Distributed Algorithms

LECTURER: V.A. ZAKHAROV

Лекция 9.

Leader election problem.
Lower bounds.
Optimal election.
Gallager–Humblet–Spira Algorithm (GHS).
Global description of GHS.
Deteailed descriptoion of GHS.
Corach–Katten–Moran Algorithm.

Leader Election Problem

The leader election problem is to find an algorithm which starts from a configuration where all processes are in the same state, and arrives at a configuration where exactly one process is in a state **leader** and all other processes are in the state **lost** .

Definition

A **leader election algorithm** is an algorithm which complies with the following requirements.

1. Every process executes the same local algorithm.
2. The algorithm is decentralized, i.e., a computation can be initialized by an arbitrary non-empty subset of processes.
3. The algorithm reaches a terminal configuration in each computation, and in each reachable terminal configuration there is exactly one process in the state **leader** and all other processes are in the state **lost**.

Lower bounds

Theorem 9.1.
Any comparison election algorithm for arbitrary networks has a (worst-case and average-case) message complexity of at least $\Omega(|E| + N \log N)$.

Corollary.
A decentralized wave algorithm for arbitrary networks without neighbor knowledge has a message complexity of at least $\Omega(|E| + N \log N)$.

Lower bounds

Proof.
The $\Omega(N \log N)$ term is a lower bound because arbitrary networks include rings, for which an $\Omega(N \log N)$ lower bound holds.

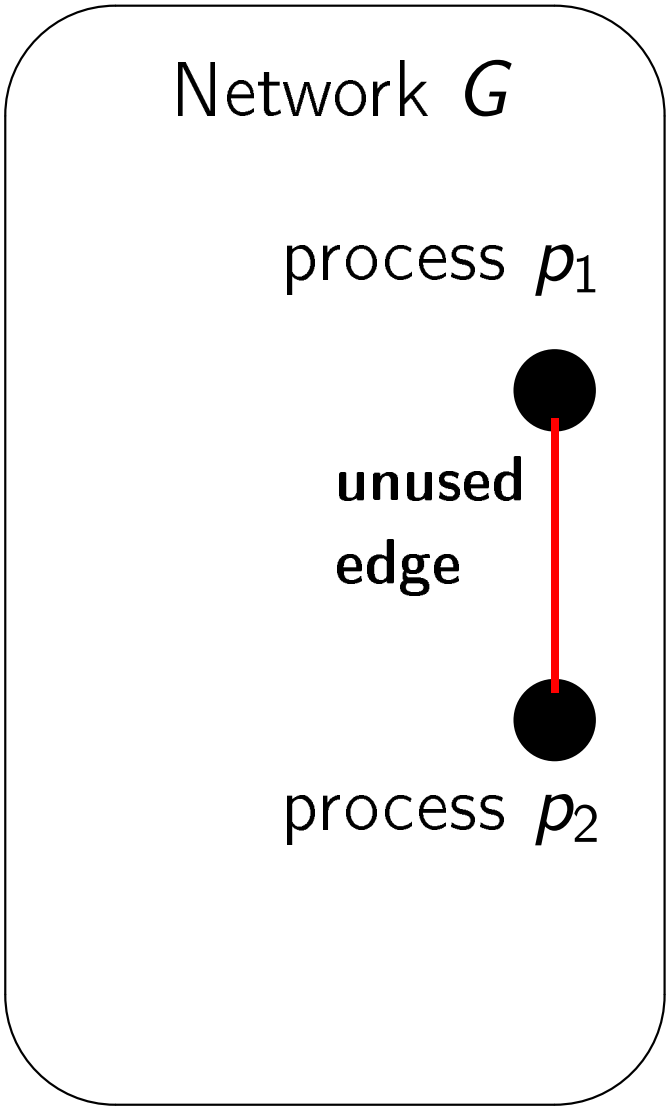
Lower bounds

Proof.
The $\Omega(N \log N)$ term is a lower bound because arbitrary networks include rings, for which an $\Omega(N \log N)$ lower bound holds.

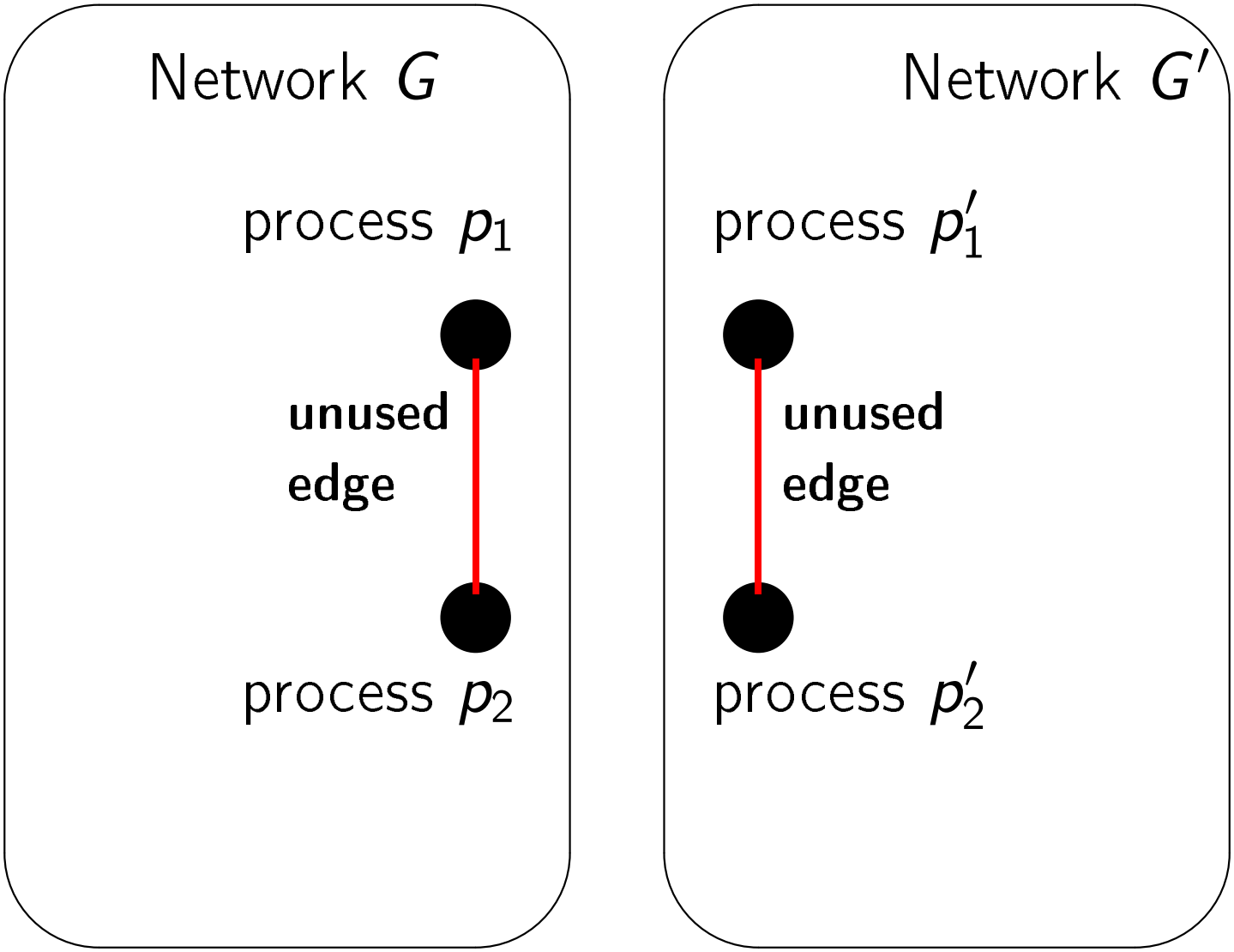
To see that $|E|$ messages is a lower bound, even in the best of all computations, assume that an election algorithm A has a computation C on network G in which fewer than $|E|$ messages are exchanged.

Construct a network G' by connecting two copies of G with one edge between nodes inserted on an edge that is not used in C .

