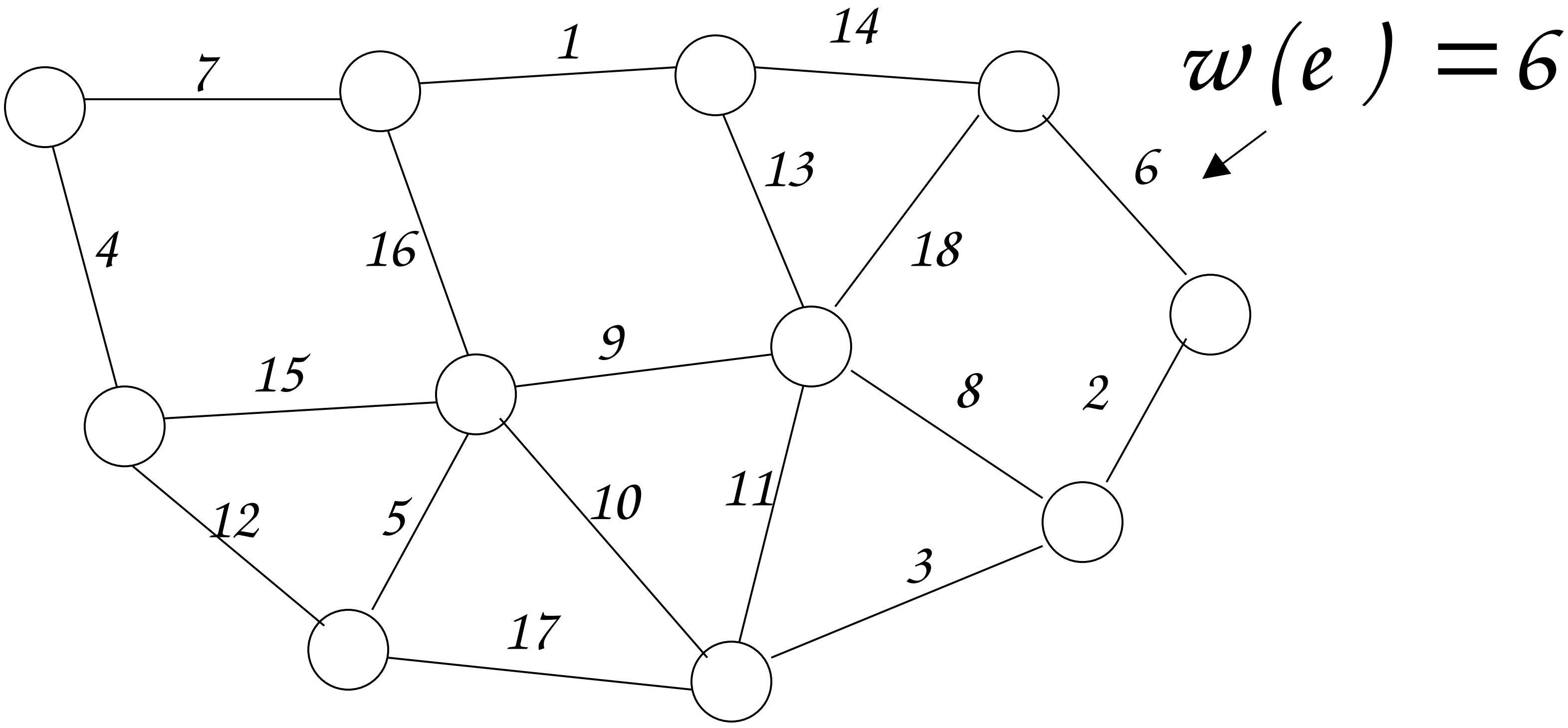
The Minimum Spanning Tree Problem

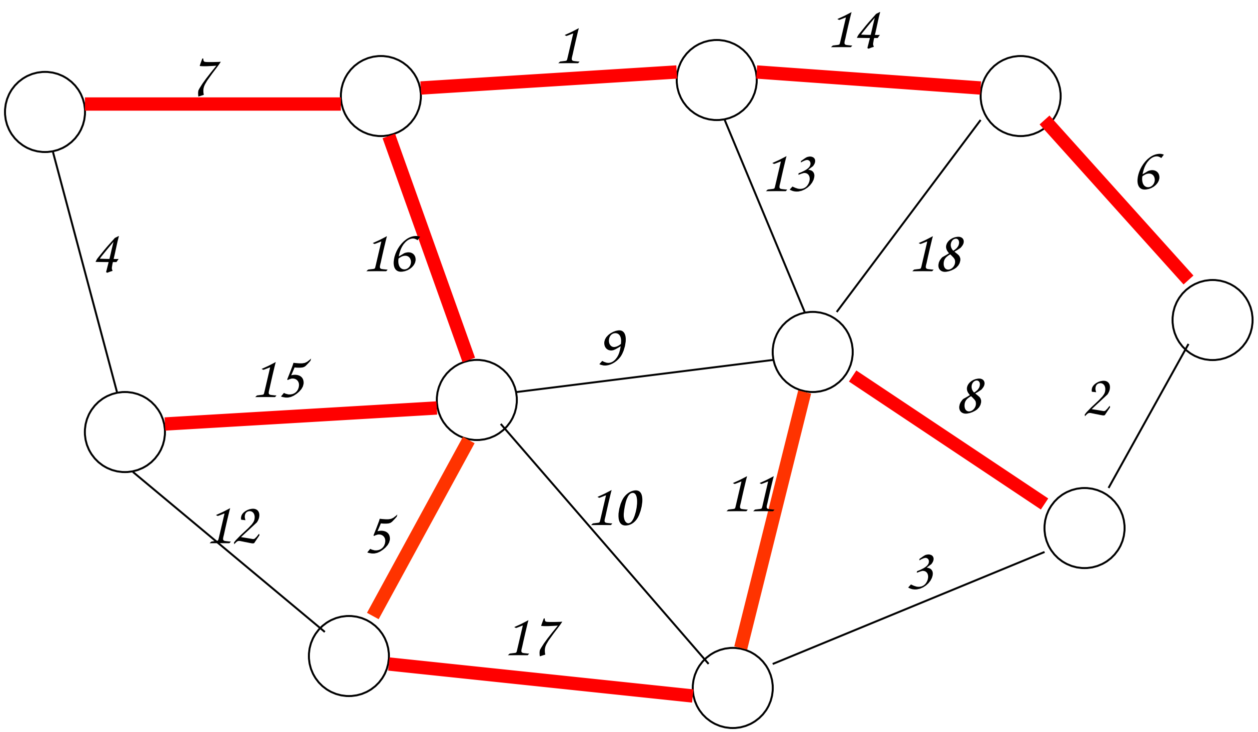
Distributing Prim's and Kruskal's Algorithm

Weighted Graph $G=(V,E,w)$, $|V|=n$, $|E|=m$



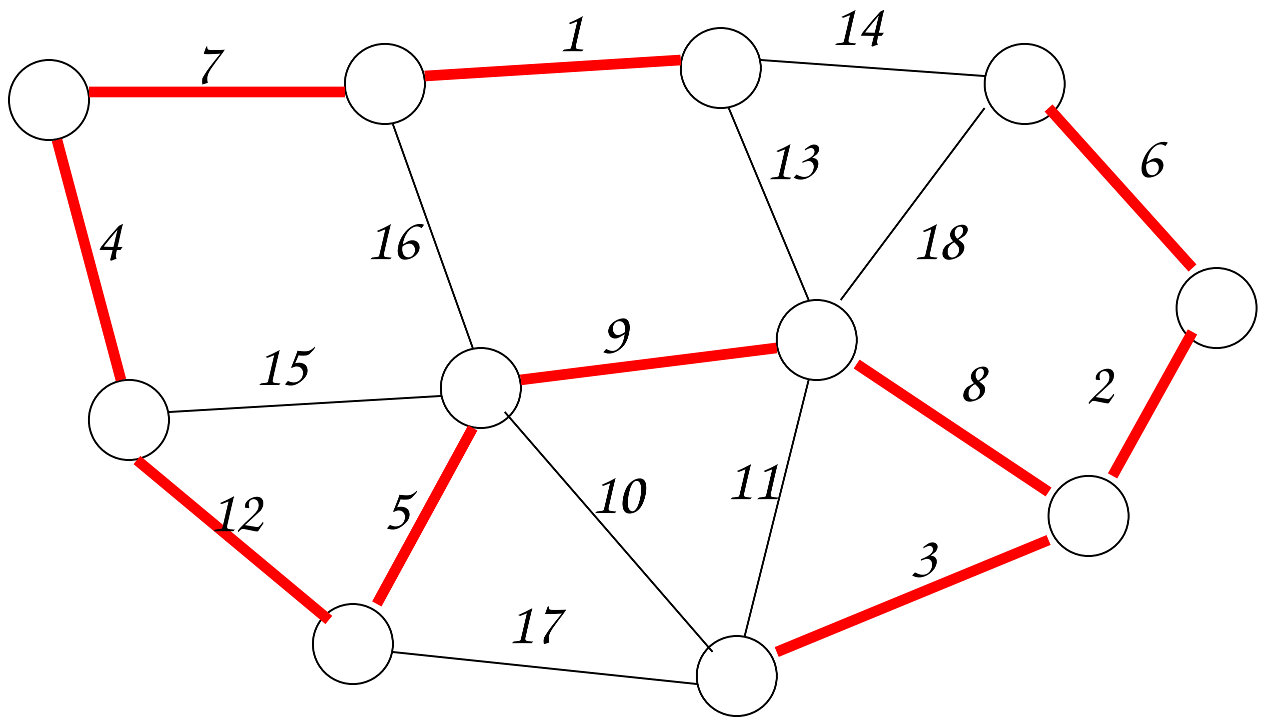
For the sake of simplicity, we assume that weights are positive integers

Spanning tree



Any tree T=(V,E') (connected acyclic graph) spanning all the nodes of G

Minimum-weight spanning tree (MST)

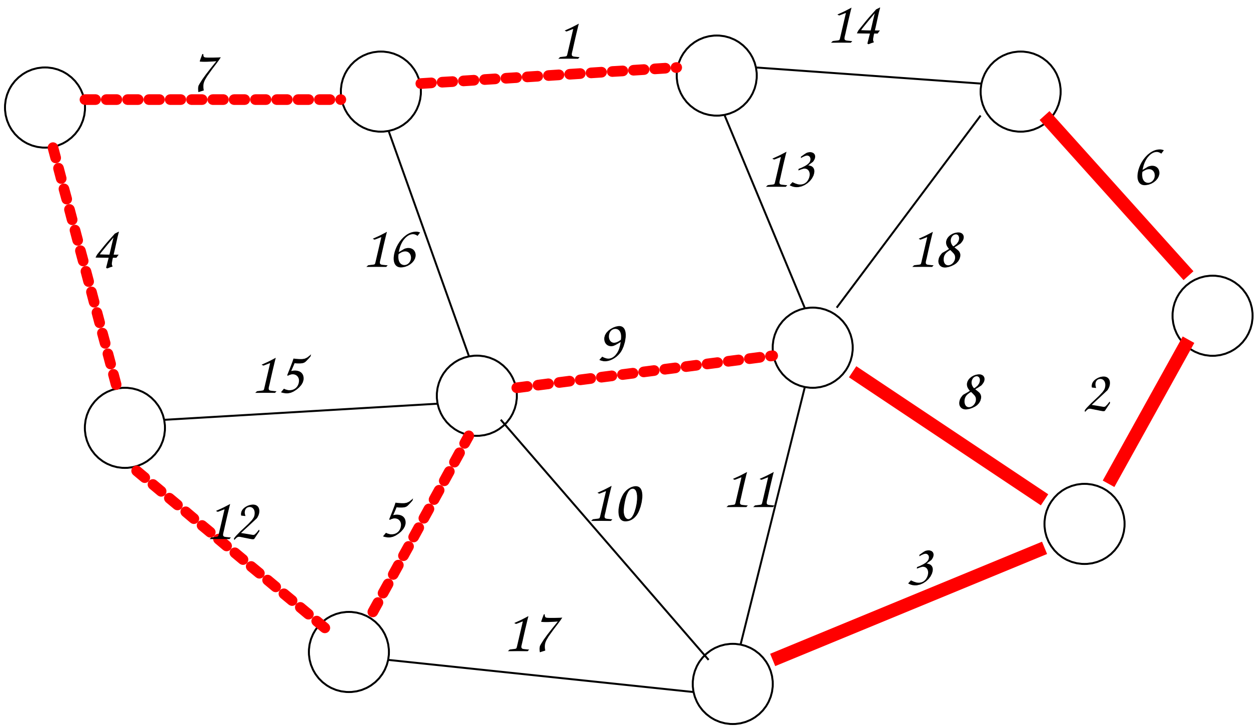


A spanning tree s.t. the sum of its weights is minimized:

$$MST\ \mathcal{T}^{*} := \arg\min \{ w(\mathcal{T}) = \Sigma_{e \in \mathcal{E}(\mathcal{T})} w(e) \mid \mathcal{T} \text{ is a spanning tree of } \mathcal{G} \}$$

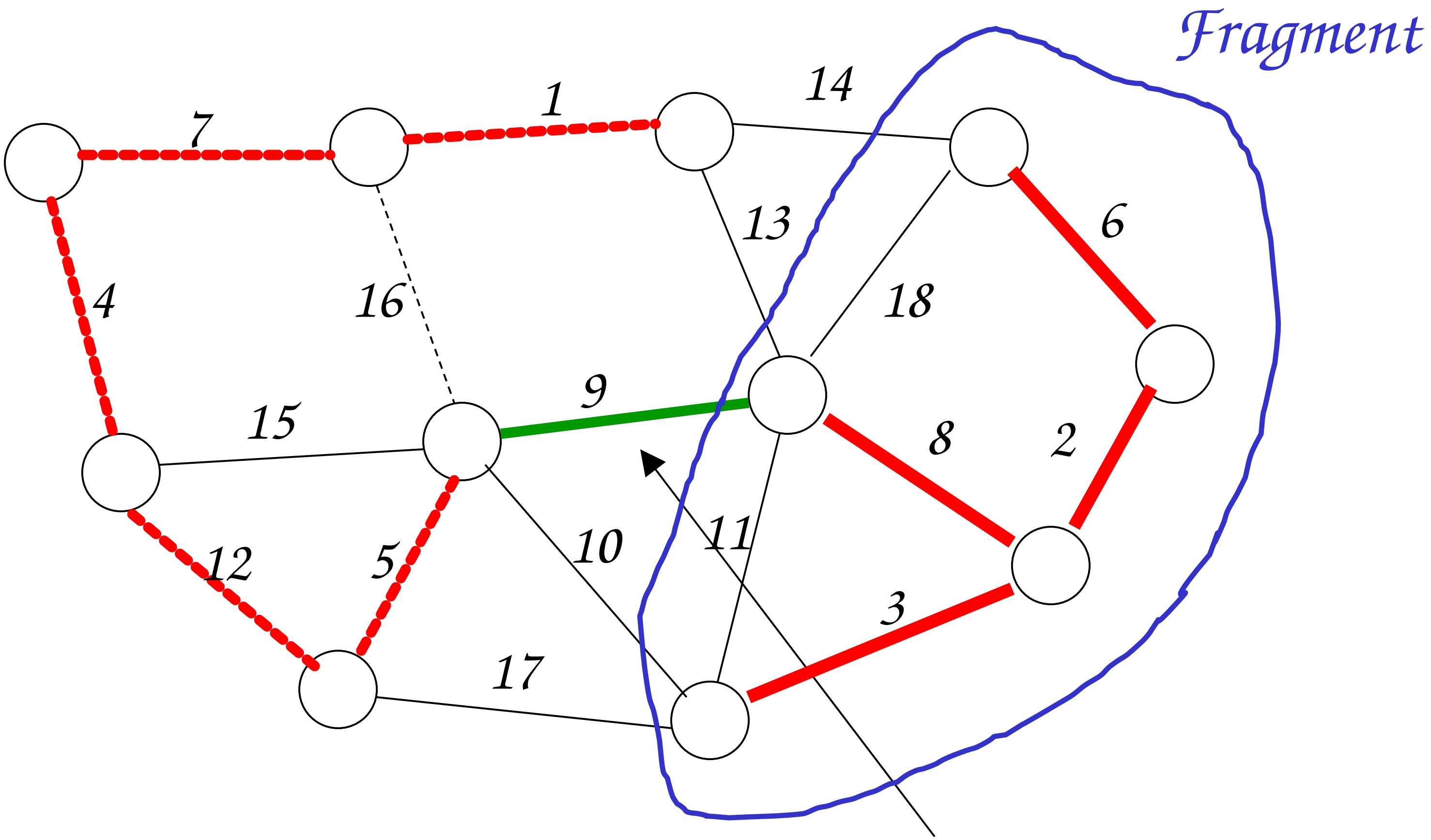
In general, the MST is not unique.

MST fragment:



Any (connected) sub-tree of a MST

Minimum-weight outgoing edge (MOE) of a fragment



An edge incident to a single node of the fragment and having smallest weight (notice it does not create any cycles in the fragment)

Two important properties for building a MST

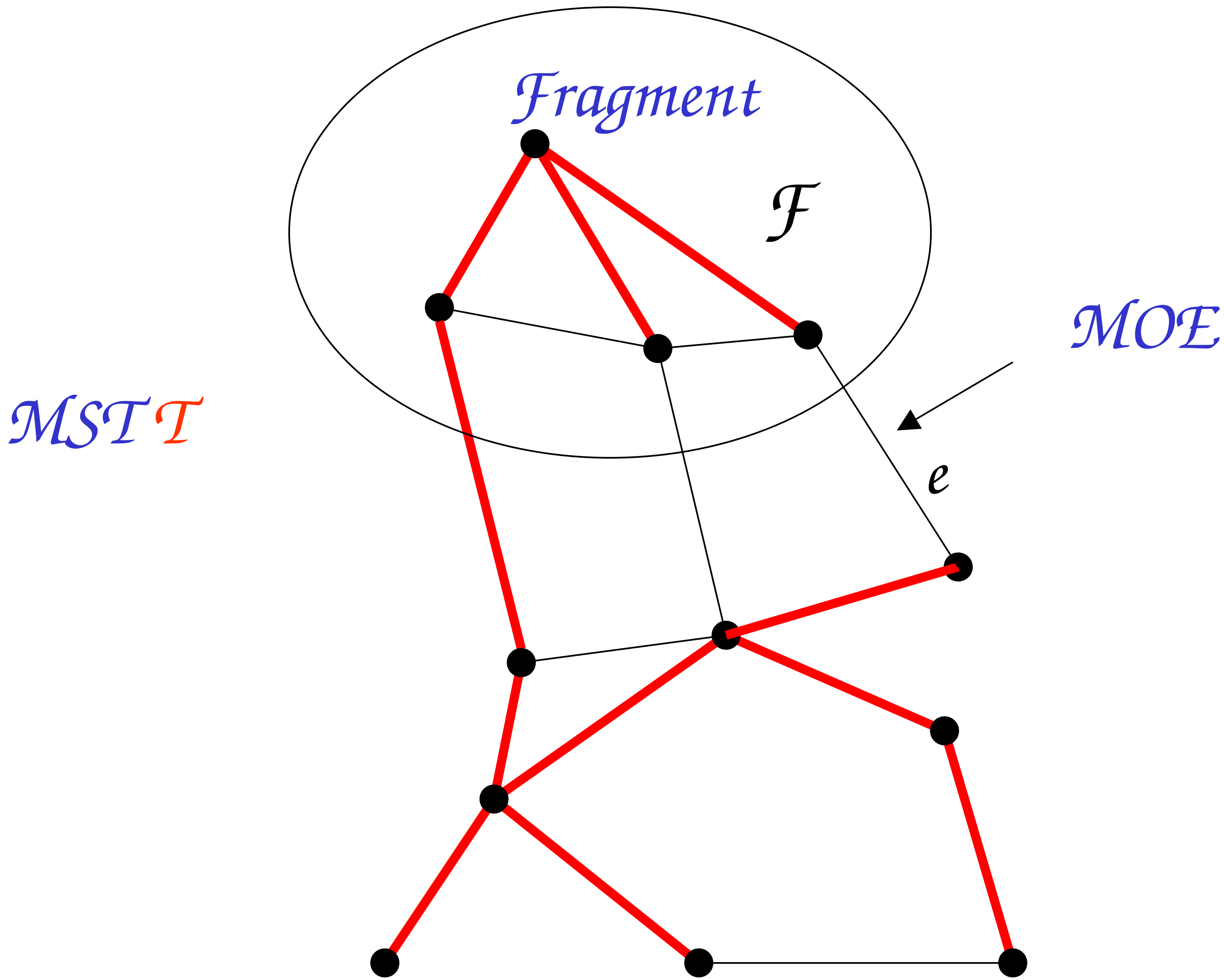
Property 1: *The union of a MST fragment and any of its MOE is a fragment of some MST (so called* ***blue rule****).*

Property 2: *If the edge weights are distinct then the MST is unique*

<u>Property 1:</u> *The union of a MST fragment $F \subseteq T$ and any of its MOE is a fragment of some MST.*

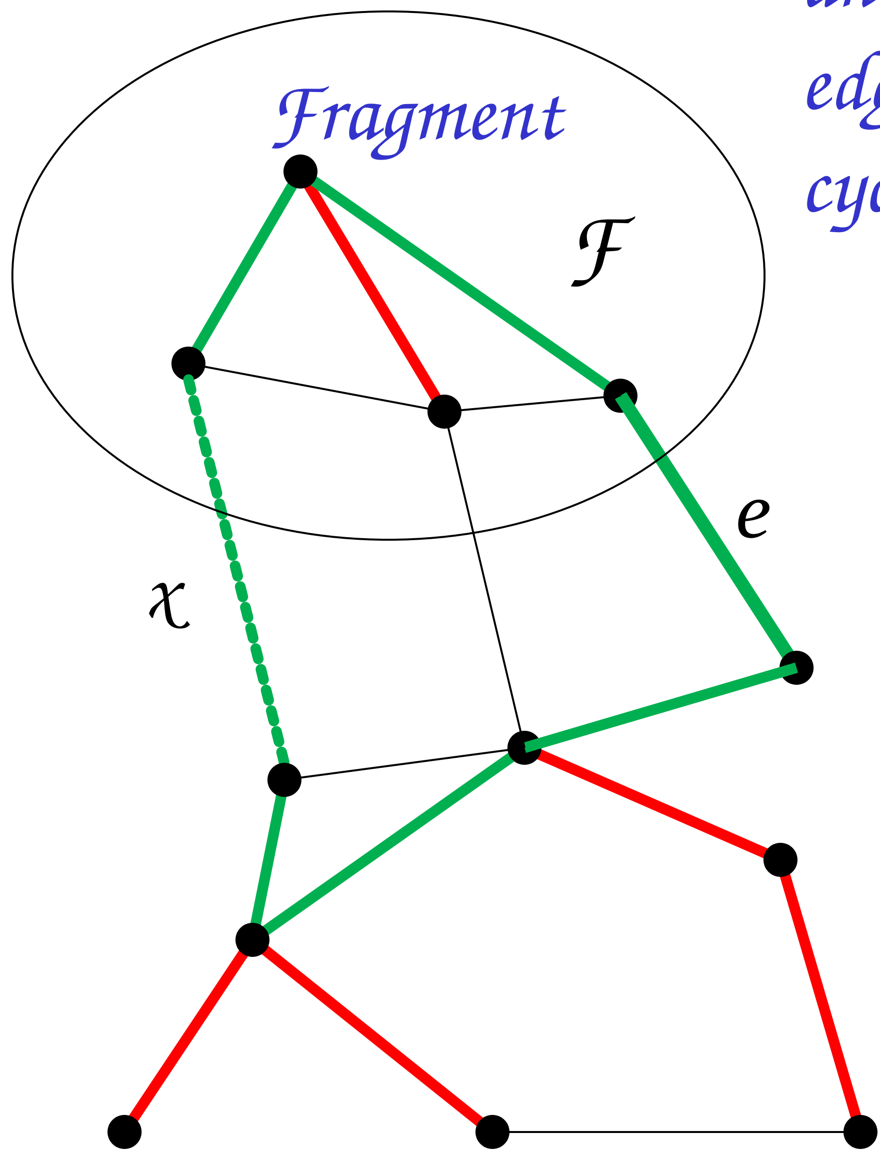
Proof: *Remind that in general the MST is not unique. Let e be a MOE of F, and for the sake of contradiction, assume that F∪{e} is not a fragment of any MST of G, and then in particular, this means that e does not belong to T.*

$e \notin \mathcal{T}$



Then add e *to* $\mathcal{T}$ *(thus forming a cycle) and remove* χ *(any edge of* $\mathcal{T}$ *in such a cycle exiting from* $\mathcal{F}$*)*

MST $\mathcal{T}$



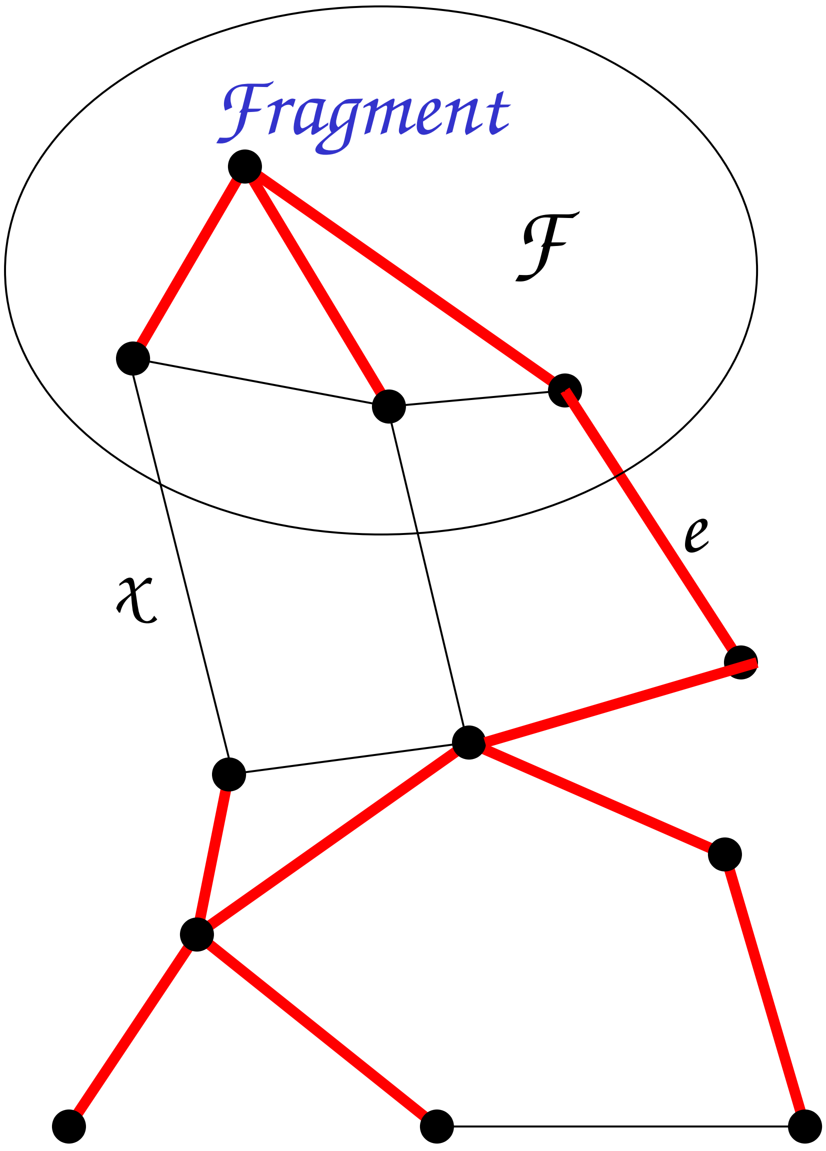
$$w(e) \leq w(\chi)$$

Obtain T'

