
Traversal and DFS Algorithms

28 April 2010

Lecture 7

Topics for Today

- Overview of traversal algorithms
 - What is a traversal algorithm
 - F-traversal algorithms
 - Overview
 - Traversal algorithms for specific topologies
 - Tarry's traversal for arbitrary connected networks
- Depth-first Search
 - Depth-first search trees
 - Time complexity
 - one-time complexity
 - Overview of DFS-algorithms
 - Distributed depth first search tree construction
 - Classical
 - Awerbuch

Source: Tel 6.1-6.3

Traversal algorithms

A traversal algorithm is an algorithm with the following properties:

1. In each computation there is one initiator, which starts the algorithm by sending out exactly one message.
2. A process, upon receipt of a message, either sends out **one** message or **decides**.
 - Hence in each finite computation exactly one process decides, and the algorithm is said to terminate in that process.
3. The algorithm terminates in the initiator and when this happens, each process has sent a message **at least once**.
 - In other words a traversal algorithm hands over a *token* from one process to the next in such a way that all processes hold the token at least once and upon termination the token has returned to the initiator.

F-traversal algorithms

An algorithm is an f-traversal algorithm when

1. It is a traversal algorithm
2. In each computation at least $\min(|V|, x+1)$ processes have been *visited* after $f(x)$ token passes.

A process has been visited, when it is or has been in possession of the token.

Prop. A process that has performed a send has been visited.

Prop. A process that has been visited and does not currently hold the token has performed at least one send event.

Traversal algorithms overview

Algorithm	Topology	Centralized?	Messages	Time
Ring	Ring	C	$ V $	$ V $
Sequential polling	Clique	C	$2 V -2$	$2 V -2$
Torus	Torus	C	$ V $	$ V $
Hypercube	Hypercube	C	$2 E $	$2 E $
Tarry	Any	C	$2 E $	$2 E $
Classic DFS	Any	C	$2 E $	$2 E $
Classic + NK	Any	C	$2 V -2$	$2 V -2$

Ring Algorithm

var $Next_p$: node

{p is an initiator}

begin send <tok> to $Next_p$;

 receive <tok>;

decide

end

{p is not an initiator}

begin receive <tok>;

 send <tok> to $Next_p$;

end

Polling Algorithm

For networks with at least one node v such that $\deg(v) = |V|-1$. One of these nodes must be the unique initiator.

var rec_p : int **init** 0; (* for the initiator only *)

If p is initiator **then**

begin

forall $q \in Neigh_p$ **do** send **<tok>** to q ;

while $rec_p < \#Neigh_p$ **do**

begin receive **<tok>**; $rec_p := rec_p + 1$ **end**;

decide

end

else (* bounce the token *)

begin receive **<tok>** from q ; send **<tok>** to q **end**

Sequential Polling Algorithm

Move the send action into the while-loop.

```
var  $rec_p$  : int init 0; (* for the initiator only *)
```

```
if  $p$  is initiator then
```

```
  begin
```

```
    while  $rec_p < \#Neigh_p$  do
```

```
      begin send <tok> to  $q_{rec_p + 1}$ ;
```

```
        receive <tok>;  $rec_p := rec_p + 1$  end;
```

```
    decide
```

```
  end
```

```
else (* bounce the token *)
```

```
  begin receive <tok> from  $q$ ; send <tok> to  $q$  end
```

Clique Traversal

Torus Traversal

Thm. The sequential polling algorithm is a 2x-traversal algorithm for cliques

For the initiator, execute once:

send $\langle \mathbf{num}, 1 \rangle$ to Up

For each process, upon receipt of the token $\langle \mathbf{num}, k \rangle$:

begin

if $k = n^2$ **then** *decide*

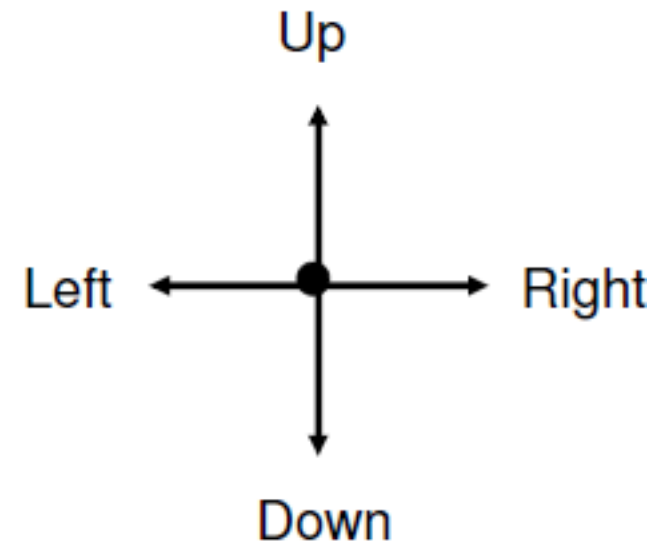
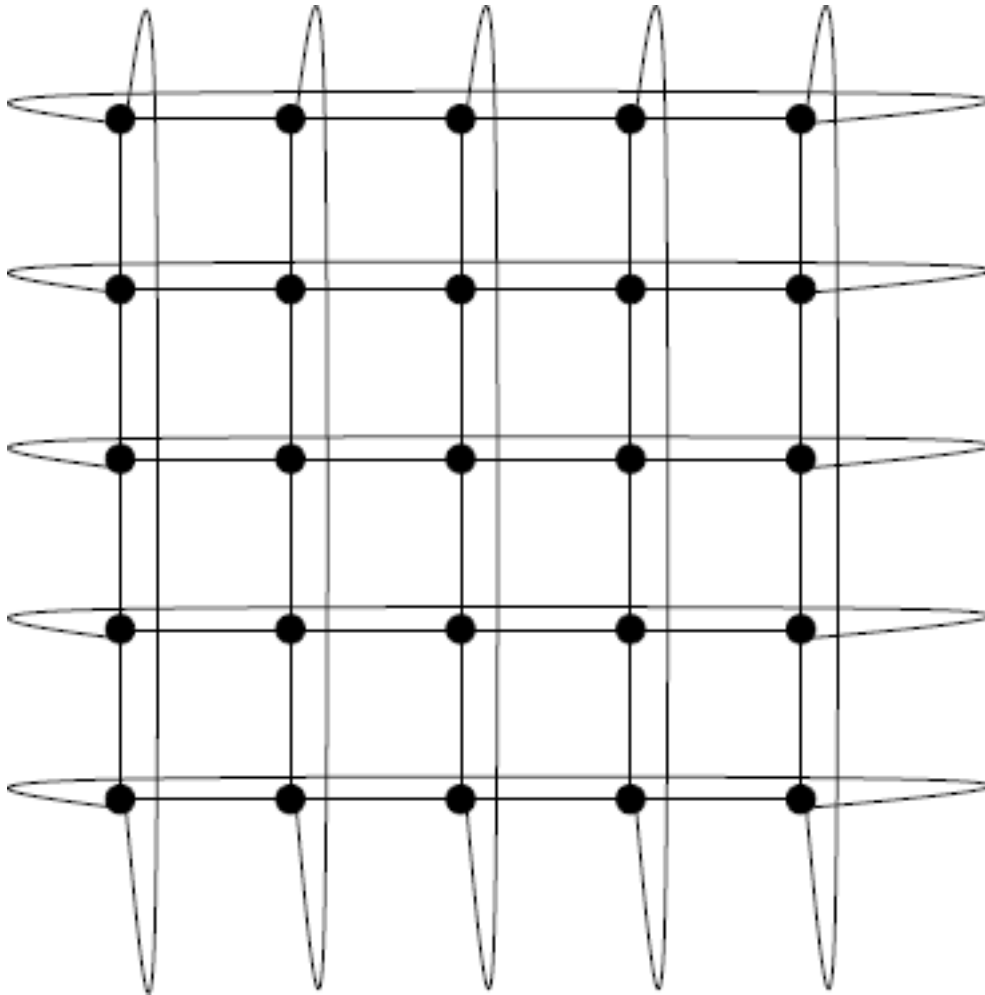
else if $n \mid k$

then send $\langle \mathbf{num}, k+1 \rangle$ to Up

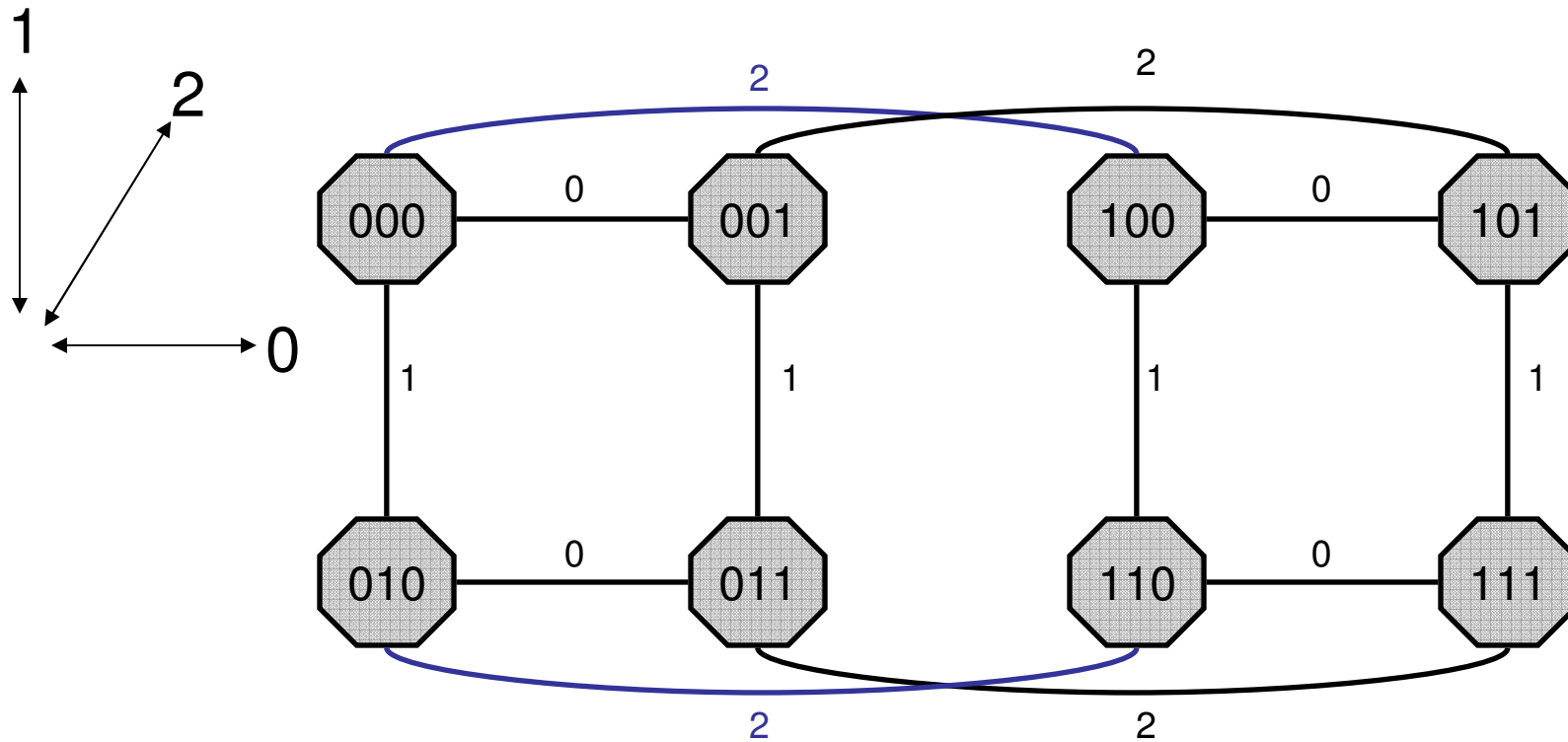
else send $\langle \mathbf{num}, k+1 \rangle$ to *Right*