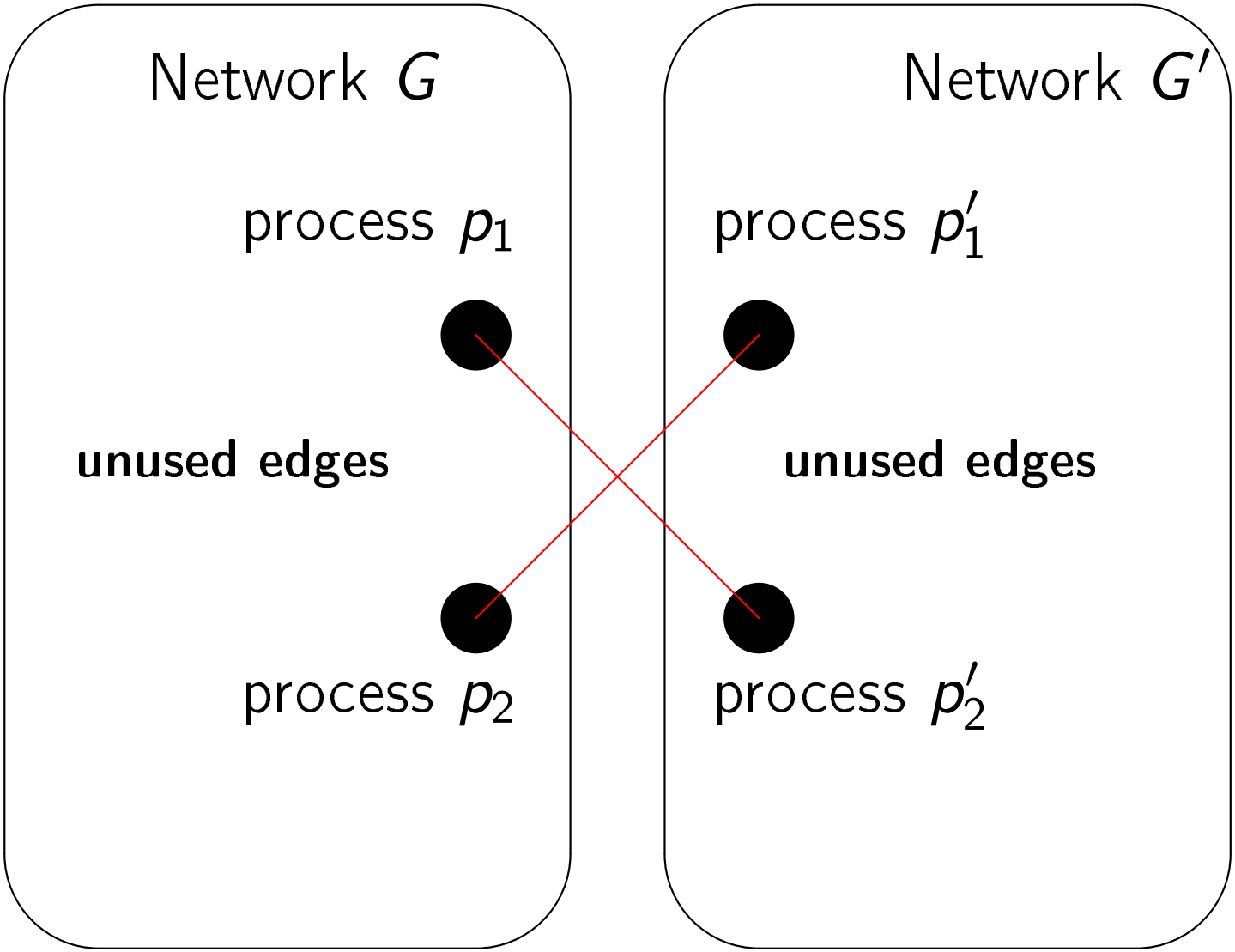
Lower bounds



Lower bounds



Lower bounds



Lower bounds

Proof.

The identities in both parts of the network $G \cup G'$ have the same relative order as in G .

Therefore, computation C can be simulated simultaneously in both parts G and G', yielding a computation of $G \cup G'$ in which two processes in both parts become elected.

□

Optimal election

TOPIC FOR RESEARCH.

Is it possible to carry out leader election in arbitrary networks using only

$$O(|E| + N \log N)$$

message exchanges?

Optimal election

TOPIC FOR RESEARCH.

Is it possible to carry out leader election in arbitrary networks using only

$$O(|E| + N \log N)$$

message exchanges?

Leader election problem is closely related with the problem of computing a spanning tree with a decentralized algorithm.

Let

C_E — message exchange complexity of leader election,
and

C_T — message exchange complexity of spanning tree computing.

Optimal election

Leader election in trees requires only $O(N)$ message exchanges.

This implies $C_E \leq C_T + O(N)$.

Optimal election

Leader election in trees requires only $O(N)$ message exchanges.

This implies $C_E \leq C_T + O(N)$.

If a leader is elected then a spanning tree can be computed with only $2|E|$ message exchanges with help of a centralized network traversal algorithm.

This implies $C_T \leq C_E + 2|E|$.

Optimal election

Leader election in trees requires only $O(N)$ message exchanges.

This implies $C_E \leq C_T + O(N)$.

If a leader is elected then a spanning tree can be computed with only $2|E|$ message exchanges with help of a centralized network traversal algorithm.

This implies $C_T \leq C_E + 2|E|$.

Thus, we can carry out an optimal leader election iff we can make an optimal computation of a spanning tree.

Gallager–Humblet–Spira (GHS) algorithm builds a spanning tree using $2|E| + 5N \log N$ message exchanges.

Gallager–Humblet–Spira (GHS) Algorithm

The GHS algorithm relies on the following assumptions.

Gallager–Humblet–Spira (GHS) Algorithm

The GHS algorithm relies on the following assumptions.

1. Every edge has a unique weight $\omega(e)$. All weights are totally ordered.

Gallager–Humblet–Spira (GHS) Algorithm

The GHS algorithm relies on the following assumptions.

1. Every edge has a unique weight $\omega(e)$. All weights are totally ordered.
2. All nodes though initially asleep awaken before they start the execution of the algorithm.
   Some nodes awaken spontaneously (if execution of the algorithm is triggered by circumstances occurring in these nodes), others may receive a message of the algorithm while they are still asleep.
   In the latter case a node upon receiving the first message begins it work with executing the local initialization procedure, and then processes the message.

Minimal spanning tree

Let $G = (V, E)$ be a weighted graph, where $\omega(e)$ denotes the weight of an edge e .
Weights of all edges are pairwise different and pairwise comparable.

The weight of a spanning tree T in a graph G is the sum of weights of all $N - 1$ edges contained in T .

We say that T is a minimal spanning tree , or, briefly, MST if no tree has a smaller weight than T .

Minimal spanning tree

Proposition 9.1.

If all edge weights are different, there is only one MST.

Minimal spanning tree

Proposition 9.1.

If all edge weights are different, there is only one MST.

Proof.

Suppose that there are two MST T_1 and T_2 such that $T_1 \neq T_2$. Consider the lowest-weight edge e that is in one of the trees, but not in both. Assume, without loss of generality, that $e \in T_1$.

Minimal spanning tree

Proposition 9.1.

If all edge weights are different, there is only one MST.

Proof.

Suppose that there are two MST T_1 and T_2 such that $T_1 \neq T_2$. Consider the lowest-weight edge e that is in one of the trees, but not in both. Assume, without loss of generality, that $e \in T_1$.

Then the graph $T_2 \cup \{e\}$ has a cycle, and, since the tree T_1 has no cycles, at least one edge of the cycle, say e' , is not contained in T_1 . By the choise of e it is true that $\omega(e) < \omega(e')$.

Minimal spanning tree

Proposition 9.1.

If all edge weights are different, there is only one MST.

Proof.

Suppose that there are two MST T_1 and T_2 such that $T_1 \neq T_2$. Consider the lowest-weight edge e that is in one of the trees, but not in both. Assume, without loss of generality, that $e \in T_1$.

Then the graph $T_2 \cup \{e\}$ has a cycle, and, since the tree T_1 has no cycles, at least one edge of the cycle, say e' , is not contained in T_1 . By the choise of e it is true that $\omega(e) < \omega(e')$.

Then the tree $T_2 \cup \{e\} \setminus \{e'\}$ has the weight less than the weight of T_2 , which contradicts that T_2 is MST. $\square$

Minimal spanning tree

A fragment is any subtree of MST. An edge e is called outgoing edge of a fragment F if one end of the edge e is in F , whereas the other is not.

Minimal spanning tree

A fragment is any subtree of MST. An edge e is called outgoing edge of a fragment F if one end of the edge e is in F , whereas the other is not.

Proposition 9.2.

If F is a fragment of MST, and e is the least-weight outgoing edge of F then $F \cup \{e\}$ is also a fragment of the MST.

Minimal spanning tree

A fragment is any subtree of MST. An edge e is called outgoing edge of a fragment F if one end of the edge e is in F , whereas the other is not.

Proposition 9.2.

If F is a fragment of MST, and e is the least-weight outgoing edge of F then $F \cup \{e\}$ is also a fragment of the MST.

Proof.

HELP YOURSELF

Minimal spanning tree

A fragment is any subtree of MST. An edge e is called outgoing edge of a fragment F if one end of the edge e is in F , whereas the other is not.

Proposition 9.2.

If F is a fragment of MST, and e is the least-weight outgoing edge of F then $F \cup \{e\}$ is also a fragment of the MST.

Proof.

