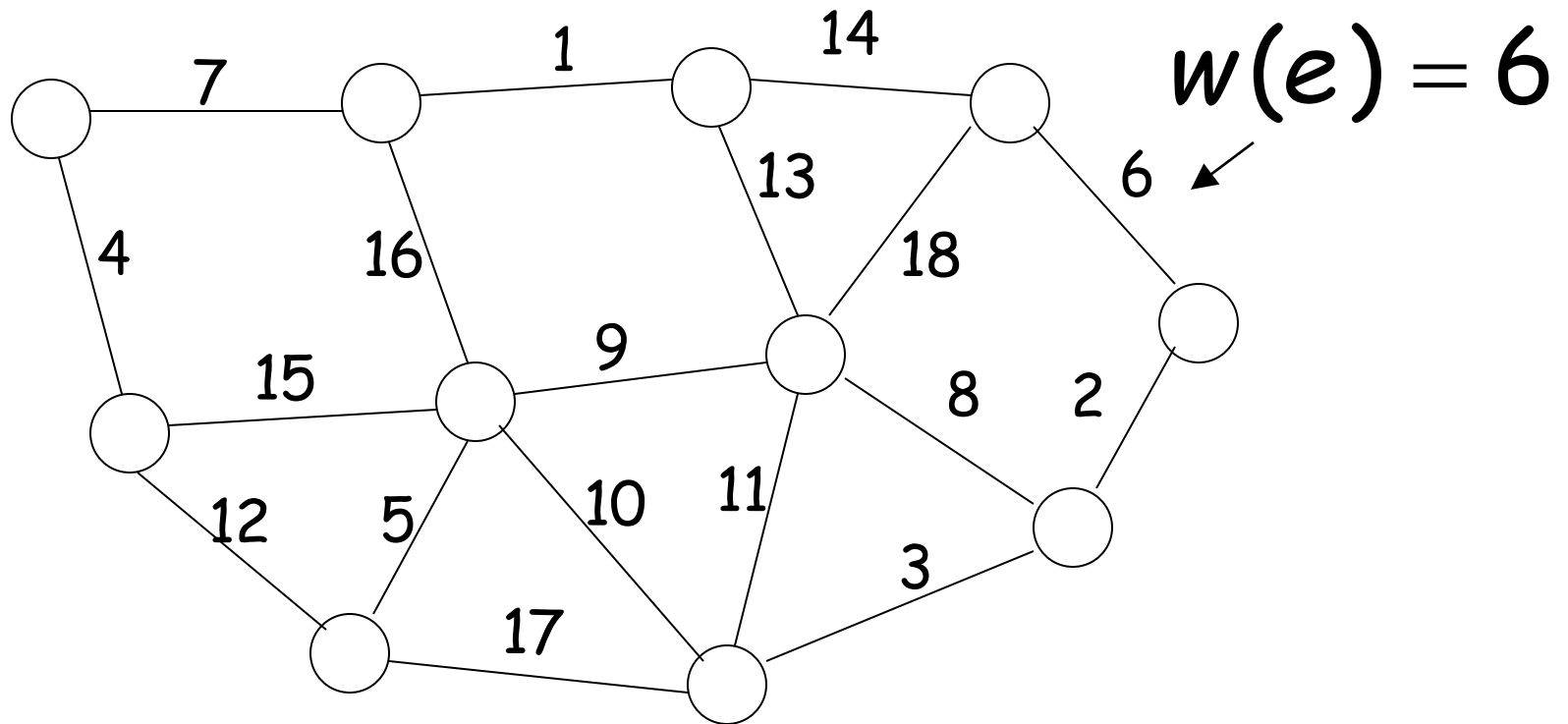


The Minimum Spanning Tree Problem

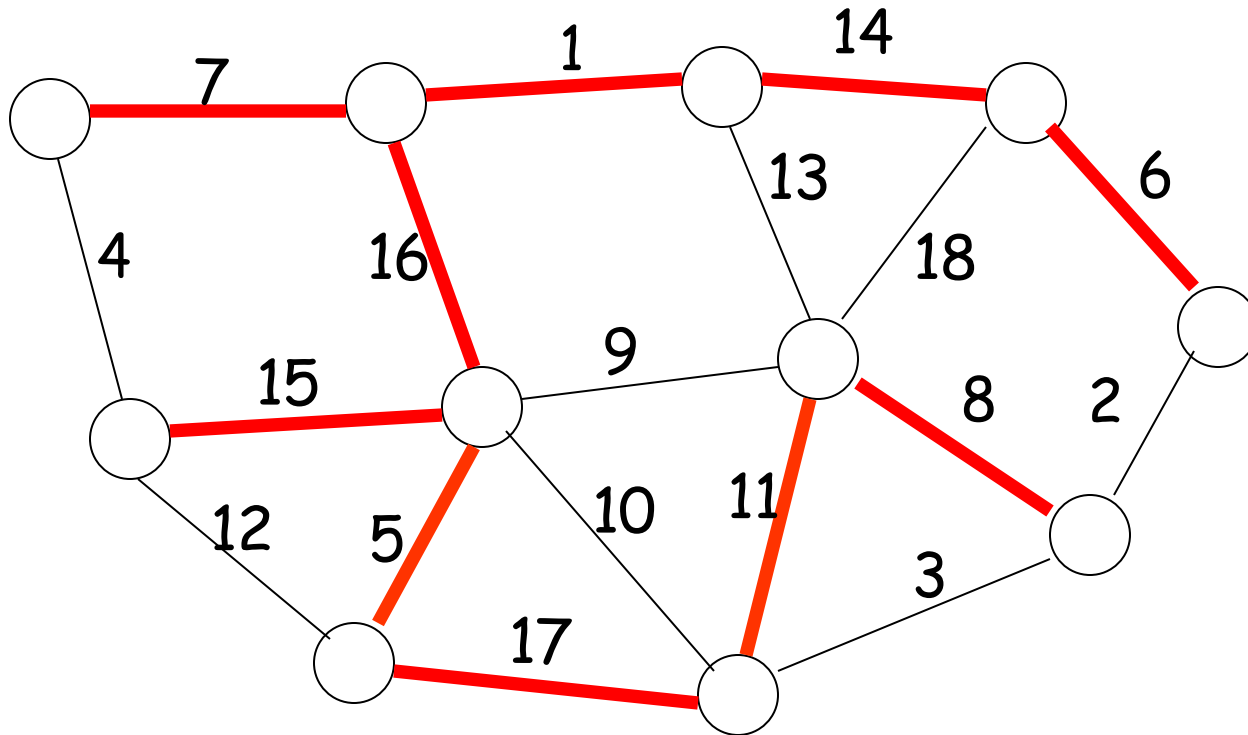
Distributing Prim's and Kruskal's
Algorithm

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



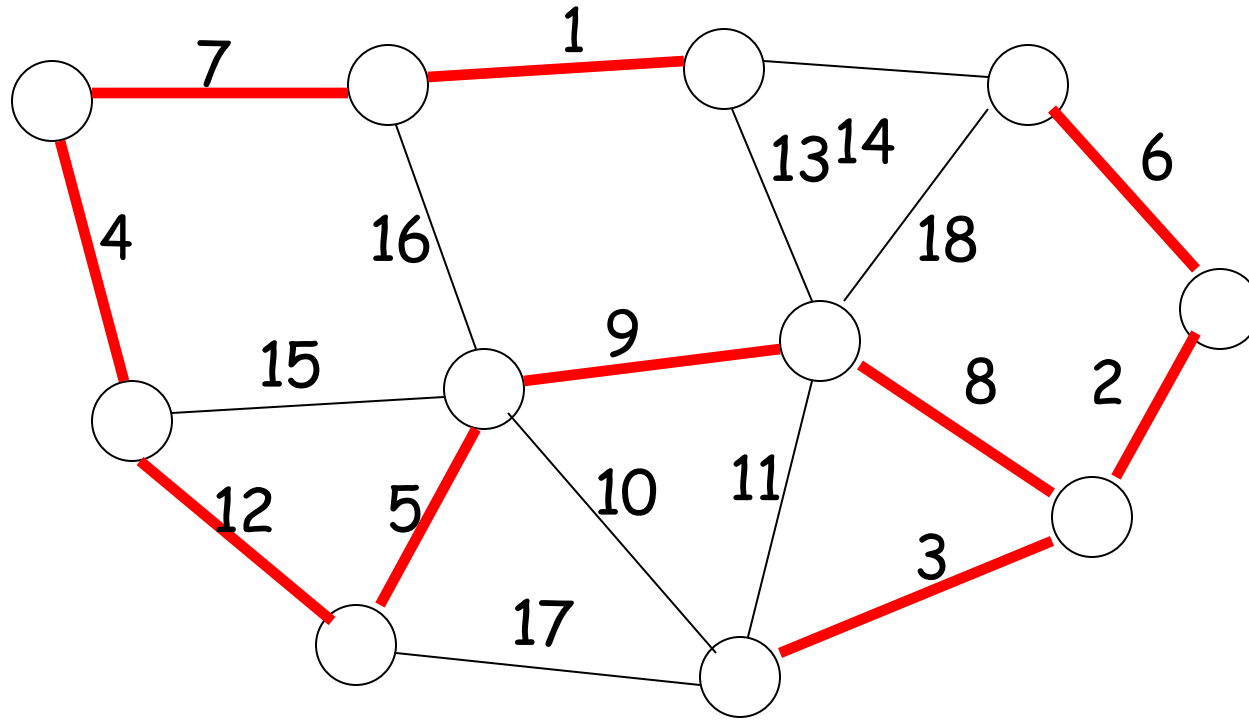
For the sake of simplicity, we assume that weights are positive integers