end

5 x 5 – Torus traversal



Hypercube traversal (dim n=3)



Hypercube traversal (dim n)

For the initiator

send $\langle \mathbf{num}, 1 \rangle$ to $channel[n-1]$

For each process, upon receipt of the token $\langle \mathbf{num}, k \rangle$:

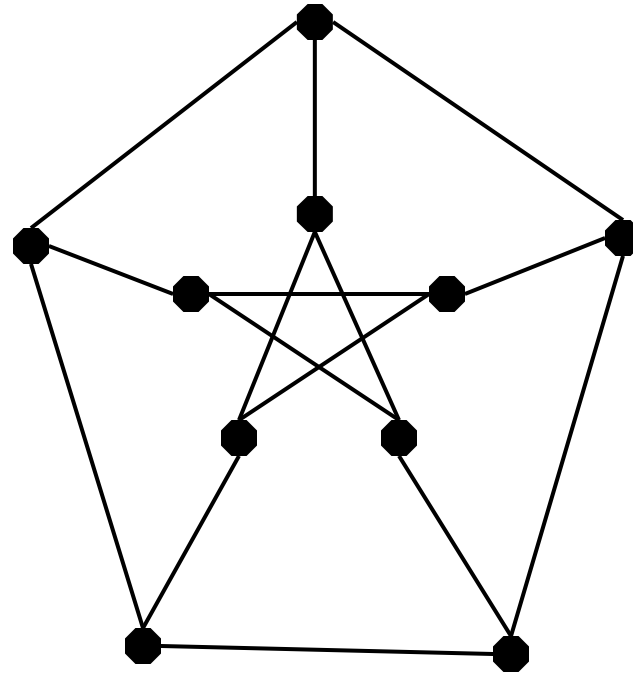
```
begin    if  $k = 2^n$  then decide
          else    begin  $l := \max\{m \mid k \bmod 2^m = 0\}$ ;
                  send  $\langle \mathbf{num}, k+1 \rangle$  to  $channel[l]$ 
          end
end
```

Tarry's traversal algorithm

Forwarding rules:

1. A process never forwards the token twice through the same channel.
2. A non-initiator forwards the token to its father only if there is no other channel left according to rule 1.

Tarry on Petersen's Graph



Tarry's traversal algorithm

```
var     $used_p[q]$  : boolean    init false for each  $q$  in  $Neigh_p$   
       $father_p$ : node          init undef
```

For the initiator execute once:

```
begin   $father_p := p$ ; choose  $q$  in  $Neigh_p$ :  
         $used_p[q] := true$ ; send <tok> to  $q$   
end
```

For each process, upon receipt of <tok> from q_0 :

```
begin  if  $father_p = undef$  then  $father_p := q_0$ ;  
        if for all  $q$  in  $Neigh_p$  :  $used_p[q] = true$   
          then decide  
        else ForwardToken()  
end
```

Tarry's traversal algorithm

Procedure *ForwardToken* (p: node)

(* pre: exists q in $Neigh_p$: $\neg used_p[q]$ *)

begin

if $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$

then begin $q \in \{r \in Neigh_p \mid r \neq father_p \wedge \neg used_p[r]\}$

$used_p[q] := true$; send **<tok>** to q

end

else begin

$used_p[father_p] := true$;

send **<tok>** to $father_p$

end

end

Forwarding rules:

1. A process never forwards the token twice through the same channel.
2. A non-initiator forwards the token to its father only if there is no other channel left according to rule 1.

Tarry = traversal

1. Unique initiator starts with single send
 - see program text
2. Upon receive, send once or decide
 - see program text
3. Termination
 - only the initiator decides
 - in a terminal configuration:
 1. all channels incident to the **initiator** have been used **once in each direction**
 2. all channels incident to a **visited** node have been used **once in each direction**
 3. **all processes** have been visited and **each channel** has been used once in both directions

The final father-son graph → Spanning Tree

Types of Complexity

- Message complexity
 - The total number of **messages** sent by all nodes, or
 - The total number of **bits** sent (in case of long messages)
- Time complexity
 - From the perspective of an **external observer**
 - Totally unpredictable due to unbounded link delays
 - Different for each execution of the algorithm
- Ideal time complexity
 - Processing time **zero**, transmission time at most **one unit**
 - **-OR-** Synchronous mode of operation
- Causal time complexity
 - Length of the longest chain of **causally related messages** transmissions in any execution sequence

Time Complexity

Def. The time complexity of a distributed algorithm is the maximum time taken by a computation under the following assumptions:

1. a process can execute any finite number of events in zero time,
2. the time between sending and receipt of a message is at most one time unit.

Lemma. For traversal algorithms the time complexity equals the message complexity.