and since

$w(e) \leq w(x)$

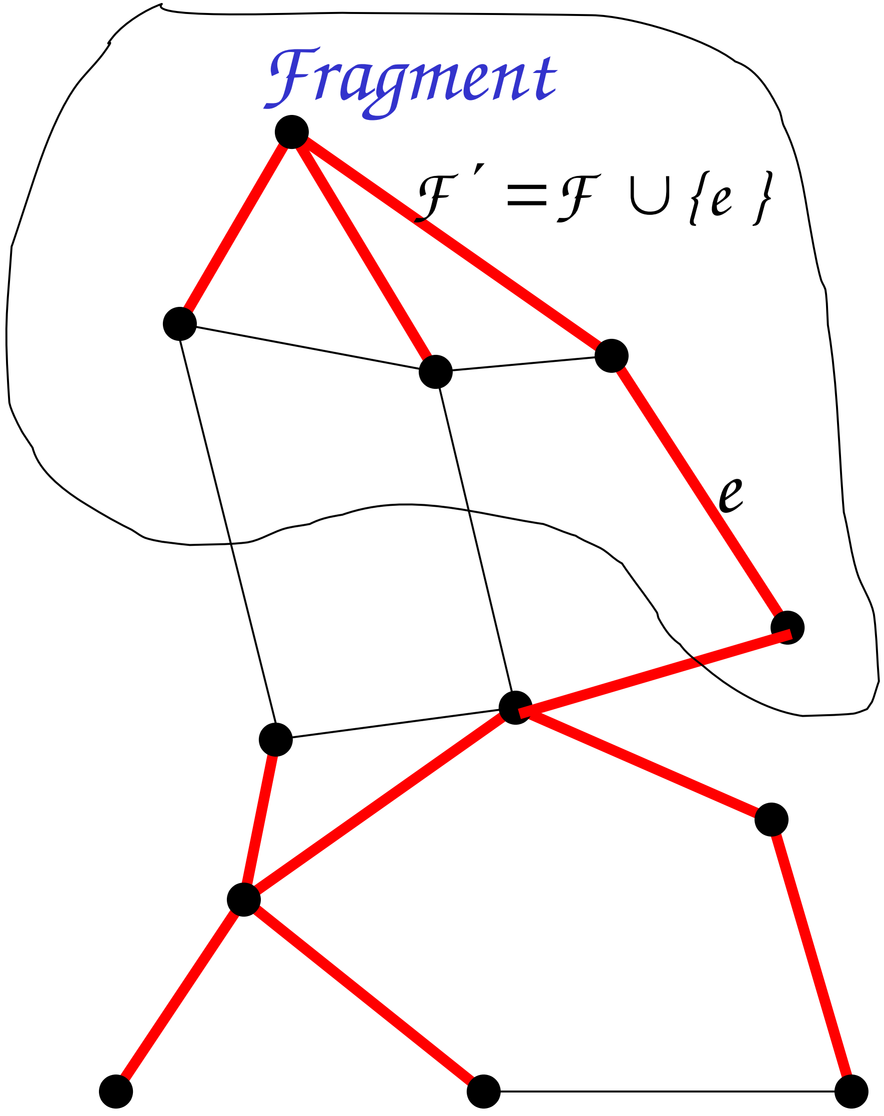
$\Rightarrow w(T') \leq w(T)$



But $w(T') \geq w(T)$,

since T is an MST

$\Rightarrow w(T')=w(T)$, i.e., T' is an MST



MST *T'*

thus $F \cup \{e\}$ *is a fragment of MST* *T '*

$\Rightarrow$ *contradiction!*

END OF PROOF

Property 2: *If the edge weights are distinct then the MST is unique*

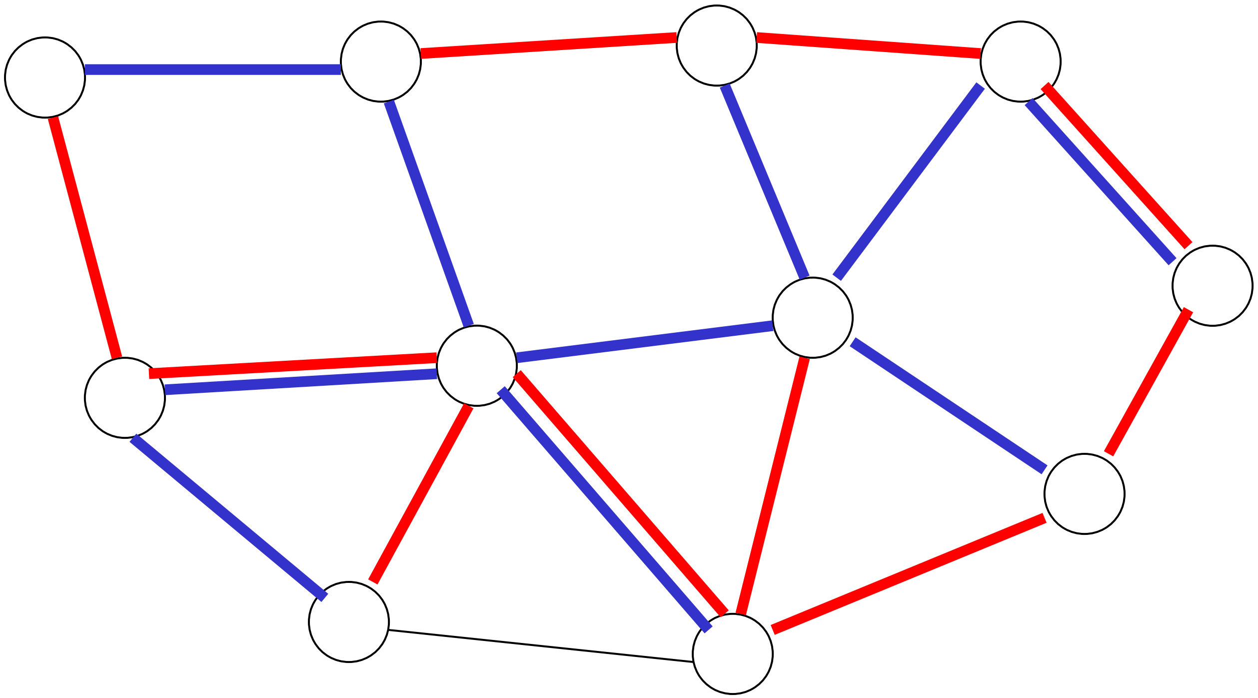
Proof: *Basic Idea:*

Suppose there are two MSTs

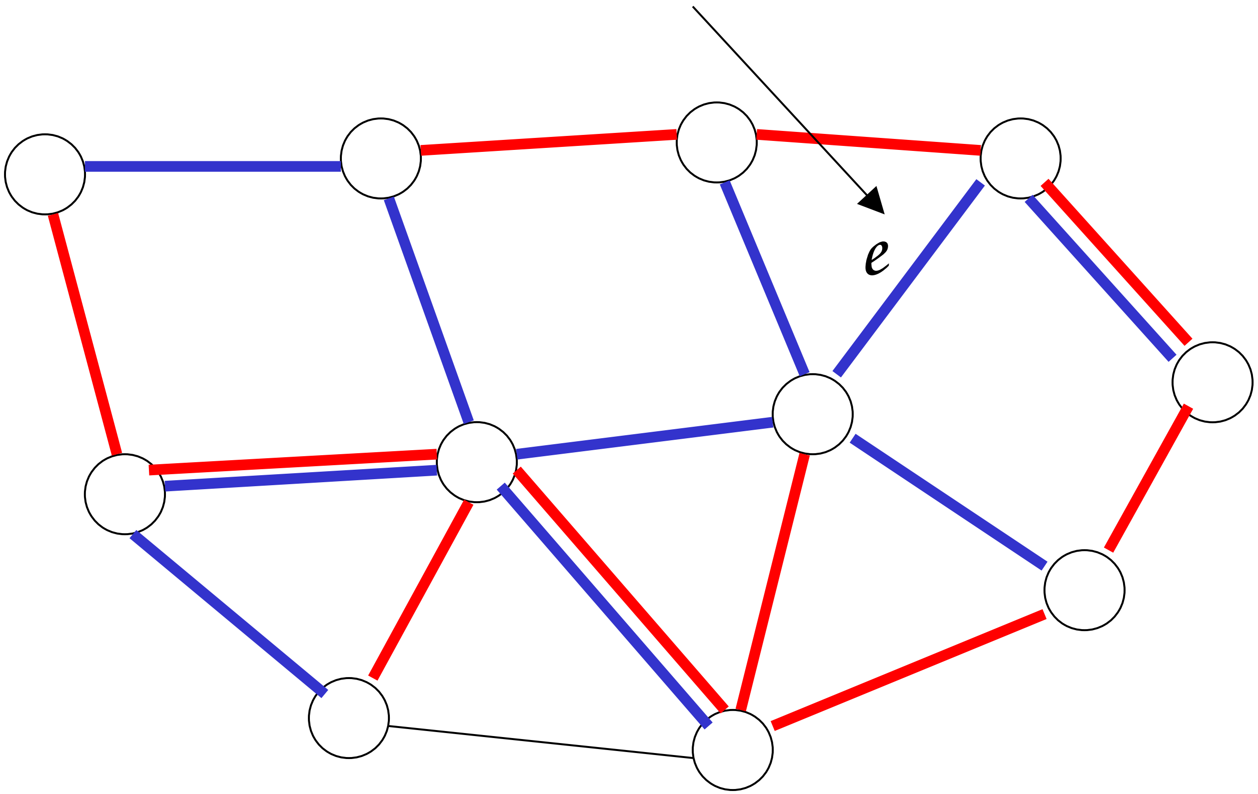
Then we prove that there is another spanning tree of smaller weight

$\Rightarrow$ *contradiction!*

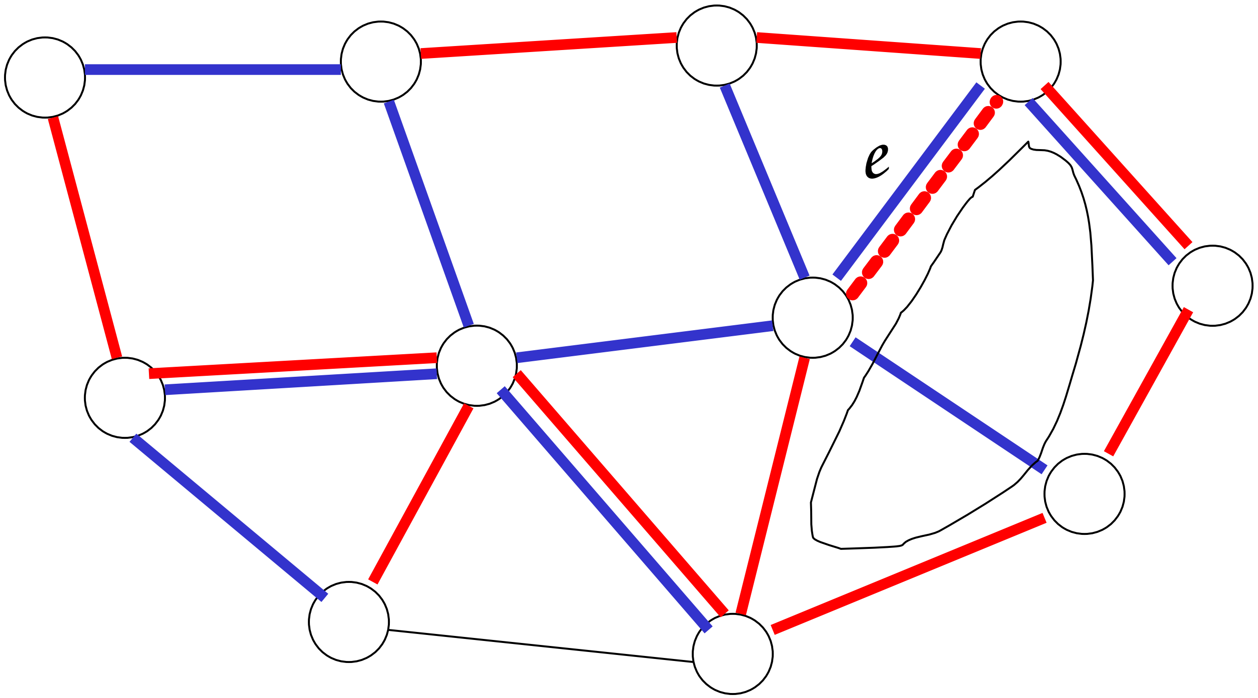
Suppose there are two MSTs



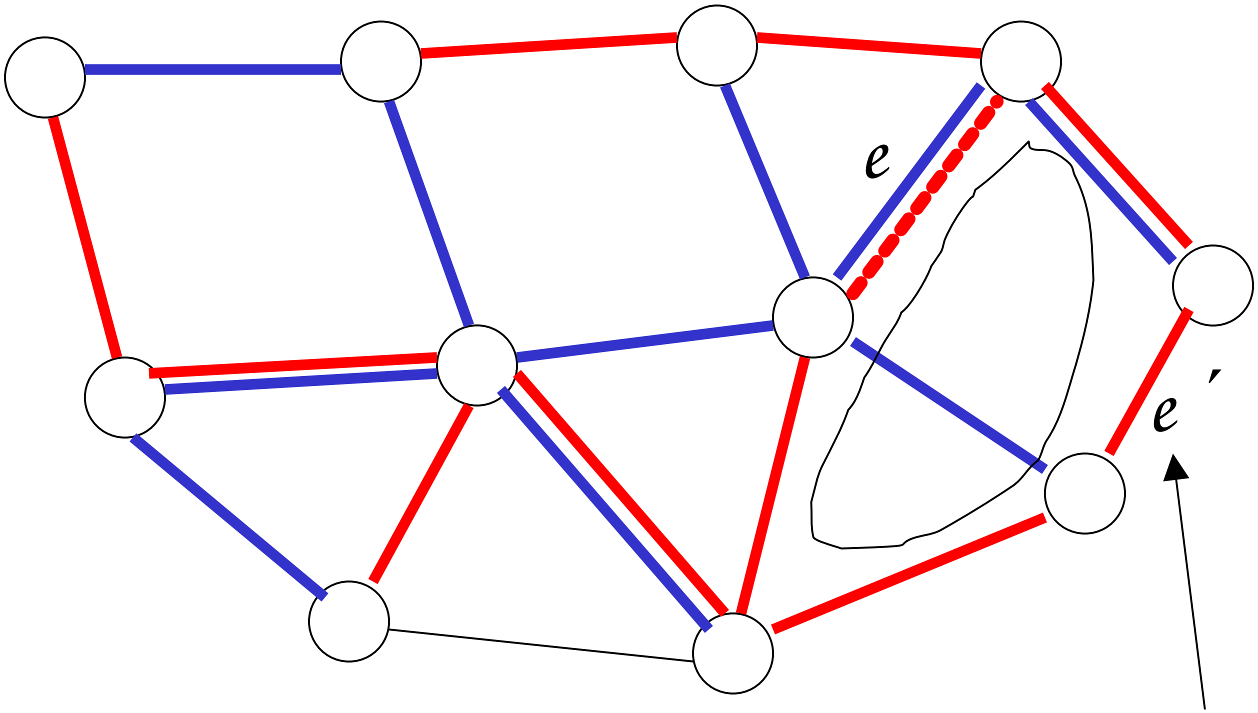
Take the smallest-weight edge not in the intersection, and assume w.l.o.g. it is blue



Cycle in RED MST



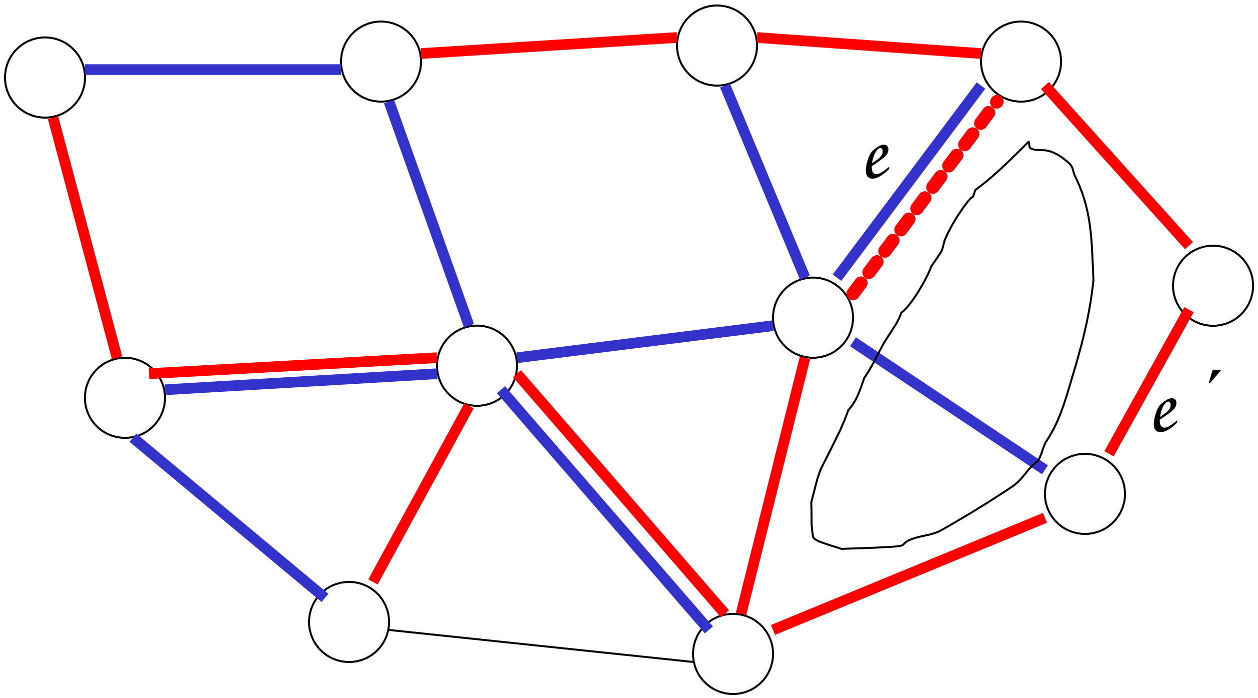
Cycle in RED MST



e': any red edge in the cycle not in BLUE MST

(∃ since blue tree is acyclic)

Cycle in RED MST

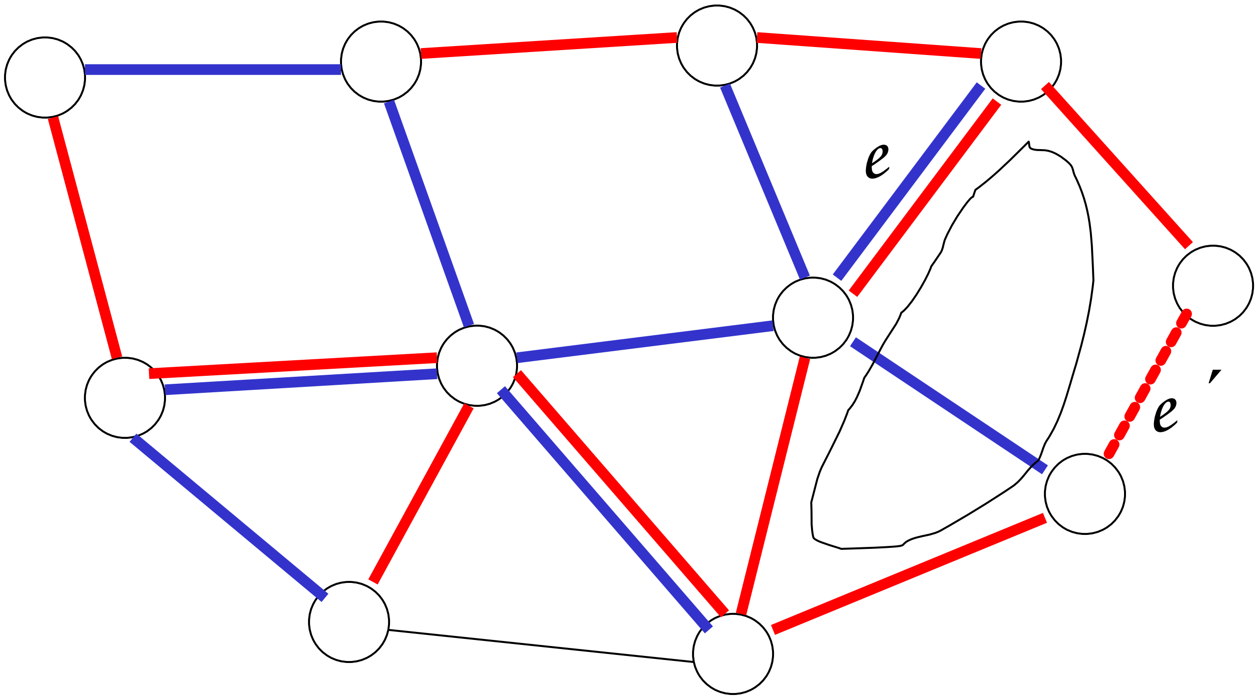


Since e' is not in the intersection, $w(e) < w(e')$

(weights are distinct and the weight of e is the smallest among edges not in the intersection)

$w(e) < w(e')$

Cycle in RED MST



Delete e' and add e in RED MST

$\Rightarrow$ we obtain a new tree with smaller weight

$\Rightarrow$ contradiction!

END OF PROOF

Overview of MST distributed algos

There exist algorithms only when nodes have unique ids. We will evaluate them according to their message (and time) complexity. Upcoming results follow:

- Distributed Prim:
  - Asynchronous (uniform): $O(n^2)$ messages
  - Synchronous (uniform): $O(n^2)$ messages, and $O(n^2)$ rounds
- Distributed Kruskal (so-called Gallagher-Humblet-Spira (GHS) algorithm) (distinct weights):
  - Synchronous (non-uniform): $O(m+n \log n)$ messages, and $O(n \log n)$ rounds
  - Asynchronous (uniform): $O(m+n \log n)$ messages

Prim's Algorithm (sequential version)

Start with a node as an initial fragment, say $\mathcal{F}$, and repeatedly apply the blue rule

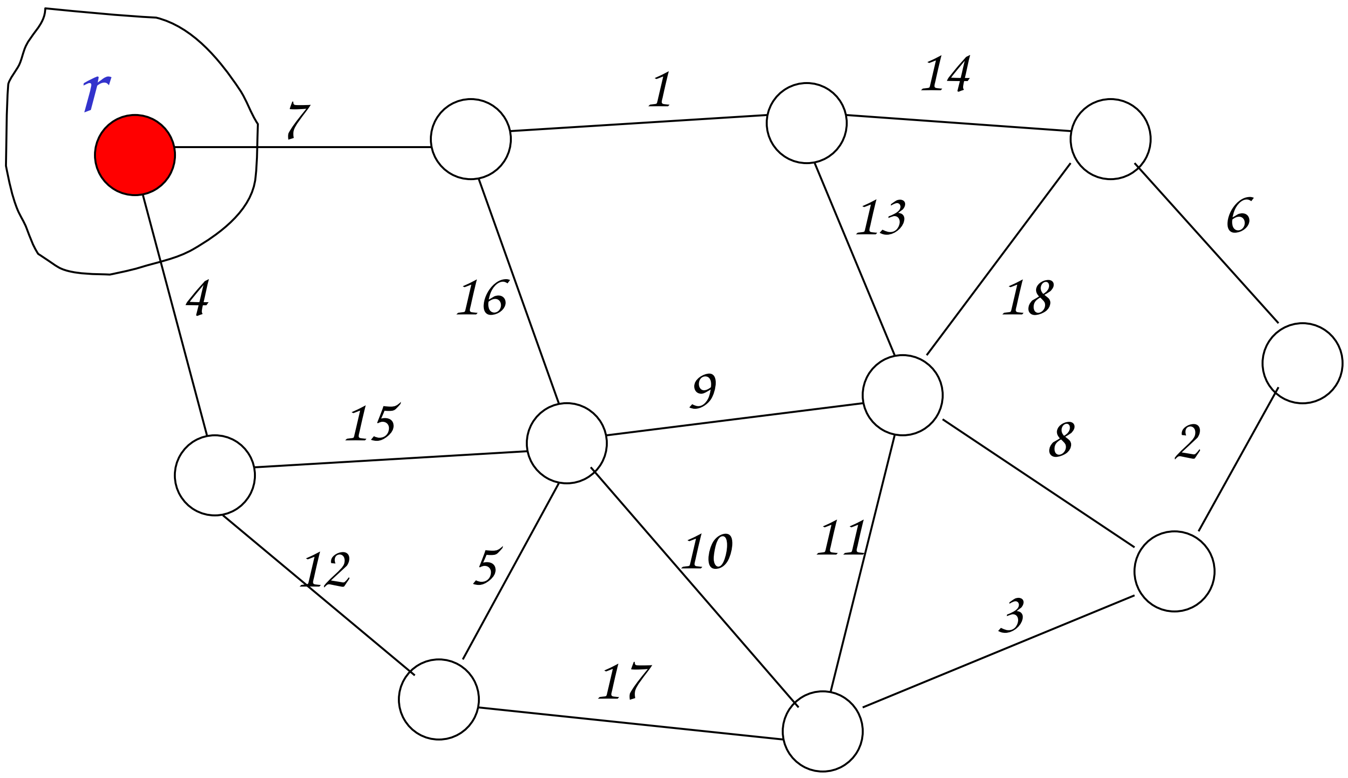
$\mathcal{F}=\{r\in V(G)\}$

Repeat

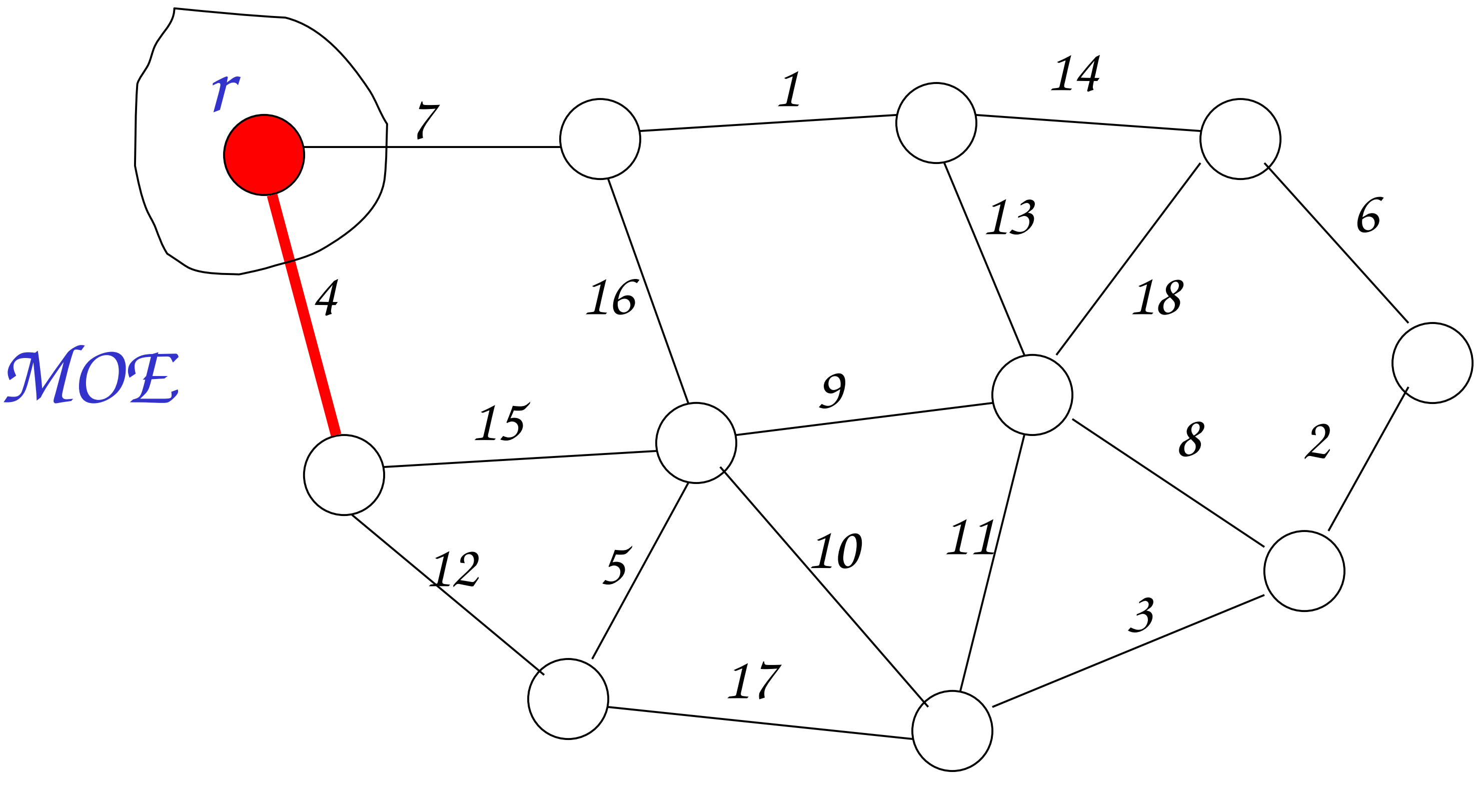
Augment fragment $\mathcal{F}$ with a MOE