HELP YOURSELF

Well-known sequential algorithms for computing an MST are the algorithms of Prim and Kruskal. Kruskal's algorithm starts with a collection of single-node fragments and merges fragments by adding the lowest-weight outgoing edge of some fragment.

Global description of GHS algorithm

A computation of the GHS algorithm proceeds according to the following steps.

1. A collection of fragments is maintained, such that the union of all fragments contains all nodes.
2. Initially this collection contains each node as a one-node fragment.
3. The nodes in a fragment cooperate to find the lowest-weight outgoing edge of the fragment.
4. When the lowest-weight outgoing edge of a fragment is known, the fragment will be combined with another fragment by adding the outgoing edge, in cooperation with the other fragment.
5. The algorithm terminates when only one fragment remains.

Global description of GHS algorithm

The efficient implementation of these steps requires the introduction of some notation and mechanisms.

Global description of GHS algorithm

The efficient implementation of these steps requires the introduction of some notation and mechanisms.

1. Fragment name.

To determine the lowest-weight outgoing edge it must be possible to see whether an edge is an outgoing edge or leads to a node in the same fragment.

To this end each fragment will have a name, which will be known to the processes in that fragment.

Processes test whether an edge is internal or outgoing by a comparison of their fragment names.

Global description of GHS algorithm

2. Combining large and small fragments.

When two fragments are combined, the fragment name of the processes in at least one of the fragment changes, which requires an update to take place in every node of at least one of the two fragments.

To keep this update efficient, the combining strategy is based on the idea that the smaller of two fragments combines into the larger of the two by adopting the fragment name of the larger fragment.

Global description of GHS algorithm

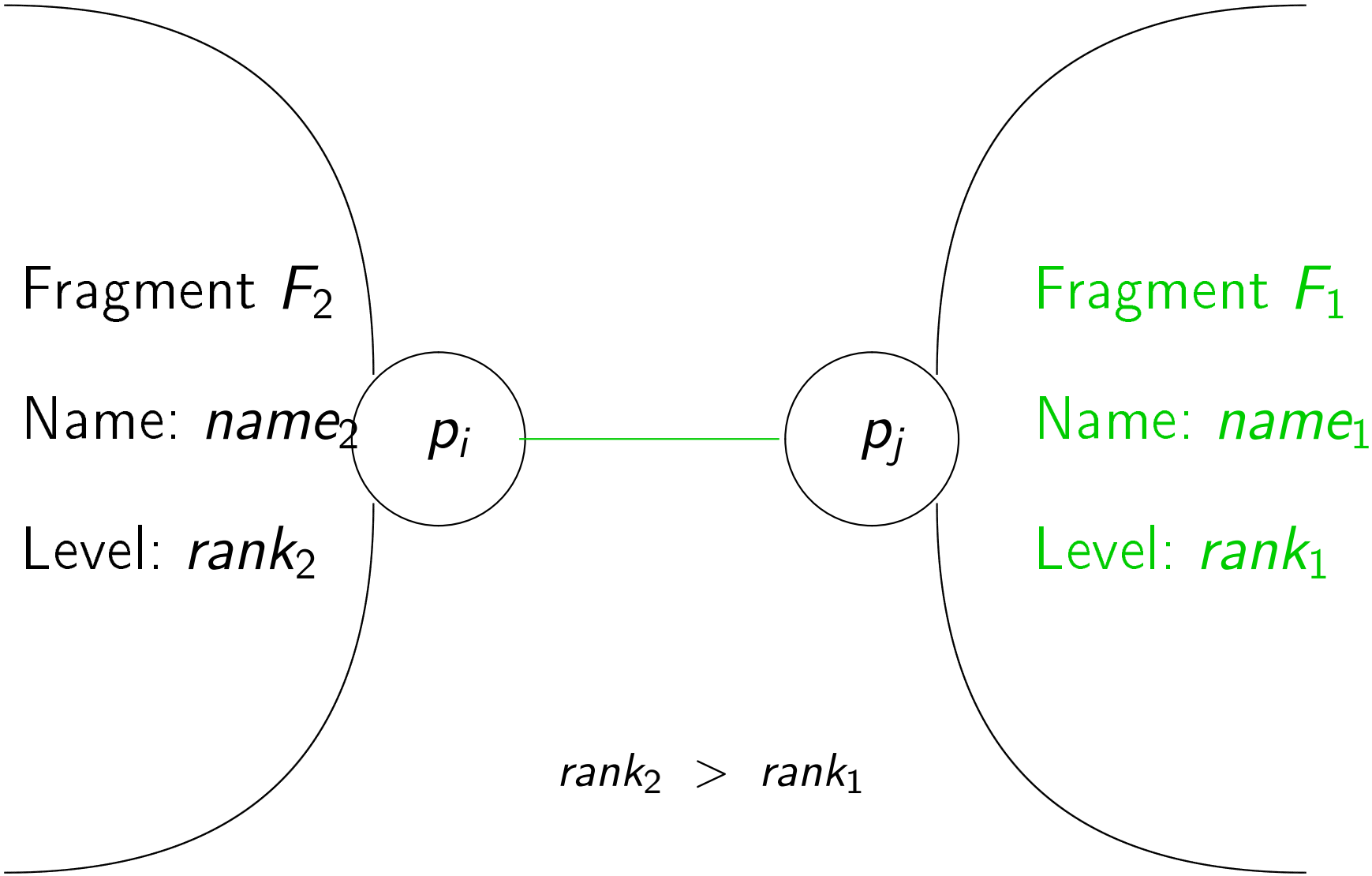
3. Fragment levels.

Each fragment is assigned a level , which is 0 for an initial one-node fragment.

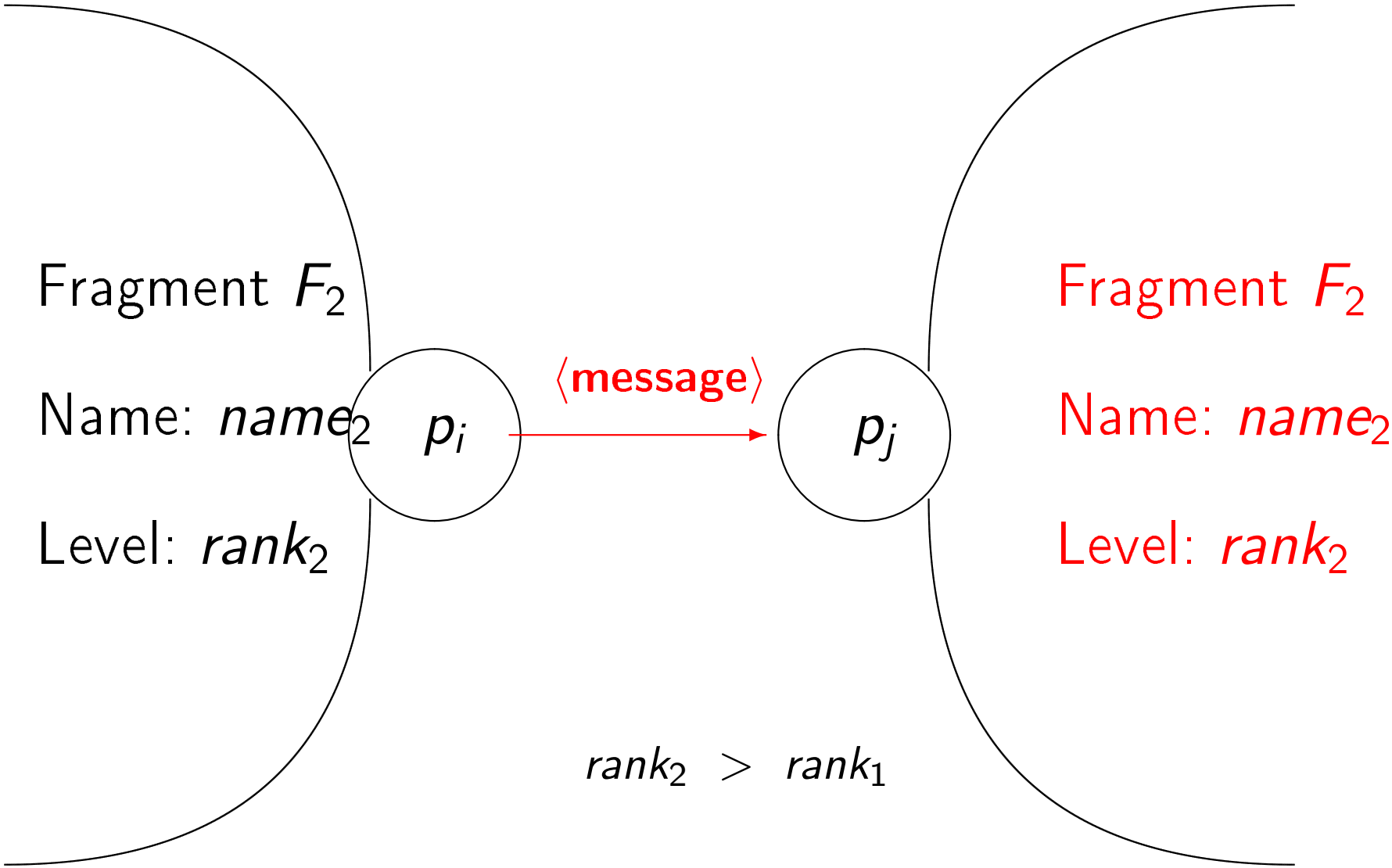
It is allowed that a fragment F_1 combines into a fragment F_2 with higher level, after which the new fragment $F_1 \cup F_2$ has the level of F_2 .