Spanning tree



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

Minimum-weight spanning tree (MST)

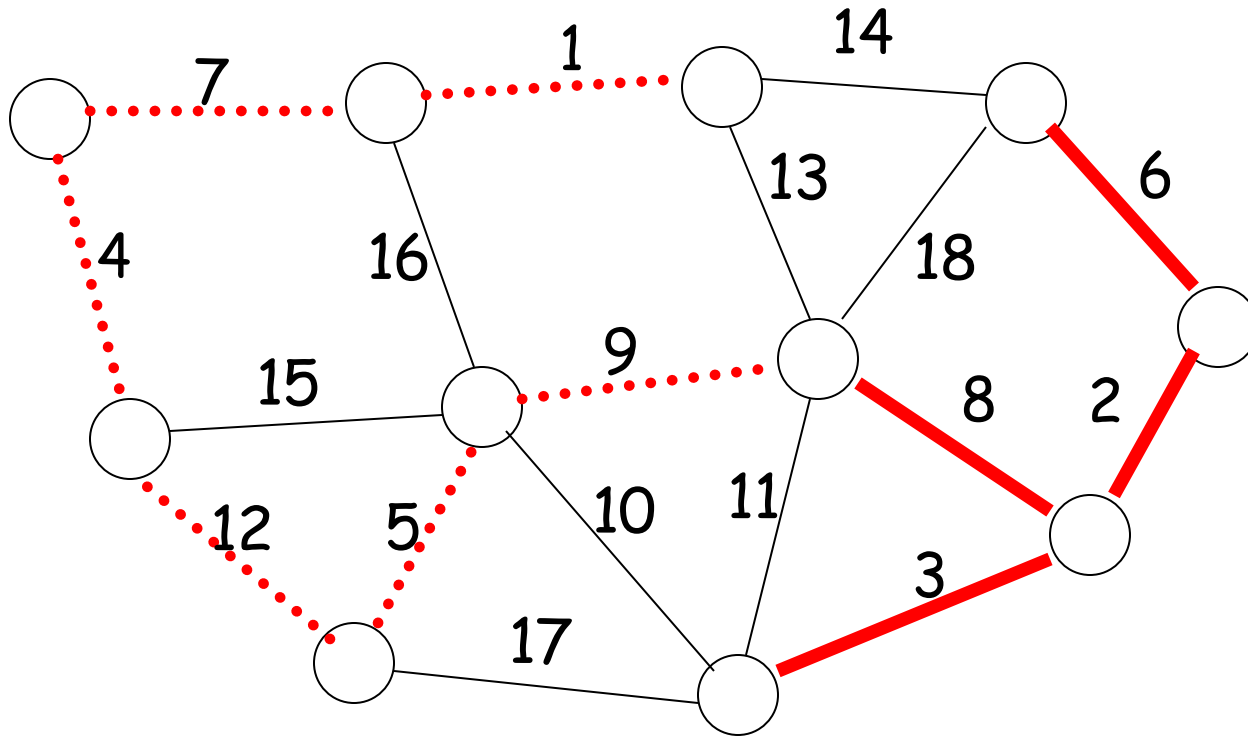


A spanning tree s.t. the sum of its weights is minimized:

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

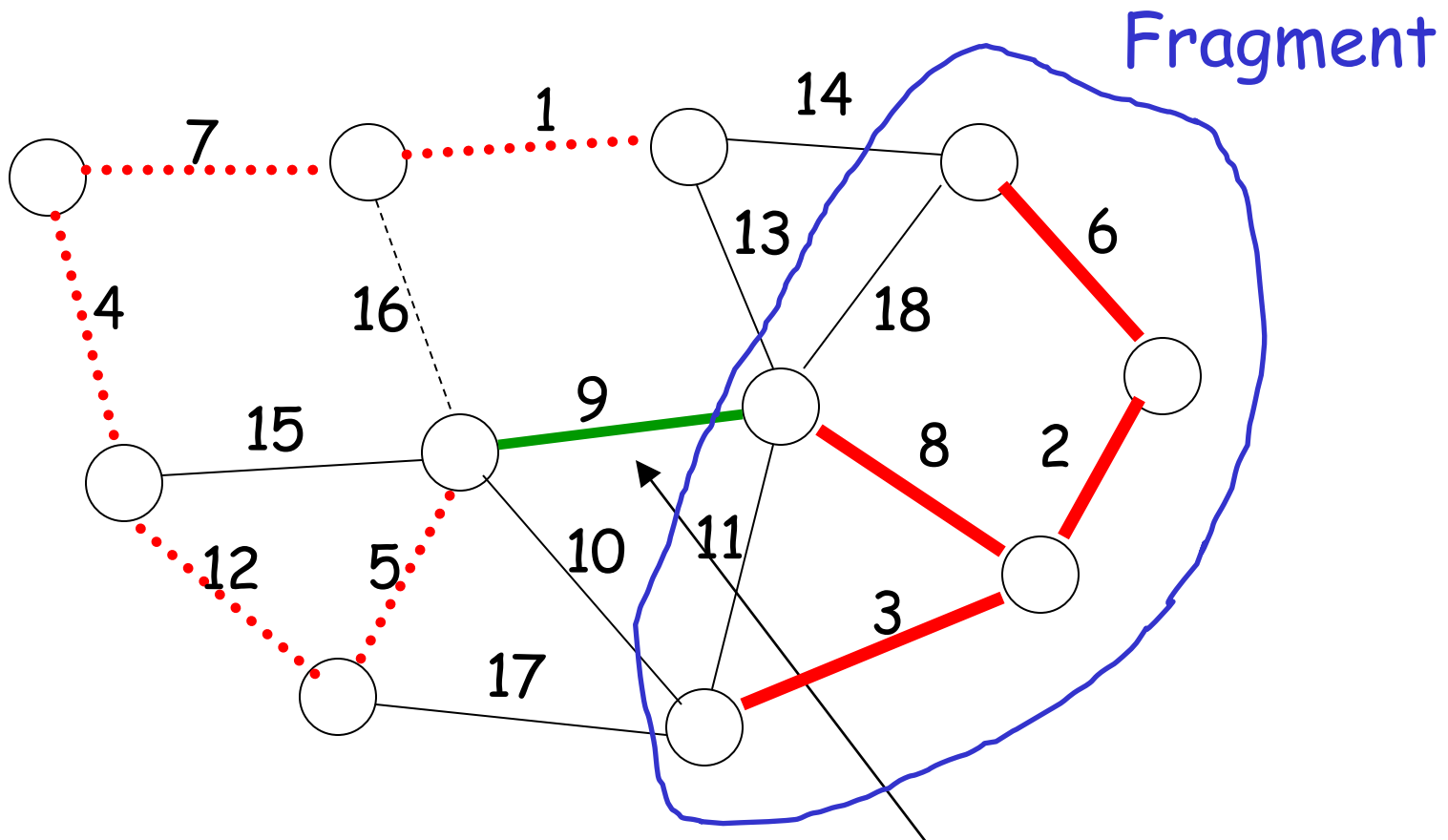
In general, the MST is **not unique**.