Until no other edge can be added to $\mathcal{F}$

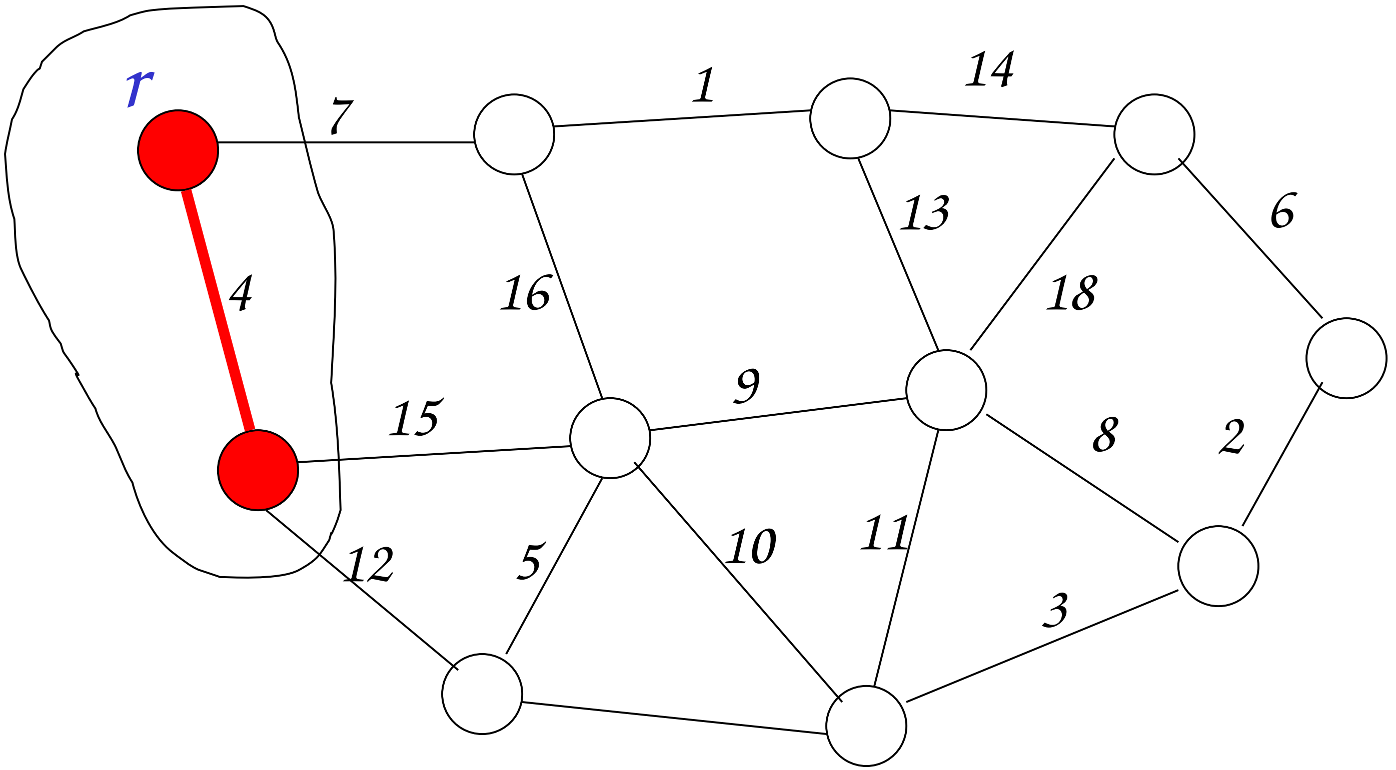
Fragment $\mathcal{F}$



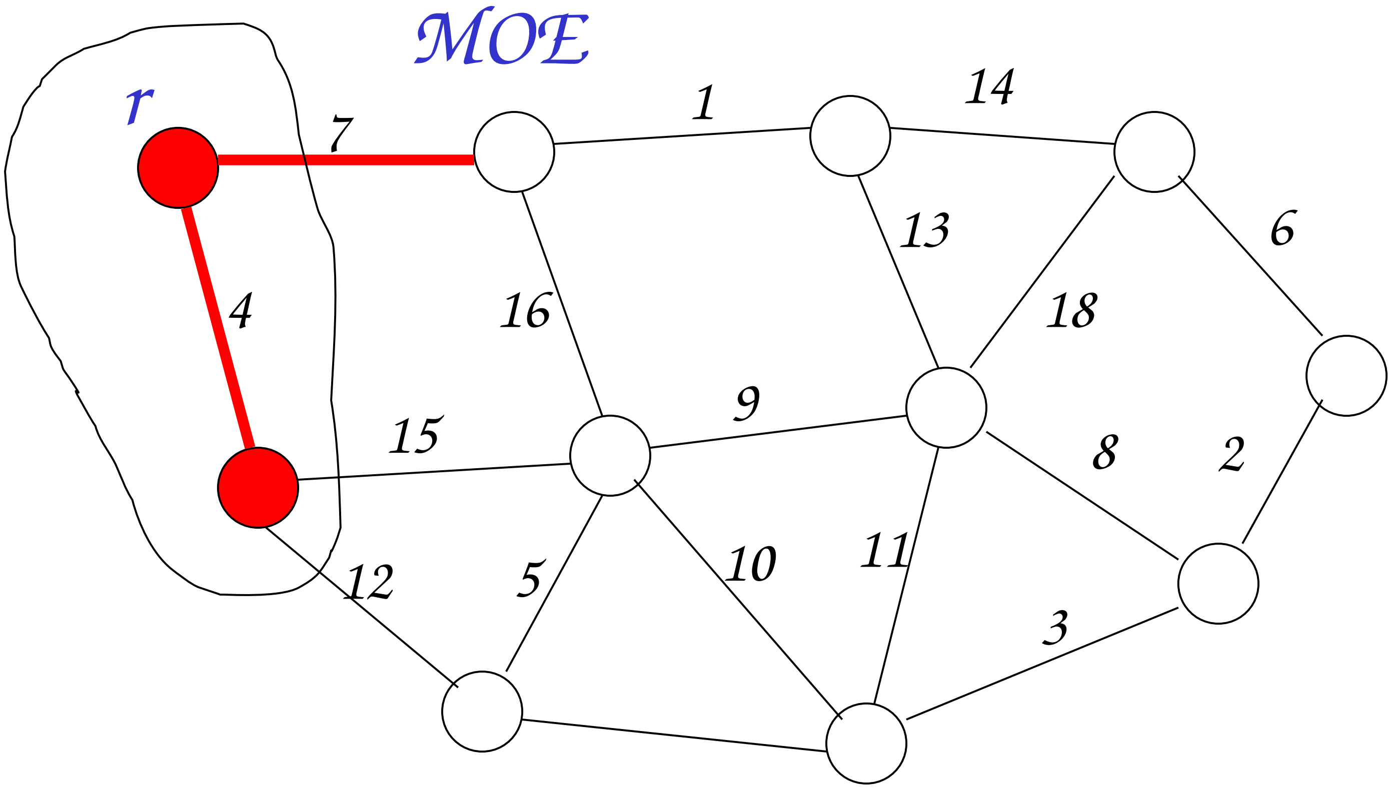
Fragment $\mathcal{F}$



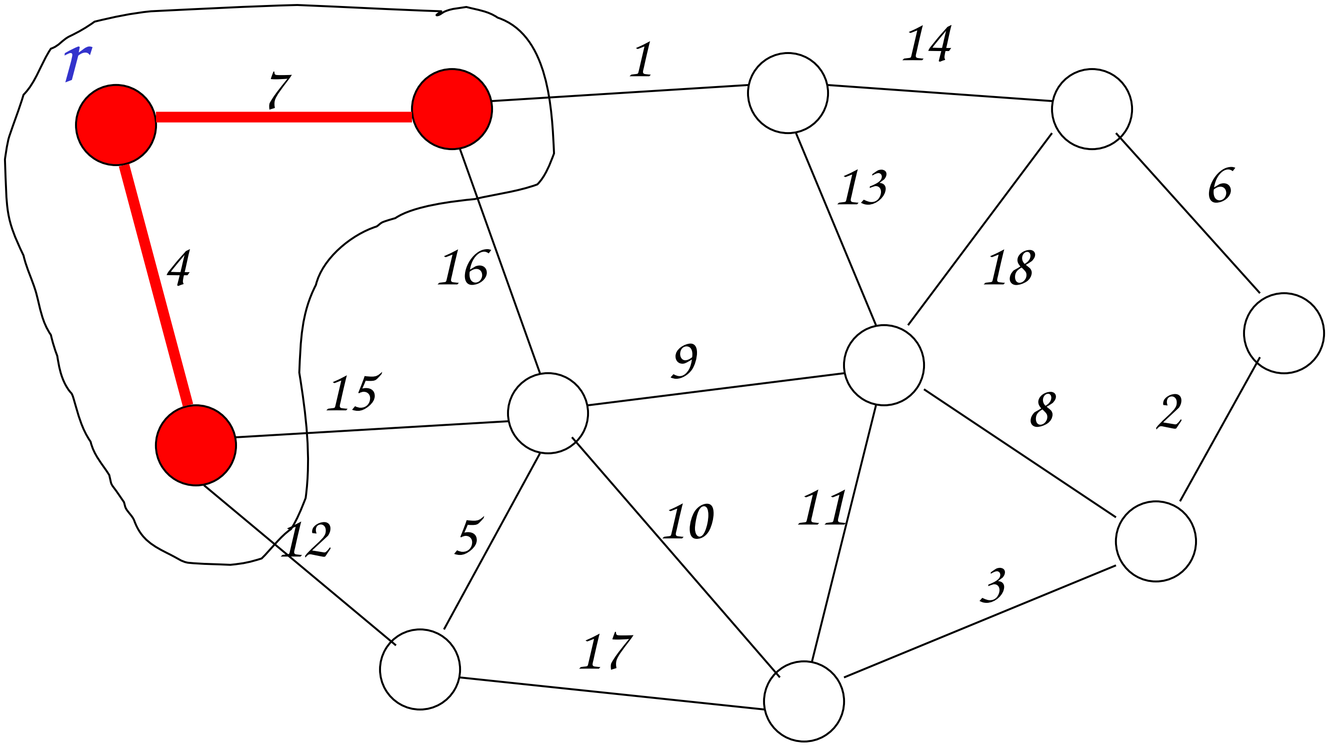
Augmented fragment $\mathcal{F}$



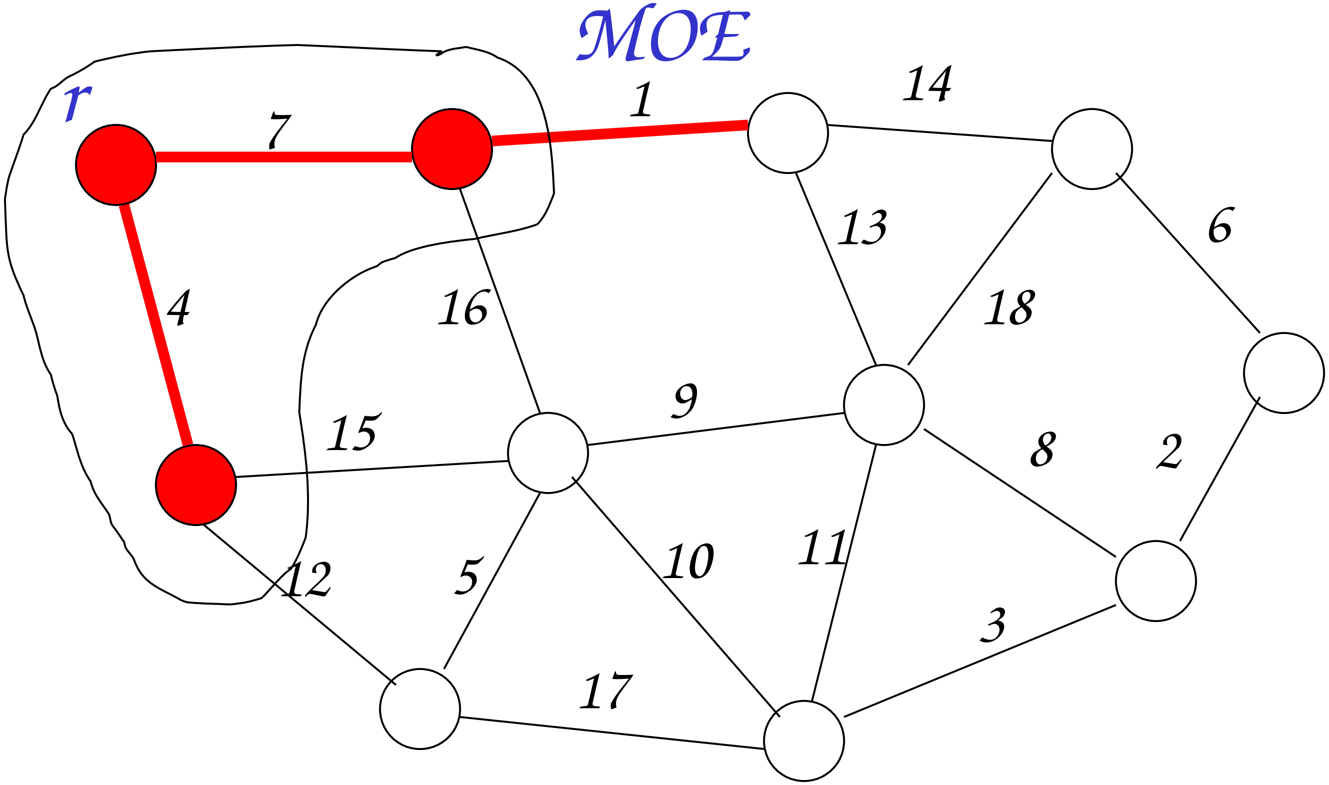
Fragment $\mathcal{F}$



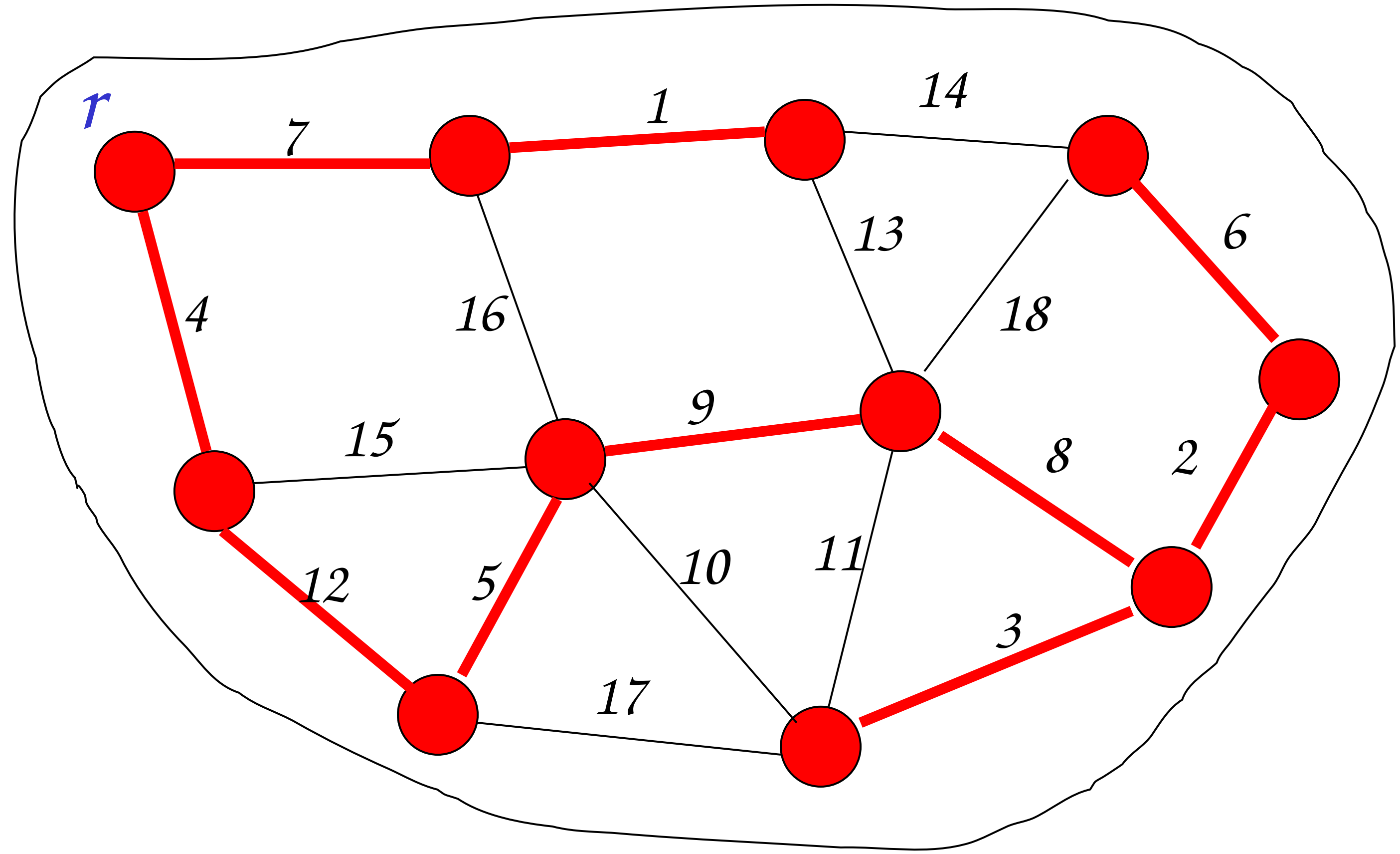
Augmented fragment $\mathcal{F}$



Fragment $\mathcal{F}$



Final MST



Theorem: Prim's algorithm gives an MST

Proof: Use Property 1 repeatedly

END OF PROOF

Prim's algorithm (distributed version)

Works with both asynchronous and synchronous non-anonymous, uniform models (and with non-distinct weights)

Algorithm (asynchronous high-level version):

Let vertex r be the **root** as well as the first fragment (notice that r should be provided by a leader election algorithm, and this is why we ask for non-anonymity, although the actual algorithm will not make use of ids)

REPEAT (phase)

- r broadcasts a message on the current fragment to search for a MOE of the fragment (i.e., each vertex in the fragment searches for its local (i.e., incident) MOE)
- Starting from the leaves of the fragment, apply the following bottom-up procedure: each leaf reports the weight of its local MOE (if any) to its parent, while an internal node reports to its parent the weight of the MOE of its appended subfragment, i.e., the minimum between the weight of its local MOE and the weight of the MOEs received by its children (in other words, it reports the minimum among the weights of all the local MOEs of the nodes in the subfragment rooted in it (ties are broken arbitrarily);
- the MOE of the fragment is then selected by r and added to the fragment, by sending an add-edge message on the appropriate path
- finally, the root is notified the edge has been added

UNTIL the fragment spans all the nodes

Local description of asynchronous Prim

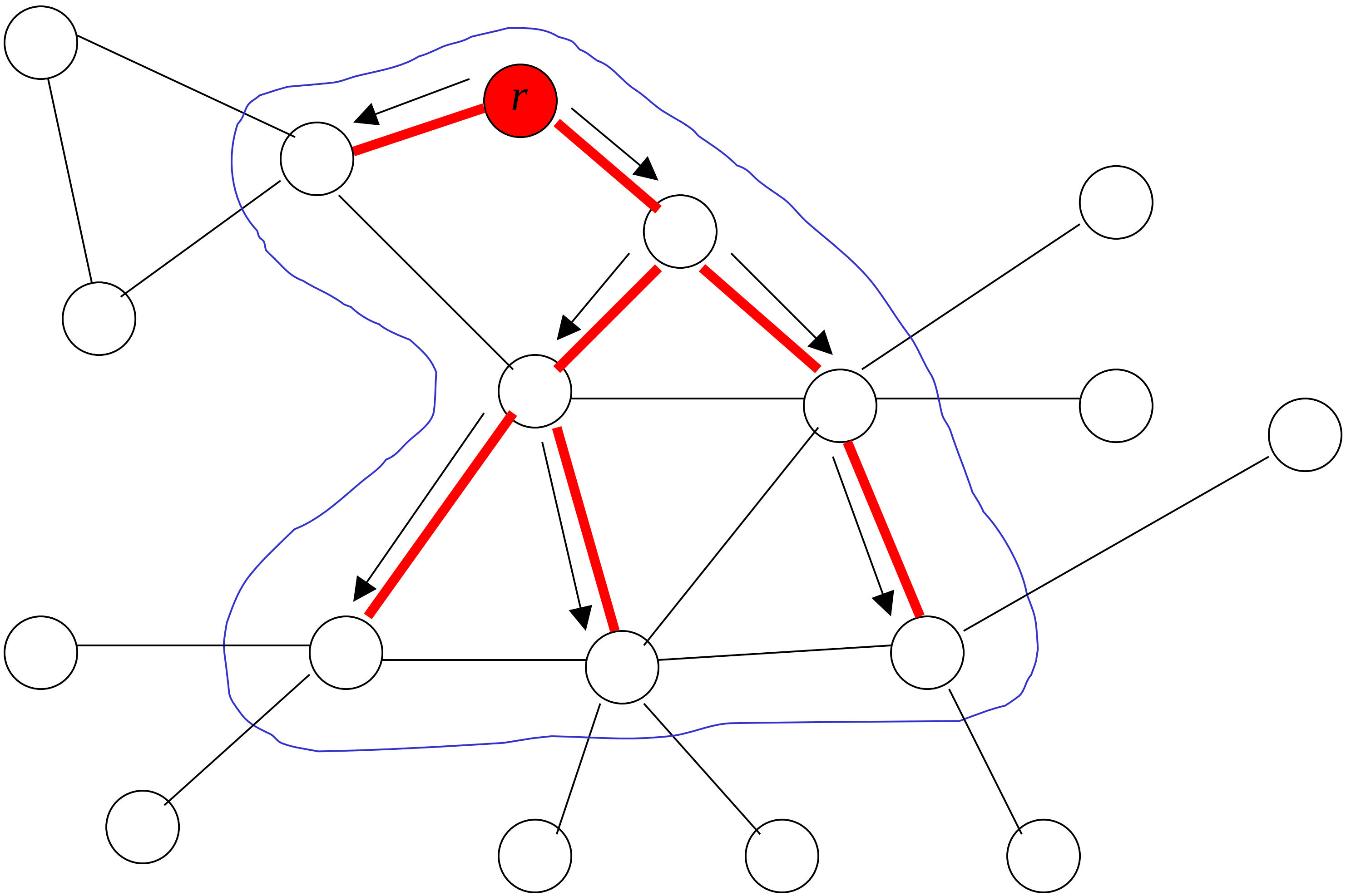
Each processor stores:

1. The status of any of its incident edges, which can be either of {basic, branch, reject}; initially all edges are basic
2. Its own status, which can be either {in, out} of the fragment; initially all nodes are out
3. Parent channel (route towards the root)
4. Children channels (routes towards the children)
5. Local (incident) MOE
6. MOE for each children channel
7. MOE channel (route towards the MOE of its appended subfragment)

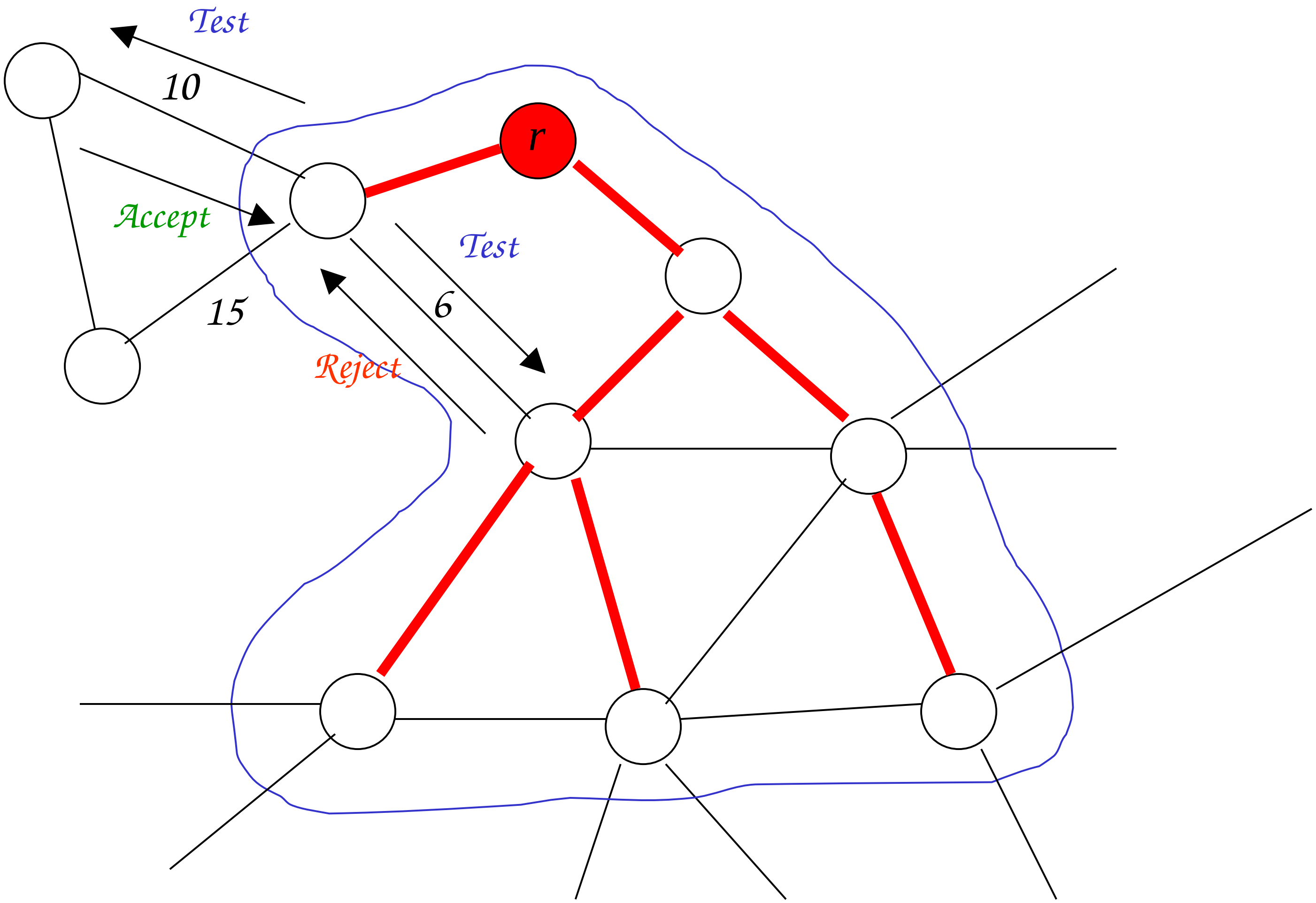
Types of messages in asynchronous Prim

1. ***Search_MOE:*** *coordination message initiated by the root, that will flood top-down towards all the nodes of the fragment*
2. ***Test:*** *originated by a node in the fragment that checks the status of its basic edges in increasing order of weight (if any)*
3. ***Reject****: the response of an in node to a* ***Test*** *message;*
4. ***Accept****: the response of an out node to a* ***Test*** *message;*
5. ***Report(weight):*** *originated by a node that reports to the parent node the weight of the MOE of the appended subfragment (it can be nil)*
6. ***Add_edge:*** *initiated by the root, it will descend the path in the fragment towards the node adjacent to the fragment's MOE, in order to add it*
7. ***Connect:*** *sent by the end-node incident to the found MOE to its adjacent on the MOE, in order to add it to the fragment (this changes the status of the other end-node from out to in, and of the MOE from basic to branch)*
8. ***Connected:*** *originated by the just added node, will travel back up to the root to notify it that connection has taken place*

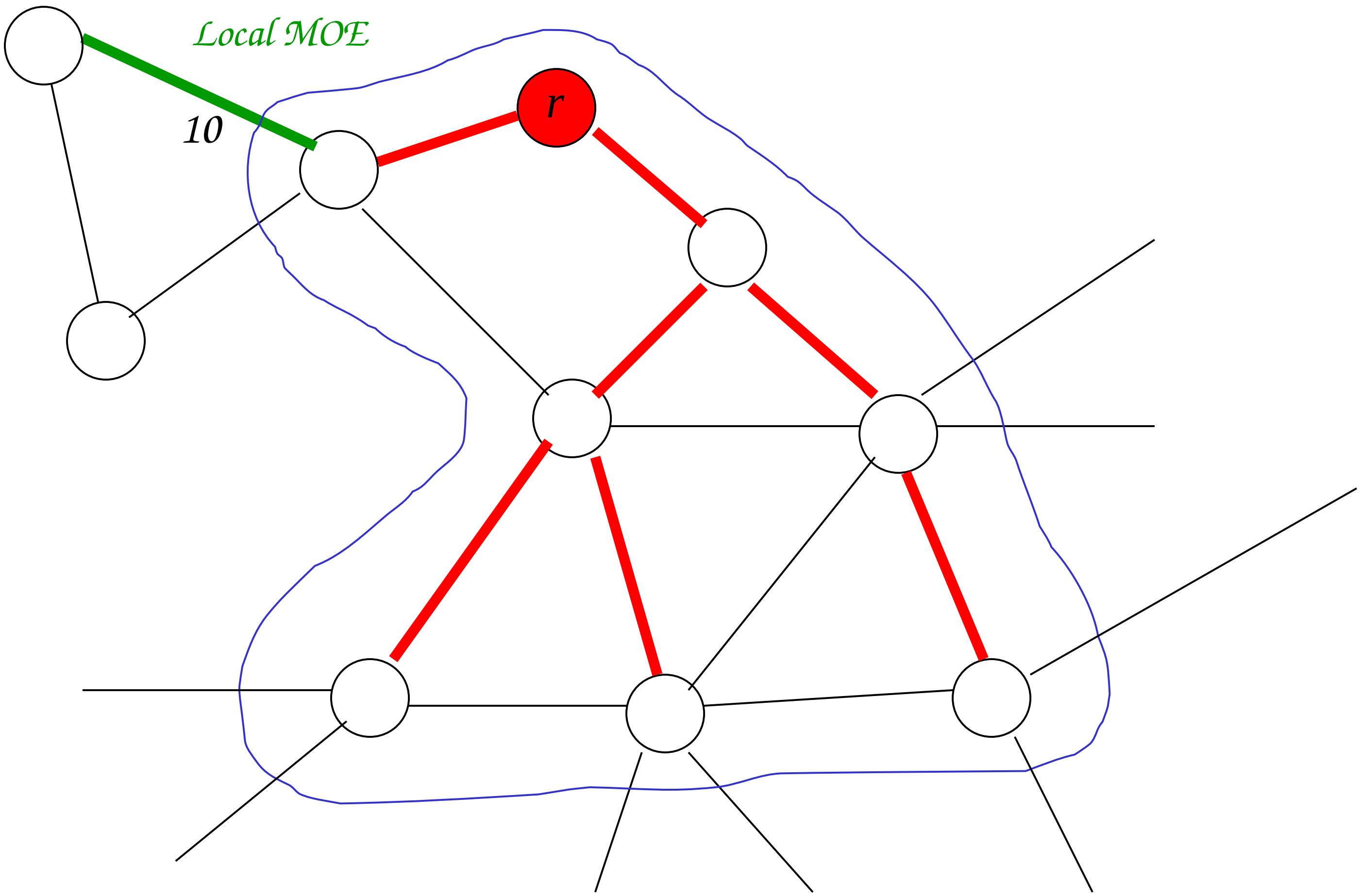
Sample execution: *At the beginning of a phase, the root sends a* ***Search_MOE*** *message, and the message floods along the fragment, so that each node in fragment starts looking for its local MOE*



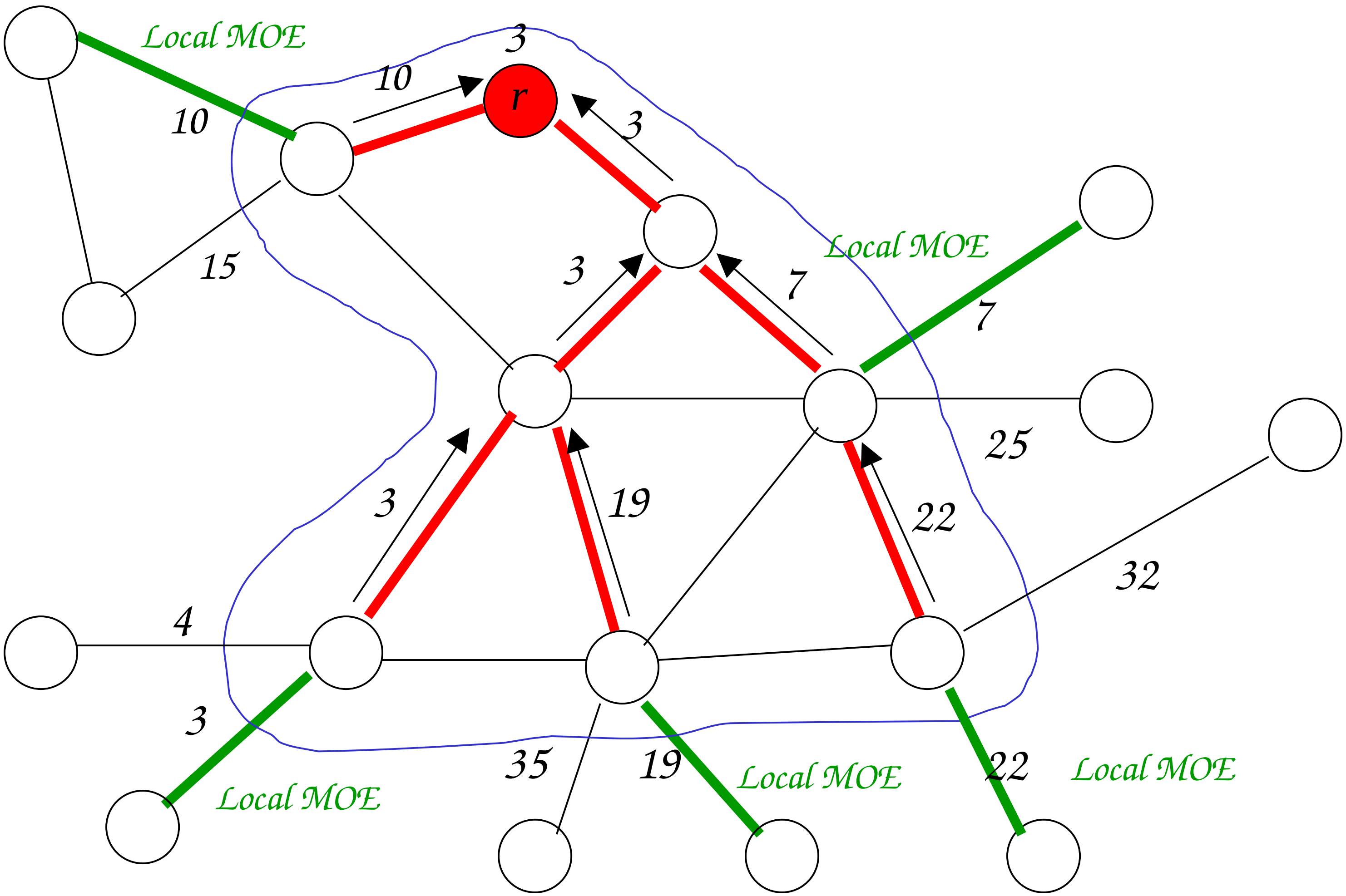
*To discover its local MOE, each node sends a **Test** message over its basic edges in increasing order of weight, until it receives an **Accept.** Rejected tests turn to reject the status of the corresponding edge.*



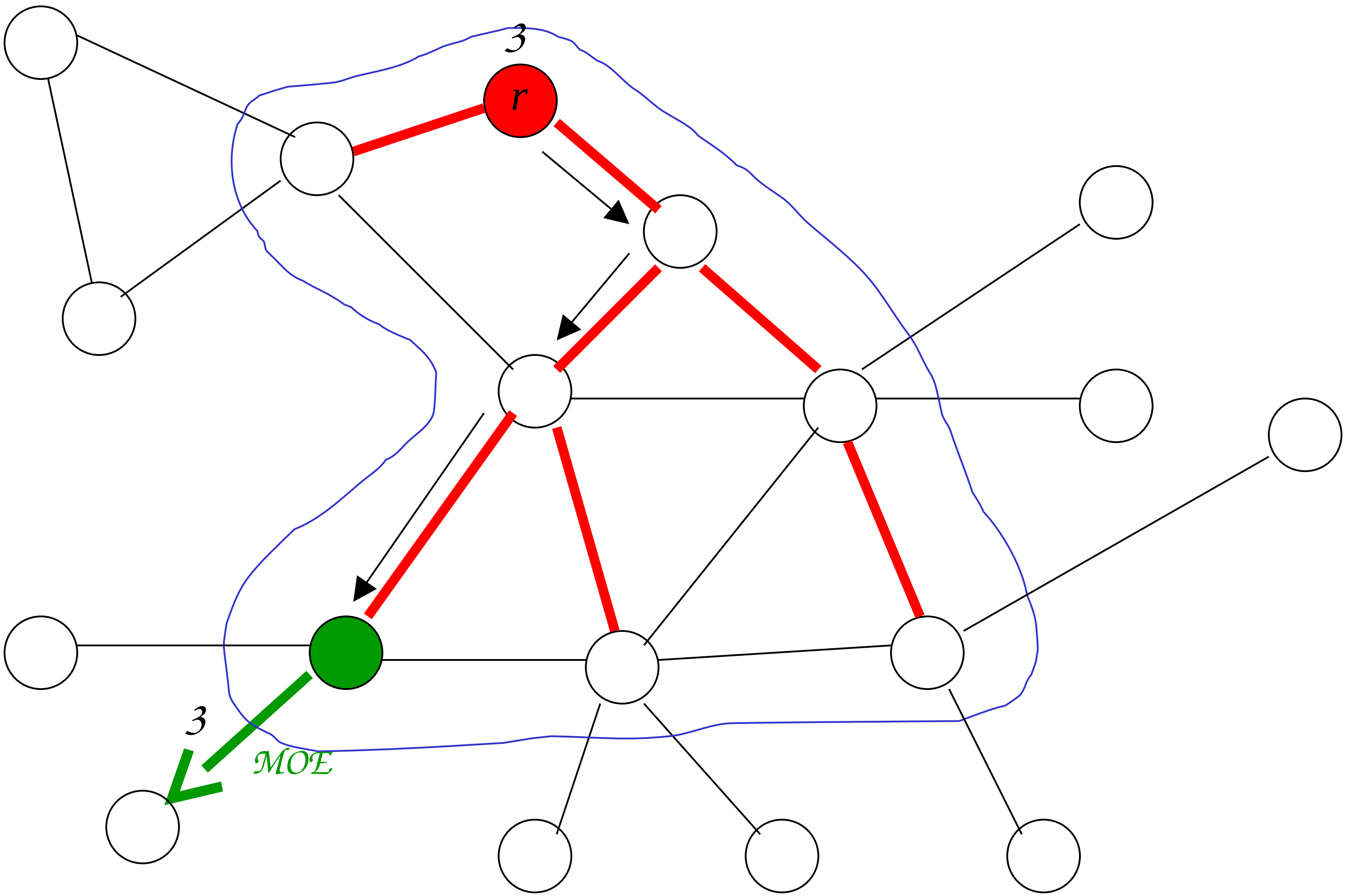
Then it knows its local MOE (notice this can be void)