The new fragment also has the fragment name of F_2 , so no updates are necessary for the nodes in F_2 .

Global description of GHS algorithm



Global description of GHS algorithm



Global description of GHS algorithm

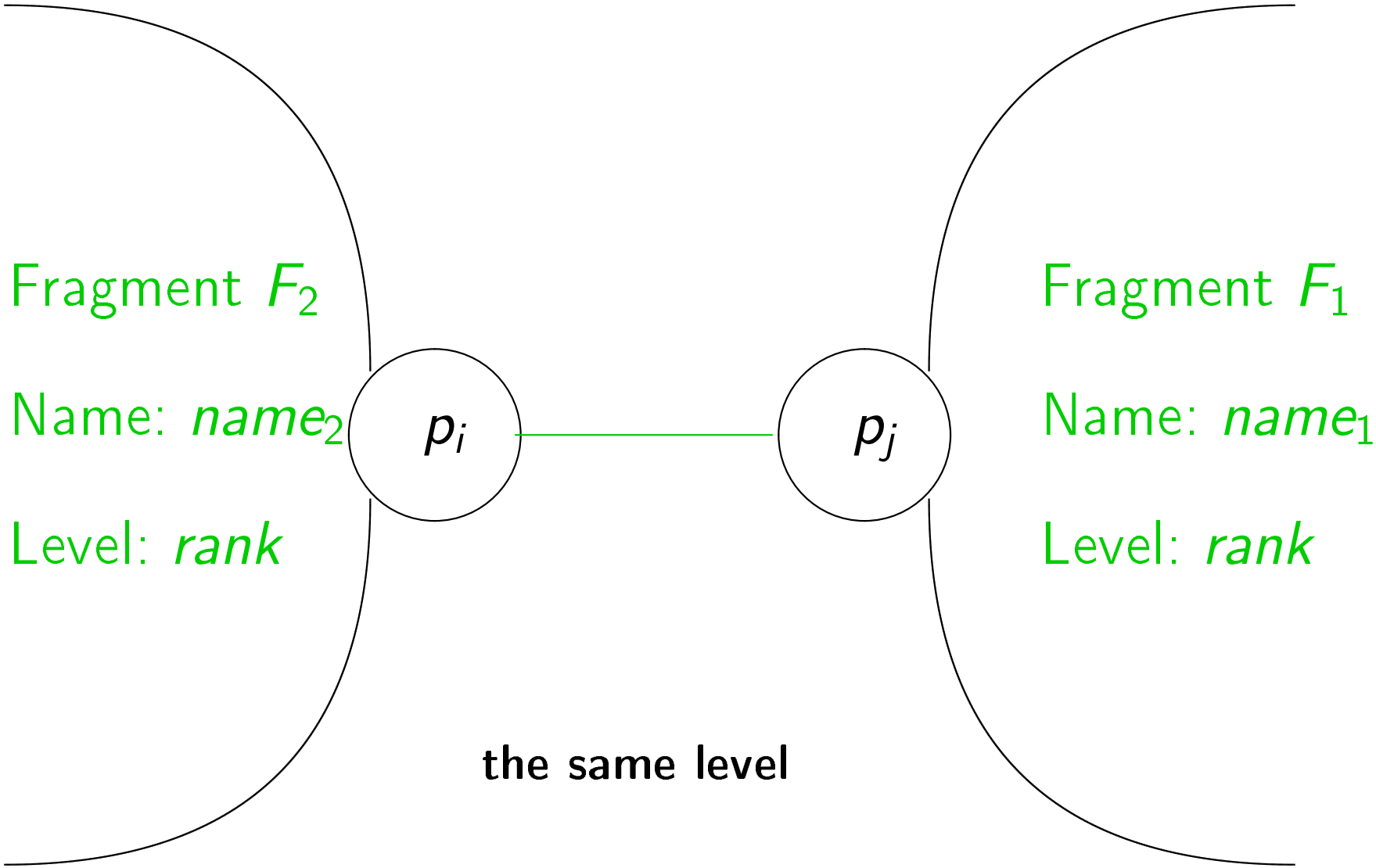
3. Fragment levels.

It must also be possible for two fragments of the same level to combine; in this case the new fragment has a new name and its level is one higher than the level of the combining fragments.

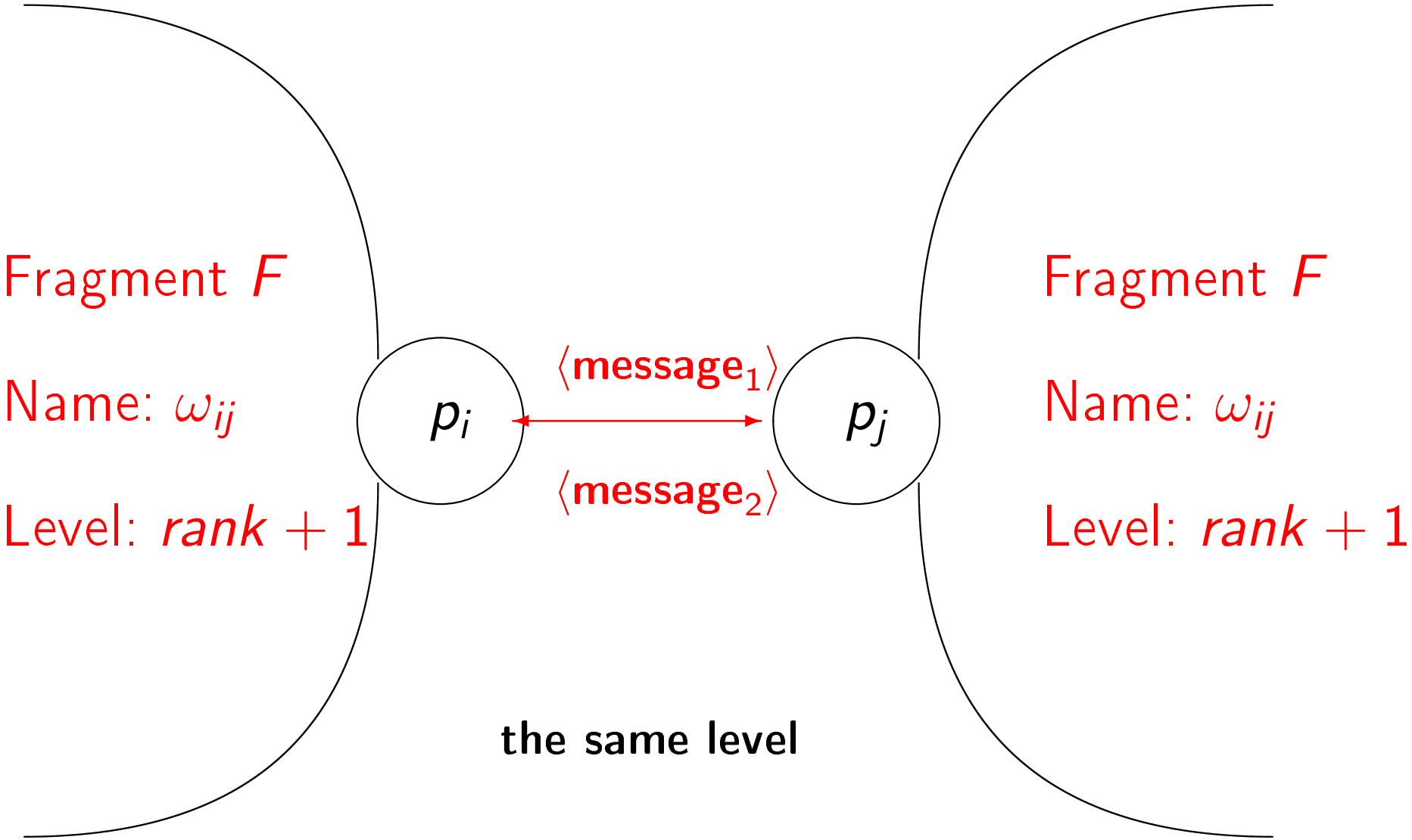
The new name of the fragment is the weight of the edge by which the two fragments are combined, and this edge is called the core edge of the new fragment.