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

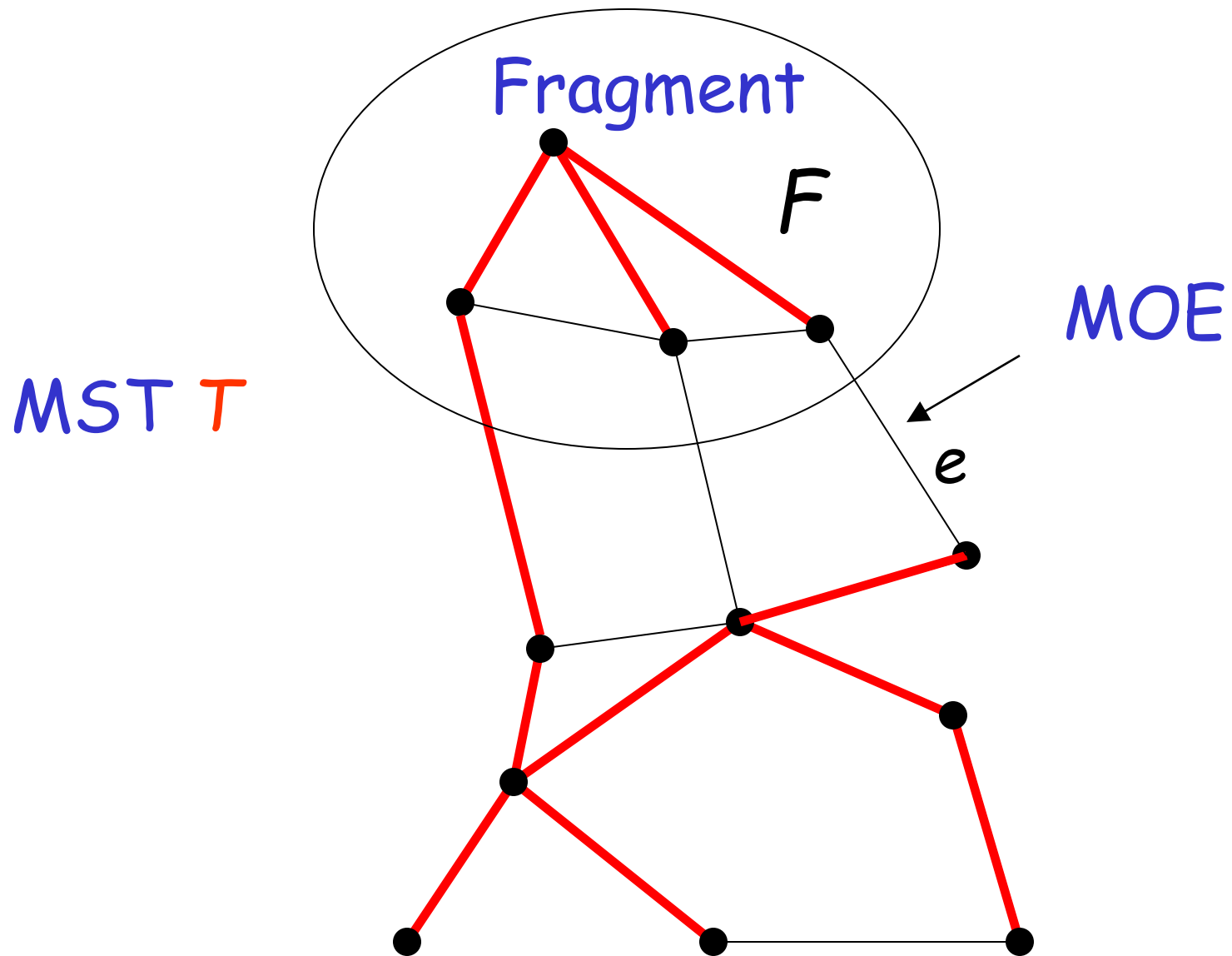
Property 1: The union of a MST fragment and any of its MOE is a fragment of some MST (so called **blue rule**).

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

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

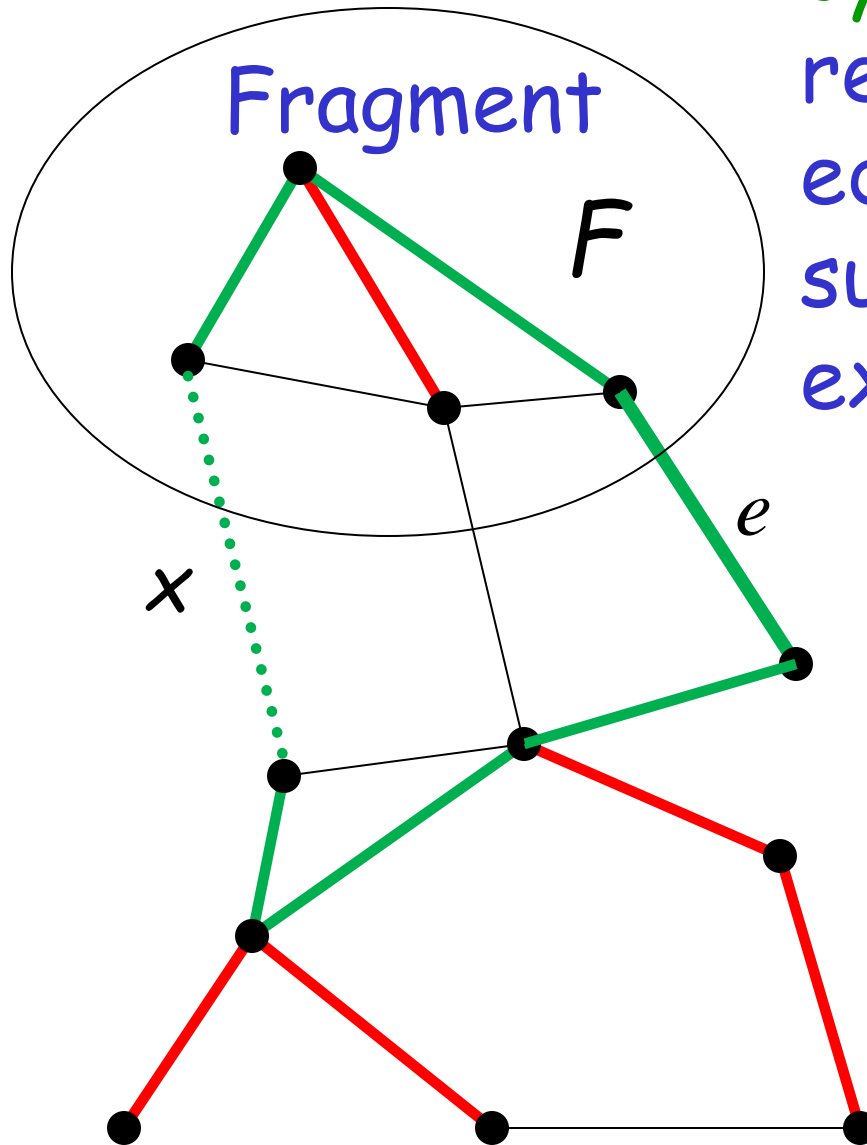
Proof: Remind that in general the MST is **not unique**. Then, for the sake of contradiction, assume that a MOE of a fragment F of a MST T , say e , does not belong to any MST. In particular, it does not belong to T .

$e \notin T$



Then add e to T (thus forming a cycle) and remove x (any edge of T in such a cycle exiting from F)