So Far

- Overview of traversal algorithms
 - What is a traversal algorithm
 - F-traversal algorithms
 - Overview
 - Traversal algorithms for specific topologies
 - Tarry's traversal for arbitrary connected networks
- **Depth-first Search**
 - Depth-first search trees
 - Time complexity
 - one-time complexity
 - Overview of DFS-algorithms
 - Distributed depth first search tree construction
 - Classical
 - Awerbuch
 - Cidon

DFS-trees

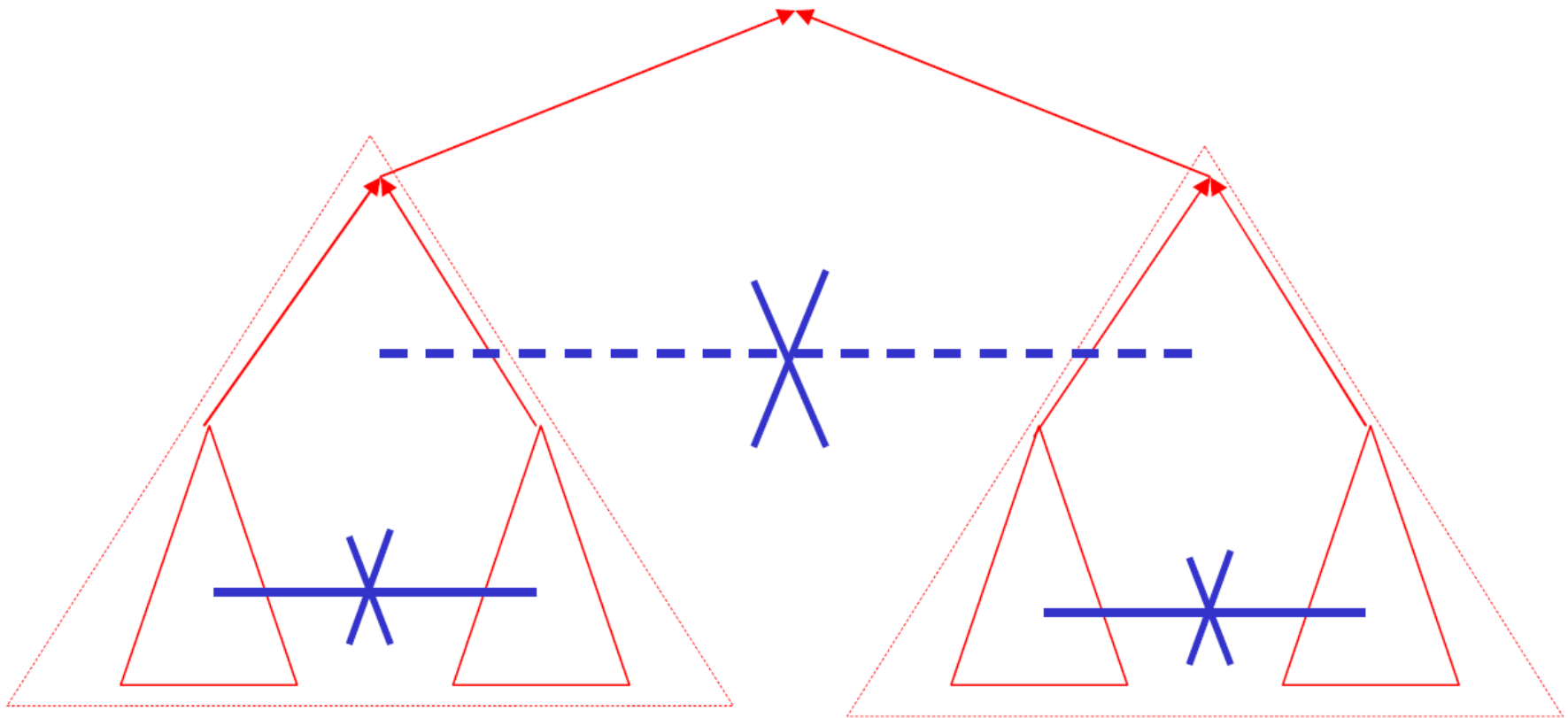
Let T be a rooted spanning tree of graph $G = (V, E)$.

Fronde edges are edges of G that do not belong to T .

For any $p \in V$ let $A[p]$ be the set of **ancestors** of p (w.r.t. T), and let $D[p]$ be the set of **descendants** of p .