The two nodes connected by the core edge are called the core nodes .

Global description of GHS algorithm



Global description of GHS algorithm



Global description of GHS algorithm

Proposition 9.3.

If these combining rules are obeyed, the number of times a process changes its fragment name or level is at most $N \log N$.

Global description of GHS algorithm

Proposition 9.3.

If these combining rules are obeyed, the number of times a process changes its fragment name or level is at most $N \log N$.

Proof.

The level of a process never decreases, and only when it increases does the process change its fragment name.

By induction on L it is easy to show that every fragment of a level L contains at least 2^L nodes; so the maximum level is $\log N$.

This implies that each individual process increases its fragment level at most $\log N$ times.
Hence, the overall total number of fragment name and level changes is bounded by $N \log N$.

Combining strategy

Let a fragment $F = (FN, L)$ has a name FN and a level L , and let e_F be the lowest-weight outgoing edge of F .

Combining strategy

Let a fragment $F = (FN, L)$ has a name FN and a level L , and let e_F be the lowest-weight outgoing edge of F .

Rule A. If e_F leads to a fragment $F' = (FN', L')$ with $L < L'$, then F joins to F' , and the new fragment will have name FN' and level L' .

Combining strategy

Let a fragment $F = (FN, L)$ has a name FN and a level L , and let e_F be the lowest-weight outgoing edge of F .

Rule A. If e_F leads to a fragment $F' = (FN', L')$ with $L < L'$, then F joins to F' , and the new fragment will have name FN' and level L' .

Rule B. If e_F leads to a fragment $F' = (FN', L')$ with $L = L'$ and $e_{F'} = e_F$, then these fragments merge into a new fragment which will have name $\omega(e_F)$ and level $L + 1$.

Combining strategy

Let a fragment $F = (FN, L)$ has a name FN and a level L , and let e_F be the lowest-weight outgoing edge of F .

Rule A. If e_F leads to a fragment $F' = (FN', L')$ with $L < L'$, then F joins to F' , and the new fragment will have name FN' and level L' .

Rule B. If e_F leads to a fragment $F' = (FN', L')$ with $L = L'$ and $e_{F'} = e_F$, then these fragments merge into a new fragment which will have name $\omega(e_F)$ and level $L + 1$.

Rule C. In all other cases (when $L > L'$ or $L = L'$ and $e_{F'} \neq e_F$) the fragment F must wait until rule A or B applies.

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$
- $bestwt_p$
- $level_p$
- $father_p$
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$ — operating mode of a process $\{\textbf{sleep}, \textbf{find}, \textbf{found}\}$;
- $stach_p[q]$
- $name_p$
- $bestwt_p$
- $level_p$
- $father_p$
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$ — link status $\{\textbf{basic}, \textbf{branch}, \textbf{reject}\}$;
- $name_p$
- $bestwt_p$
- $level_p$
- $father_p$
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$ — name of a fragment;
- $bestwt_p$
- $level_p$
- $father_p$
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$
- $bestwt_p$ — the lowest weight of outgoing edge;
- $level_p$
- $father_p$
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$
- $bestwt_p$
- $level_p$ — process level;
- $father_p$
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$
- $bestwt_p$
- $level_p$
- $father_p$ — edge leading toward core node;
- $testch_p, bestch_p$
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$
- $bestwt_p$
- $level_p$
- $father_p$
- $testch_p, bestch_p$ — promising edges toward neighbor fragments;
- rec

Detailed description of GHS algorithm

Variables.

- $state_p$
- $stach_p[q]$
- $name_p$
- $bestwt_p$
- $level_p$
- $father_p$
- $testch_p, bestch_p$
- rec — counter.

Detailed description of GHS algorithm

Variables.

- $state_p$ — operating mode of a process $\{\textbf{sleep}, \textbf{find}, \textbf{found}\}$;
- $stach_p[q]$ — link status $\{\textbf{basic}, \textbf{branch}, \textbf{reject}\}$;
- $name_p$ — name of a fragment;
- $bestwt_p$ — the lowest weight of outgoing edge;
- $level_p$ — process level;
- $father_p$ — edge leading toward core node;
- $testch_p, bestch_p$ — promising edges toward neighbor fragments;
- rec — counter.

Detailed description of GHS algorithm

Messages.

- $\langle$**connect**, *level*$\rangle$
- $\langle$**initiate**, *level*, *name*, *state*$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**connect**, *level*$\rangle$ [invitation to merge]— a message that is sent via the edge of the smallest weight outgoing from the fragment, indicating the level of this fragment;
- $\langle$**initiate**, *level*, *name*, *state*$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**connect**, *level*$\rangle$
- $\langle$**initiate**, *level*, *name*, *state*$\rangle$ [order to merge]— a message that is sent via the edge of the smallest weight outgoing from the fragment,
  when performing the **rule A** —attaching a fragment of a lower level, and also
  when performing the **rule B** — joining of two fragments of the same level;

Detailed description of GHS algorithm

Messages.

- $\langle$**connect**, *level*$\rangle$ [invitation to merge]— a message that is sent via the edge of the smallest weight outgoing from the fragment, indicating the level of this fragment;
- $\langle$**initiate**, *level*, *name*, *state*$\rangle$ [order to merge]— a message that is sent via the edge of the smallest weight outgoing from the fragment,
  when performing the **rule A** —attaching a fragment of a lower level, and also
  when performing the **rule B** — joining of two fragments of the same level;

Detailed description of GHS algorithm

Messages.

- $\langle$**test**, *level*, *name*$\rangle$
- $\langle$**reject**$\rangle$
- $\langle$**accept**$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**test**, *level*, *name*$\rangle$ [test an edge] — a message that is sent via a «fresh» edge of the smallest weight, outgoing from the node, indicating the name and the level of the fragment to which the node belongs;
- $\langle$**reject**$\rangle$
- $\langle$**accept**$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**test**, *level*, *name*$\rangle$
- $\langle$**reject**$\rangle$ — a message that is sent via a edge when this edge connects two nodes from the same fragment;
- $\langle$**accept**$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**test**, *level*, *name*$\rangle$
- $\langle$**reject**$\rangle$
- $\langle$**accept**$\rangle$ — a message that is sent via a edge when this edge connects two nodes from different fragments;

Detailed description of GHS algorithm

Messages.

- $\langle$**test**, *level*, *name*$\rangle$ [test an edge] — a message that is sent via a «fresh» edge of the smallest weight, outgoing from the node, indicating the name and the level of the fragment to which the node belongs;
- $\langle$**reject**$\rangle$ — a message that is sent via a edge when this edge connects two nodes from the same fragment;
- $\langle$**accept**$\rangle$ — a message that is sent via a edge when this edge connects two nodes from different fragments;

Detailed description of GHS algorithm

Messages.

- $\langle$**report**, *bestwt*$\rangle$
- $\langle$**changeroot**$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**report**, *bestwt*$\rangle$ — a message that is sent towards the core node, indicating the smallest weight of a «fresh» edge which outgoes from a «subordinate» node;
- $\langle$**changeroot**$\rangle$

Detailed description of GHS algorithm

Messages.

- $\langle$**report**, *bestwt*$\rangle$

- $\langle$**changeroot**$\rangle$ — a message which is sent via the tree to the lowest-weight outgoing edge.

Detailed description of GHS algorithm

Messages.

- $\langle$**report**, *bestwt*$\rangle$ — a message that is sent towards the core node, indicating the smallest weight of a «fresh» edge which outgoes from a «subordinate» node;
- $\langle$**changeroot**$\rangle$ — a message which is sent via the tree to the lowest-weight outgoing edge.

Detailed description of GHS algorithm

(1) The first action of every process
is the initialization of this algorithm:
begin let pq be a channel of the process p,
which has the smallest weight ;
$stach_p[q] := branch$; $level_p := 0$;
$state_p := found$; $rec_p := 0$;
send $\langle \mathbf{connect}, 0 \rangle$ to q
end

Each process begins the computation
with assigning itself the lowest rank,
finding the outgoing edge of the smallest weight,
and announcing its readiness to join the neighboring fragment.

Detailed description of GHS algorithm

(2) After receiving a message $\langle \textbf{connect}, L \rangle$ from q:

begin if $L < level_p$ **then** (* Combine with Rule A *)
begin $stach_p[q] := branch$;
send $\langle \textbf{initiate}, level_p, name_p, state_p \rangle$ to q
end
else if $stach_p[q] = basic$ or $L > level_p$
then (* Rule C *)
process this message later
else (* Rule B *)
send $\langle \textbf{initiate}, level_p + 1, \omega(pq), find \rangle$ to q
end

After receiving the proposal to merge fragments, the node that received this proposal checks the level of the neighboring fragment,...