MST T



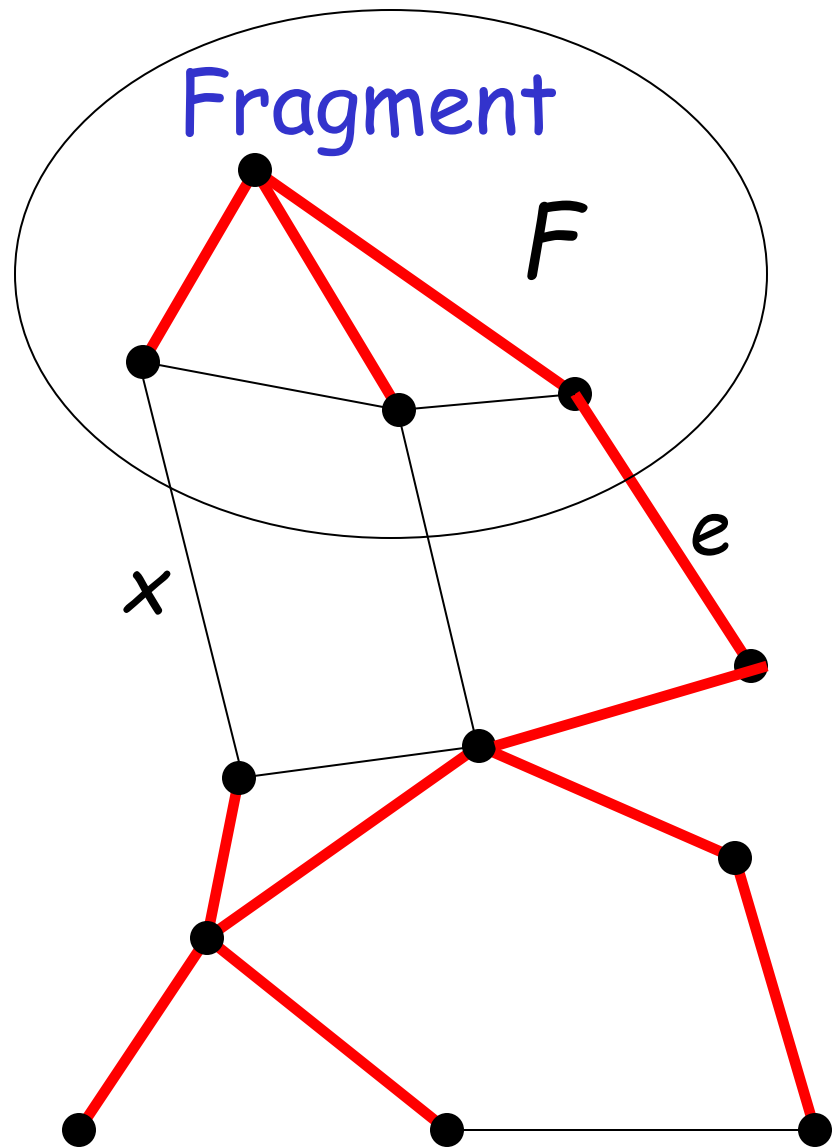
$$w(e) \leq w(x)$$

Obtain T'
and since
 $w(e) \leq w(x)$

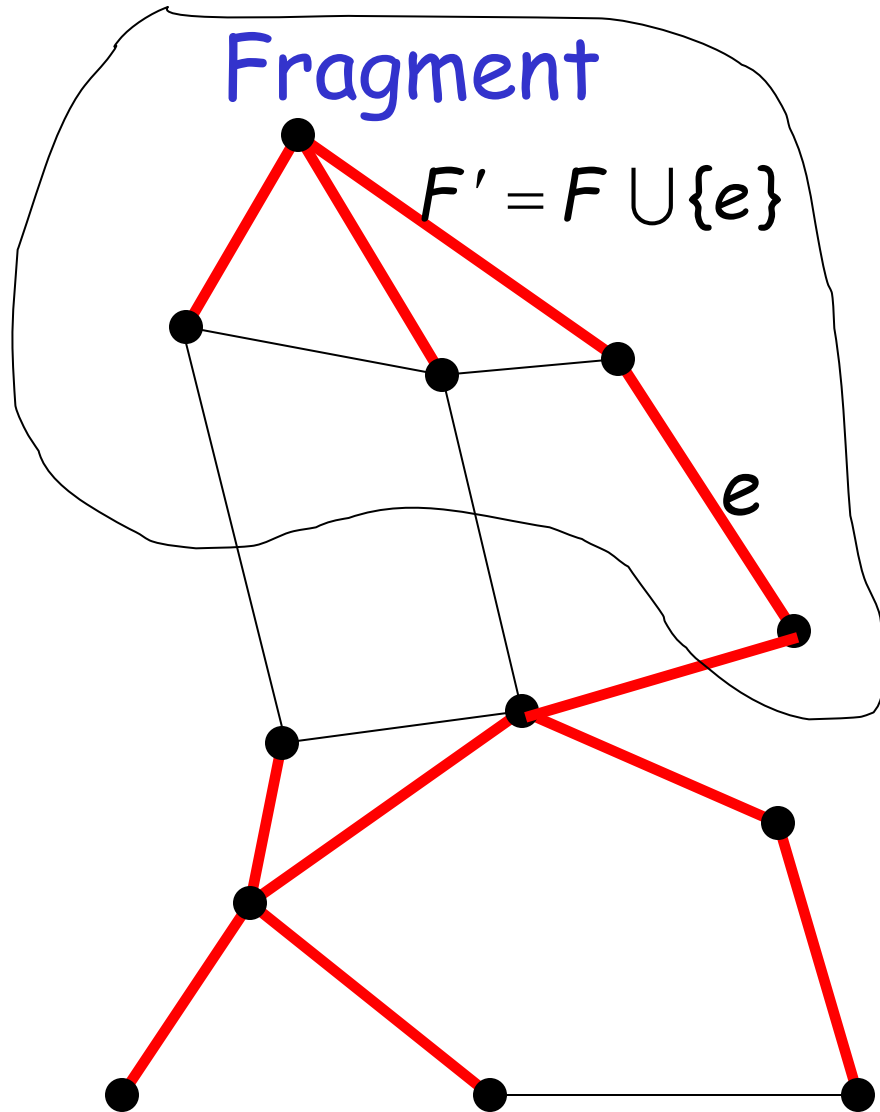
$\Rightarrow w(T') \leq w(T)$

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 12

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

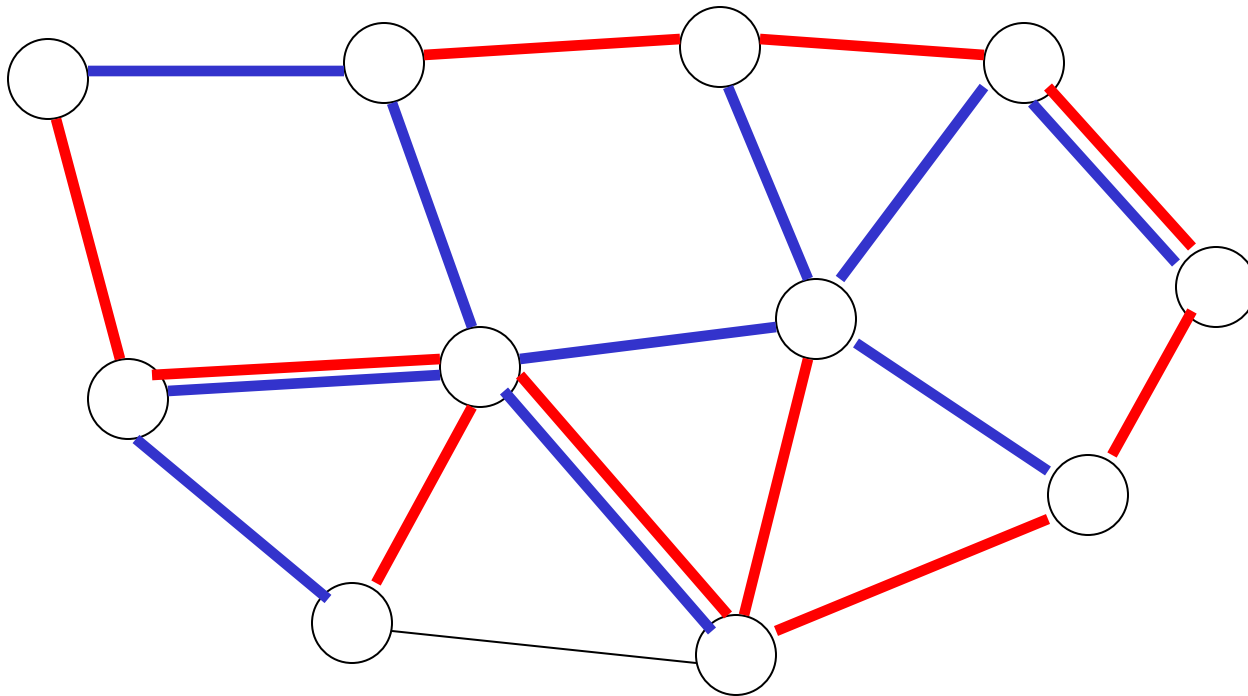
Proof: Basic Idea:

Suppose there are two MSTs

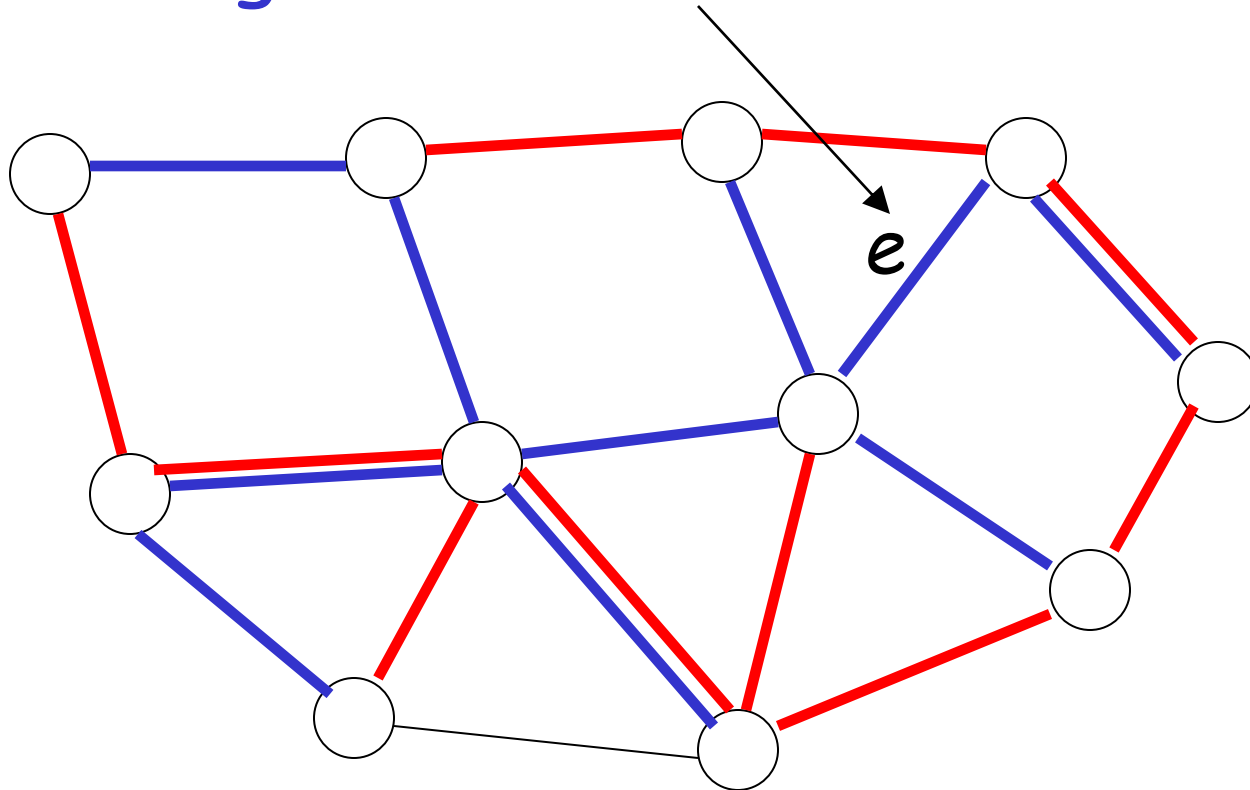
Then we prove that there is another spanning tree of smaller weight

⇒ contradiction!