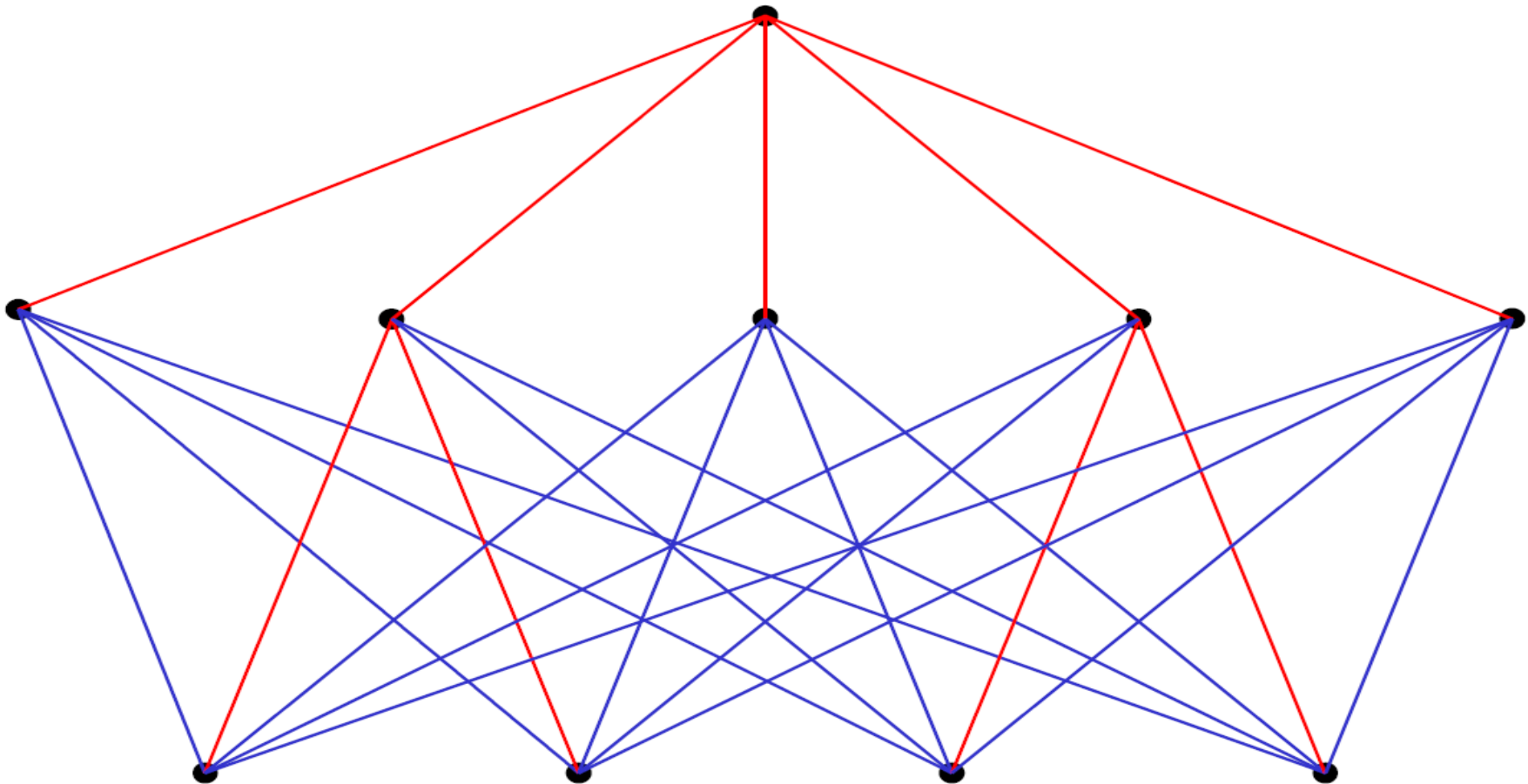
A rooted spanning tree of G is a *depth-first search tree* if, for each fronde edge pq , $p \in A[q] \vee q \in A[p]$.