Detailed description of GHS algorithm

(2) After receiving a message $\langle \textbf{connect}, L \rangle$ from q:

begin if $L < level_p$ **then** (* Combine with Rule A *)

begin $stach_p[q] := branch$;

send $\langle \textbf{initiate}, level_p, name_p, state_p \rangle$ to q

end

else if $stach_p[q] = basic$ **or** $L > level_p$

then (* Rule C *)

process this message later

else (* Rule B *)

send $\langle \textbf{initiate}, level_p + 1, \omega(pq), find \rangle$ to q

end

and if the level of the neighboring fragment is less than the level of the fragment the node p belongs to, then the node p sends an order to the neighboring fragment to join as a «subordinate», taking the name and the level of the senior fragment.

Detailed description of GHS algorithm

(2) After receiving a message $\langle \textbf{connect}, L \rangle$ from q:

begin if $L < level_p$ **then** (* Combine with Rule A *)
begin $stach_p[q] := branch$;
send $\langle \textbf{initiate}, level_p, name_p, state_p \rangle$ to q
end
else if $stach_p[q] = basic$ **or** $L > level_p$
then (* Rule C *)
process this message later
else (* Rule B *)
send $\langle \textbf{initiate}, level_p + 1, \omega(pq), find \rangle$ to q
end

Otherwise the process p checks if the edge pq has been «tested» by the process p earlier,...

Detailed description of GHS algorithm

(2) After receiving a message $\langle \textbf{connect}, L \rangle$ from q:

begin if $L < level_p$ **then** (* Combine with Rule A *)

begin $stach_p[q] := branch$;

send $\langle \textbf{initiate}, level_p, name_p, state_p \rangle$ to q

end

else if $stach_p[q] = basic$ **or** $L > level_p$

then (* Rule C *)

process this message later

else (* Rule B *)

send $\langle \textbf{initiate}, level_p + 1, \omega(pq), find \rangle$ to q

end

and if this edge has not been «tested» by the process p , then the processing of the proposal to merge is delayed until the process p starts evaluating the edge pq .

Detailed description of GHS algorithm

(2) After receiving a message $\langle \textbf{connect}, L \rangle$ from q:

begin if $L < level_p$ **then** (* Combine with Rule A *)
begin $stach_p[q] := branch$;
send $\langle \textbf{initiate}, level_p, name_p, state_p \rangle$ to q
end
else if $stach_p[q] = basic$ **or** $L > level_p$
then (* Rule C *)
process this message later
else (* Rule B *)
send $\langle \textbf{initiate}, level_p + 1, \omega(pq), find \rangle$ to q
end

In the case when the edge pq has been evaluated by p as a «promissing for joining» then p sends an order to the neighboring fragment to merge as equal partners and to move at a higher level, taking the weight of the core edge pq as a new name for the combined fragment.

Detailed description of GHS algorithm

CHECK YOURSELF QUESTION:

And why in the last case the process p may be sure that the fragments have the same level?

Is it possible that the neighboring fragment (which includes the node q) has a level greater than the level of the fragment which contains the node p ?

And what the process p does in this case?

Detailed description of GHS algorithm

(3) After receiving a message $\langle \textbf{initiate}, L, F, S \rangle$ от q:

begin $level_p := L$; $name_p := F$; $state_p := S$; $father_p := q$;
$bestch_p := udef$; $bestwt_p := \infty$;
forall $r \in Neigh_p$: $stach_p[r] = branch \wedge r \neq q$ **do**
send $\langle \textbf{initiate}, L, F, S \rangle$ to r ;
if $state_p = find$ **then begin** $rec_p := 0$; $test$ **end**
end

After receiving an order to merge, the process p changes the name of the fragment, the level, the operating mode, and the direction toward the core edge,...

Detailed description of GHS algorithm

(3) After receiving a message $\langle \textbf{initiate}, L, F, S \rangle$ от q:

begin $level_p := L$; $name_p := F$; $state_p := S$; $father_p := q$;
$bestch_p := udef$; $bestwt_p := \infty$;
forall $r \in Neigh_p$: $stach_p[r] = branch \wedge r \neq q$ **do**
send $\langle \textbf{initiate}, L, F, S \rangle$ to r ;
if $state_p = find$ **then begin** $rec_p := 0$; $test$ **end**
end

«discards» its current data concerning the search of the outgoing edge of the smallest weight,...

Detailed description of GHS algorithm

(3) After receiving a message $\langle \textbf{initiate}, L, F, S \rangle$ от q:

begin $level_p := L$; $name_p := F$; $state_p := S$; $father_p := q$;
$bestch_p := udef$; $bestwt_p := \infty$;
forall $r \in Neigh_p : stach_p[r] = branch \wedge r \neq q$ **do**
send $\langle \textbf{initiate}, L, F, S \rangle$ to r ;
if $state_p = find$ **then begin** $rec_p := 0$; $test$ **end**
end

sends to its neighbors in the fragment an order to merge because of the formation of a new fragment,...

Detailed description of GHS algorithm

(3) After receiving a message $\langle \textbf{initiate}, L, F, S \rangle$ от q:

begin $level_p := L$; $name_p := F$; $state_p := S$; $father_p := q$;
$bestch_p := udef$; $bestwt_p := \infty$;
forall $r \in Neigh_p$: $stach_p[r] = branch \wedge r \neq q$ **do**
send $\langle \textbf{initiate}, L, F, S \rangle$ to r ;
if $state_p = find$ **then begin** $rec_p := 0$; $test$ **end**
end

and, in the case when the operating mode of the node requires to continue the search of an outgoing edge with smallest weight, join this search by invoking the procedure *test*.

Detailed description of GHS algorithm

CHECK YOURSELF QUESTION.

And why the process p «discards» its current data of the searach (variables *bestch* and *bestwt*) instead of continuing the search while preserving the achieved data?

Will the algorithm operate correctly if this «reset» of search indicators is not done?

Detailed description of GHS algorithm

(4) **procedure** *test*:
begin if $\exists q \in Neigh_p : \ stach_p[q] = basic$ **then**
begin $testch_p := q$ with $stach_p[q] = basic$
and $\omega(pq)$ minimal ;
send $\langle$**test**, $level_p, name_p\rangle$ to $testch_p$
end
else begin $testch_p := udef$; *report* **end**
end

The node p when attempting to find an outgoing edge of the smallest weight, scans one after the other in increasing weight all adjoining edges with the status *basic*.

Detailed description of GHS algorithm

(4) **procedure** *test*:
begin if $\exists q \in Neigh_p$: $stach_p[q] = basic$ **then**
begin $testch_p := q$ with $stach_p[q] = basic$
and $\omega(pq)$ minimal ;
send $\langle$**test**, $level_p, name_p\rangle$ to $testch_p$
end
else begin $testch_p := udef$; *report* **end**
end

If the node p has at least one outgoing «fresh» edge (which has the status *basic*) then the process p chooses the edge of the smallest weight, and requests via this channel a name and a level of the fragment the neighbor q belongs to.

Detailed description of GHS algorithm

(4) **procedure** *test*:
begin if $\exists q \in Neigh_p$: $stach_p[q] = basic$ **then**
begin $testch_p := q$ with $stach_p[q] = basic$
and $\omega(pq)$ minimal ;
send $\langle$**test**, $level_p, name_p\rangle$ to $testch_p$
end
else begin $testch_p := udef$; *report* **end**
end

And if all outgoing edges have been tested earlier, then the process p launches the procedure *report* to prepare the report on the results of the search.

Detailed description of GHS algorithm

(5) After receiving a message $\langle \textbf{test}, L, F \rangle$ from q:

begin if $L > level_p$ **then** (* Answer must wait! *)
process this message later
else if $F = name_p$ **then** (* internal edge *)
begin if $stach_p[q] = basic$ **then** $stach_p[q] := reject$;
if $q \neq testch_p$
then send $\langle \textbf{reject} \rangle$ to q
else *test*
end
else send $\langle \textbf{accept} \rangle$ to q
end

Upon receipt of a request for the name and rank of the fragment to which the node p belongs check the level of the fragment to which the node that sent the request belongs. If the request came from a fragment of a higher level then the response is delayed (Why??? The node p is not sure about something?).

Detailed description of GHS algorithm

(5) After receiving a message $\langle \textbf{test}, L, F \rangle$ from q:

begin if $L > level_p$ **then** (* Answer must wait! *)
process this message later
else if $F = name_p$ **then** (* internal edge *)
begin if $stach_p[q] = basic$ **then** $stach_p[q] := reject$;
if $q \neq testch_p$
then send $\langle \textbf{reject} \rangle$ to q
else *test*
end
else send $\langle \textbf{accept} \rangle$ to q
end

If the level of the fragment to which the node that sent the request belongs does not exceed the level of the fragment which includes the node p then the process p checks the name of the fragment which sends the request. And if p finds that the node q belongs to the same fragment then ...