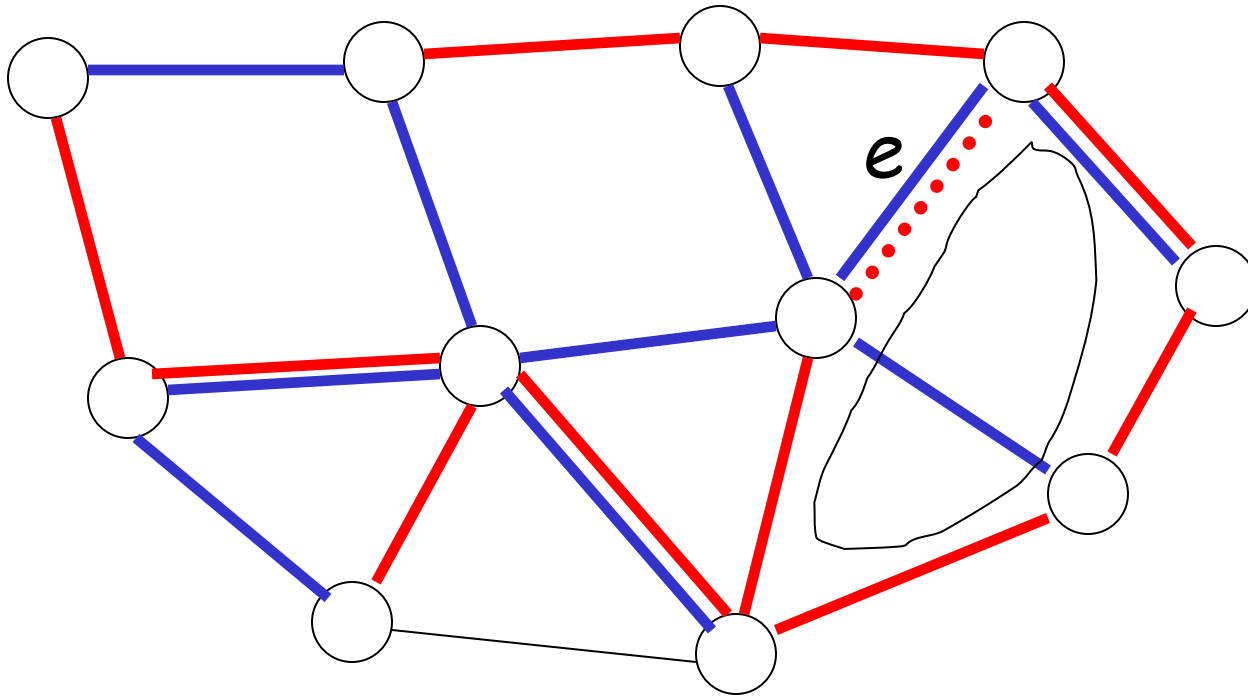
Suppose there are two MSTs



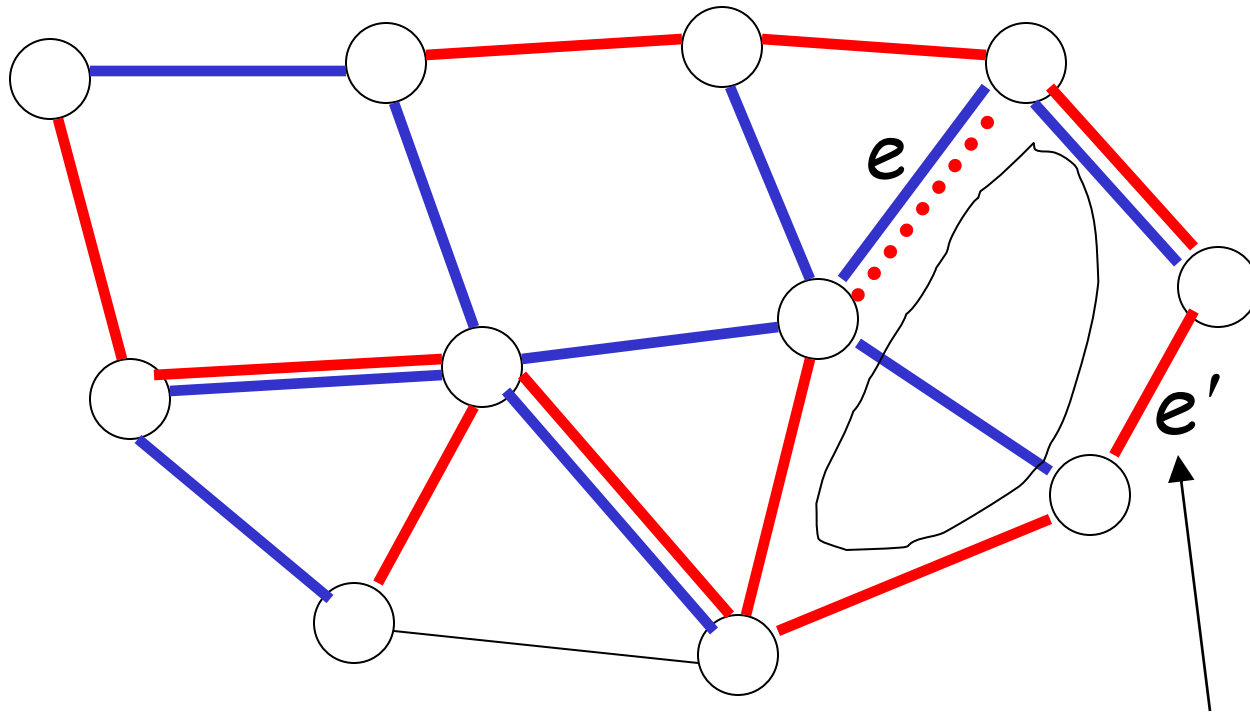
Take the smallest-weight edge
not in the intersection, and assume
w.l.o.g. it is blue



Cycle in RED MST

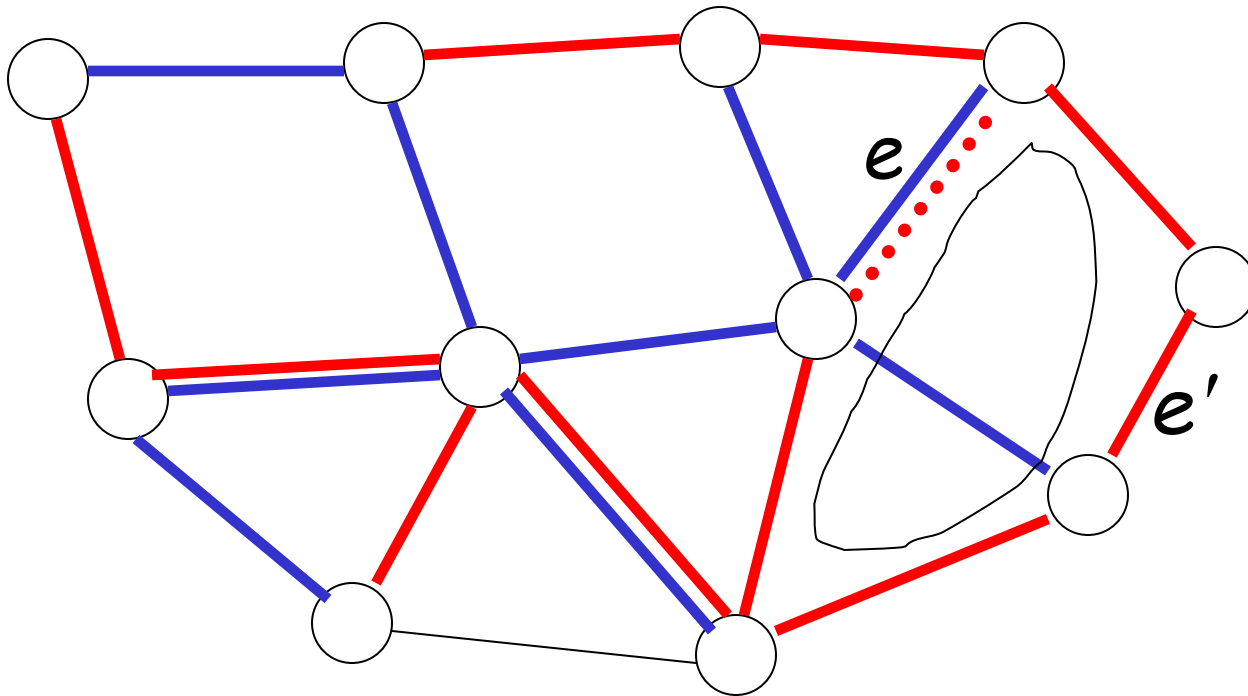


Cycle in RED MST



e' : any red edge in the cycle not in BLUE MST
(\exists since blue tree is acyclic)