Forbidden edges in DFS-trees

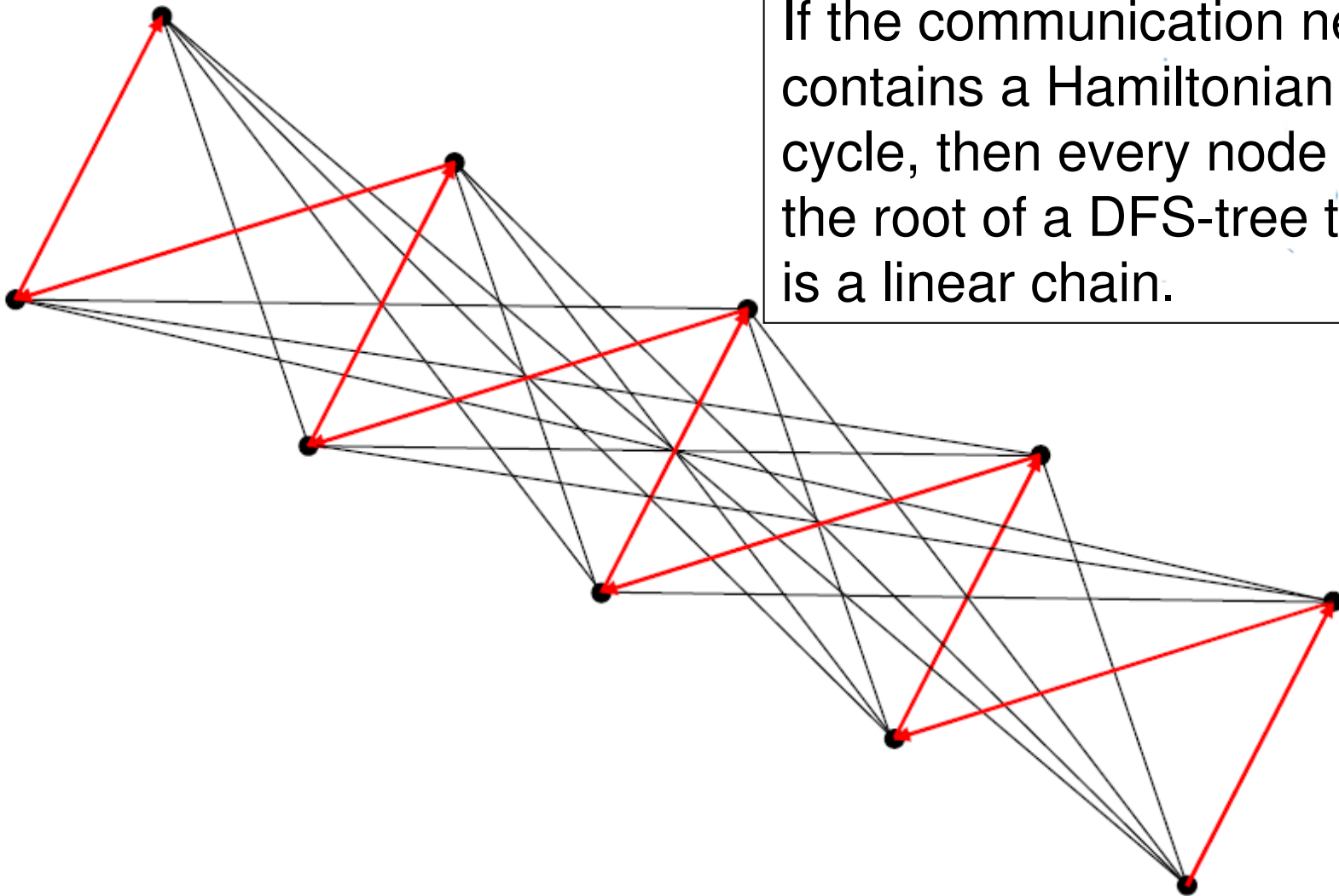


Spanning tree of $K_{5,5}$

Not a depth first search spanning tree



Spanning tree of $K_{5,5}$



If the communication network contains a Hamiltonian cycle, then every node is the root of a DFS-tree that is a linear chain.

Time Complexities

The *time complexity* of a distributed algorithm is the maximum time taken by a computation of the algorithm under the assumptions:

T1. A process can execute any finite number of events in zero time

T2. The time between sending and receipt of a message is **at most** one time unit

The *one-time complexity* of a distributed algorithm is the maximum time taken by a computation of the algorithm under the assumptions:

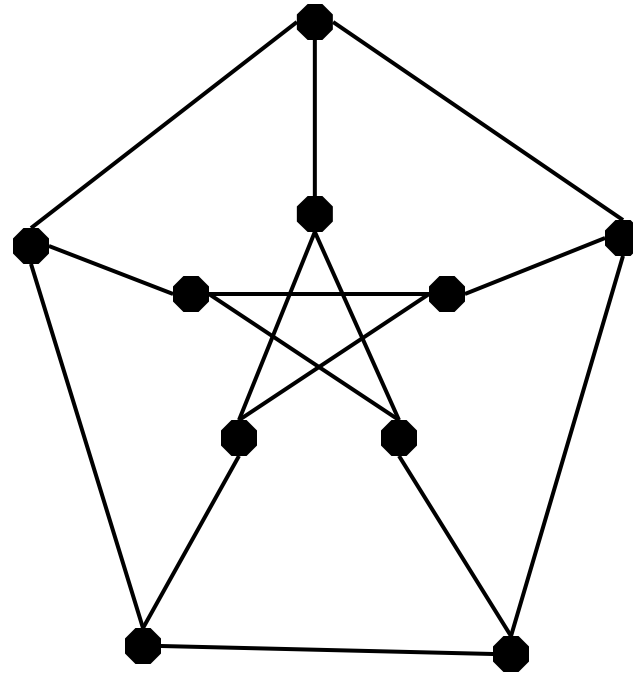
O1. A process can execute any finite number of events in zero time

O2. The time between sending and receipt of a message is **exactly** one time unit

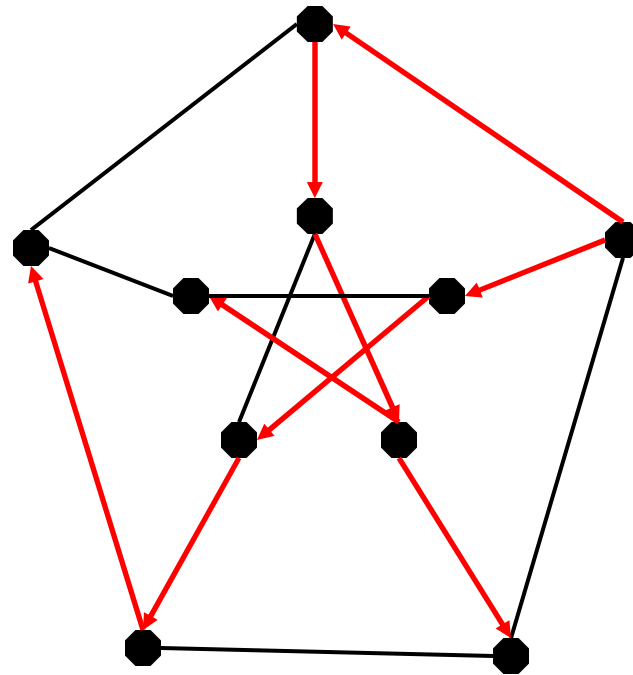
DFS-algorithms overview

<u>Algorithm</u>	<u>Topology</u>	<u>C/D</u>	<u>Messages</u>	<u>Time</u>
Classic DFS	Any	C	$2 E $	$2 E $
Awerbuch	Any	C	$4 E $	$4 V -2$
Cidon	Any	C	$4 E $	$2 V -2$
Classic DFS + Neighbor Knowledge	Any	C	$2 V -2$	$2 V -2$