Detailed description of GHS algorithm

(5) After receiving a message $\langle \textbf{test}, L, F \rangle$ from q:

begin if $L > level_p$ **then** (* Answer must wait! *)
process this message later
else if $F = name_p$ **then** (* internal edge *)
begin if $stach_p[q] = basic$ **then** $stach_p[q] := reject$;
if $q \neq testch_p$
then send $\langle \textbf{reject} \rangle$ to q
else *test*
end
else send $\langle \textbf{accept} \rangle$ to q
end

this means that the edge pq is an internal edge of the fragment, and it gets the status *reject*. After that, either a message $\langle \textbf{reject} \rangle$ about this is sent to the node q , or p launches the procedure *test* (when q is a node to which the edge of the smallest weight leads from p).

Detailed description of GHS algorithm

CHECK YOURSELF QUESTIONS.

And why otherwise the process p does not send to q a message $\langle$**reject**$\rangle$?

Does it thereby mislead q the status of the edge pq ?

And why it starts again the procedure *test* ?

Detailed description of GHS algorithm

(5) After receiving a message $\langle \textbf{test}, L, F \rangle$ from q:

begin if $L > level_p$ **then** (* Answer must wait! *)
process this message later;
else if $F = name_p$ **then** (* internal edge *)
begin if $stach_p[q] = basic$ **then** $stach_p[q] := reject$;
if $q \neq testch_p$
then send $\langle \textbf{reject} \rangle$ to q
else *test*
end
else send $\langle \textbf{accept} \rangle$ to q
end

And, finally, if the nodes p and q belong to different fragments then p informs the node q about this.

Detailed description of GHS algorithm

(6) After receiving a message ⟨**accept**⟩ from q:

begin $testch_p := udef$;
  if $\omega(pq) < bestwt_p$
    then begin $bestwt_p := \omega(pq)$; $bestch_p := q$ **end**;
  report
end

If p receives a notification that the edge pq is a channel between the nodes which belong to different fragments, then the process p completes the search of the best possible edge, it assumes that this edge is pq , and begins to prepare a report to the core node by launching the procedure *report*.

Detailed description of GHS algorithm

CHECK YOURSELF QUESTIONS.

Why is it necessary to «discard» the current value of $testch_p$?

Does the algorithm remain correct if one deletes the assignment $testch_p := udef$?

Detailed description of GHS algorithm

(7) After receiving of a message $\langle$**reject**$\rangle$ from q:
begin if $stach_p[q] = basic$ **then** $stach_p[q] := reject$;
test
end

If p receives an announcement that the edge pq connects the nodes which belong to the same fragment, then p takes into account this fact, and begins to search another outgoing edge by launching the procedure *test*.

Detailed description of GHS algorithm

(8) **procedure** *report*:
begin if $rec_p = \#\{q : \ stach_p[q] = branch \wedge q \neq father_p\}$
and $testch_p = udef$ **then**
begin $state_p := found$; send $\langle$**report**, $bestwt_p\rangle$ to $father_p$ **en**
end

If p receives from all its neighbors in the fragment (except those which are located on the path to the core nodes) data on the best possible edges, and, moreover, the process p is not engaged in the search of the best possible outgoing edge, then it sends towards the core edge a message which indicate the weight of the best possible edge.

Detailed description of GHS algorithm

(9) After receiving of a message $\langle \textbf{report}, \omega \rangle$ from q:

begin if $q \neq father_p$
then (* reply for **initiate** message *)
begin if $\omega < bestwt_p$ **then**
begin $bestwt_p := \omega$; $bestch_p := q$ **end**;
$rec_p := rec_p + 1$; *report*
end
else (* pq leads towards the core nodes *)
if $state_p = find$
then process this message later
else if $\omega > bestwt_p$
then *changeroot*
else if $\omega = bestwt_p = \infty$ **then stop**
end

If a process p receives from q (this process may be only the neighbor in the same fragment (why?)) a report on the weight of the best outgoing edge, then p checks where this edge leads to.

Detailed description of GHS algorithm

(9) After receiving of a message $\langle \textbf{report}, \omega \rangle$ from q:

begin if $q \neq father_p$

then (* reply for **initiate** message *)

begin if $\omega < bestwt_p$ **then**

begin $bestwt_p := \omega$; $bestch_p := q$ **end**;

$rec_p := rec_p + 1$; *report*

end

else (* pq leads towards the core nodes *)

if $state_p = find$

then process this message later

else if $\omega > bestwt_p$

then *changeroot*

else if $\omega = bestwt_p = \infty$ **then stop**

end

If pq is not on the path from p to the core node of the same fragment which includes p , then p estimate the weight and the location of the best possible edge, and begins to prepare a report, by launching the procedure *report*.

Detailed description of GHS algorithm

(9) After receiving of a message $\langle \textbf{report}, \omega \rangle$ from q:

begin if $q \neq father_p$

then (* reply for **initiate** message *)

begin if $\omega < bestwt_p$ **then**

begin $bestwt_p := \omega$; $bestch_p := q$ **end**;

$rec_p := rec_p + 1$; *report*

end

else (* pq leads towards the core nodes *)

if $state_p = find$

then process this message later

else if $\omega > bestwt_p$

then *changeroot*

else if $\omega = bestwt_p = \infty$ **then stop**

end

If a report delivered to p was received from a process q which is on the path from p to the core edge, then p , in the case when it did not complete the search of the best possible edge, delay the processing of this message until it finds the best possible edge.

Detailed description of GHS algorithm

(9) After receiving of a message $\langle \textbf{report}, \omega \rangle$ from q:

begin if $q \neq father_p$

then (* reply for **initiate** message *)

begin if $\omega < bestwt_p$ **then**

begin $bestwt_p := \omega$; $bestch_p := q$ **end**;

$rec_p := rec_p + 1$; *report*

end

else (* pq leads towards the core nodes *)

if $state_p = find$

then process this message later

else if $\omega > bestwt_p$

then *changeroot*

else if $\omega = bestwt_p = \infty$ **then stop**

end

If the weight of an edge found by p is less than the weight of an edge found by q then the joining of the fragment which includes p will be made via the edge which has been found by p . Therefore, p invokes the procedure *changeroot*.

Detailed description of GHS algorithm

(9) After receiving of a message $\langle \textbf{report}, \omega \rangle$ from q:

begin if $q \neq father_p$

then (* reply for **initiate** message *)

begin if $\omega < bestwt_p$ **then**

begin $bestwt_p := \omega$; $bestch_p := q$ **end**;

$rec_p := rec_p + 1$; *report*

end

else (* pq leads towards the core nodes *)

if $state_p = find$

then process this message later

else if $\omega > bestwt_p$

then *changeroot*

else if $\omega = bestwt_p = \infty$ **then stop**

end

Otherwise, the fragment which includes p will merge another fragment of the same level via an edge which locates on the q 's side of the fragment, and, therefore, the data collected by p are irrelevant.

Detailed description of GHS algorithm

(10) **procedure** *changeroot*:
begin if $stach_p[bestch_p] = branch$
then send $\langle$**changeroot**$\rangle$ to $bestch_p$
else begin send $\langle$**connect**, $level_p\rangle$ to $bestch_p$;
$stach_p[bestch_p] := branch$
end
end

This procedure initiates the merging of the fragment along the found outgoing edge of the smallest weight. A message $\langle$**changeroot**$\rangle$ is forwarded towards the outgoing edge of the smallest weight, until...

Detailed description of GHS algorithm

(10) **procedure** *changeroot*:
begin if $stach_p[bestch_p] = branch$
then send $\langle$**changeroot**$\rangle$ to $bestch_p$
else begin send $\langle$**connect**, $level_p\rangle$ to $bestch_p$;
$stach_p[bestch_p] := branch$
end
end

this message arrives at the node which has the outgoing edge of the smallest weight. Then, from this node, a merge proposal $\langle$**connect**, $level_p\rangle$ is sent to the neighboring fragment.

Detailed description of GHS algorithm

(11) Upon receipt of a message ⟨**changeroot**⟩:
begin *changeroot* **end**

No comments.

Thus, every process in the network may perform 11 actions (1)–(11) with the using of procedures *test*, *report*, and *changeroot*.

Detailed description of GHS algorithm

Theorem 9.3.

GHS Algorithm computes the minimal spanning tree using at most $5N \log N + 2|E|$ message exchanges.

Proof.

From the detailed description of the algorithm it follows that a message $\langle \textbf{connect}, L \rangle$ can be sent only via an edge of the smallest weight outgoing from the fragment. This implies that MST will be constructed correctly if every fragment really sends such a message and merges another fragment, despite the processing delay for some messages induced by this algorithm. All we need to show is that the algorithm will never reach a deadlock.

Deadlock potentially arises in situations where nodes or fragments must wait until some condition occurs in another node or fragment. The waiting introduced for $\langle \textbf{report}, \omega \rangle$ messages on the core edge does not lead to a deadlock because each core node eventually receives reports from all sons (unless the fragment as a whole waits for another fragment), after which the message will be processed.