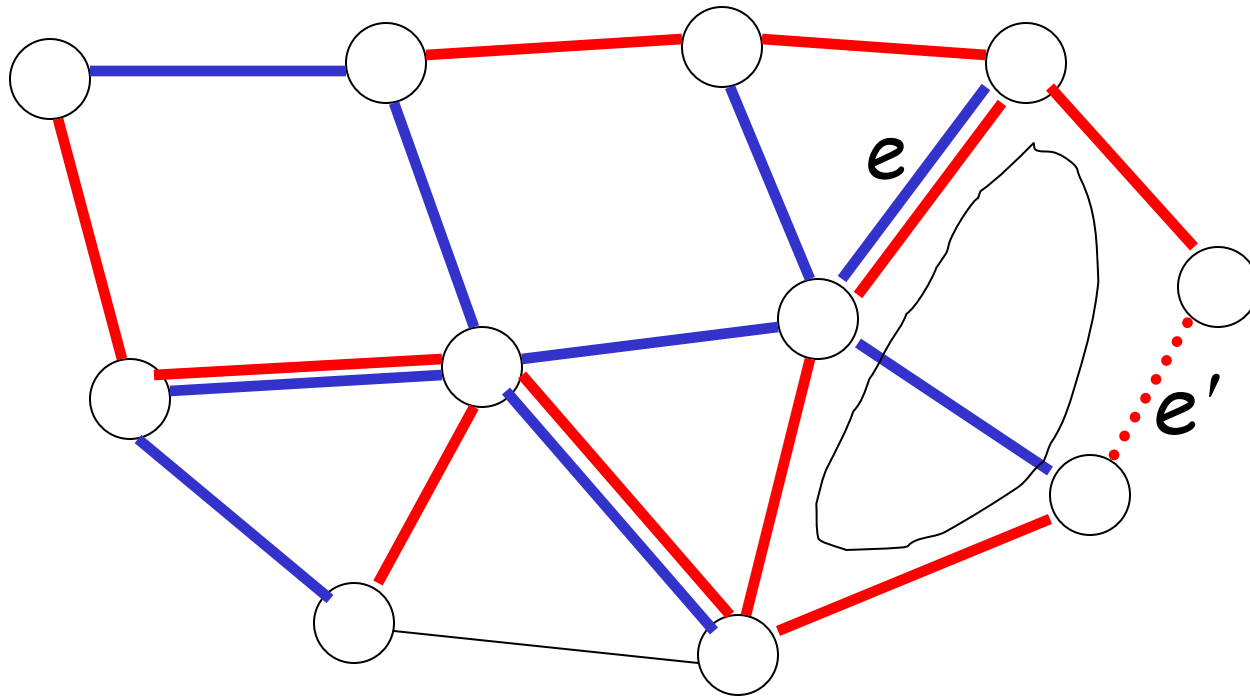
Cycle in RED MST



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 19

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 Gallager-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 F , and repeatedly apply the blue rule