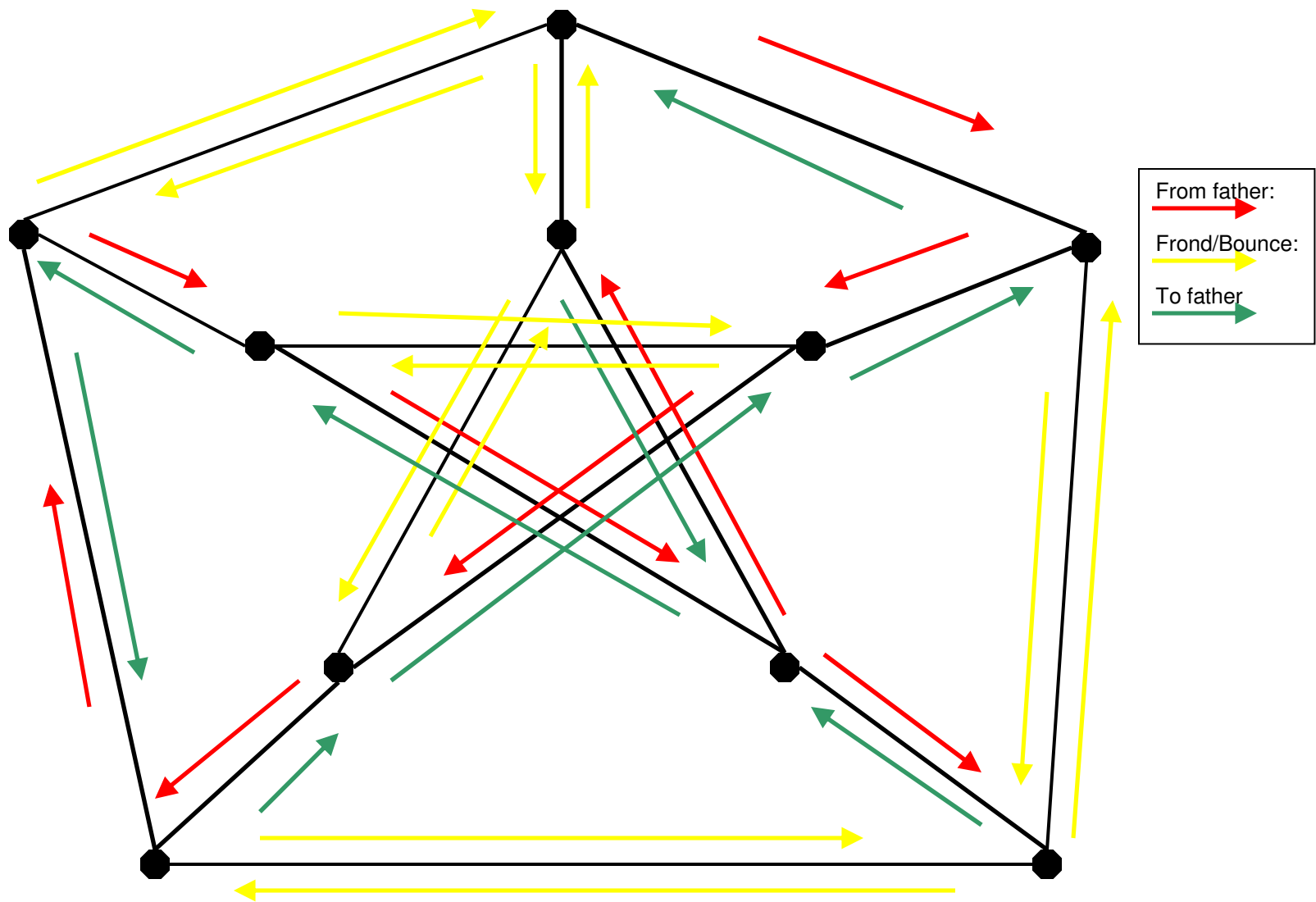
Tarry on Petersen's Graph



Tarry on Petersen's Graph



Classical DFS on Peterson's graph



Classical DFS-algorithm

```
var     $used_p[q]$  : boolean    init false for each  $q$  in  $Neigh_p$   
       $father_p$ : node          init undef
```

For the initiator execute once:

```
begin   $father_p := p$ ; choose  $q$  in  $Neigh_p$ :  
       $used_p[q] := true$ ; send <tok> to  $q$   
end
```

For each process, upon receipt of <tok> from q_0 :

```
begin  if  $father_p = undef$  then  $father_p := q_0$ ;  
      if for all  $q$  in  $Neigh_p$  :  $used_p[q] = true$   
        then decide  
      else ForwardToken()  
end
```

So far identical to Tarry's traversal algorithm

Classical DFS-algorithm (ctnd)

Forwarding rules:

1. A process never forwards the token twice through the same channel.
2. A non-initiator forwards the token to its father only if there is no other channel left according to rule 1.
3. **When a process receives the token it sends it back through the same channel if this is allowed by rules 1 and 2.**

Classical DFS-algorithm (forwarding)

Procedure *ForwardToken* (p, q_0 : node)

begin

if $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$

then begin

if $father_p \neq q_0 \wedge \neg used_p[q_0]$

then $q := q_0$;

else $q \in \{r \in Neigh_p \mid r \neq father_p \wedge \neg used_p[r]\}$

$used_p[q] := true$; send **<tok>** to q

end

else begin

$used_p[father_p] := true$;

send **<tok>** to $father_p$

end

end

Awerbuch DFS

The **problem** with Classic DFS is that the token runs on each edge **serially**

- That makes the time complexity also relative to the number of edges
- The token passes along each tree edge and each **frond edge**

We can reduce the time complexity by not passing the token over frond edges

- How?