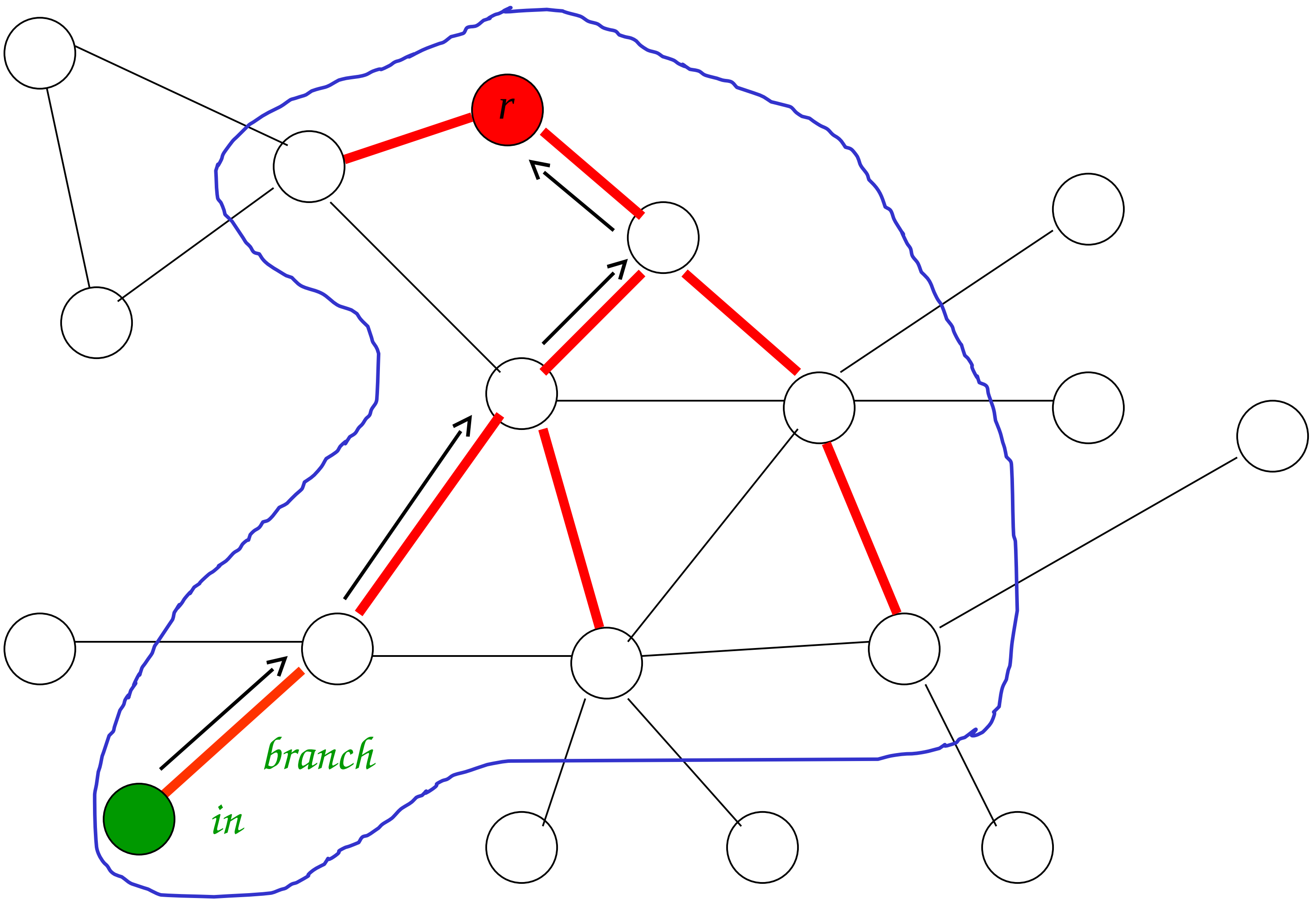
Then, if a node is a leaf, it sends a ***Report*** *with the weight of its MOE (if any) to its parent, while if a node has children, it waits for a* ***Report*** *from each child, and then selects a global minimum between the weight of its local MOE and the weights of the reported MOEs, which will be then* ***reported*** *to its parent; in this way, each internal node stores and reports the weight of the MOE of its appended subfragment*



*The root selects the minimum among received MOEs and sends along the appropriate path an **Add_edge** message, which will become a **Connect** message at the proper node*



Added node changes its status to in, and connecting edge becomes branch. Finally, a ***Connected*** *message is sent back along the appropriate path up to the root, which then starts a new phase by resuming the* ***Search_MOE*** *procedure*



Algorithm Message Complexity

Thr: *Asynchronous Prim requires $O(n^2)$ msgs.*

Proof: *We have the following messages:*

1. ***Test-Reject*** *msgs: at most 2 for each edge, namely $O(m)=O(n^2)$ messages of this type.*
2. *In each phase, each node:*
   - *sends at most **a single** message of the following type: **Report, Add_edge, Connect,** and **Connected**;*
   - *receives at most **a single** **Search_MOE** message;*
   - *sends and then receives at most **a single** **Test** followed by an **Accept**;*

which means that in each phase globally circulate $O(n)$ messages. Since we have $n-1$ phases, the claim follows.

END OF PROOF

Synchronous Prim

It will work in $O(n^2)$ rounds, can you see why?

Basically, each phase takes $O(n)$ rounds; indeed, the (only) fragment has height (longest root-leaf path, in terms of edges) at most n-1, and so all the root-nodes (and backwards) messages requires $O(n)$ rounds; moreover, each each node has at most n-1 incident edges, and so the **local MOE selection** requires $O(n)$ rounds; since we have $O(n)$ phases, the $O(n^2)$ bound follows

Kruskal's Algorithm (sequential version)

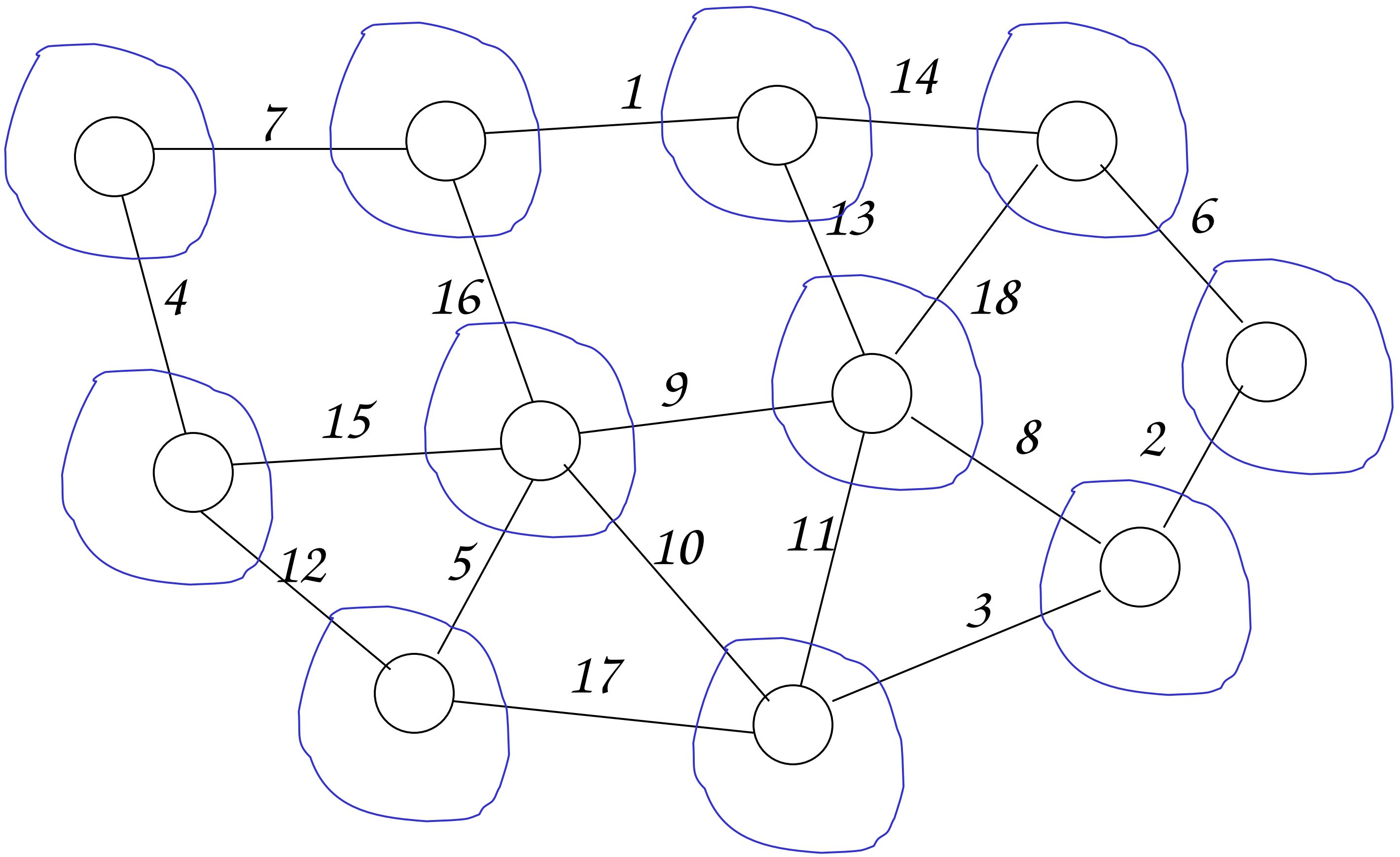
Initially, each node is a fragment

Repeat

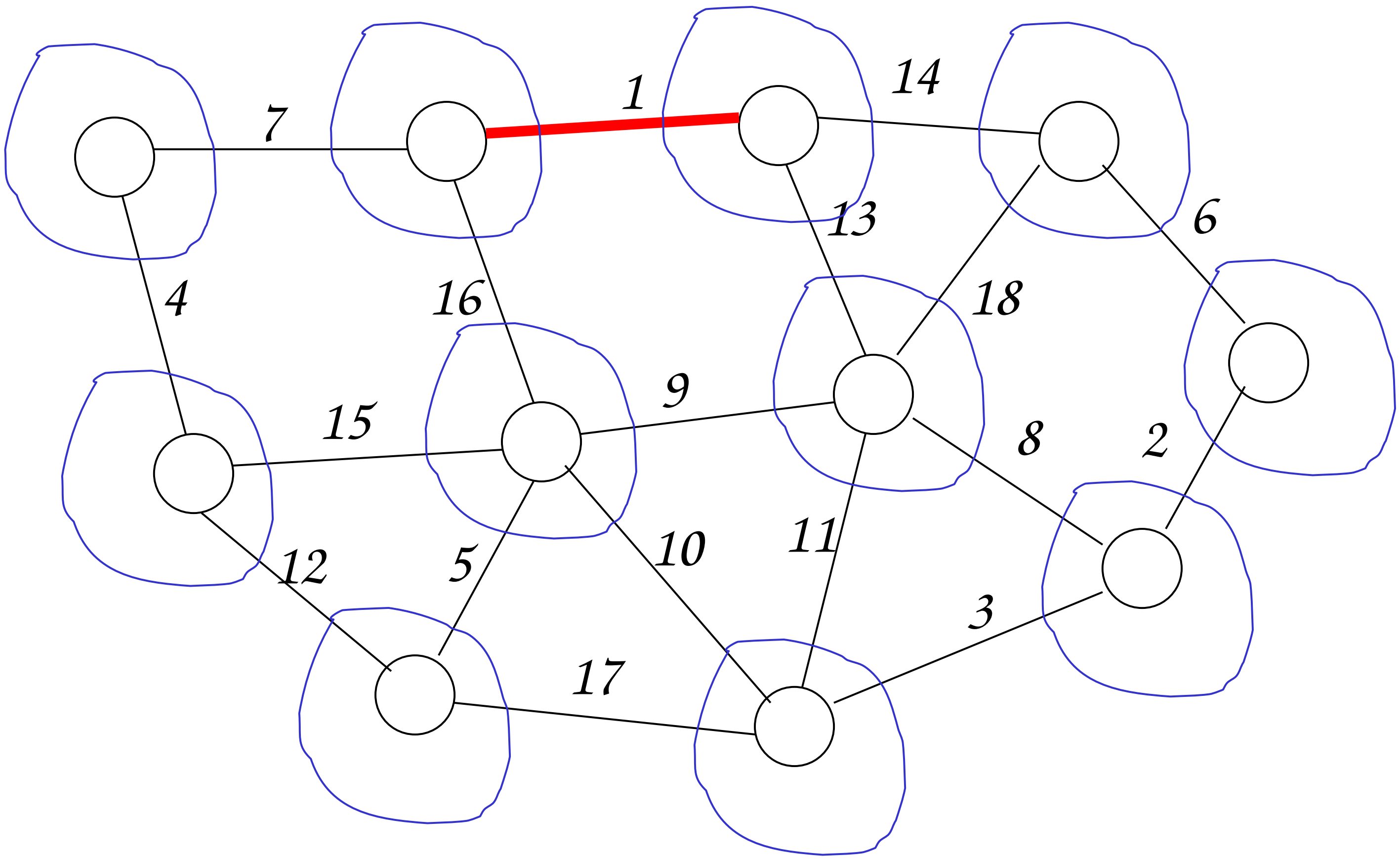
- Find the smallest MOE *e* of all current fragments
- Merge the two fragments adjacent to *e*

Until there is only one fragment left

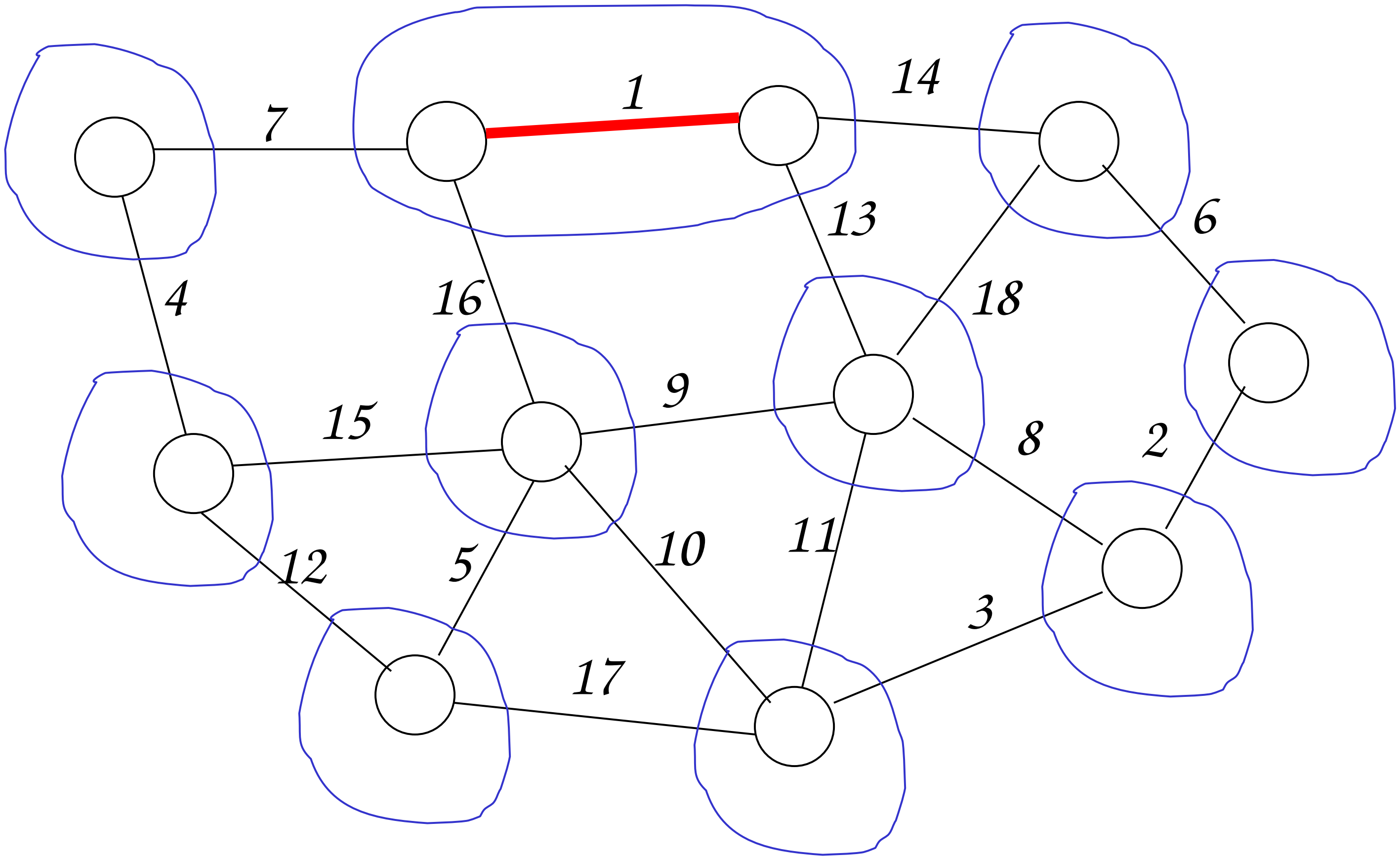
Initially, every node is a fragment



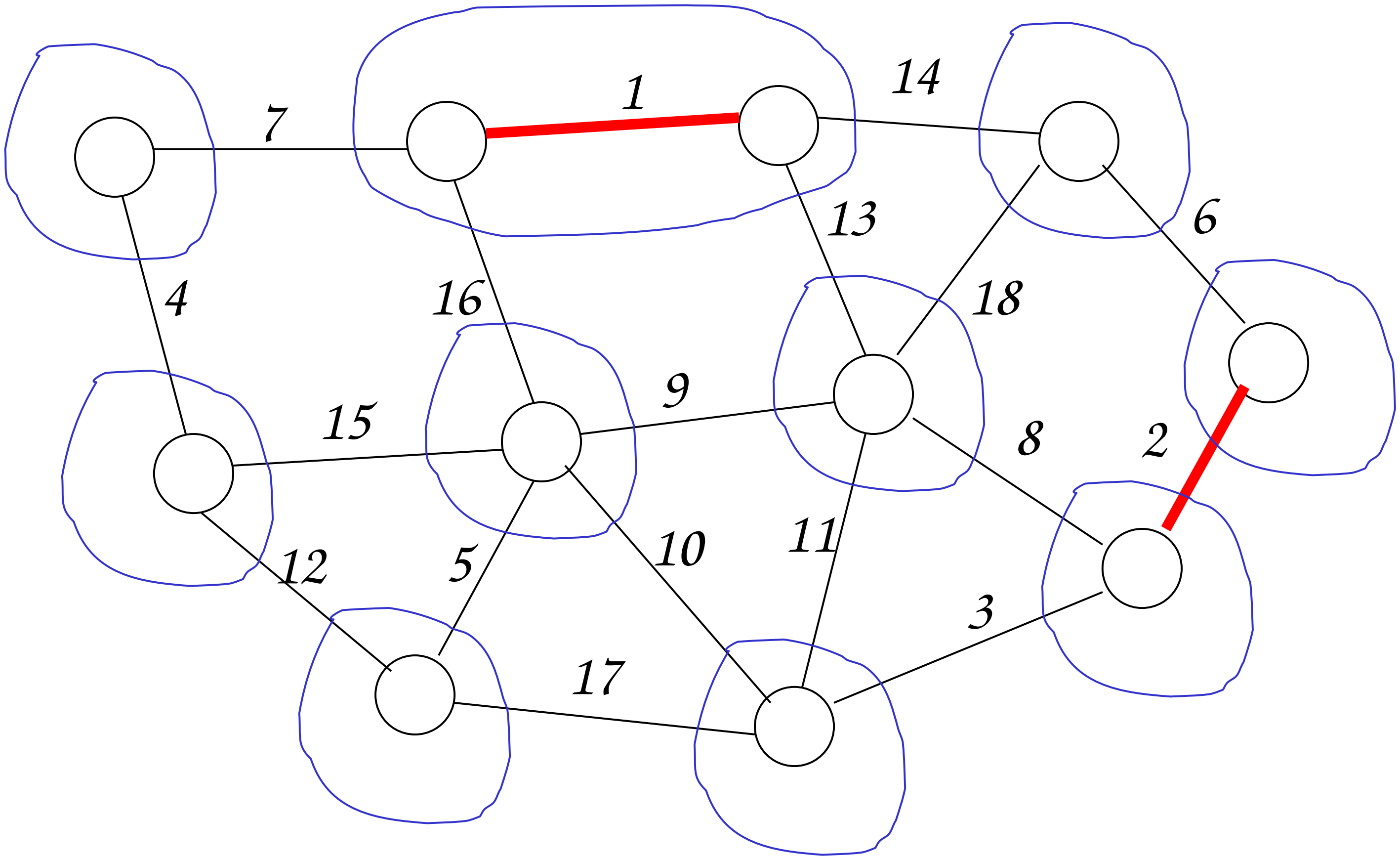
Find the smallest MOE



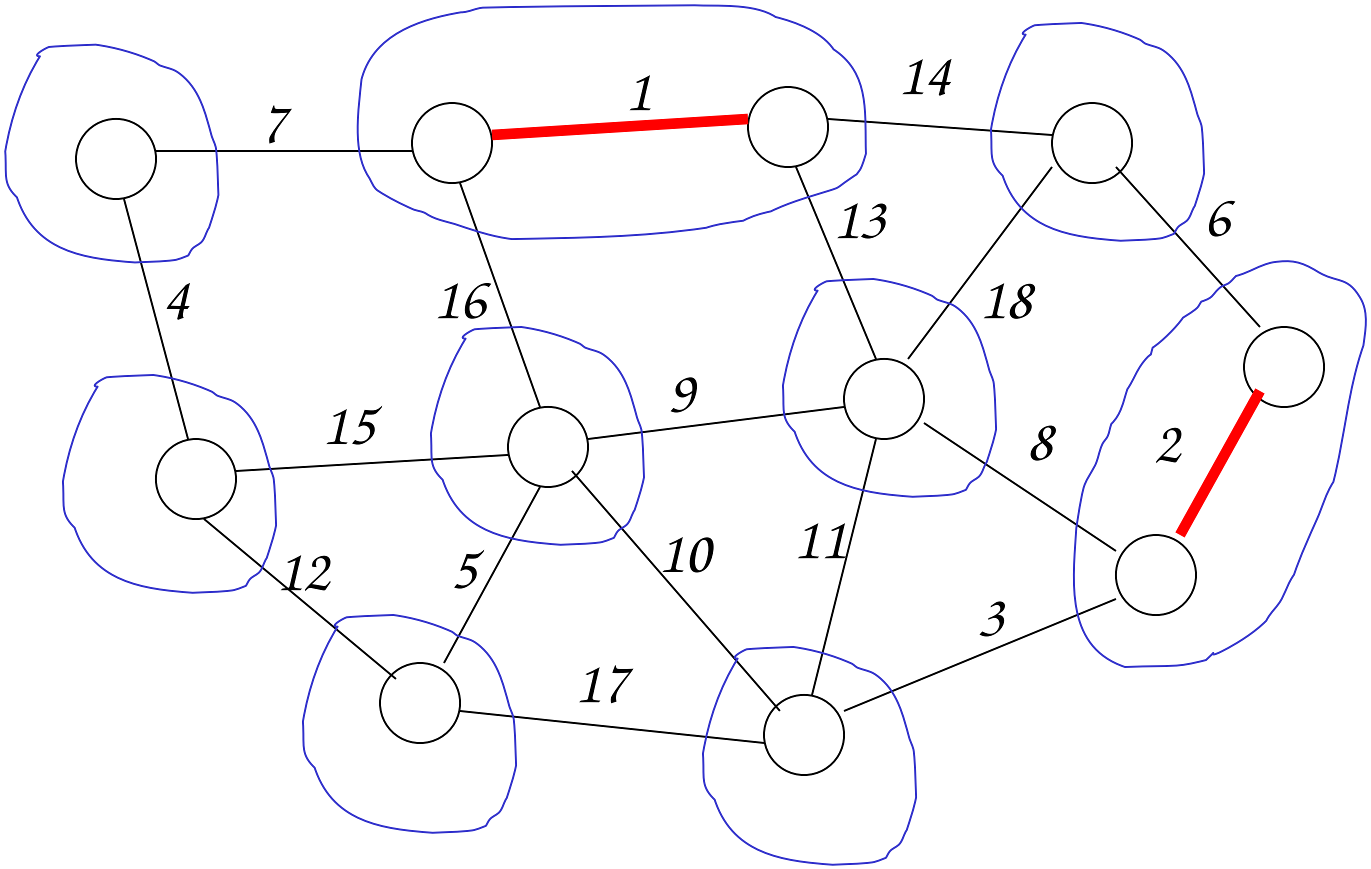
Merge the two fragments



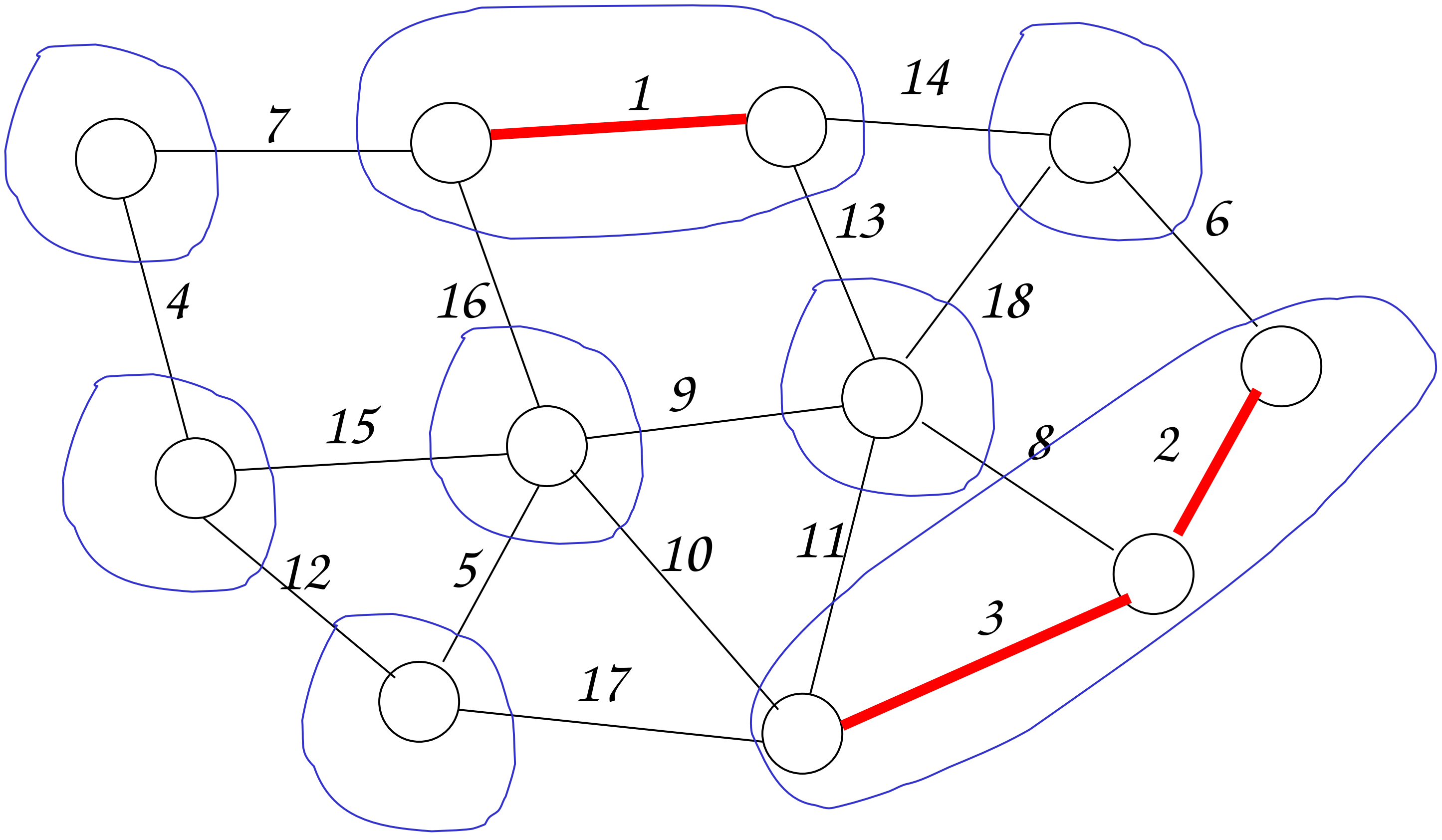
Find the smallest MOE



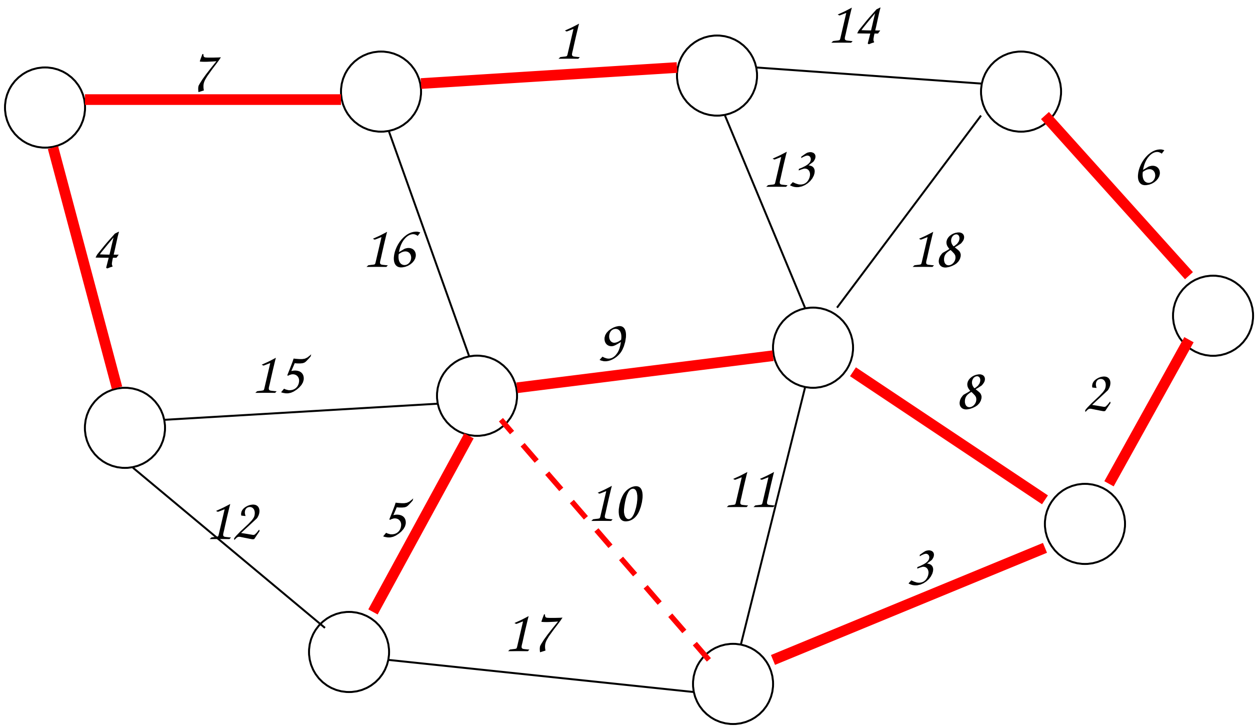
Merge the two fragments



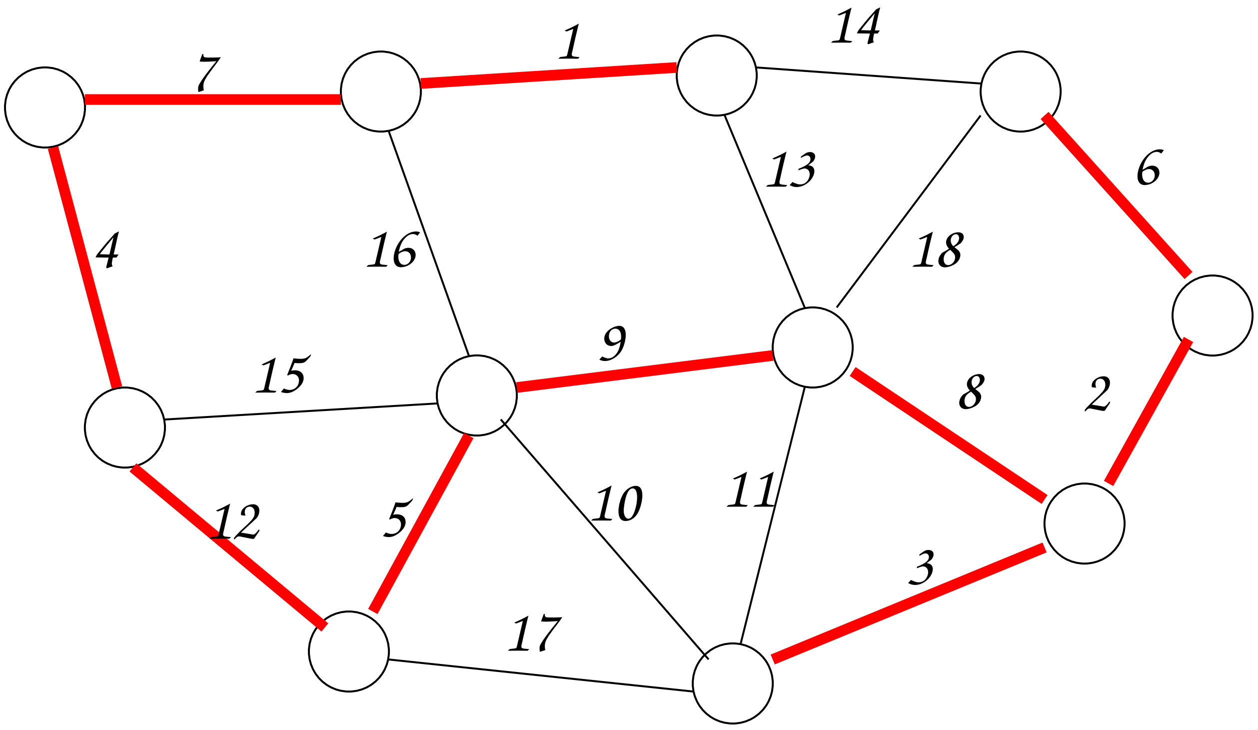
…go on by merging fragments…



…and discarding edges forming a cycle…



…until arriving to the resulting MST



Theorem: *Kruskal's algorithm gives an MST*

Proof: *Use Property 1, and observe that no cycle is created (indeed, we always select a MOE).*