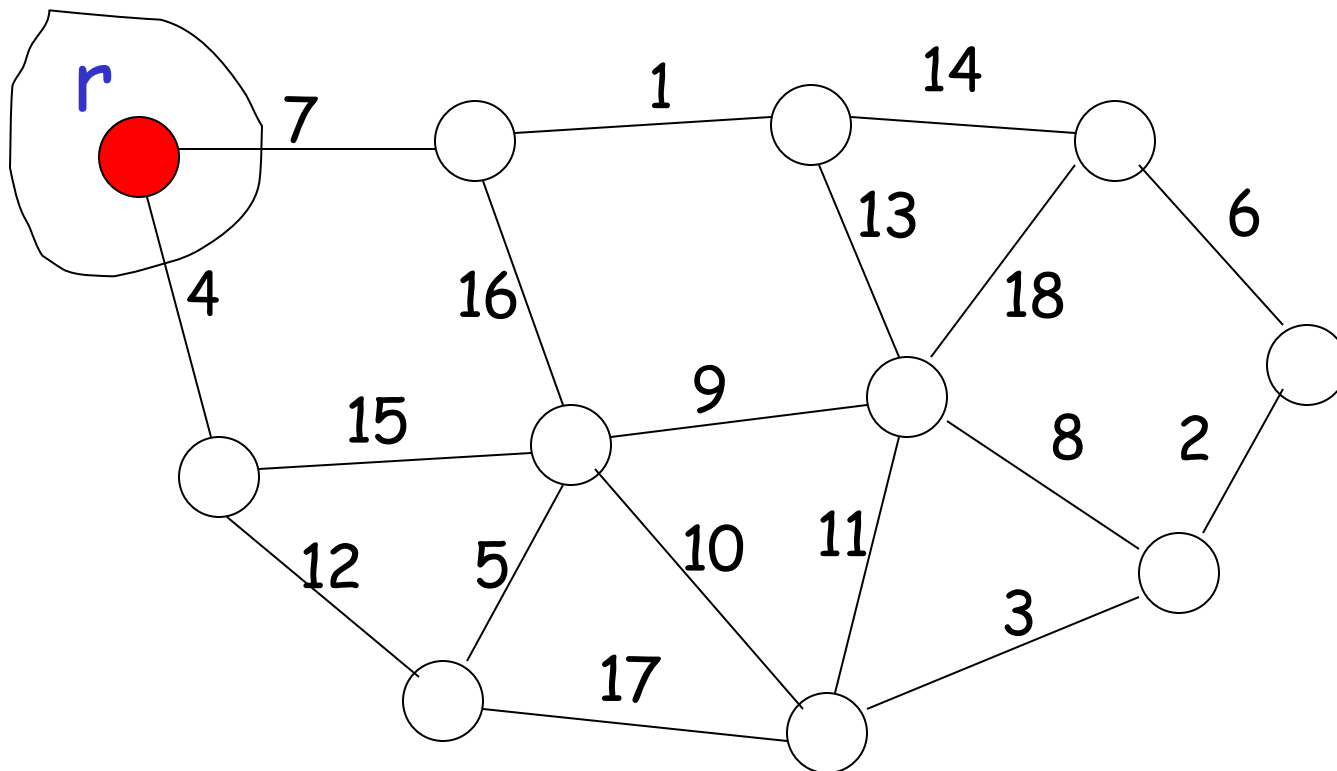
$$F = \{r \in V(G)\}$$

Repeat

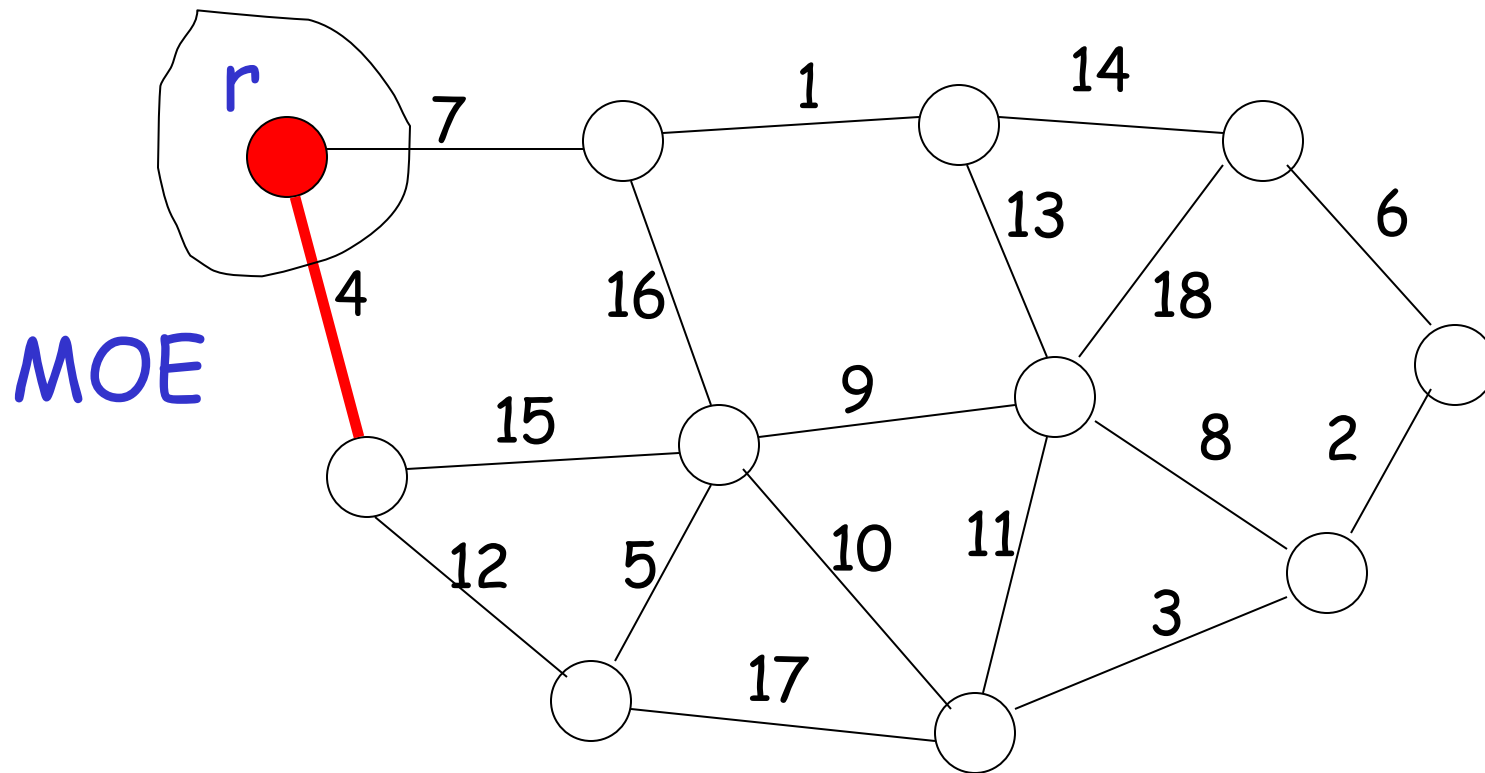
Augment fragment F with a MOE

Until no other edge can be added to F

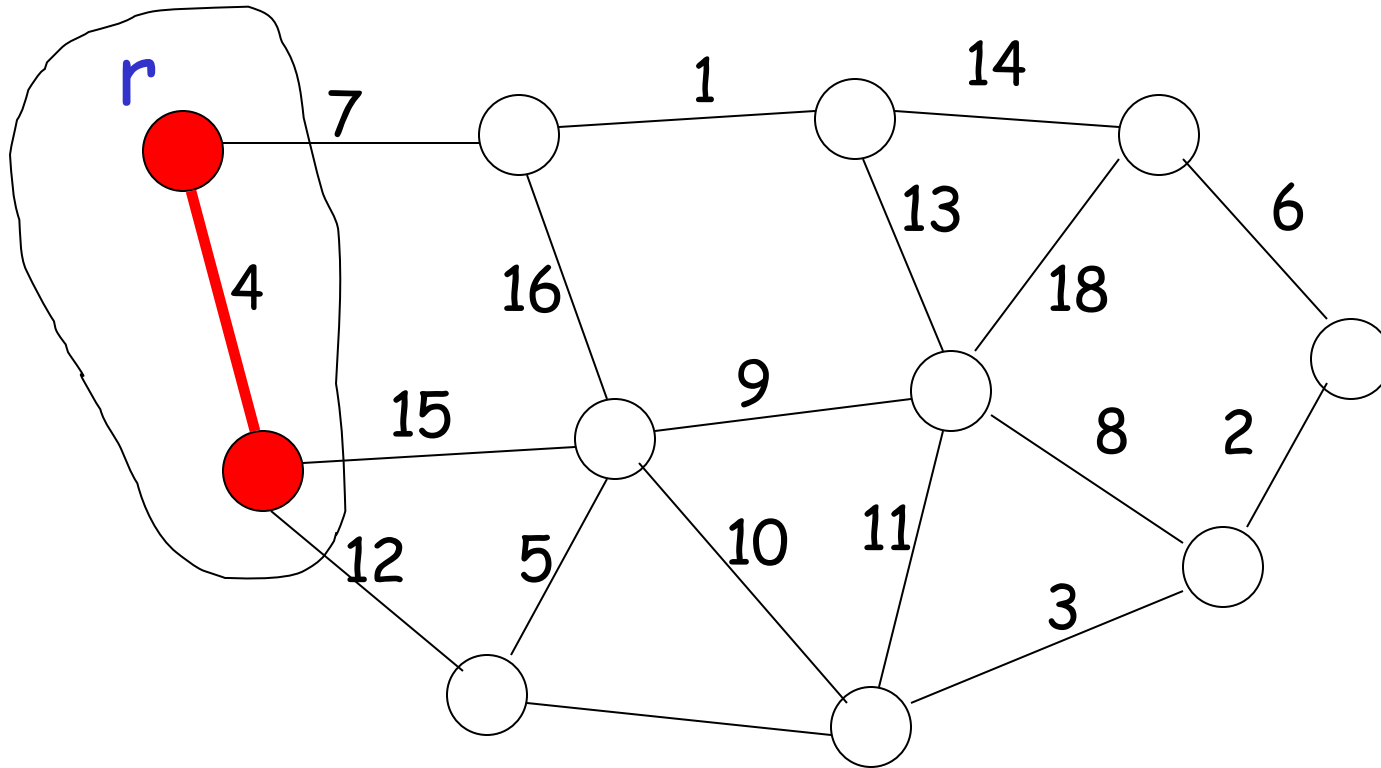
Fragment F



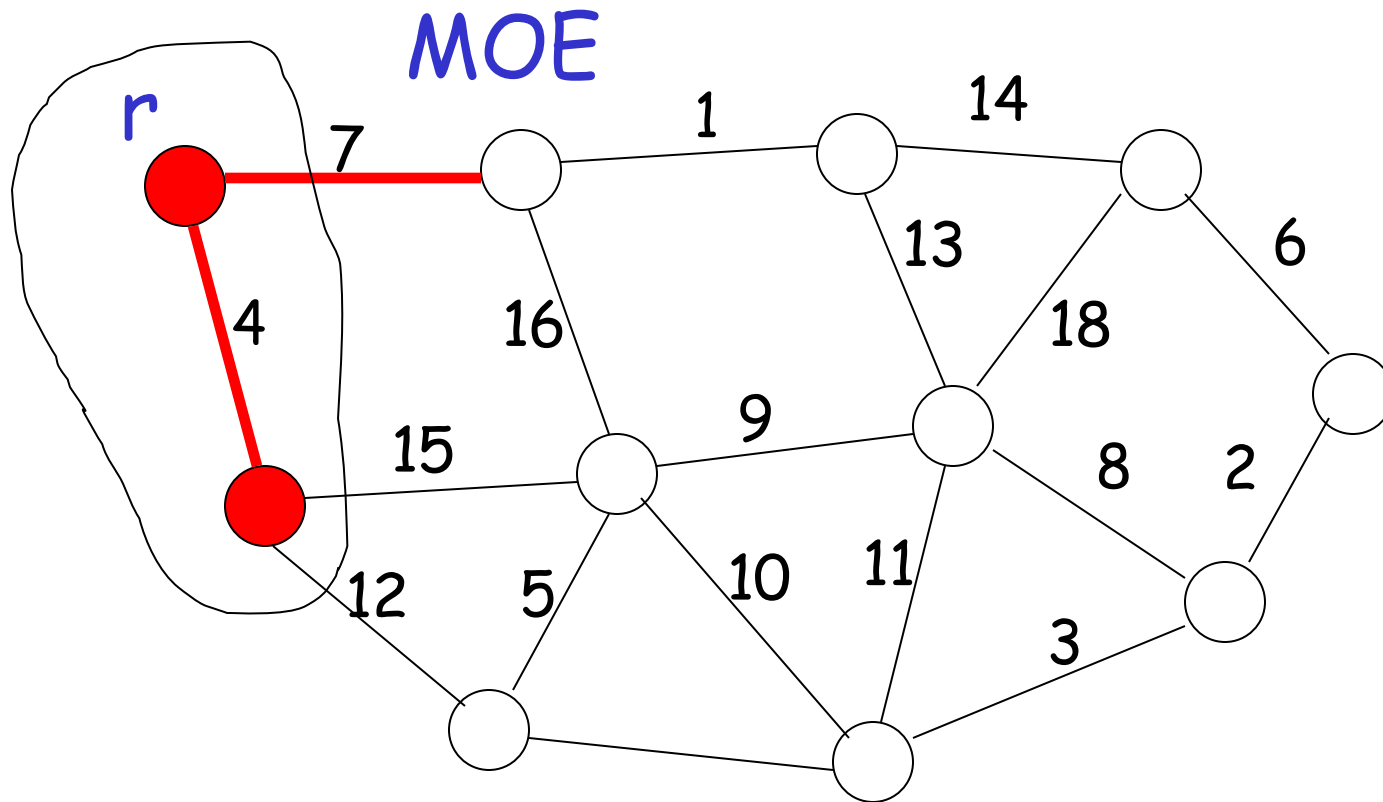
Fragment F