Solution:

- Introduce **receipt acknowledgements**
 - Each node informs all of its neighbors (except its father) when it has received the token for the first time (<vis>/<ack>)
 - Nodes forward when they have received all acks
 - Possible improvement – don't <vis> to the one you will send to

Awerbuch's DFS-algorithm

```
var     $used_p$     : boolean      init false
        $father_p$   : node         init undef;
```

For the initiator execute once:

```
begin  $father_p := p; q \in Neigh_p;$ 
```

```
  forall  $r \in Neigh_p \setminus \{father_p\}$  do send  $\langle vis \rangle$  to  $r$ ;
```

```
  forall  $r \in Neigh_p \setminus \{father_p\}$  do receive  $\langle ack \rangle$  from  $r$ ;
```

```
end     $used_p := true; send \langle tok \rangle$  to  $q$ ;
```

For each process, upon receipt of $\langle tok \rangle$ from q_0 :

```
begin if  $father_p = undef$  then
```

```
  begin  $father_p := q_0$ ;
```

```
    forall  $r \in Neigh_p \setminus \{father_p\}$  do send  $\langle vis \rangle$  to  $r$ ;
```

```
  end    forall  $r \in Neigh_p \setminus \{father_p\}$  do receive  $\langle ack \rangle$  from  $r$ ;
```

```
  if  $p$  is initiator  $\wedge \forall q \in Neigh_p : used_p[q]$ 
```

```
  then decide else ForwardToken( $p, q_0$ );
```

```
end
```

Awerbuch's DFS-algorithm (ctnd)

For each process, upon receipt of **<vis>** from q_0
begin $used_p[q_0] = \text{true}$; send **<ack>** to q_0 **end**

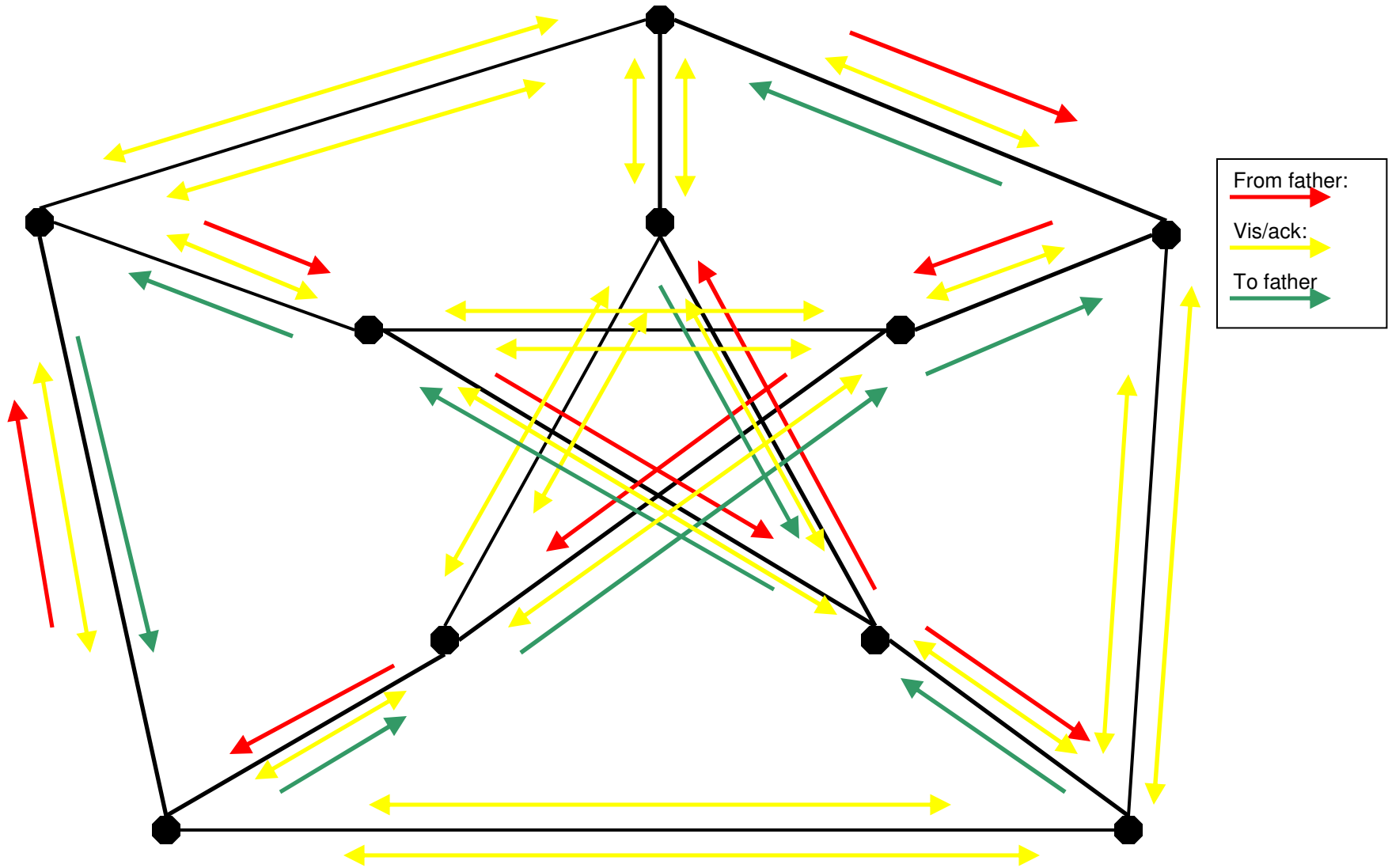
Over each **frond** edge the following messages are sent:

1. One (1) **<vis>** message in each direction.
2. One (1) **<ack>** message in each direction

Over each **tree** edge the following messages are sent:

1. One (1) **<tok>** message in each direction
2. One (1) **<vis>** message from **father to son**
3. One (1) **<ack>** message from **son to father**

Awerbuch DFS on Peterson's graph



Conclusion

- Overview of traversal algorithms
 - What is a traversal algorithm
 - F-traversal algorithms
 - Overview
 - Traversal algorithms for specific topologies
 - Tarry's traversal for arbitrary connected networks
- Depth-first Search
 - Depth-first search trees
 - Time complexity
 - one-time complexity
 - Overview of DFS-algorithms
 - Distributed depth first search tree construction
 - Classical
 - Awerbuch

Recitation Graph