END OF PROOF

Distributed version of Kruskal's (i.e., Boruvka) Algorithm: Gallagher-Humblet-Spira (GHS) Algorithm (1983)

- We start by providing a synchronous version, working under the following **restrictions**: non-anonymous, non-uniform MPS, distinct weights; for the sake of simplicity, we will assume a synchronous start, but it is not really needed
- Works in phases, by repeatedly applying the **blue rule** to multiple fragments
- Initially, each node is a fragment, and phases proceed by implementing the following steps:

Repeat a phase (These phases need to be synchronized, as we will see later)

- Each fragment – coordinated by a fragment root node - finds its MOE
- Merge fragments by using the found MOEs

Until there is only one fragment left

Local description of synchronous GHS

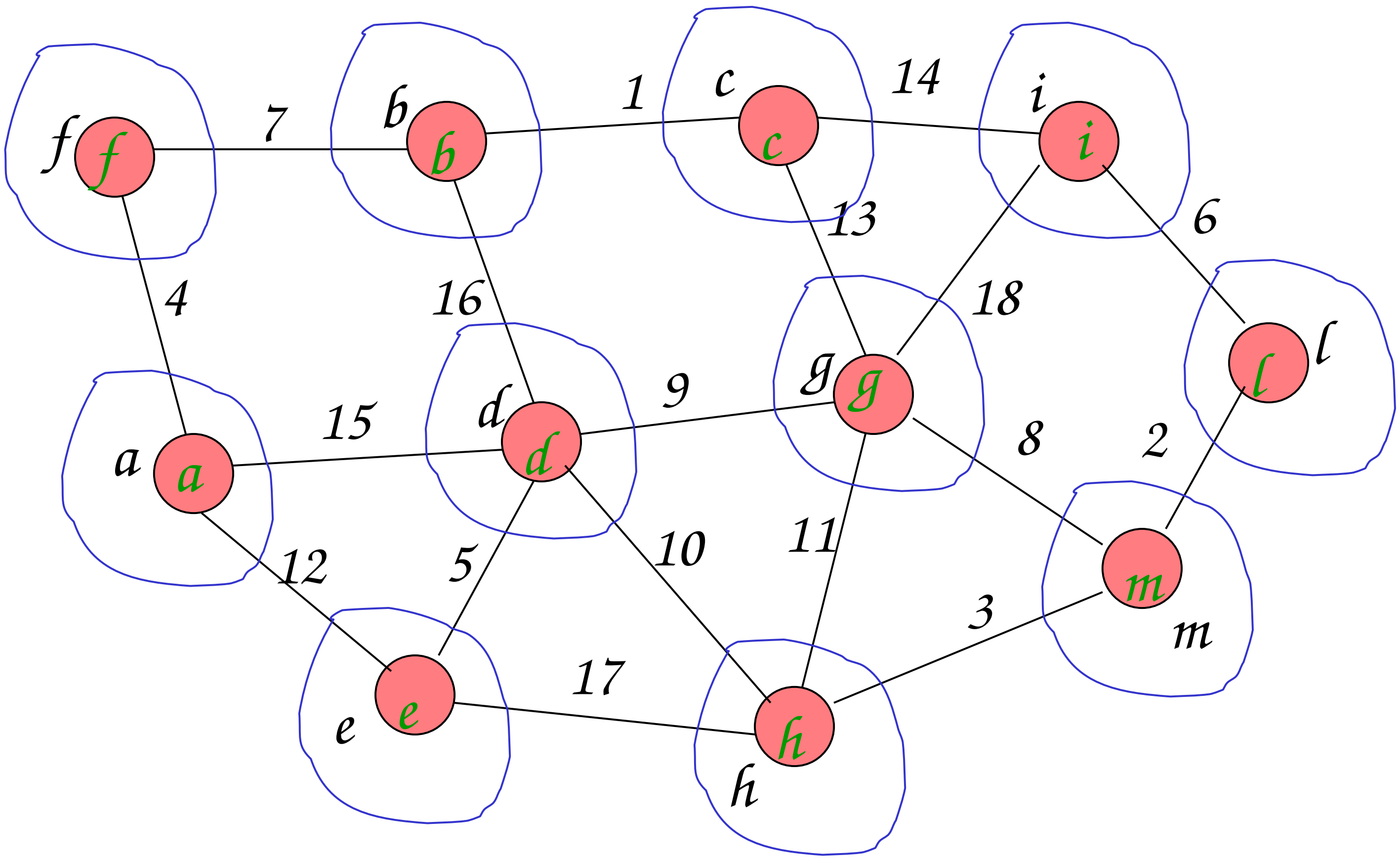
Each processor stores, besides its own ID:

1. *The **status** of any of its incident edges, which can be either of {basic, branch, reject}; initially all edges are basic*
2. *Fragment identity; initially, when each fragment is done by a single node, this is equal to the **node ID**, but then it will be equal to the ID of **some node** in the fragment*
3. *Root channel (**current** route towards the fragment root)*
4. *Children channels (**current** routes towards the descending fragment leaves)*
5. *Local (incident) MOE*
6. *MOE for each children channel*
7. *MOE channel (route towards the MOE of the appended subfragment)*

Types of messages of synchronous GHS

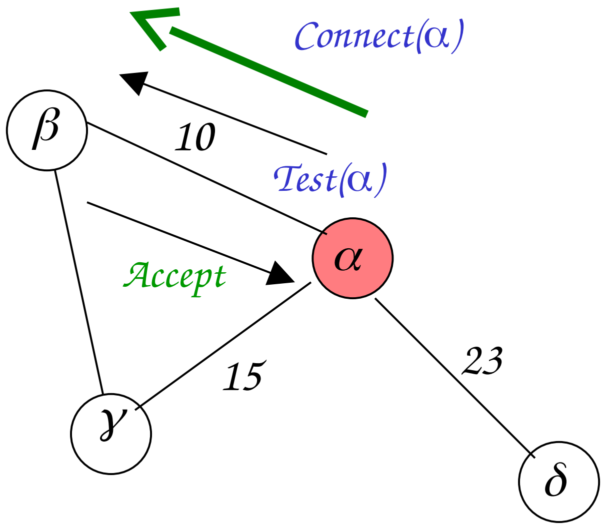
1. ***Test(fragment identity):*** *each node starts a phase by checking the status of its basic edges in increasing weight order (if any)*
2. ***Reject**, **Accept**: response to **Test***
3. ***Report(weight):*** *for reporting to the parent node the MOE of the appended subfragment*
4. ***Merge** (this was called **Add_edge** in Prim)**:** sent by the root to the node incident to the MOE to activate the merging of fragments*
5. ***Connect(fragment identity):*** *sent by the node incident to the MOE to perform the merging; as we will see, this message will be sent in the **very same round** by all the involved nodes; in the immediately next round, merges took place, and a new root for each fragment is selected, in a way that will be specified later*
6. ***New_fragment(fragment identity):*** *coordination message sent by the new root of a just created fragment at the end of a phase, and containing the new fragment identity (this will be specified later)*

Initially, every node is a fragment…

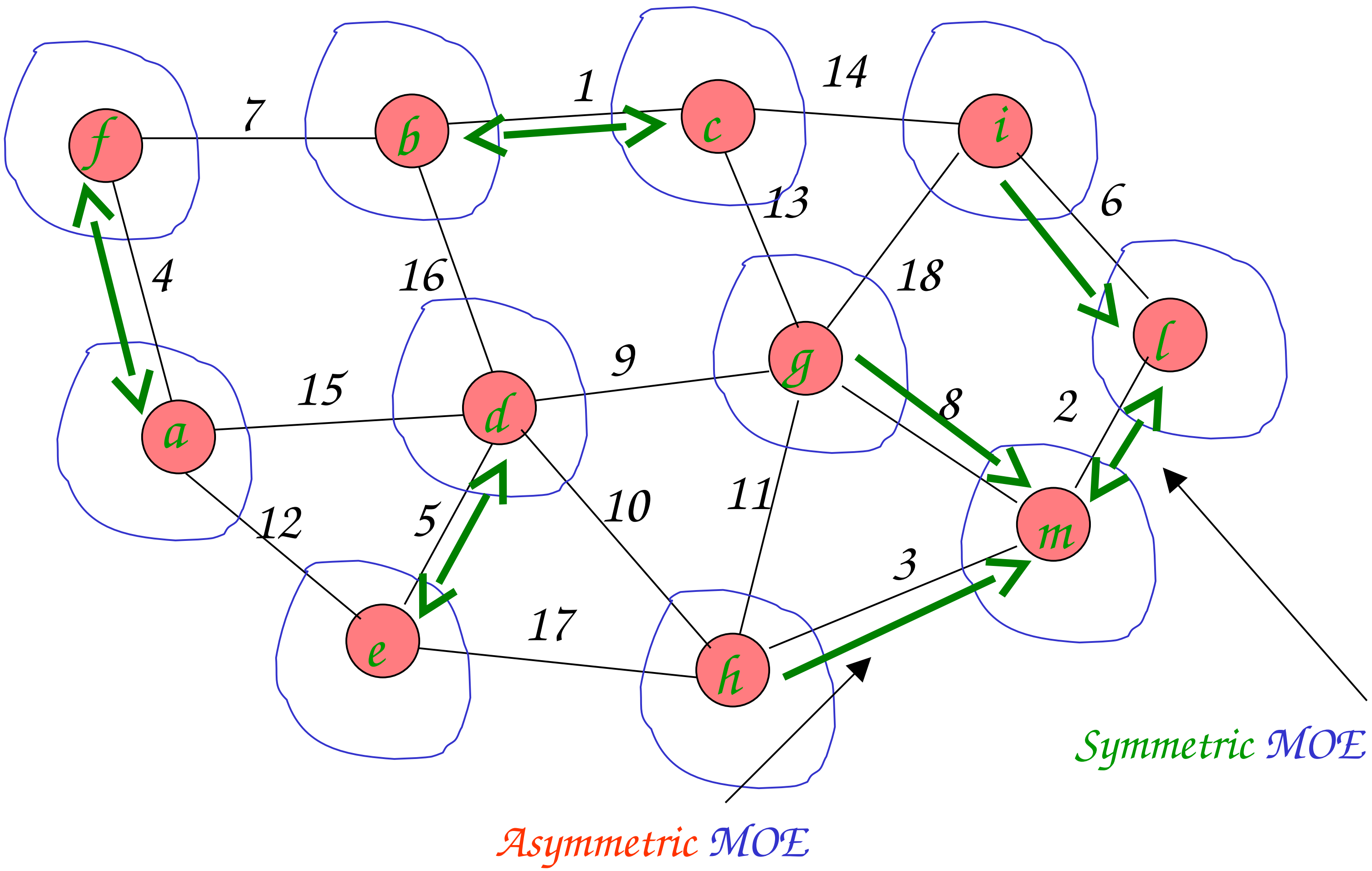


… and every node is the root of a fragment, and the fragment identity (reported within the nodes) is the node ID (reported nearby the nodes), and so non-anonimity is required

Phase 1: In this very first phase, to discover its own MOE, a node sends a ***Test*** *message containing its fragment identity over its* *basic* *edge of minimum weight, and it will certainly receives an* ***Accept****; then, it will send a* ***Connect*** *message containing its fragment identity (notice that* ***Report*** *and* ***Merge*** *messages are not needed in this first phase since the root is directly adjacent to the MOE)*

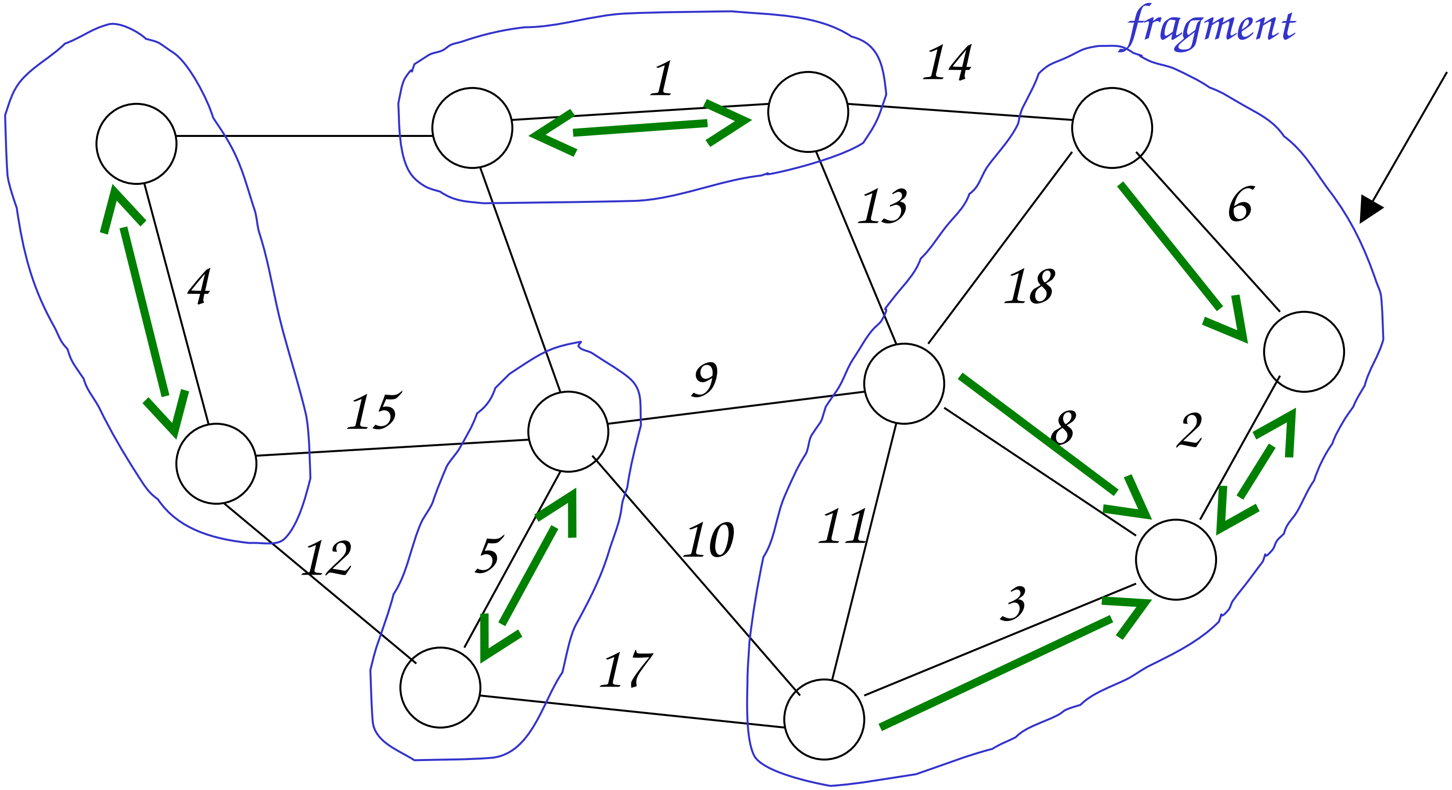


Phase 1: In our example, each node finds its MOE and sends a ***Connect*** *message (arrows denote the direction of the* ***Connect*** *message)*



*Phase 1: Merge the nodes and select a **new root***

Notice: Several nodes can be merged into a single new fragment



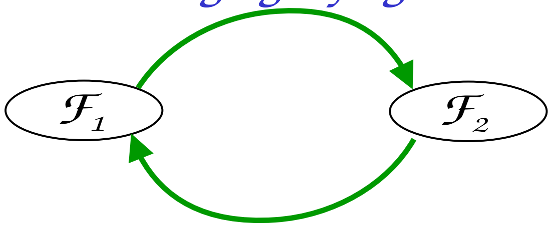
***Question**: How do we select the new roots?*

Selecting a new root: a useful property

Proposition: In merged fragments there is exactly one symmetric MOE.

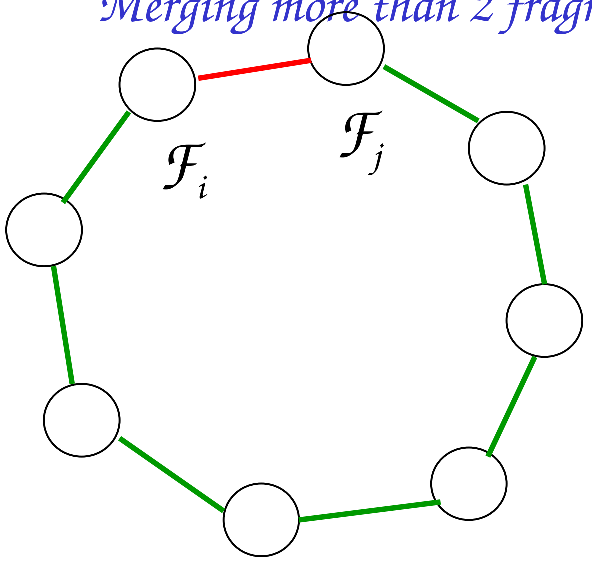
Proof: Recall that each merging fragment has exactly 1 MOE. Assuming that fragments find their MOE correctly (we will prove formally this later), we claim that since edge weights are distinct, then no cycles are created during merging. Indeed, for the sake of contradiction, assume this is false. We can have two cases:

Merging 2 fragments



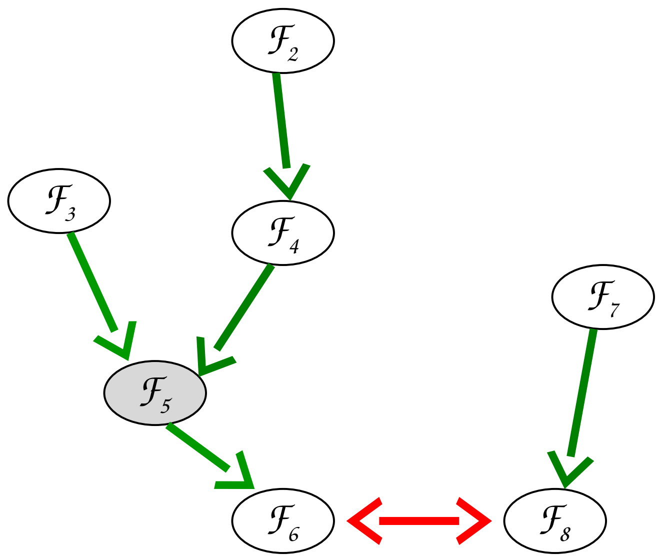
Impossible: either $\mathcal{F}_1$ or $\mathcal{F}_2$ is choosing a wrong MOE!

Merging more than 2 fragments



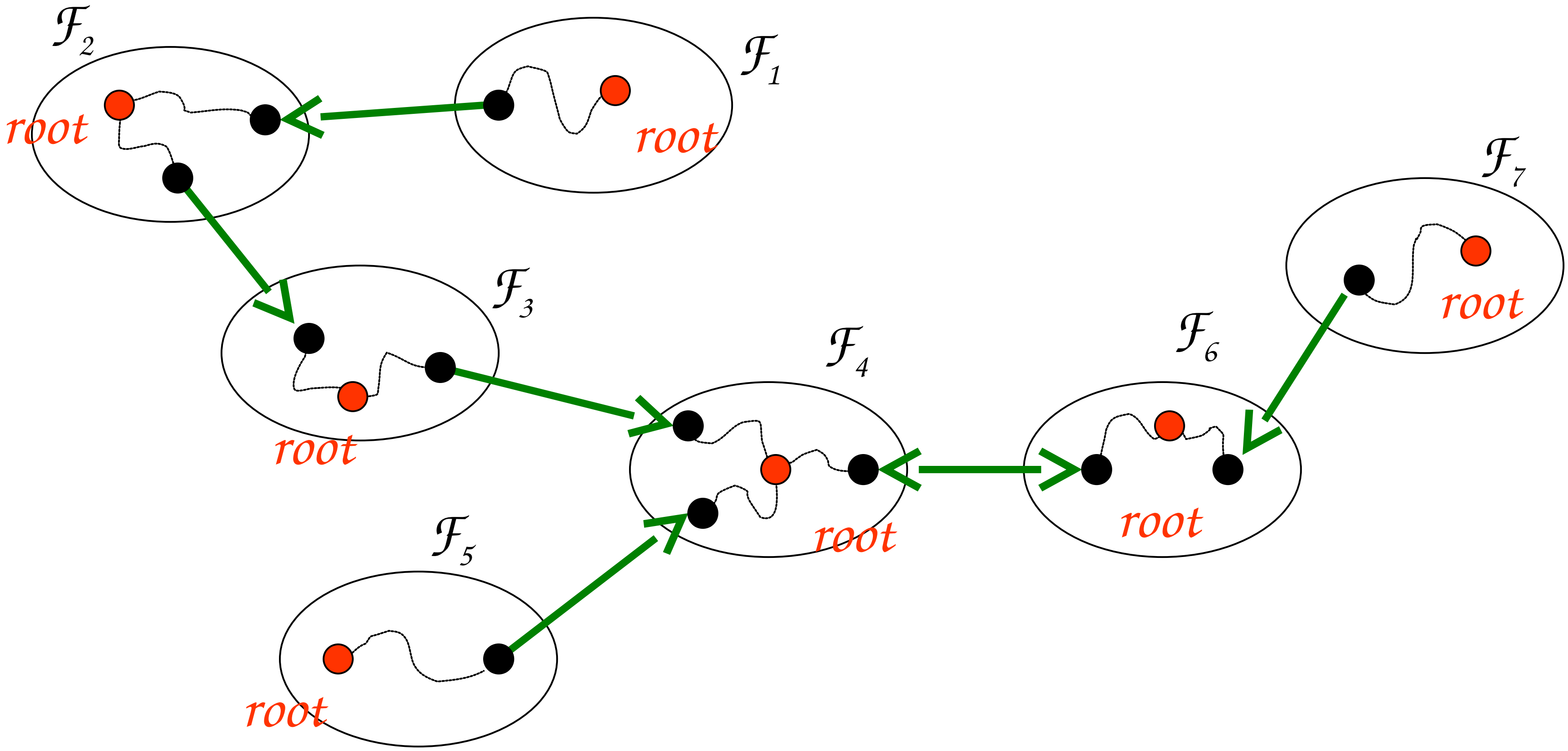
Impossible: let $(\mathcal{F}_i,\mathcal{F}_j)$ be the (only) max-weight edge on the cycle; then either $\mathcal{F}_i$ or $\mathcal{F}_j$ is choosing a wrong MOE!

(Proof cont'd) *Then, since no cycles are created, we have that if* k *fragments are merged,* k*-1 edges need to be used to perform the merge (to guarantee connectivity and acyclicity). These edges contain exactly* k *arrows, one for each fragment, and so there must be exactly one edge with two arrows, i.e., a symmetric edge.*

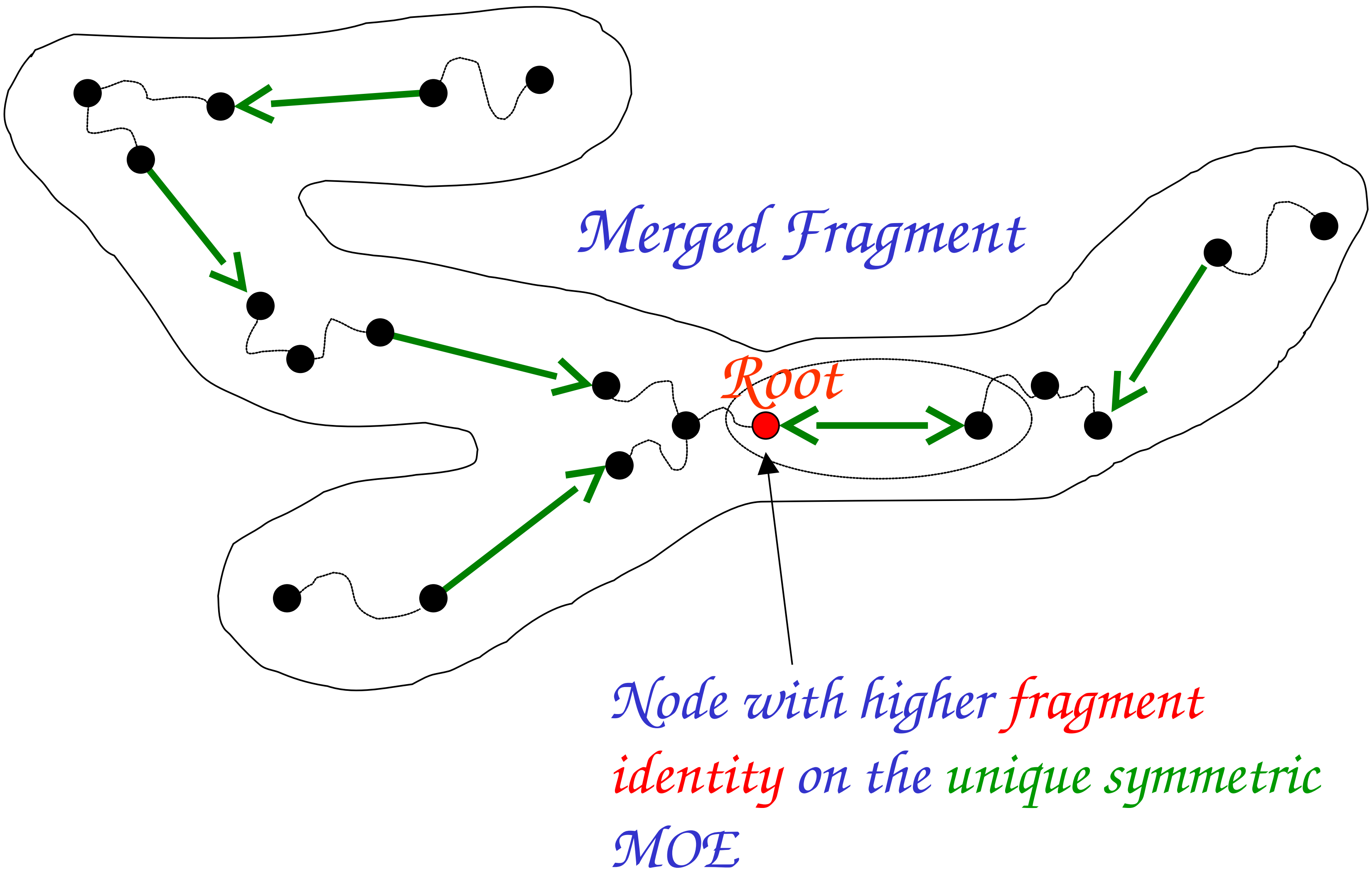


QED

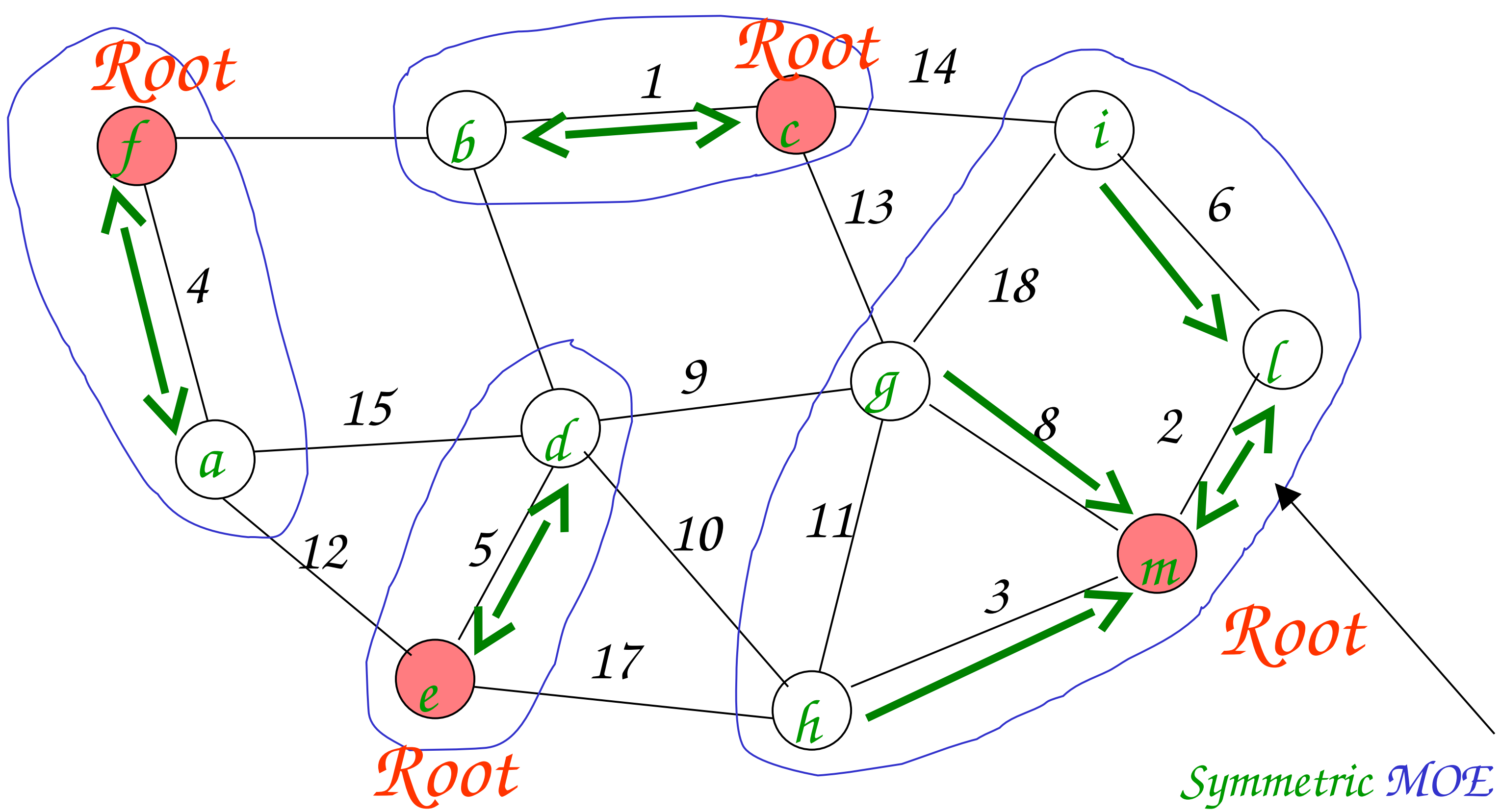
*Rule for selecting a **new root** in a fragment*



