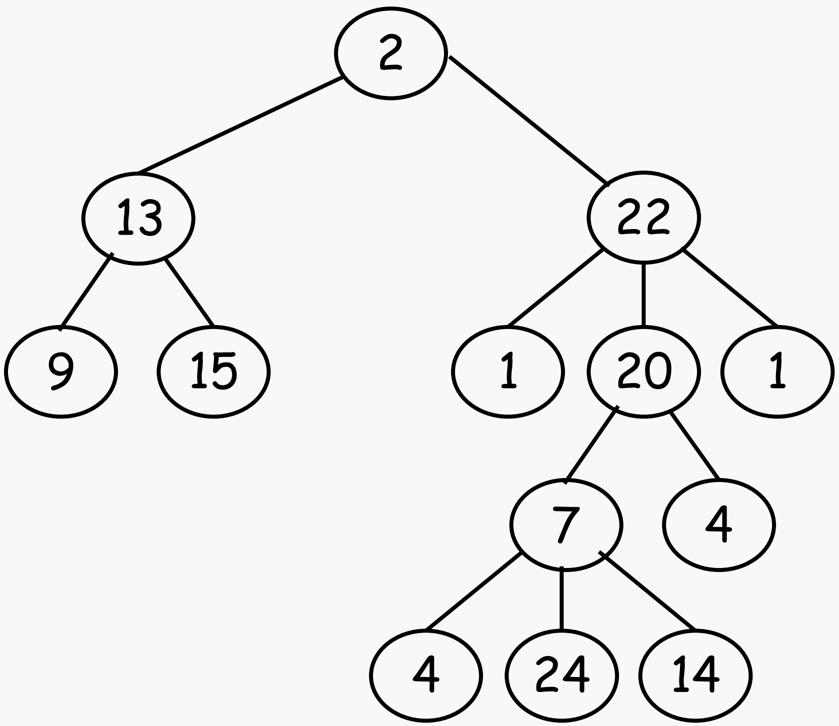
Saturation + Minimum Notification

$2n+k^*-2$ $\qquad$ $n - 2$

$3n + k^* - 4$

Distributed Function Evaluation

A generalization of minimum finding



Entity x has in input value(x)

Goal: we want to compute a function **F** whose arguments are distributed across the nodes

At the end each entity should know the computed value

Semigroups

Assumptions:
our values together with our function are a commutative semigroup

Semigroup: An algebraic structure consisting of a set **S** and a binary operator **#** which is **associative**

a # (b # c) = (a # b) # c

Commutative semigroup: A semigroup where F is also commutative

a # b = b # a

Examples

- S = natural numbers	# = +
- S = integers	# = x
- S = strings	# = concat
- S = natural numbers	# = min

Extensions to Saturation Algorithm 1

```
PROCESSING
receiving(Notification)
        send(Notification)  to N(x) - parent
        result := Received_Value;
        become DONE;
```

```
Initialize
        if v(x) is not nil then
            result := f(v(x))
        else
            result := nil

Prepare_Message
        M := ("Saturation", result);
```

Extensions to Saturation Algorithm 2

```
Process_Message
        if Received_value is not nil then
            if result is not nil then
                result := f(result, Received_value)
            else
                result := f(Received_value)


Resolve
        Notification := ("Resolution", result);
        send(Notification) to N(x) - parent;
        become DONE;
```

Message Complexity

Saturation + Notification

$2n+k^*-2$ $\qquad\qquad$ $n - 2$

$3n + k^* - 4$

Convergecast (Rooted Tree)

Additional assumption: the network a rooted tree

the root will be the unique saturated node and will start the resolution stage

We simplify the saturation stage

1. a leaf sends its message to its parent
2. each internal node waits for the messages of **all** children, then it sends a message to its parent

Broadcast + Termination Detection

Convergecast can be used every time we have a Rooted Spanning Tree

Broadcast + Convergecast:

1. The initiator s uses Flood + Reply to construct a rotted spanning tree T
2. The leaves of T start a Convergecast toward s

Cost M: **M(**Flood + Reply**)** + **M(**Convegecast**)**