Detailed description of GHS algorithm

Proof

Consider the case where a message of a fragment $F_1 = (level_1, name_1)$ arrives at some node of a fragment $F_2 = (level_2, name_2)$.

Detailed description of GHS algorithm

Proof

Consider the case where a message of a fragment $F_1 = (level_1, name_1)$ arrives at some node of a fragment $F_2 = (level_2, name_2)$.

A message $\langle \textbf{connect}, level_1 \rangle$ must wait if $level_1 \geq level_2$ and no $\langle \textbf{connect}, level_2 \rangle$ message has been sent through the same edge by fragment F_2 (see. (2)).

Detailed description of GHS algorithm

Proof

Consider the case where a message of a fragment $F_1 = (level_1, name_1)$ arrives at some node of a fragment $F_2 = (level_2, name_2)$.

A message $\langle \textbf{connect}, level_1 \rangle$ must wait if $level_1 \geq level_2$ and no $\langle \textbf{connect}, level_2 \rangle$ message has been sent through the same edge by fragment F_2 (see. (2)).

A test message $\langle \textbf{test}, level_1, name_1 \rangle$ must wait if $level_1 > level_2$ (see. (5)).

Detailed description of GHS algorithm

Proof.

In all cases where F_1 waits for F_2 , one of the following holds.:

1. $level_1 > level_2$;
2. $level_1 = level_2 \wedge \omega(e_{F_1}) > \omega(e_{F_2})$;
3. $level_1 = level_2 \wedge \omega(e_{F_1}) = \omega(e_{F_2})$ and F_2 is still searching for its lowest-weight outgoing edge. (Since e_{F_1} is an edge outgoing from F_2 , the inequality $\omega(e_{F_2}) > \omega(e_{F_1})$ does not hold.)

Thus, no deadlock cycle may occur.

Detailed description of GHS algorithm

Proof.

It remains only to estimate the complexity of the algorithm.

HOMETASK 1.

- Estimate how many messages of each type are sent by processes as the algorithm runs on the network with N nodes and E edges.
- Based on these estimates, show that the complexity of the GHS algorithm in terms of the number of message exchanges does not exceed $2N \log N + 5|E|$.

Corach–Katten–Moran Algorithm

There is a close relationship between the traversal of networks and leader election problem. Therefore, one can build an efficient leader election algorithm for an arbitrary class of networks based on any network traversal algorithm.

The Korach–Kutten–Moran algorithm uses the ideas of

- **the extinction construction**,
- **the Peterson/Dolev–Klawe–Rodeh algorithm**.

Corach–Katten–Moran Algorithm

The resemblance to the **extinction construction** is that initiators of the election start a network traversal with a token labeled with their IDs.

Processes can absorb tokens engaged in traversals launched by the initiators. If some traversal completes successfully then its initiator becomes a leader.

To this end only one traversal should be completed; all others must be interrupted.

Corach–Katten–Moran Algorithm

To interrupt redundant traversals the algorithm operates in **levels** that can be compared to the rounds of the Peterson/Dolev–Klawe–Rodeh algorithm.

If at least two traversals are started, the tokens will arrive at a process that has already been visited by another token. If this situation arises the traversal carried out by the arriving token will be aborted.

The goal of the algorithm is bring two tokens together in one process, where they will be removed («killed») and a new traversal initiated.

Corach–Katten–Moran Algorithm

Principle 1.

The notion of rounds is replaced in the Korach–Kutten–Moran algorithm by the notion of levels ; two tokens will give rise to a new traversal only if they have the same level, and the newly generated token has a level that is one higher.

A level of a node is a level of a token whose traversal is provided by this node. A level of a node may change when other tokens visit this node.

A token is represented by (q, ℓ) , where q is the ID of an initiator of the token and ℓ is its level.

If a token meets with a token of higher level, or arrives at a node already visited by a token of higher level, the arriving token is simply removed without influencing the token at higher level.

Corach–Katten–Moran Algorithm

Principle 2.

In order to bring tokens of the same level together in one process, each token can be in one of three modes: **annexing** , **chasing** , or **waiting** .

Every token is initiated in **annexing** mode and in this mode it obeys the traversal algorithm

The leader election algorithm interacts with the traversal algorithm by a call to the function *trav* ; this function returns either the neighbor to which the token must be forwarded, or **decide** if the traversal terminates.

Corach–Katten–Moran Algorithm

Principle 3.

A token (q, ℓ) in the **annexing** mode obeys the traversal algorithm until

Corach–Katten–Moran Algorithm

Principle 3.

A token (q, ℓ) in the **annexing** mode obeys the traversal algorithm until

1. the traversal algorithm terminates: then its initiator becomes a leader;

Corach–Katten–Moran Algorithm

Principle 3.

A token (q, ℓ) in the **annexing** mode obeys the traversal algorithm until

1. the traversal algorithm terminates: then its initiator becomes a leader;
2. the token is removed at the node of a higher level;

Corach–Katten–Moran Algorithm

Principle 3.

A token (q, ℓ) in the **annexing** mode obeys the traversal algorithm until

1. the traversal algorithm terminates: then its initiator becomes a leader;
2. the token is removed at the node of a higher level;
3. the token arrives in a node where a token of the same level ℓ is waiting: then both tokens are removed and a new traversal is started from that node;

Corach–Katten–Moran Algorithm

Principle 3.

A token (q, ℓ) in the **annexing** mode obeys the traversal algorithm until

1. the traversal algorithm terminates: then its initiator becomes a leader;
2. the token is removed at the node of a higher level;
3. the token arrives in a node where a token of the same level ℓ is waiting: then both tokens are removed and a new traversal is started from that node;
4. the token arrives at a node with the same level ℓ that has been most recently visited by a token of ID $p > q$ or by a **chasing** token: then the token becomes **waiting** in that node;

Corach–Katten–Moran Algorithm

Principle 3.

A token (q, ℓ) in the **annexing** mode obeys the traversal algorithm until

1. the traversal algorithm terminates: then its initiator becomes a leader;
2. the token is removed at the node of a higher level;
3. the token arrives in a node where a token of the same level ℓ is waiting: then both tokens are removed and a new traversal is started from that node;
4. the token arrives at a node with the same level ℓ that has been most recently visited by a token of ID $p > q$ or by a **chasing** token: then the token becomes **waiting** in that node;
5. the token arrives at a node of the same level ℓ that has been most recently visited by an **annexing** token with ID $p < q$: then the token becomes **chasing** and it is forwarded via the same channel as the previous token.

Corach–Katten–Moran Algorithm

Principle 4.

A **chasing** token (q, ℓ) is forwarded in each node via the channel through which the most recently passed token was sent, until

Corach–Katten–Moran Algorithm

Principle 4.

A **chasing** token (q, ℓ) is forwarded in each node via the channel through which the most recently passed token was sent, until

1. the token arrives in a process of a level greater than ℓ : then the token is removed;

Corach–Katten–Moran Algorithm

Principle 4.

A **chasing** token (q, ℓ) is forwarded in each node via the channel through which the most recently passed token was sent, until

1. the token arrives in a process of a level greater than ℓ : then the token is removed;
2. the token arrives at a process with a **waiting** token of the same level ℓ : both tokens are removed and a new traversal is started in this node;

Corach–Katten–Moran Algorithm

Principle 4.

A **chasing** token (q, ℓ) is forwarded in each node via the channel through which the most recently passed token was sent, until

1. the token arrives in a process of a level greater than ℓ : then the token is removed;
2. the token arrives at a process with a **waiting** token of the same level ℓ : both tokens are removed and a new traversal is started in this node;
3. the token arrives at a process of the same level ℓ where the most recently passed token was *chasing*: then the token becomes **waiting**.

Corach–Katten–Moran Algorithm

Principle 5.

A **waiting** token (q, ℓ) resides in a process until

Corach–Katten–Moran Algorithm

Principle 5.

A **waiting** token (q, ℓ) resides in a process until

1. a token of a higher level arrives at this node: then the waiting token is removed;

Corach–Katten–Moran Algorithm

Principle 5.

A **waiting** token (q, ℓ) resides in a process until

1. a token of a higher level arrives at this node: then the waiting token is removed;
2. a token of the same level ℓ arrives at this node: then both tokens are removed and a traversal of level $\ell + 1$ is started from this node.

Corach–Katten–Moran Algorithm

HOMETASK 2.

- Build a pseudo-code of Corach–Katten–Moran Algorithm
- Prove that Corach–Katten–Moran Algorithm is a leader election algorithm .
- Estimate the message exchange complexity of Corach–Katten–Moran Algorithm assuming that the procedure *trav* implements Tarry's Traversal Algorithm .

END OF LECTURE 9.