Rule for selecting a new root in a fragment (2)

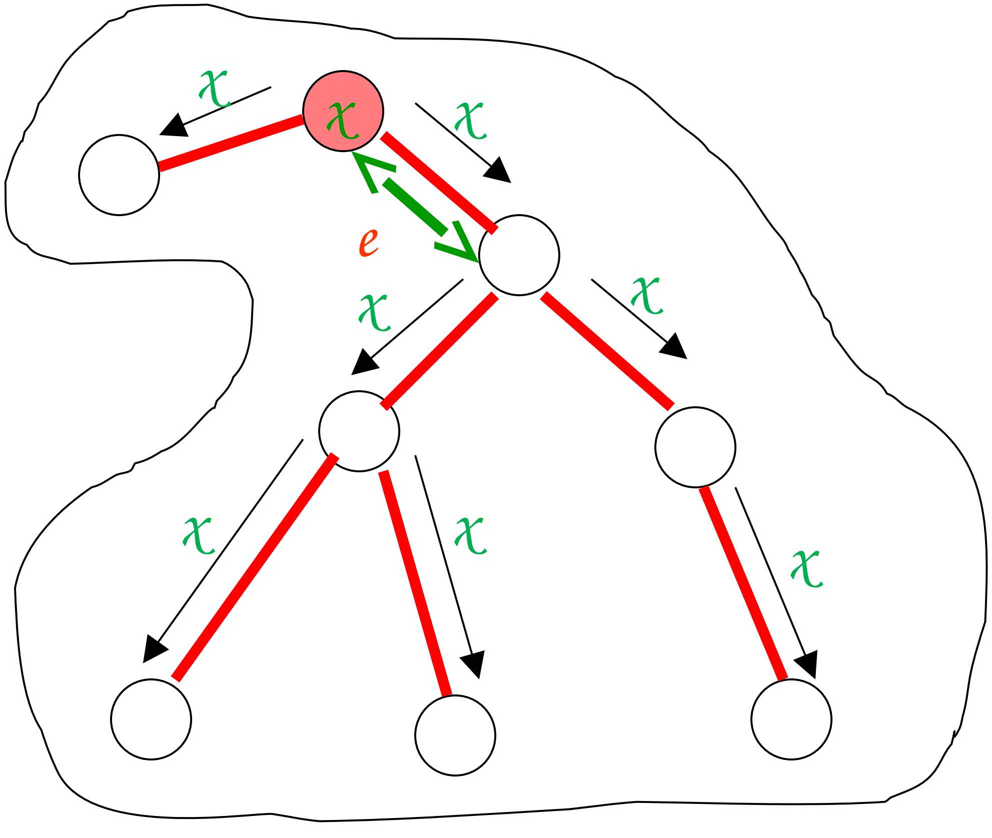


In our example, after the merges, each fragment has its new root



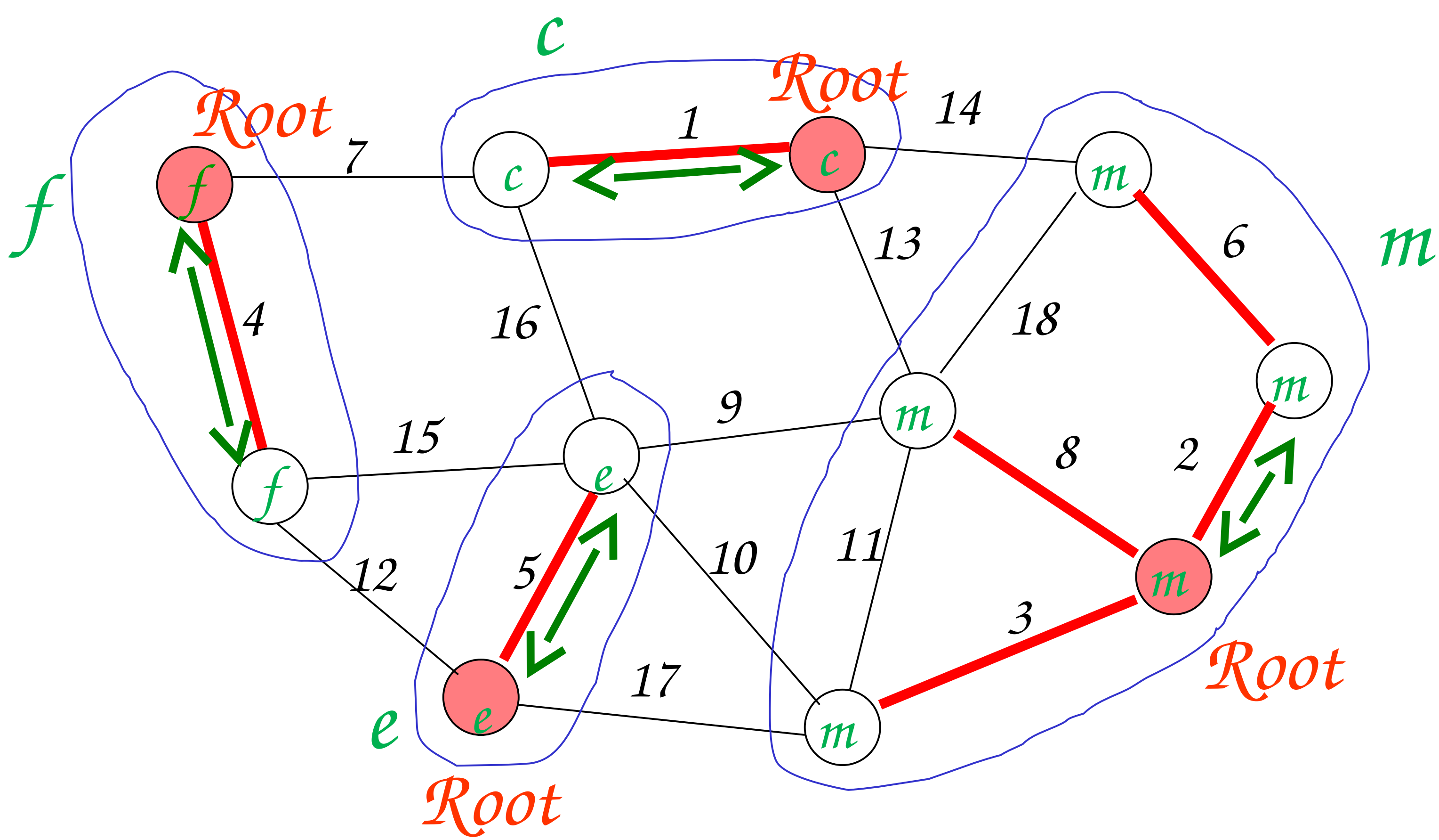
In a generic phase, after that merging has taken place, the new root broadcasts ***New_fragment(x)*** *to all the nodes in the new fragment, where x was the fragment identity of the new root in its previous fragment, and* ***once this notification is completed*** *a new phase starts*

e is the symmetric MOE of the merged fragments, and x is the identity of the fragment the red node was belonging to before the merge



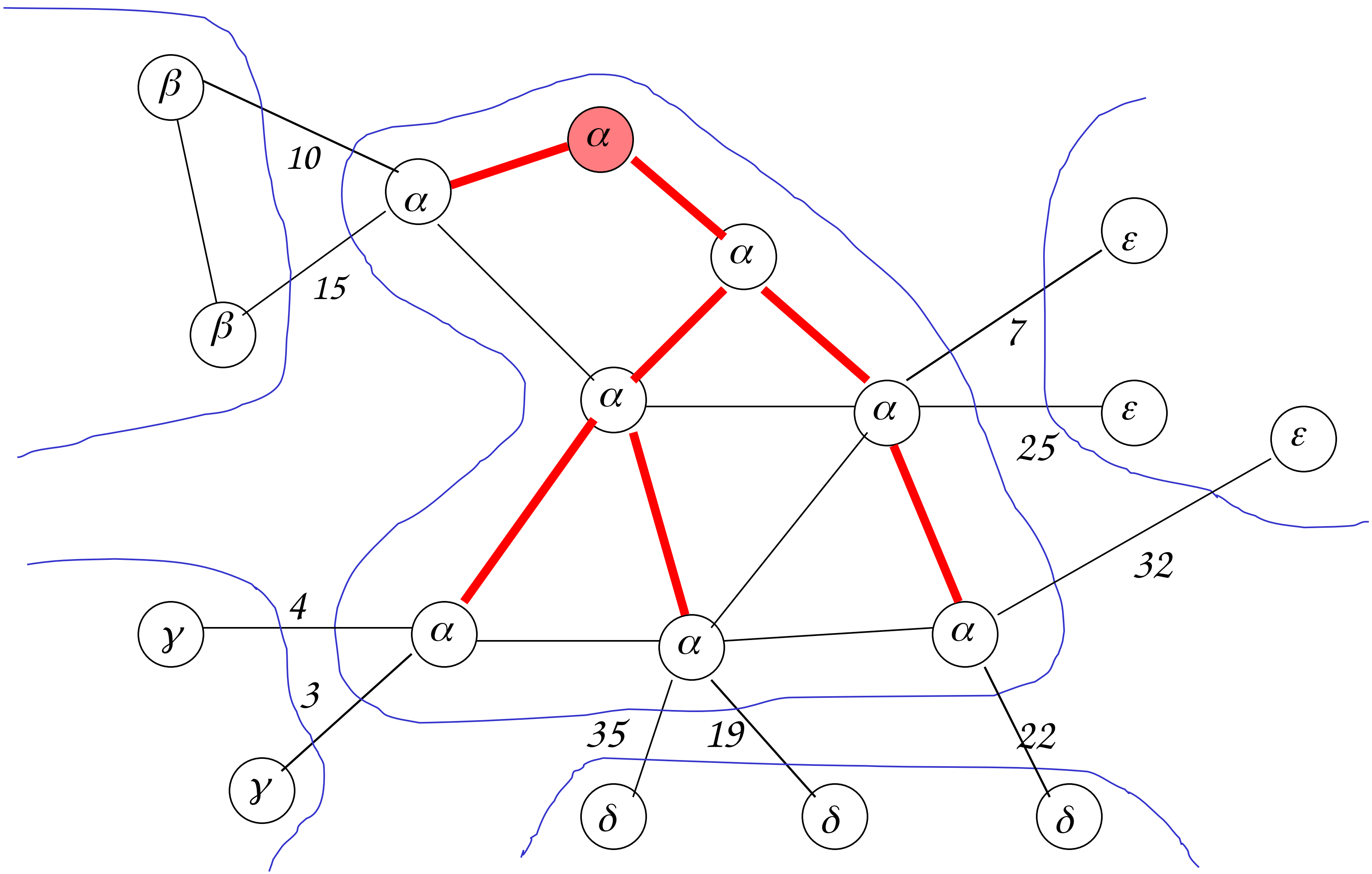
x is the identity of the new fragment

In our example, at the end of phase 1 **every node in every fragment** has its new fragment identity, and a new phase can start

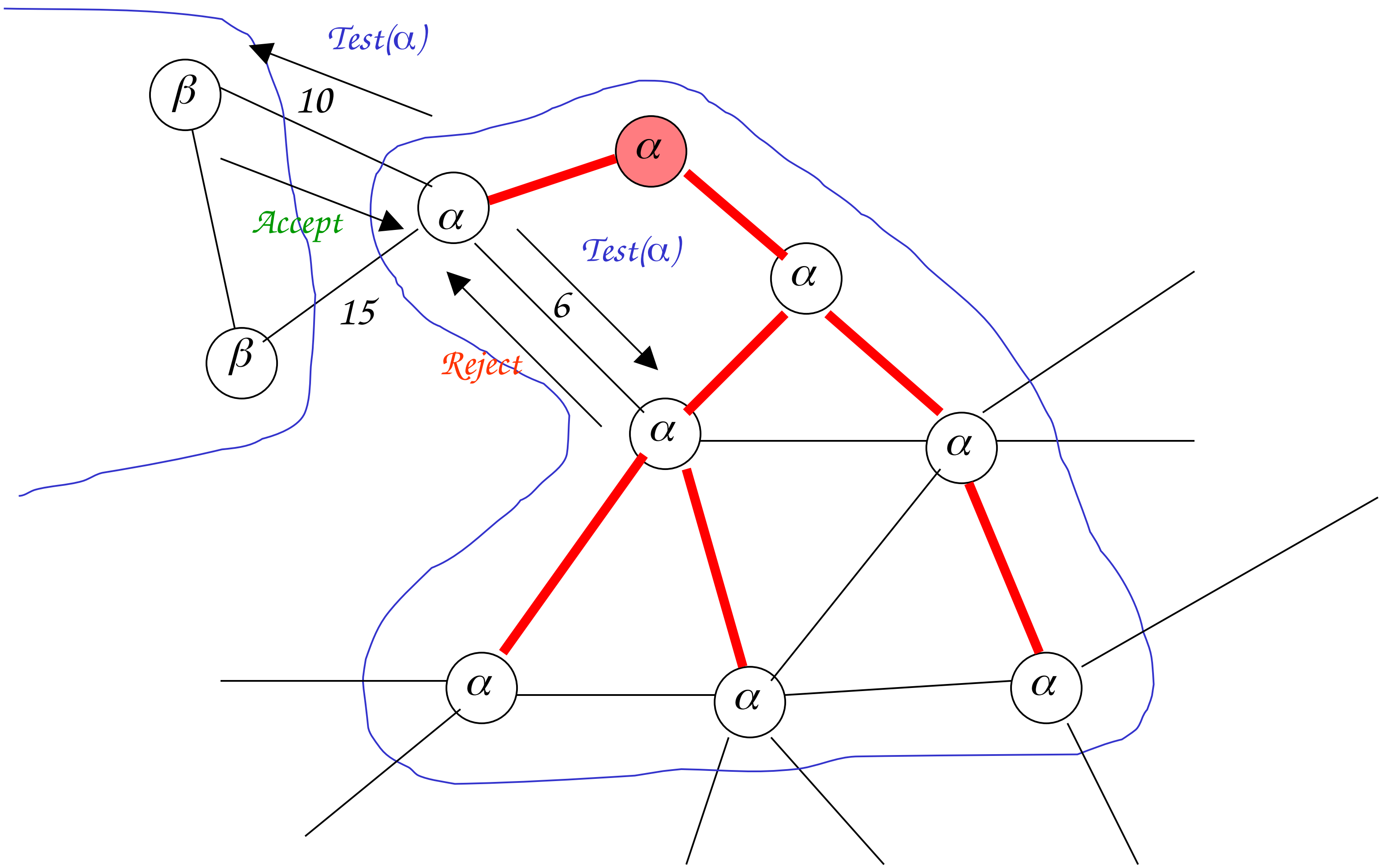


End of phase 1 (notice that the fragment identity is equal to the ID of some node in the fragment)

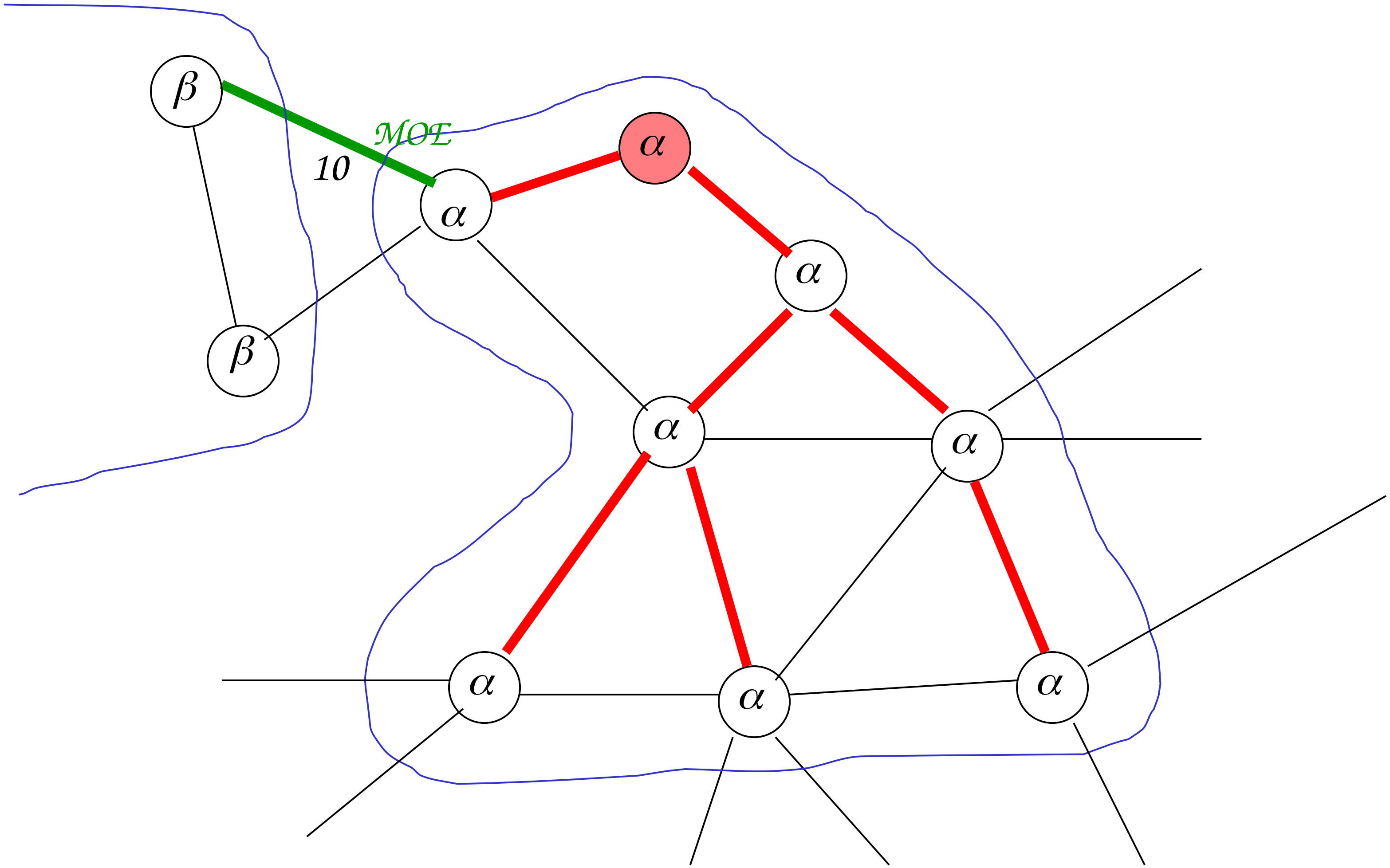
At the beginning of a generic phase, each node in a fragment starts finding its local MOE: the fact that all nodes in the graph have their actual identity guarantees that the correct MOE of each fragment is found



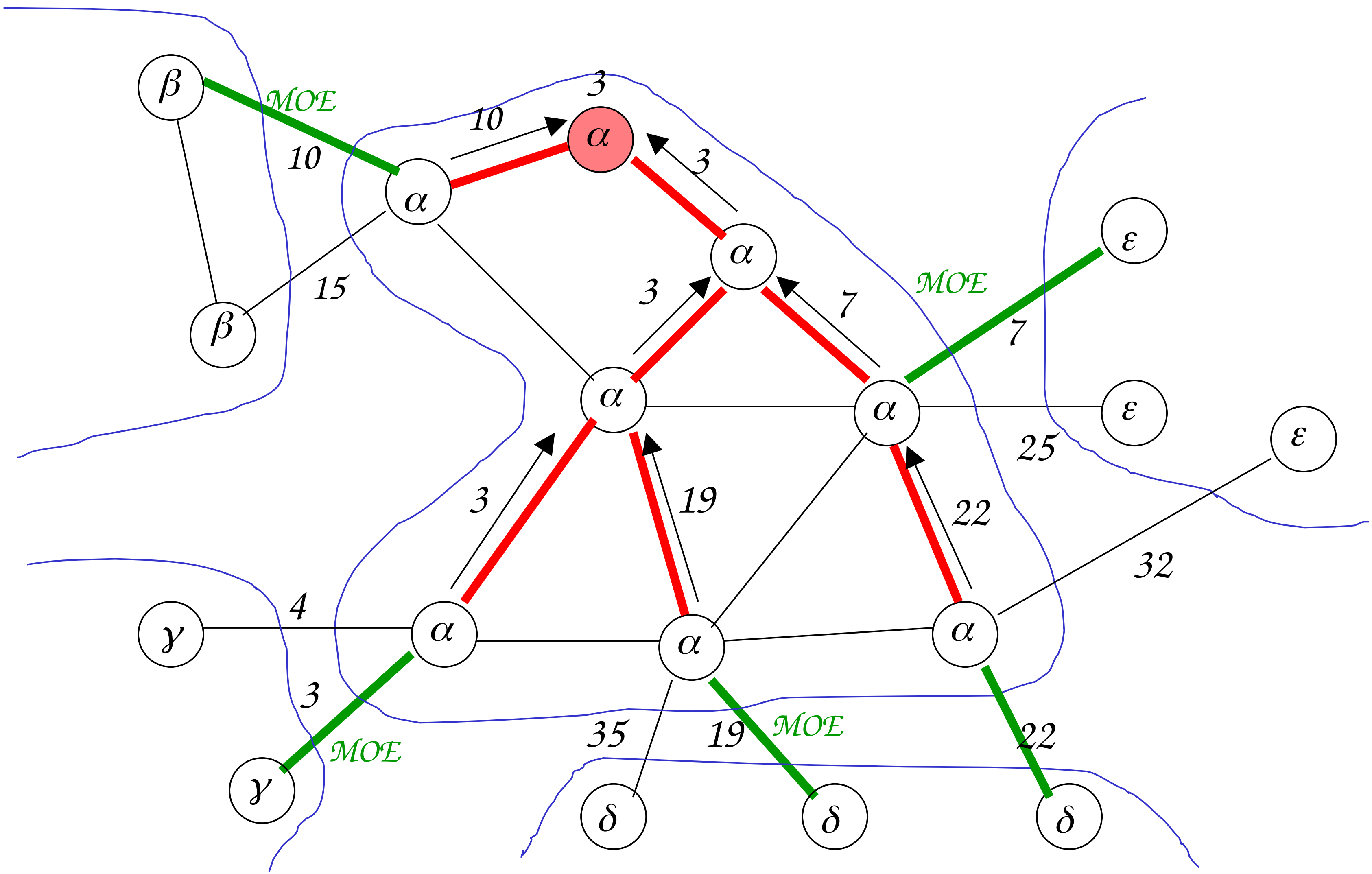
To discover its own MOE, each node sends a ***Test*** *message containing its fragment identity over its basic edges in increasing order of weight, until it receives an* ***Accept***



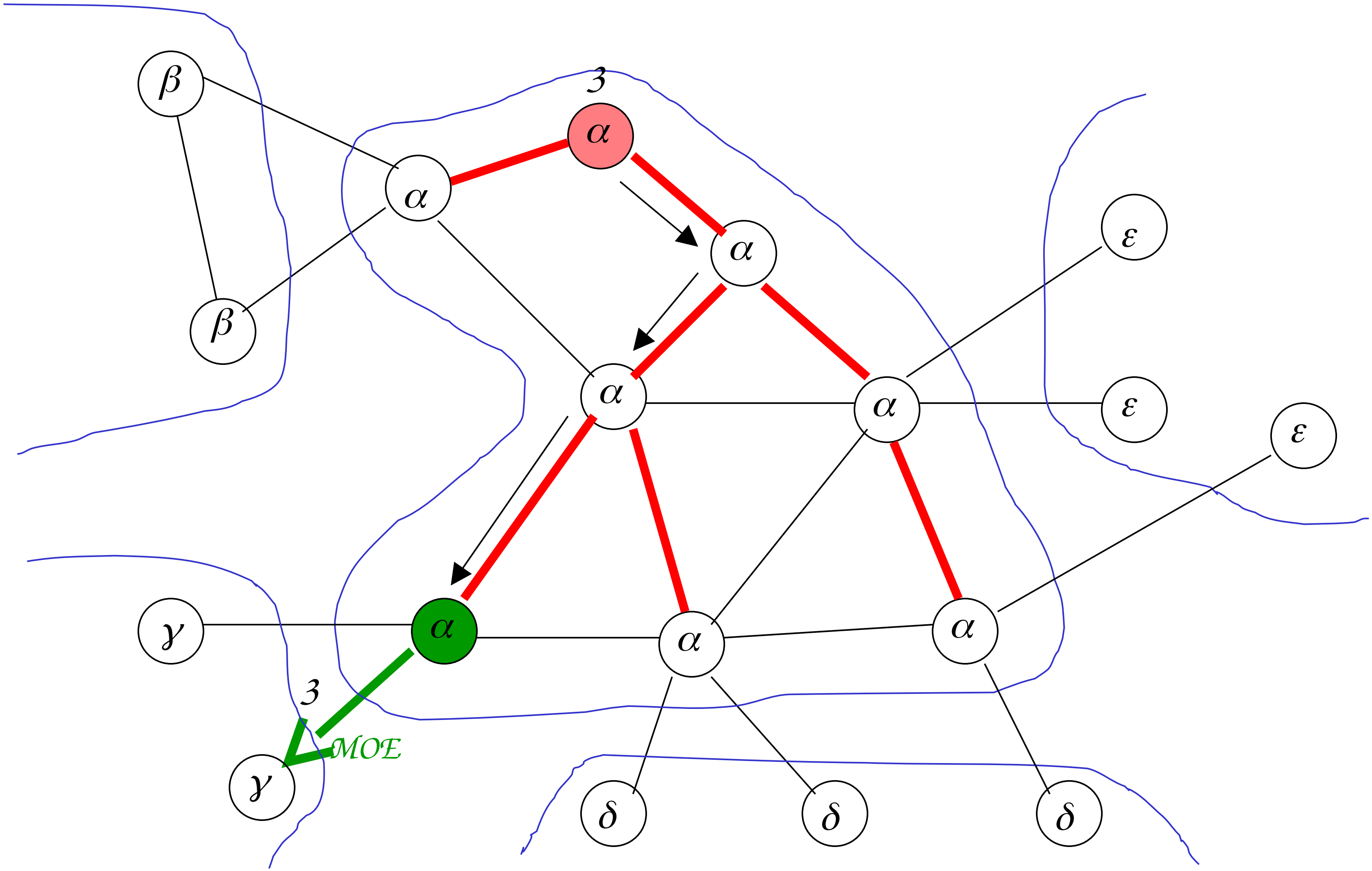
Then it knows its local MOE (notice this can be void)



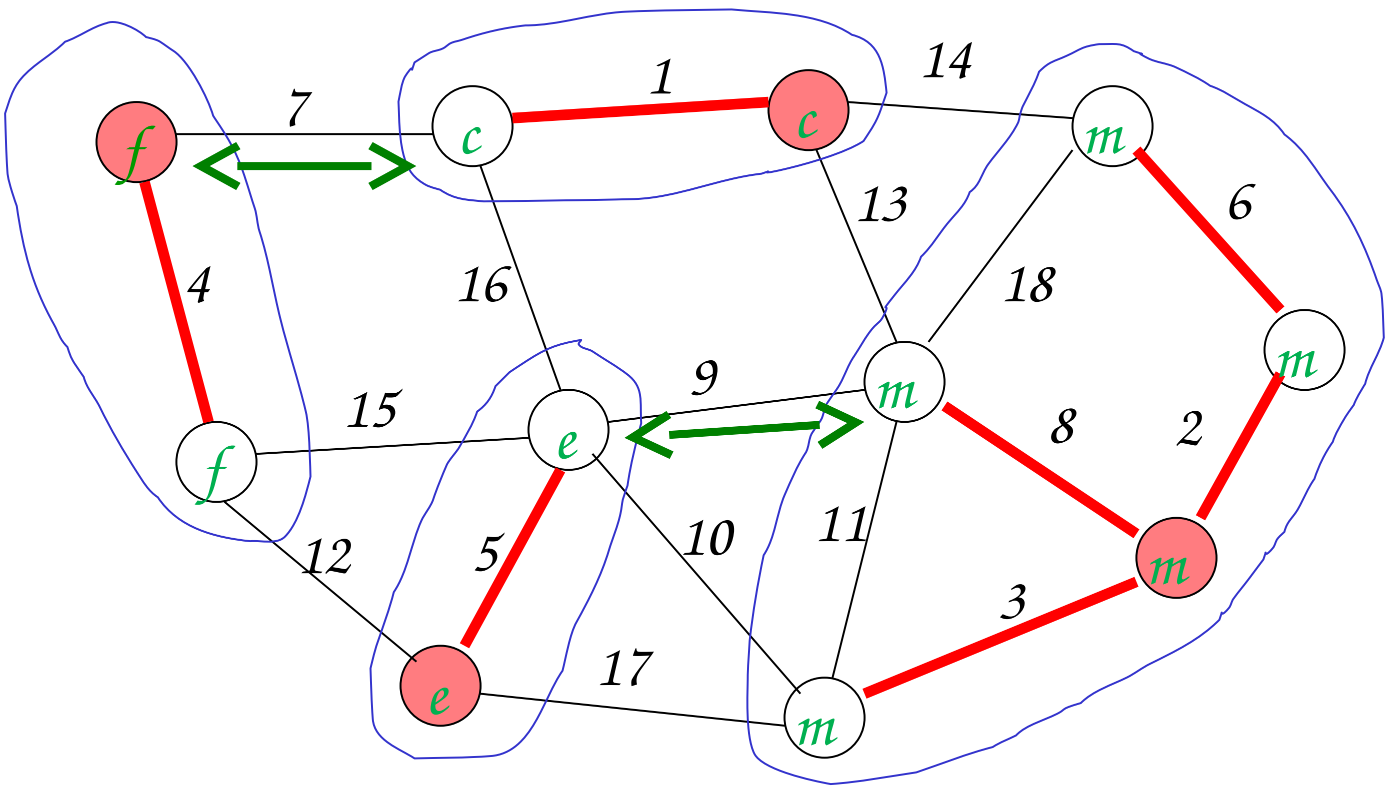
*After receiving the **Report** from each child, a node sends its own **Report** to its parent with the MOE of the appended subfragment (the global minimum survives in propagation towards the root)*



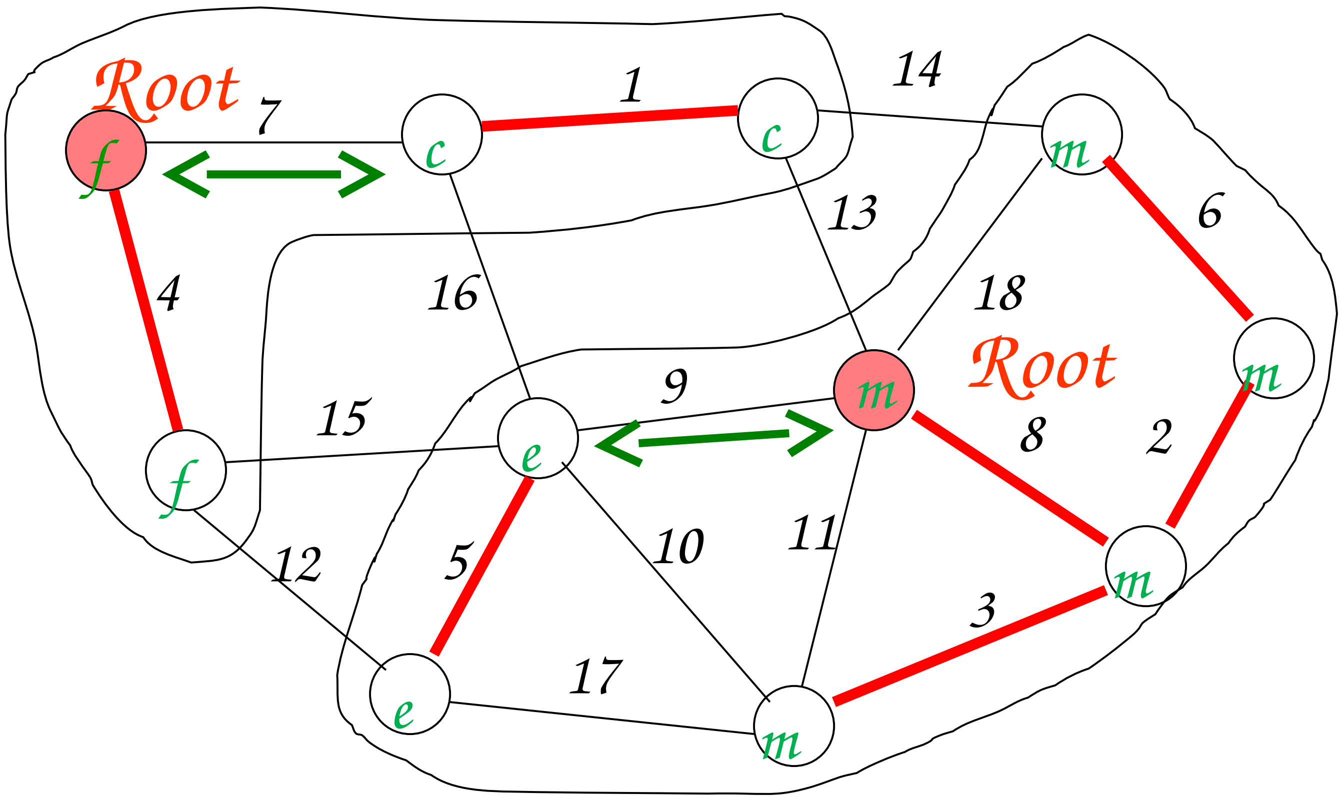
After receiving the ***Report*** *from each child, the root selects the minimum MOE and sends along the appropriate path a* ***Merge*** *message, which will become a* ***Connect(α)*** *message at the proper node (which possibly becomes a new root)*



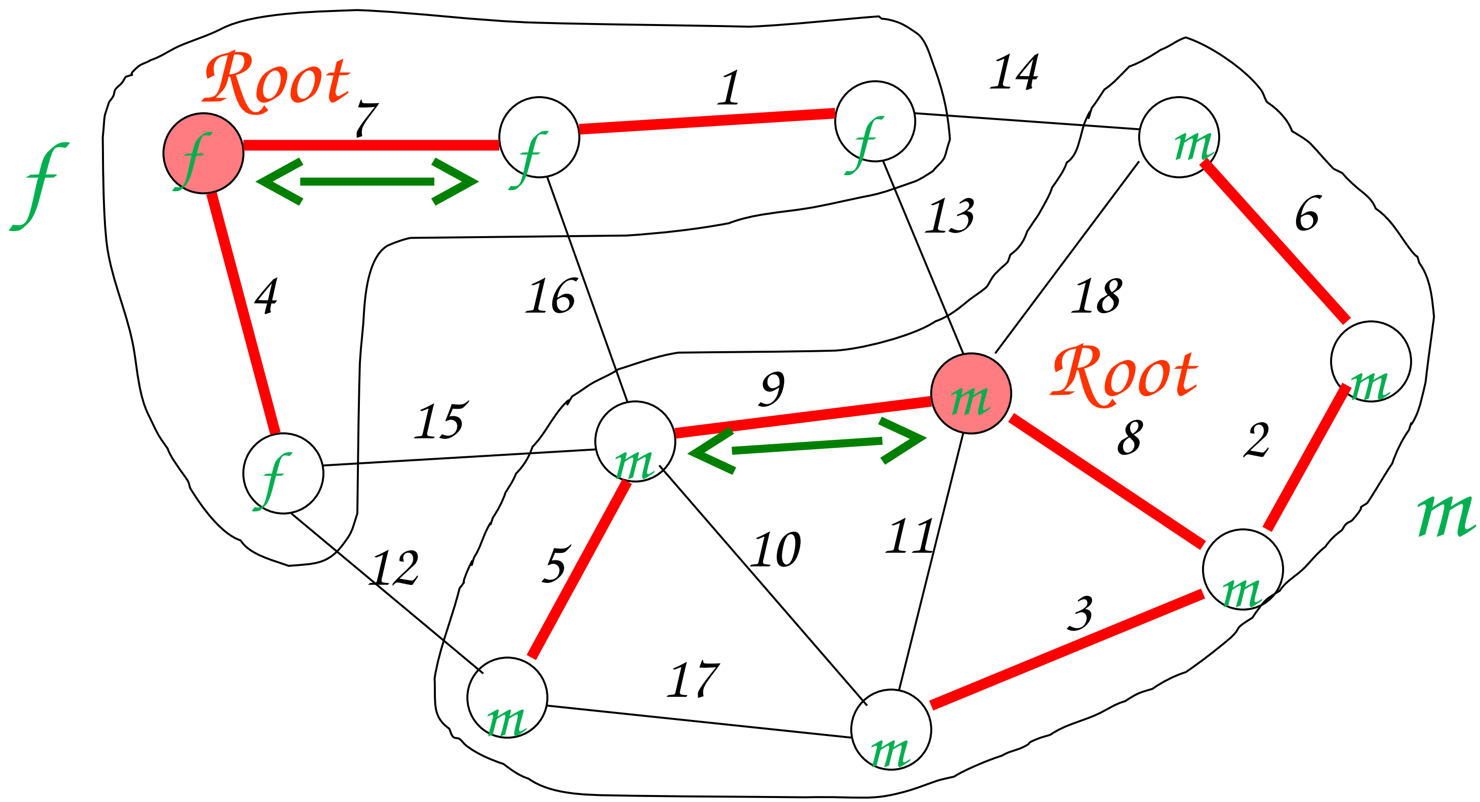
Phase 2 of our example: After receiving the new fragment identity at the end of the previous phase, find again the MOE for each fragment



Phase 2: Merge the fragments and select a new root

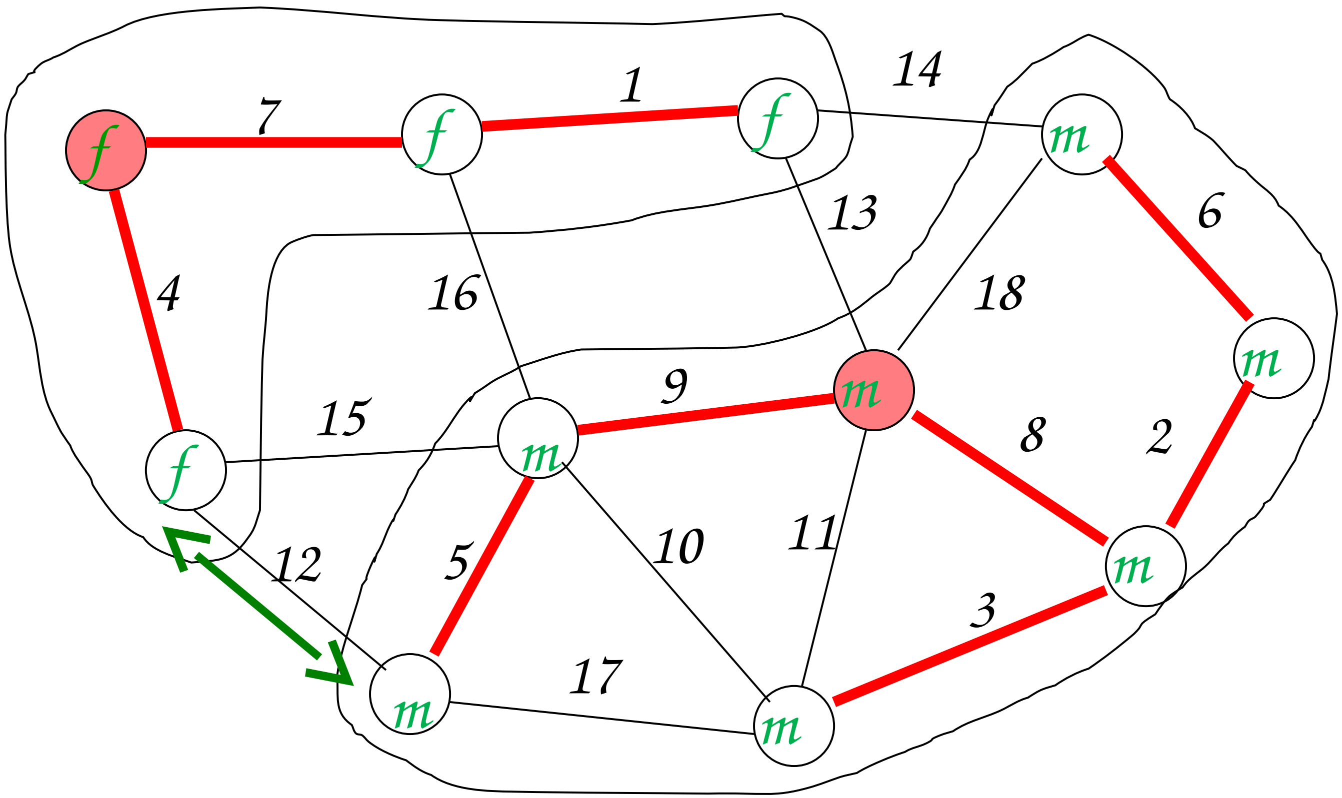


Broadcast the ***new fragment identity*** *to all the nodes in the new fragment, and so at the end of phase 2 each node knows its own new fragment identity.*

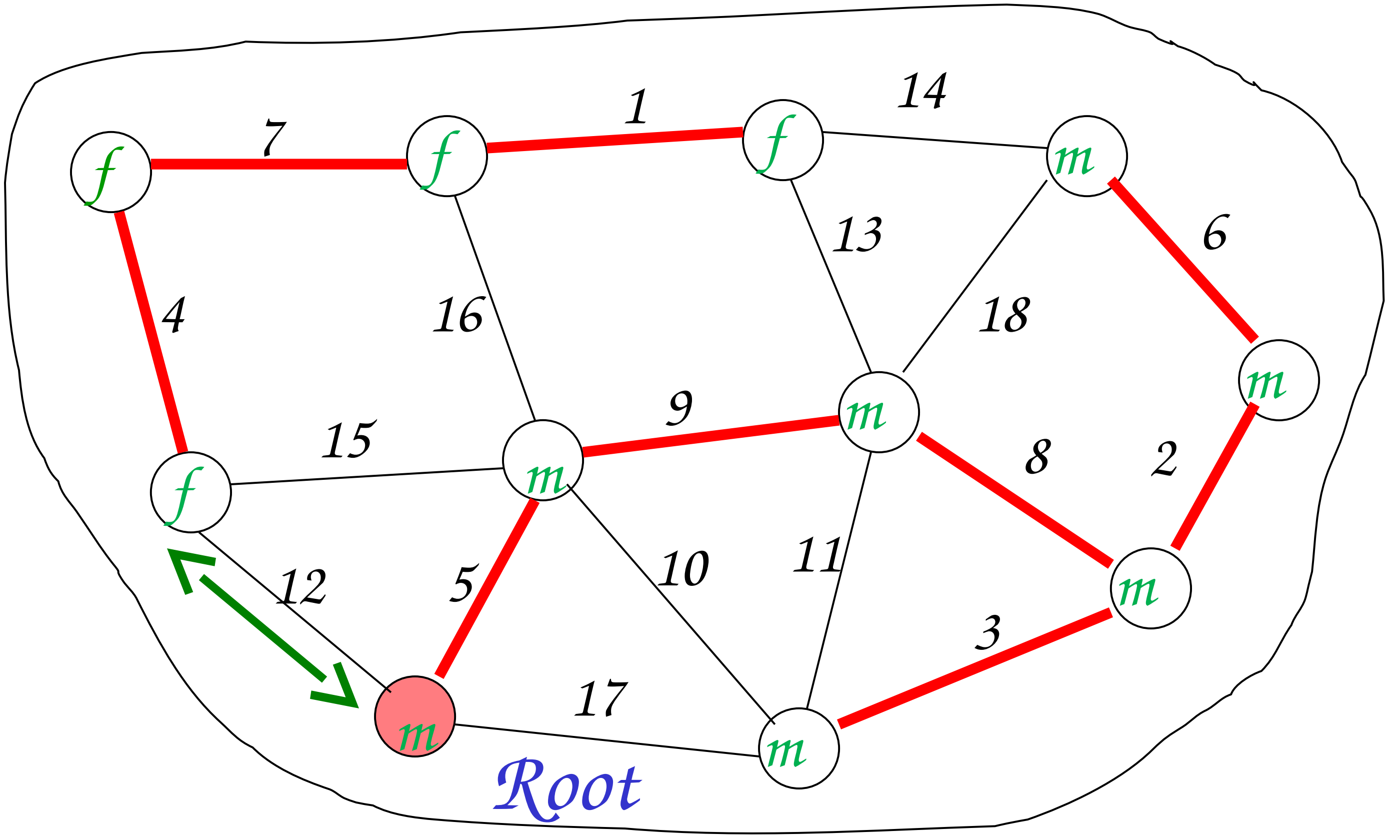


End of phase 2

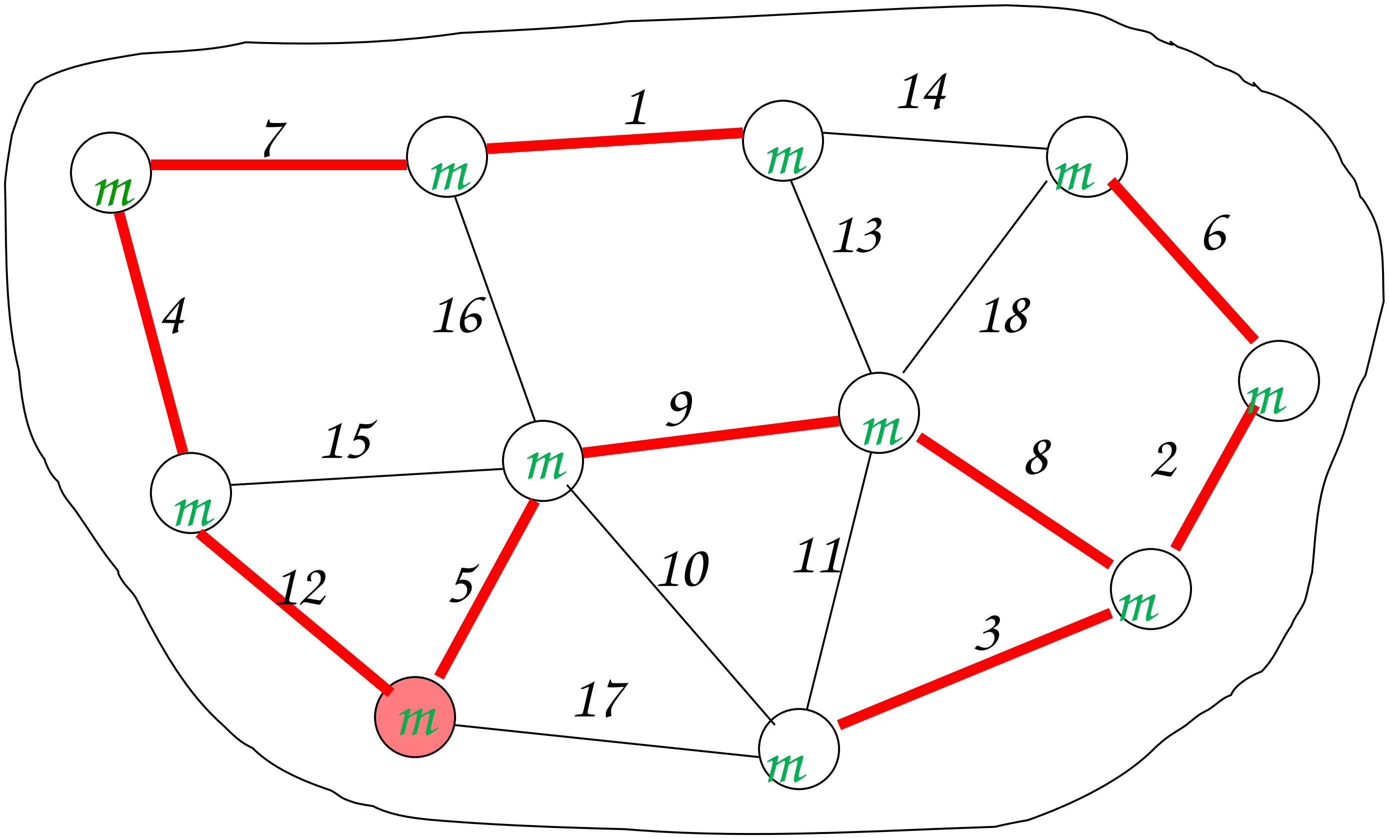
Phase 3: Find the MOE for each fragment



Phase 3: Merge the fragments

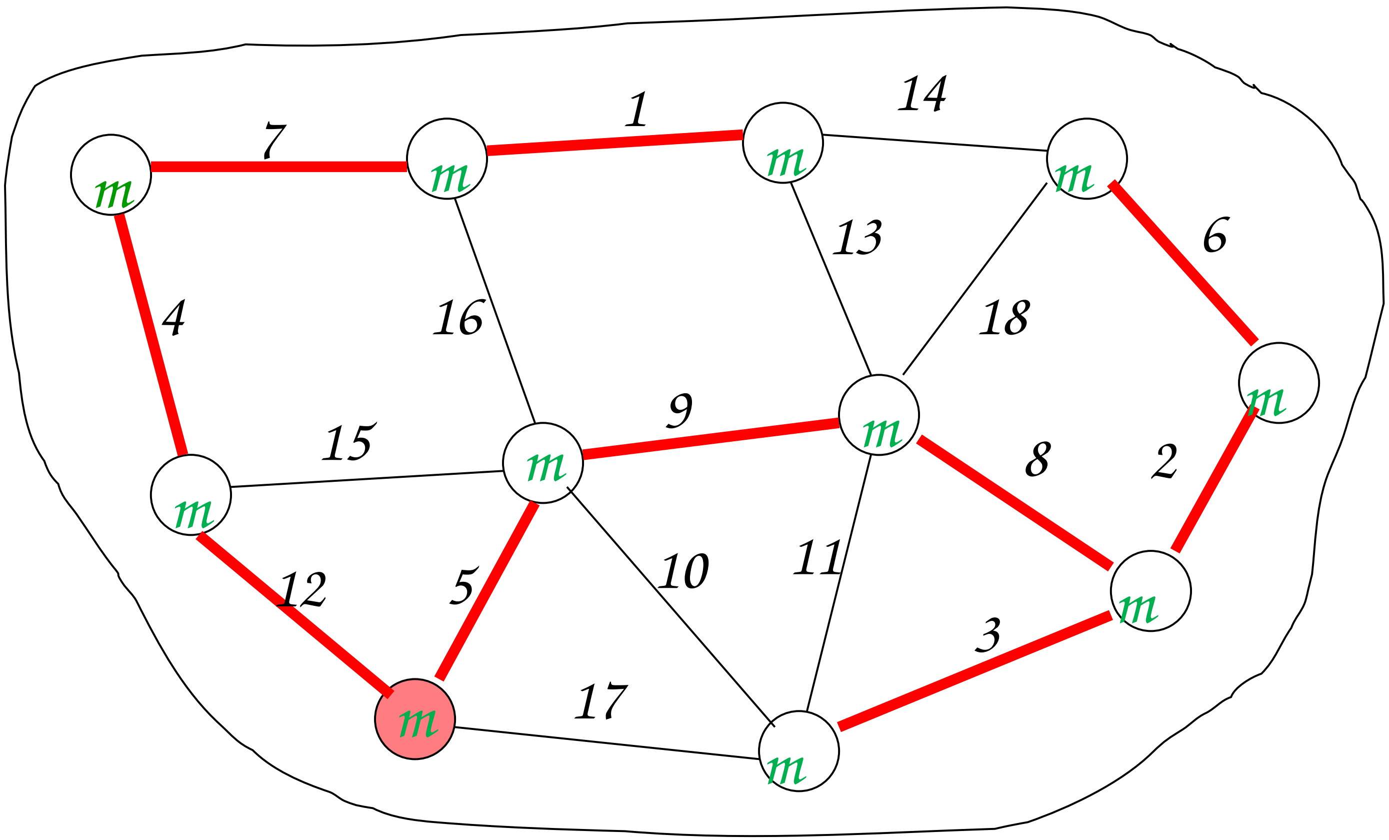


Broadcast the ***new fragment identity*** *to all the nodes in the new fragment, and so at the end of phase 3 each node knows its own new fragment identity.*



End of phase 3

At the beginning of Phase 4, every node will start searching its MOE, but all the edges will be rejected, and so each node will report nil



⇒ the root node realizes this is the FINAL MST

Syncronicity

- *To guarantee correctness, selection of local MOEs must start when all the nodes know their new fragment identity (notice the difference w.r.t. Prim)*
- *But at the beginning of a phase, each fragment can have a different number of nodes, and thus the broadcasting of the new fragment identity can take different times ⇒ fragments and phases need to be "synchronized"*

Phases' synchronization

- *First of all, assume that all the nodes start the **local MOE selection** at the very same round (we will convince ourselves about that on next slide)**;** observe that each node has at most n-1 incident edges, and so the **local MOE selection** requires at most 2(n-1)+1 rounds (convince yourself… notice that each **Test** span 3 rounds to be completed)⇒ we assign exactly 2n-1 rounds to the local MOE selection (this means that if a node discovers a local MOE in less rounds, it will wait in any case till 2n-1 rounds have passed before proceeding to the **Report** step)*

Syncronicity (2)

- *Moreover, each fragment has height (longest root-leaf path, in terms of edges) at most n-1, and so the **Report** activity requires at most n rounds; this means, the root node will find the MOE of the fragment in at most n rounds. Again, it could take less than n round, but it will wait in any case till n rounds have passed before proceeding to the **Merge** step*
- *Similarly, the **Merge** message requires at most n rounds to reach the proper node, and so we assign exactly n rounds to this step, which means that the node which is incident to the MOE will send the **Connect** message **exactly** at round $4n$-1 of a phase*
- *Finally, exactly at round $4n$ a node knows whether it is a new root, and if this is the case it sends a **New_fragment** message which will take at most n rounds to "flood", and so again we assign exactly n rounds to this step*

⇒ *A **fixed** number of **$5n$** total rounds are used for each phase (in some rounds nodes do nothing…)!*

Algorithm Time Complexity (# rounds)

End of phase	Smallest Fragment size (#nodes)
1	2
2	4
$\vdots$	$\vdots$
i	2^i

Algorithm Time Complexity (# rounds)

Maximum possible fragment size $2^i = n$

Number of nodes

$\Rightarrow$ Maximum # phases: $i = \log_2 n$

$\Downarrow$

$O(\log n)$

Total time = Phase time • #phases = $O(n \log n)$

$5n$ rounds, i.e., $\Theta(n)$ rounds

Algorithm Message Complexity

Thr: Synchronous GHS requires $O(m+n \log n)$ msgs.

Proof: We have the following messages:

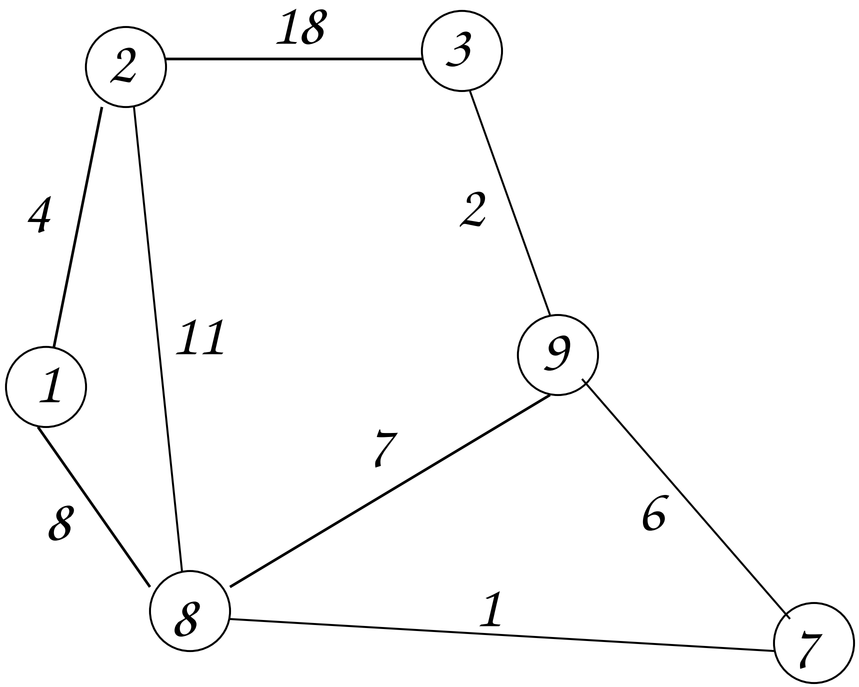
1. **Test-Reject** msgs: at most 2 for each edge, namely $O(m)$ messages of this type.
2. In each phase, each node:
   - sends at most **a single Report, Merge, Connect** message;
   - receives at most **a single New_Fragment** message;
   - sends and then receives at most **a single Test** followed by an **Accept;**

which means that globally circulate $O(n)$ messages in each phase (and in particular, on the branch edges of a fragment, circulate a constant number of messages in each phase). Since we have at most $\log n$ phases, the claim follows.

END OF PROOF

Homework

1. Execute synchronous GHS on the following graph:



2. What is a best-possible execution of synchronous GHS? (Provide a class of instances on which the number of rounds and the number of messages is asymptotically minimum)

Asynchronous Version of GHS Algorithm

- Simulates the synchronous version, but it is even stronger: Works with uniform models and asynchronous start (but still requires non-anonymity and distinct edge weights)
- As before, we have fragments which are coordinated by a root node, and which are merged together through their MOEs
- However, we have two types of merges now, depending on the "size" of the merging fragments, as we will describe soon: absorption and join

Local description of asynchronous GHS

A node stores the same information as in the synchronous case, but now:

1. A fragment (i.e., each node in it) is identified by a pair:

(fragment identity (id), level)

where the fragment identity is again the ID of some node in the fragment, and the level is a non-negative (and monotonically increasing) integer; at the beginning, each node is a fragment with identity (node ID, 0); during the execution, the fragment identity changes and the level increases as a consequence of absorptions and joins

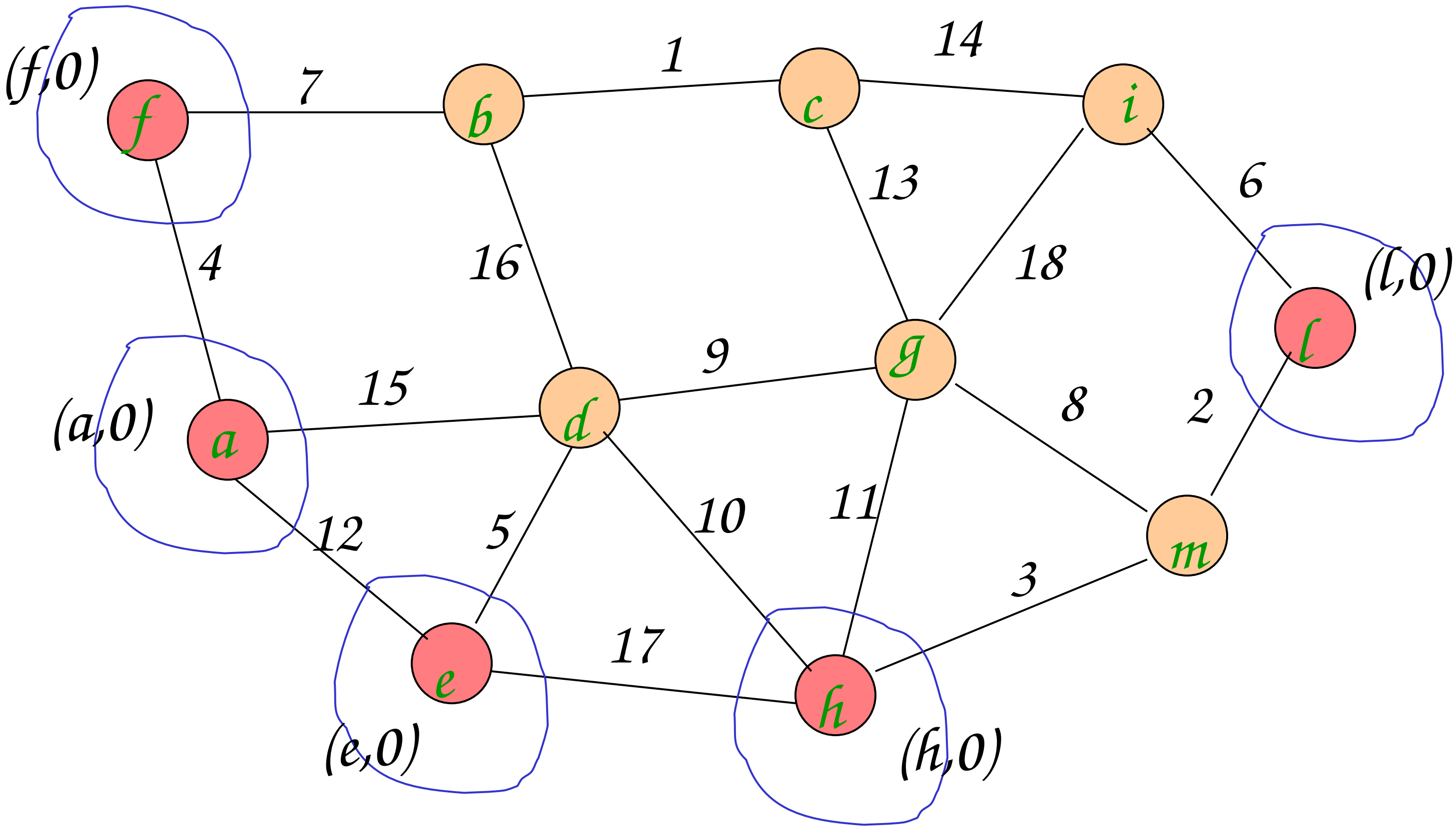
2. A node has also a **status** which describes what is currently doing w.r.t. the search of the MOE of its appended subfragment, and it can be either of {sleeping, finding, found}

Type of messages of asynchronous GHS

Similar to the synchronous case, but now:

1. ***New_fragment(id, level, node status):*** *coordination message flooding in the fragment just after a merge; this is originated by a node onto which the merge was taking place*
2. ***Test(id,level):*** *to test a basic edge (in increasing order of weight); when a node is testing an edge, it must be in a finding state, and a **Test** message is replied (Accept/Reject) if and only if the tested node has a **not smaller** level, otherwise it is freezed*
3. ***Report(weight):*** *immediately after reporting the MOE of the appended subfragment, a node put itself in a found state*
4. ***Connect(id,level):*** *to perform the merge; due to the above constraint on the **Test** message, it follows that this message will only travel from a fragment of level L to a fragment of level $L' \geq L$*

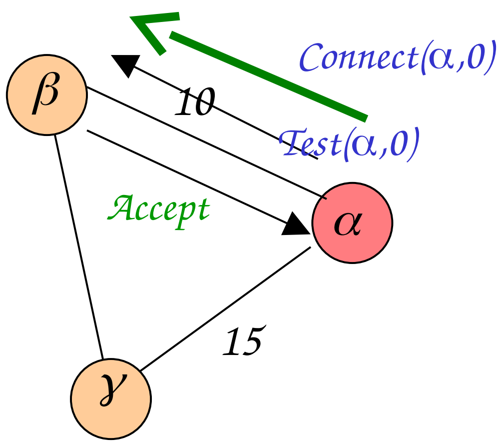
Example: *Initially, some nodes are awake and form a fragment, while some other are sleeping (these will wake-up either spontaneously or after receiving a message)*



Every non-sleeping node is the root of its fragment, and the fragment identity is the pair (node ID,0) (reported nearby the nodes), and so non-anonimity is required

Actions taken by a node of level 0

- Similarly to the synchronous case, each awake node will start searching its own MOE, by sending a **Test** message containing its fragment identity (node ID, 0) over its basic edge of minimum weight, and it will certainly receives an **Accept**, since on the other side of the edge there must be a node with a different fragment identity, and of **level ≥ 0** (i.e., not smaller than that of the testing node)



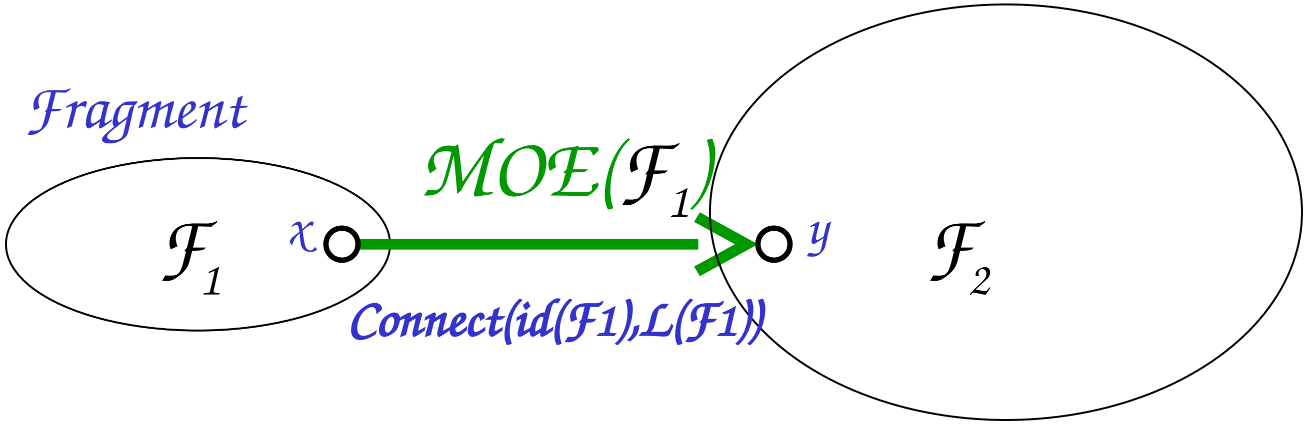
- Then, it will send on such an edge a **Connect**(node ID, 0) message, and depending on the fragment identity of the other end-node, some kind of merge will take place, as we will see soon.

Actions taken by a node of level d>0

- *Similarly to the synchronous case, nodes of a fragment are coordinated by a root node, and each of them will search its local incident MOE, by sending a **Test(id,level)** over its basic edges in increasing order of weight, until it will receive an **Accept**.*
- *If the level of the node receiving a **Test** message is smaller than that of the querying node, then as we said the reply **will be delayed**; however, once that a node receives an **Accept** (or once that all its incident basic edges have been rejected), it will wait for the **Report** messages of its children, and will then send its **Report** message towards the root.*
- *Once the root has received all the **Report** messages, it will select the MOE of the fragment, and will send a **Merge** message along the proper way, which will become a **Connect(id,level)** message at the proper node.*
- *Once again, depending on the fragment identity of the other end-node, some kind of merge will take place, as we will see on the next slide.*

Merge of two fragments

*Merges are generated by **Connect** messages:*



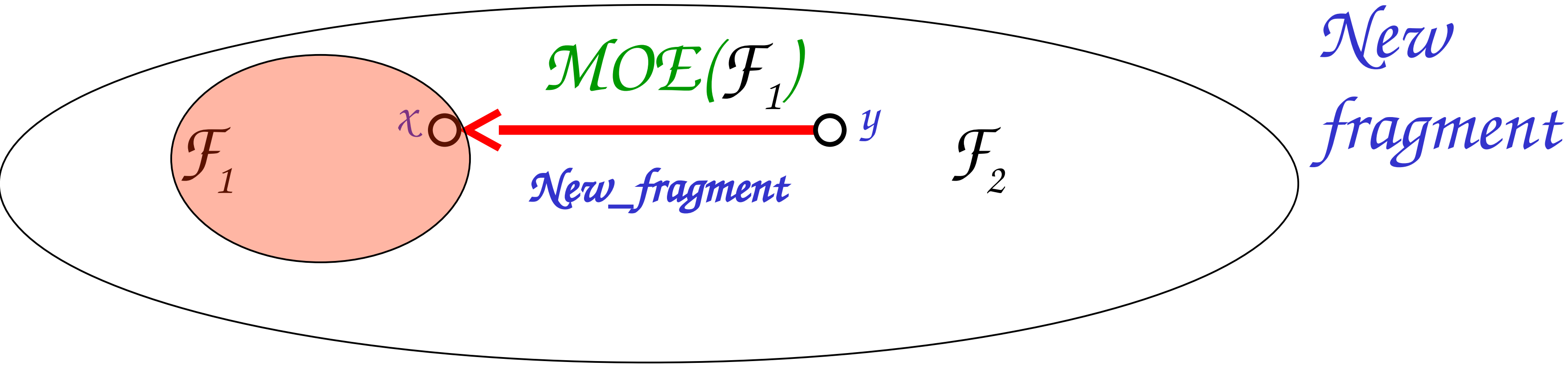
1. *If $L(\mathcal{F}_1)<L(\mathcal{F}_2)$, then $\mathcal{F}_2$ absorbs $\mathcal{F}_1$*

2. *If $L(\mathcal{F}_1)=L(\mathcal{F}_2)$ and (x,y) is also the MOE of $\mathcal{F}_2$, then $\mathcal{F}_1$ and $\mathcal{F}_2$ will join (once that $\mathcal{F}_2$ will send a **Connect** to $\mathcal{F}_1$ on the same edge), otherwise $\mathcal{F}_2$ will "freeze" the message (and later on it will absorb $\mathcal{F}_1$)*

3. *$L(\mathcal{F}_1)>L(\mathcal{F}_2)$ is instead impossible, since as we said before, node y in this case would not have replied to a previous **Test** on edge (x,y)*

Case 1: $L(\mathcal{F}_1) < L(\mathcal{F}_2) \Rightarrow$ *merge as an* ***Absorption***

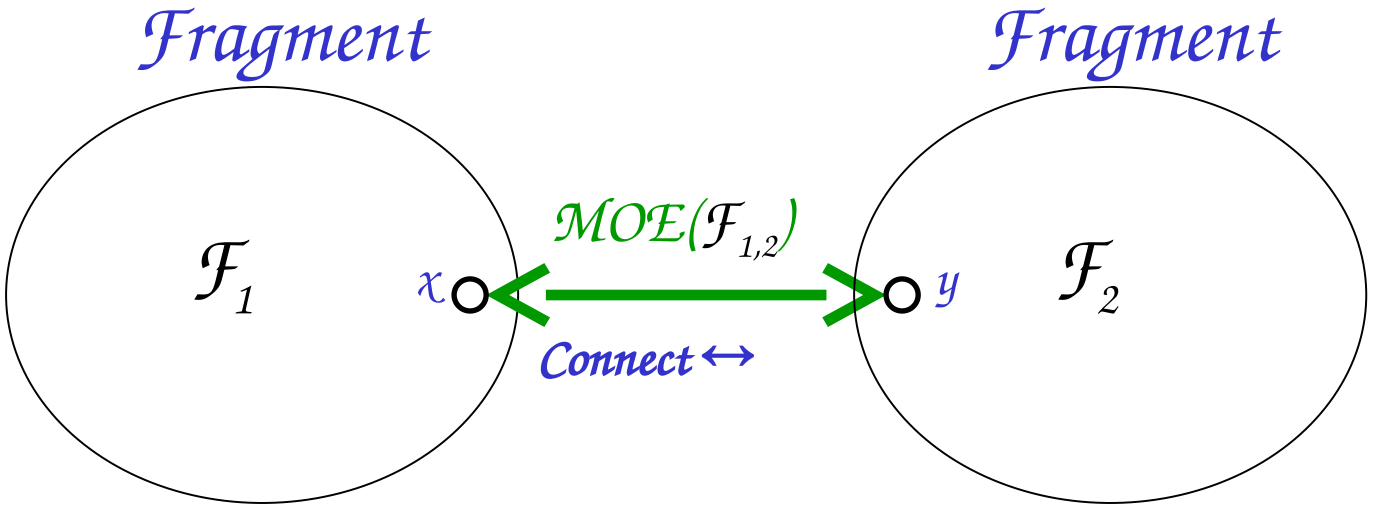
In this case, a "new" fragment is created with the *same identity* *as* $\mathcal{F}_2$



A ***New_fragment(ID($\mathcal{F}_2$),L($\mathcal{F}_2$),status(y))*** *message is* *broadcasted* *to nodes of* $\mathcal{F}_1$ *by the node y of* $\mathcal{F}_2$ *on which the merge took place*

(cost of merging, in terms of number of messages, proportional to the size of $\mathcal{F}_1$*)*

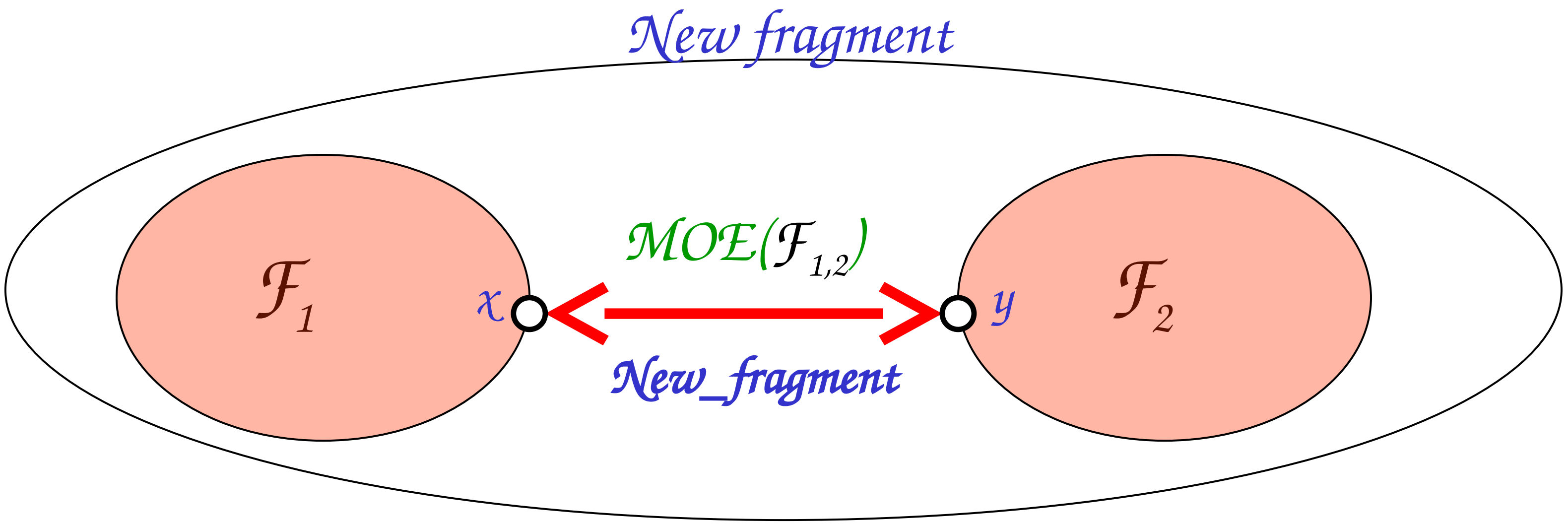
Case 2.1: $L(\mathcal{F}_1)=L(\mathcal{F}_2)$ and (x,y) is also the MOE of $\mathcal{F}_2 \Rightarrow$ merge as a ***Join***



A symmetric MOE will generate a bidirectional **Connect** message on the same edge, and in this case, $\mathcal{F}_1$ joins with $\mathcal{F}_2$. This can happen iff $L(\mathcal{F}_1)=L(\mathcal{F}_2)$, as otherwise either $\mathcal{F}_1$ or $\mathcal{F}_2$ would be locked in a **Test**

Notice that the system is asynchronous, and so the **Connect** message on the two directions may be not simultaneous (differently from sync GHS)

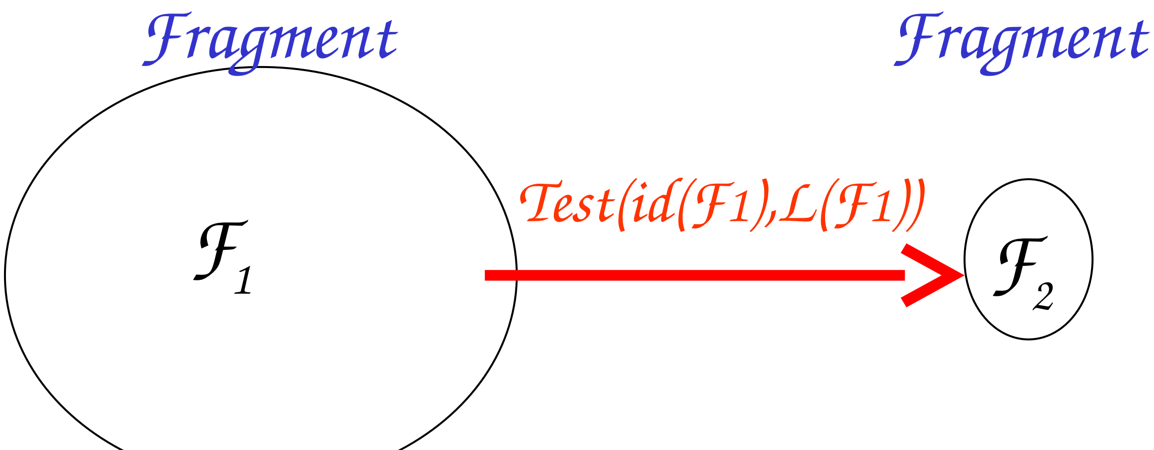
*Merge as a **Join**: the combined level is* $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}_{1,2}) + 1$



*and a **New_fragment(max(ID($\mathcal{F}_1$),ID($\mathcal{F}_2$)),$\mathcal{L}(\mathcal{F}_{1,2})+1$,finding)** message is broadcasted to all nodes of $\mathcal{F}_1$ and $\mathcal{F}_2$ by the new root, i.e., the node with max ID field in the fragment identity between x and y*

(cost of merging, in terms of number of messages, is proportional to the size of $\mathcal{F}_1$ and $\mathcal{F}_2$)

Remark: *a **Connect** message cannot travel from a fragment of **higher** level to a fragment of **lower** level (actually, this is for message-complexity efficiency reasons, since absorption has a cost proportional to the size of the absorbed fragment, as we mentioned before). Indeed, recall that a **Test** message from a fragment* $\mathcal{F}_1$ *to a fragment* $\mathcal{F}_2$ *is replied only once that* $\mathcal{L}(\mathcal{F}_1) \leq \mathcal{L}(\mathcal{F}_2)$ *(this prevents* $\mathcal{F}_1$ *to find its MOE, i.e., to ask a connection to* $\mathcal{F}_2$*, while* $\mathcal{L}(\mathcal{F}_1) > \mathcal{L}(\mathcal{F}_2)$*)*



Correctness of asynchronous GHS

Full proof is quite complicated. It must address the following general properties:

1. Termination: Response to ***Test*** are sometimes delayed $\Rightarrow$ deadlock is a priori possible!

2. Asynchronicity: Message transmission time is ***unbounded*** $\Rightarrow$ inaccurate information in a node about its own fragment is a priori possible! Replies to ***Test*** messages are really correct?

3. Absorption while the absorbing fragment is searching for a MOE: in this case, new nodes are added to the fragment, and they are dynamically involved in the on-going MOE searching process. Is that feasible?

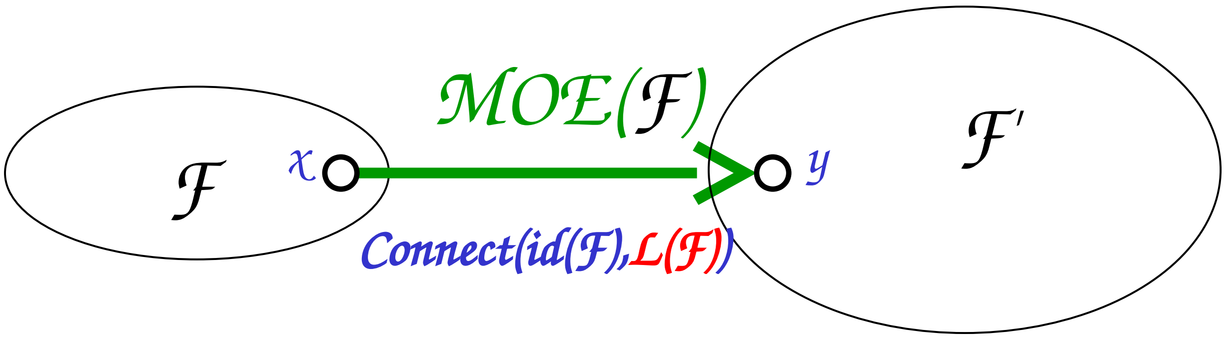
We will show formally only termination, while we only sketch the proof for point 2 and 3

1. Termination (1/2)

Lemma: *From any configuration with at least 2 fragments, eventually either absorption or join takes place.*

Proof: *Let L be the minimum level in this configuration, and let F be the (not necessarily unique) L-level fragment having the lightest MOE. Then, any* ***Test*** *message sent by F either reaches a fragment F' of level L' ≥ L or a sleeping node. In the first case, F gets a reply immediately, while in the second case the sleeping node awakes and becomes a fragment of level 0 ⇒ this creates a new configuration, onto which the argument of the proof is applied recursively ⇒ eventually, we get a configuration in which there are no sleeping nodes, where only the first case applies. This means that F will get all the needed replies, and then it will find its MOE, over which a* ***Connect*** *message will be routed. Two cases are possible:*

1. Termination (2/2)



1. *L(F')>L(F): in this case F' **absorbs** F;*
2. *L(F')=L(F): in this case, since (x,y) is a lightest MOE, then it is also the MOE of F' (recall that edge weights are distinct) and F' cannot be locked (similarly to F); then, a **join** between F and F' takes place.*

END OF PROOF

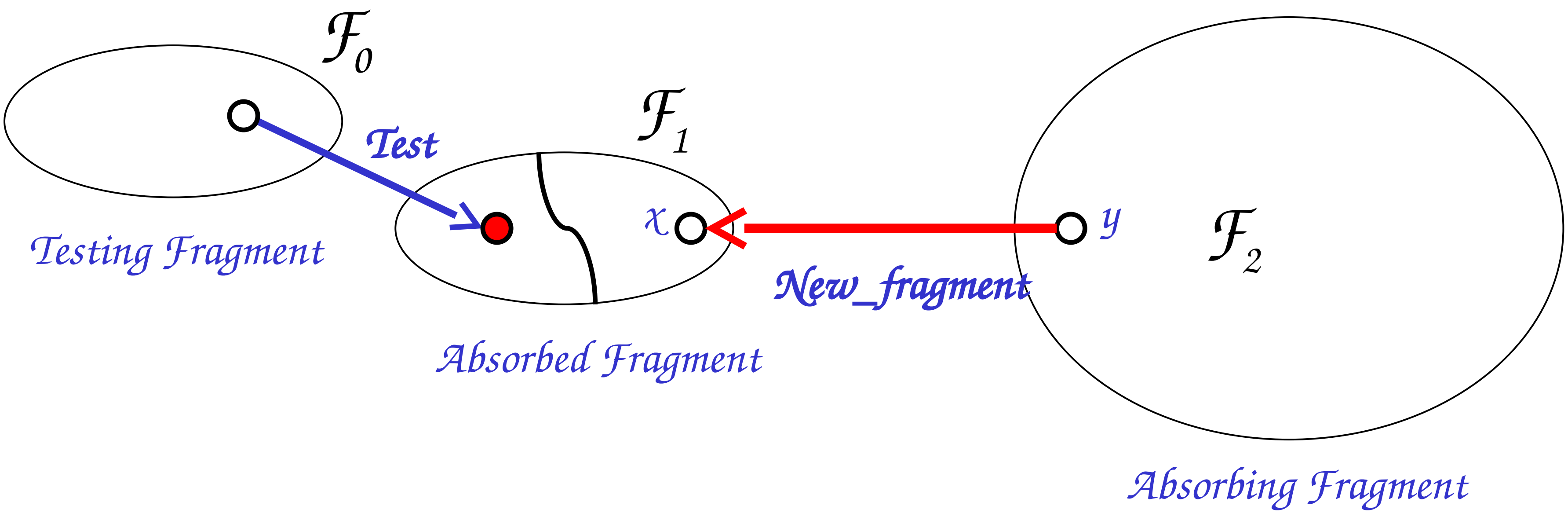
***Corollary:** Asynchronous GHS terminates.*

***Proof:** By contradiction, if not then there must be at least two fragments left; but then the above lemma guarantees their number will be progressively reduced to 1.*

END OF PROOF

2. Asynchronicity (1/3)

Message transmission time is unbounded $\Rightarrow$ a node might have inaccurate info about its status! Let us see an example in which the red node of $\mathcal{F}_1$ is tested, but its status is inaccurate since it did not yet received the new fragment identity after that $\mathcal{F}_1$ was absorbed by $\mathcal{F}_2$



We will show that an answer (Accept/Reject) given having inaccurate information will not affect the correctness of the algorithm!

2. Asynchronicity (2/3)

Claim 1: *A node p_i whose fragment identity is currently (id,L) actually belongs to a fragment of level $L' \geq L$.*

Proof: *If the identity of p_i is accurate, then $L'=L$, while if it is inaccurate, then p_i is participating in either a join or an absorption. But in both cases, $L'>L$.*

QED

Remark 1: *If a node p_i of a fragment $\mathcal{F}$ sends a test to a node p_j of a fragment $\mathcal{F}'$, then the fragment $\mathcal{F}$ is not involved in a merge, and so the only inaccurate info might be at p_j.*

Remark 2: *Reject messages are always correct.*

⇒ *Only an Accept message may be wrong, but we will see this is not the case.*

2. Asynchronicity (3/3)

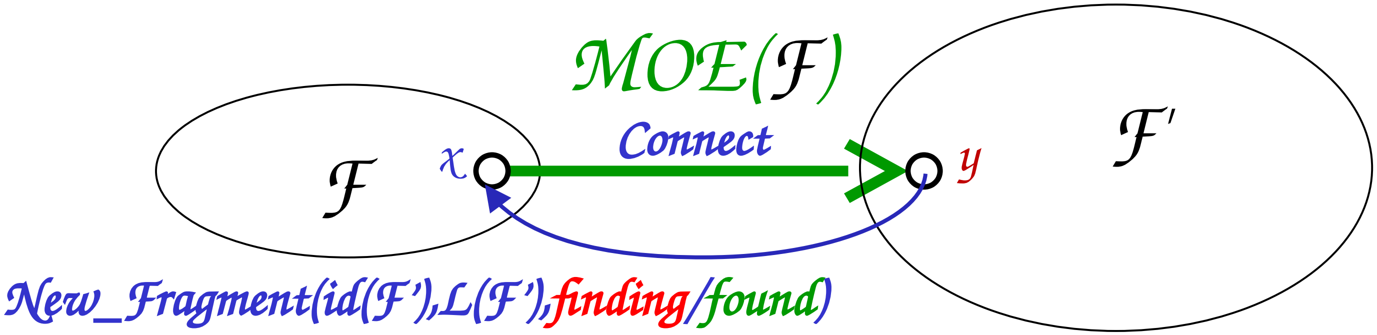
Claim 2: *If a node p_i of a fragment $\mathcal{F}_1=(id_1,\mathcal{L}_1)$ sends a test to a node p_j of a fragment $\mathcal{F}_2=(id_2,\mathcal{L}_2)$ and p_j accepts, then p_i and p_j are not in the same fragment.*

Proof: *Notice that by definition, p_j accepts iff $(id_2,\mathcal{L}_2)\neq(id_1,\mathcal{L}_1)$ and $\mathcal{L}_2\geq\mathcal{L}_1$. We then have two cases:*

1. *$\mathcal{L}_2>\mathcal{L}_1$: by Claim 1, the real level of the fragment to which p_j belongs is $\mathcal{L}'\geq\mathcal{L}_2>\mathcal{L}_1$, and so it follows that p_i and p_j are not in the same fragment (remember that by Remark 1, information holds by p_i are accurate).*
2. *$\mathcal{L}_2=\mathcal{L}_1$: again by Claim 1, the real level of the fragment to which p_j belongs is $\mathcal{L}'\geq\mathcal{L}_2=\mathcal{L}_1$, and so:*
   a) *If $\mathcal{L}'=\mathcal{L}_2=\mathcal{L}_1$, then it must be $id_2\neq id_1$, and so it follows that p_i and p_j are not in the same fragment;*
   b) *If $\mathcal{L}'>\mathcal{L}_2=\mathcal{L}_1$, i.e., $\mathcal{L}'>\mathcal{L}_1$, then see above.* ***QED***

$\Rightarrow$ *Accept messages are always correct as well!*

3. Absorption while $\mathcal{F}'$ is searching for a MOE



1. *Transmitted status is finding: in this case, nodes in $\mathcal{F}$ start searching for their local MOE, and node y will wait a **Report** from x before reporting to its parent in $\mathcal{F}'$. Apparently, it is possible that a node u in $\mathcal{F}$, after getting the new identity, tests a node v in $\mathcal{F}$ which is still not updated, and so v could wrongly reply accept. But this is **impossible**, since the level of v is less than the level of u, due to the absorption, and so v does not reply to u. (Notice the very same argument can be applied also when a **Join** takes place)*
2. *Transmitted status is found: in this case, nodes in $\mathcal{F}$ do not participate to the selection of the MOE for $\mathcal{F} \cup \mathcal{F}'$, and then it seems that edges outgoing from $\mathcal{F}$ are omitted. However, observe that y has already found the MOE of the appended subfragment, and since y is adjacent to (x,y), y must have at least another incident basic edge (y,u) s.t. w(y,u) < w(x,y), since otherwise y would be locked!. Hence, since any edge outgoing from $\mathcal{F}$ will be heavier than (x,y), no any of them can be the MOE of $\mathcal{F}'$, and so correctness is guaranteed.*

END OF PROOF

Algorithm Message Complexity

Lemma: A fragment of level L contains at least 2^L nodes.

Proof: By induction. For L=0 it is trivial. Assume it is true up to L=k-1, and let F be of level k>0. But then, either:

1. F was obtained by joining two fragments of level k-1, each containing at least 2^{k-1} nodes by inductive hypothesis ⇒ F contains at least $2^{k-1} + 2^{k-1} = 2^k$ nodes;
2. F was obtained after absorbing another fragment F' of level < k ⇒ apply recursively to F\F', until case (1) applies (observe that we have to arrive to a fragment generated by a **Join**, since k>0).

END OF PROOF

⇒ The maximum level of a fragment is log n

Algorithm Message Complexity (2)

Thr: Asynchronous GHS requires $O(m+n \log n)$ msgs.

Proof: We have the following messages:

1. **Connect**: at most 2 for each edge of the final MST, namely $\Theta(n)$ messages of this type;
2. **Test-Reject**: at most 2 for each edge, namely $O(m)$ messages of this type;
3. Each time the level of its fragment **increases**, a node receives at most **a single New_Fragment** message, sends at most **a single Merge, Report** message, and finally sends and then receives at most **a single Test** message followed by an **Accept**;

and since from previous lemma each node can change at most $\log n$ levels, it means that each of the n nodes generates $O(\log n)$ messages of type 3, and the claim follows.

END OF PROOF

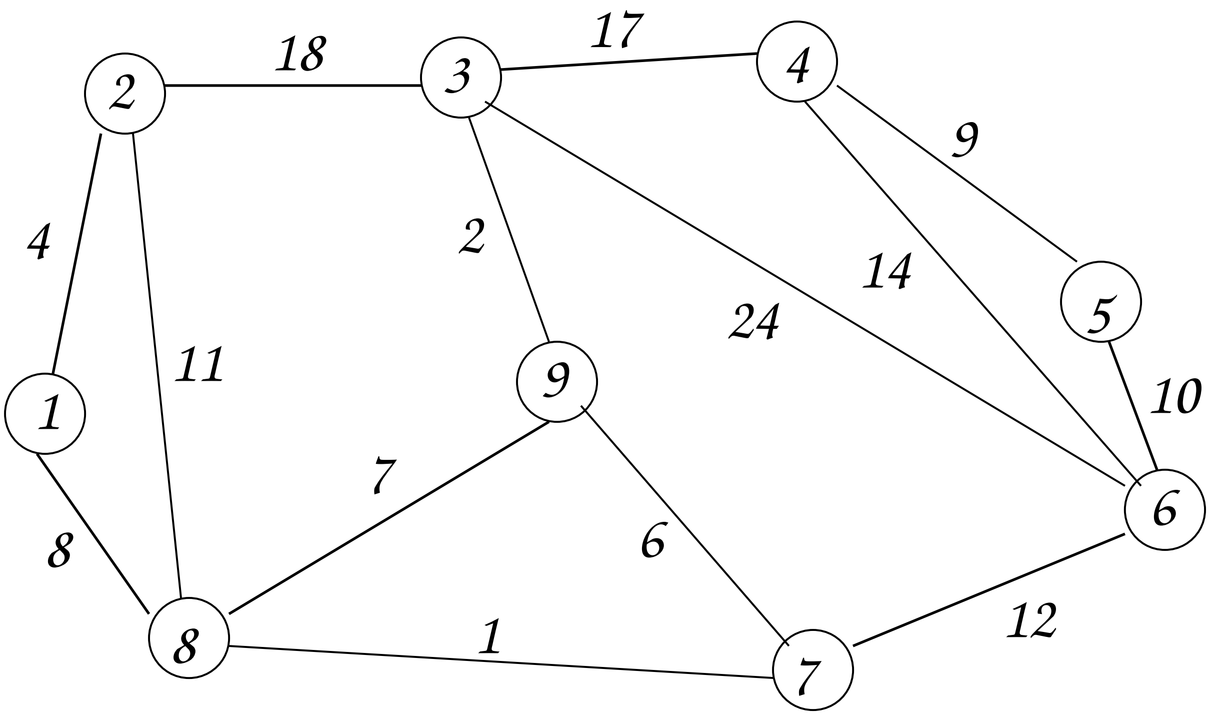
Summary of results for distributed MST

There exist algorithms only when nodes have unique ids:

- Distributed Prim (non-distinct weights):
  - Asynchronous (uniform): $O(n^2)$ messages
  - Synchronous (uniform): $O(n^2)$ messages, and $O(n^2)$ rounds
- Distributed Kruskal (GHS) (distinct weights):
  - Synchronous (non-uniform): $O(m+n \log n)$ messages, and $O(n \log n)$ rounds
  - Asynchronous (uniform): $O(m+n \log n)$ messages

Homework

Execute asynchronous GHS on the following graph:



assuming that system is pseudosynchronous: Start from 1 and 5, and messages sent from odd (resp., even) nodes are read after 1 (resp., 2) round(s)