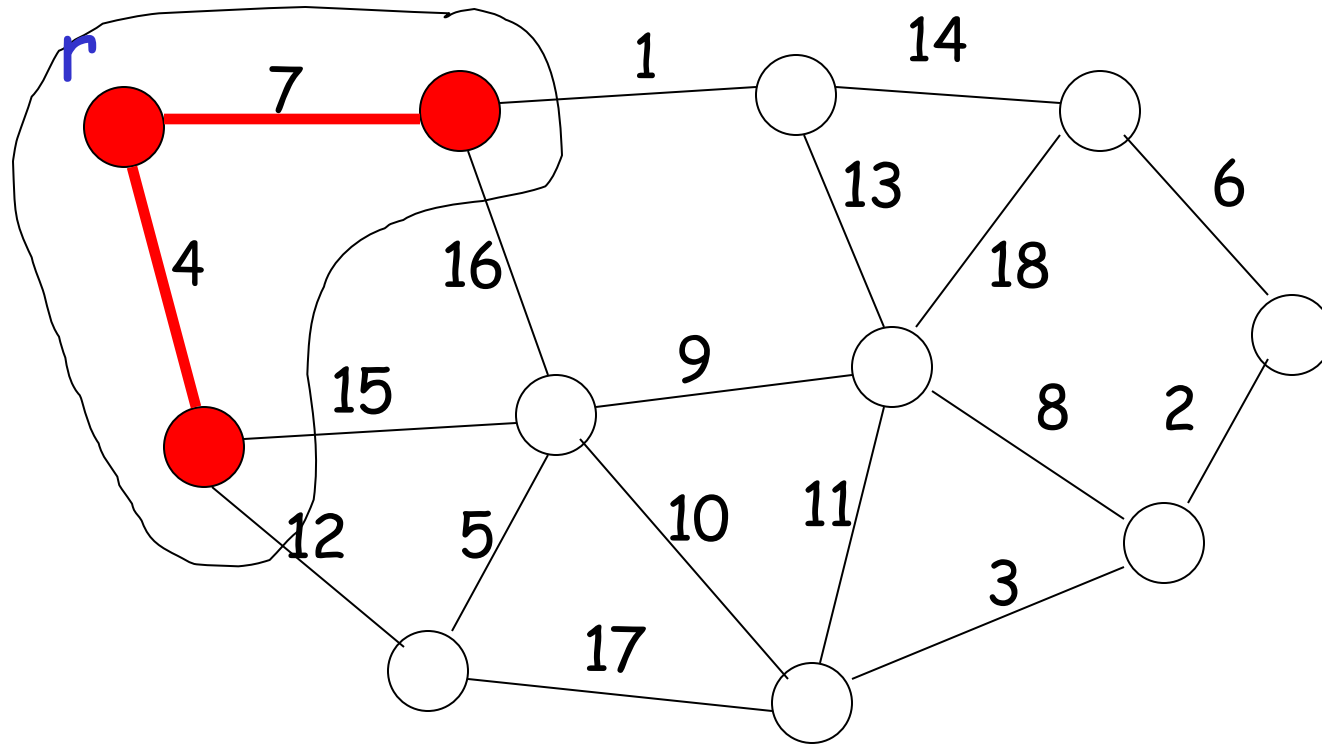
Augmented fragment F



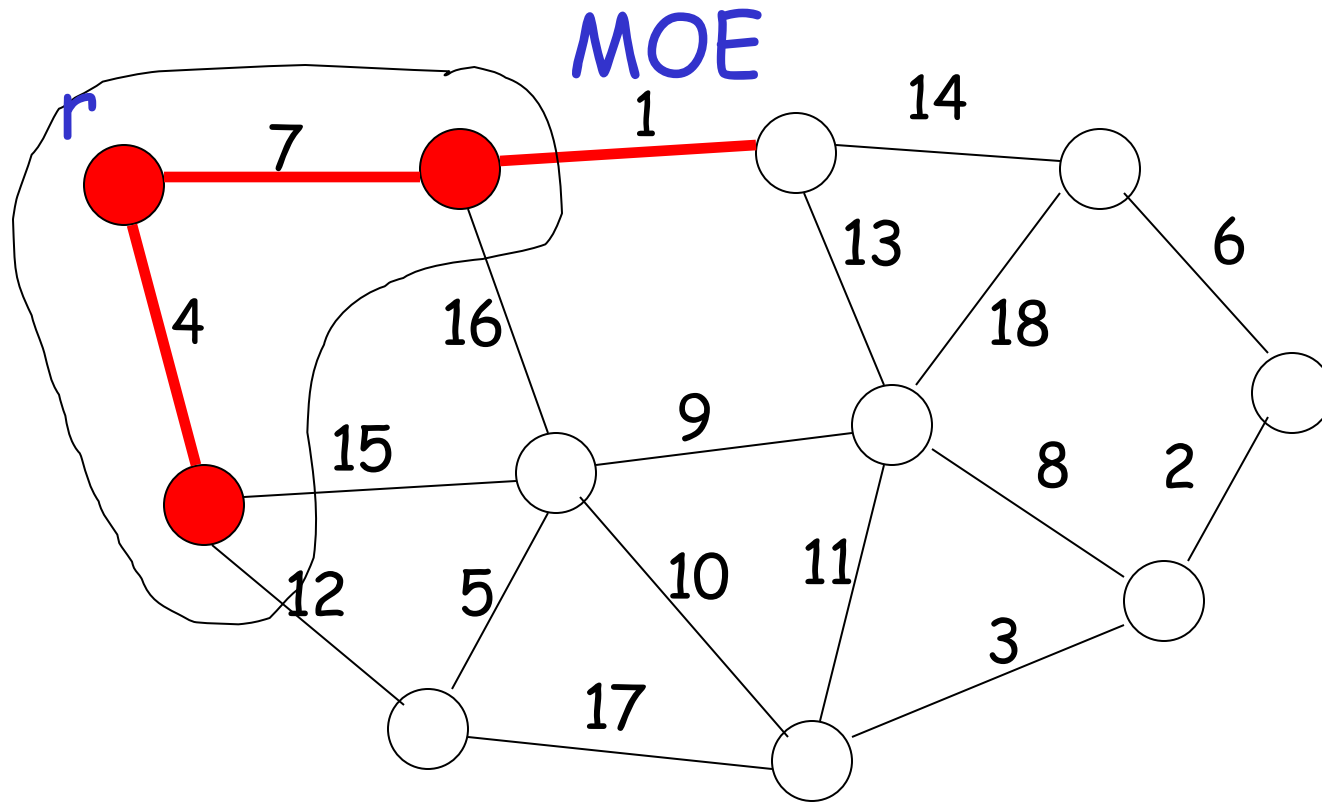
Fragment F



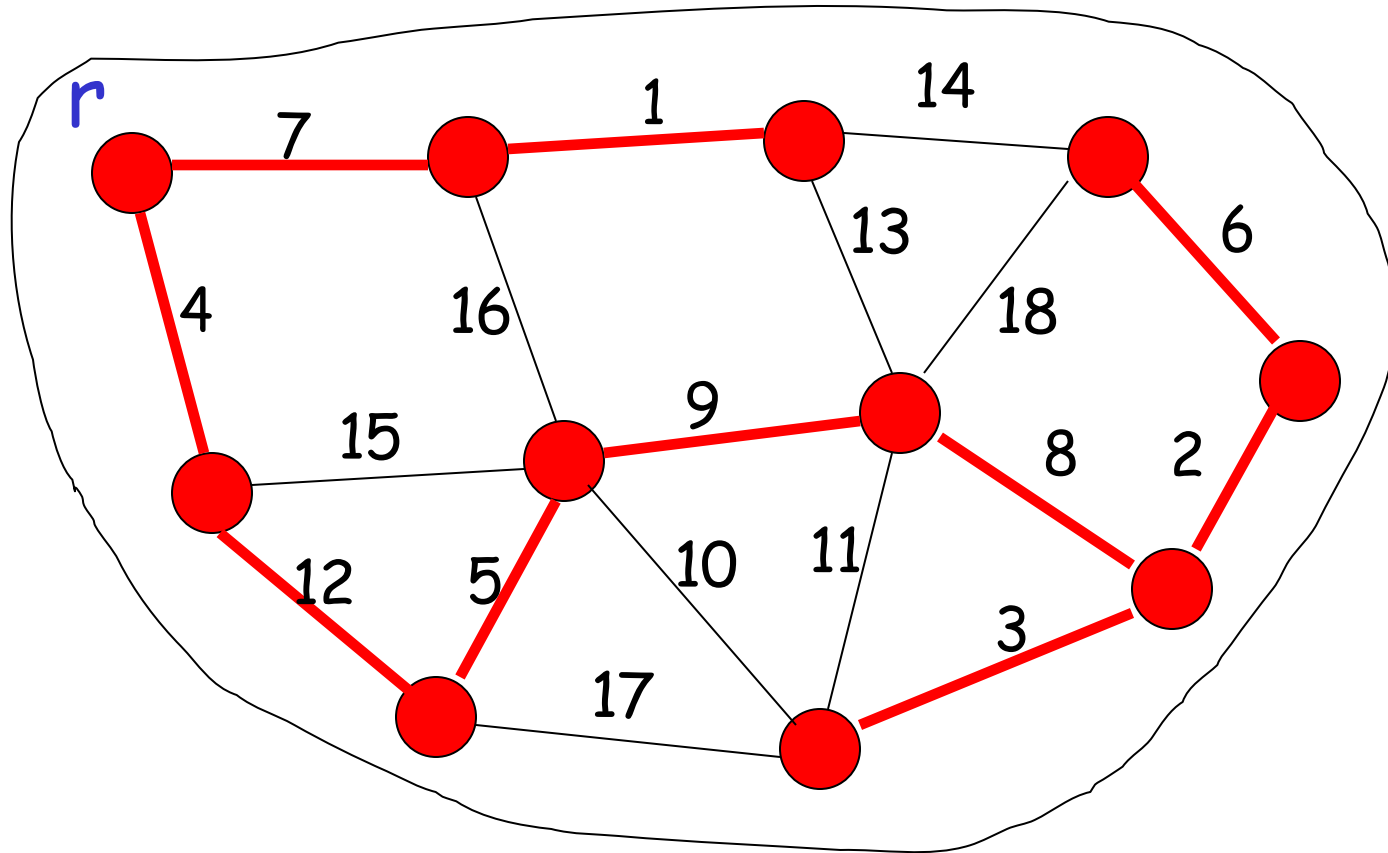
Augmented fragment F



Fragment 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

- 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

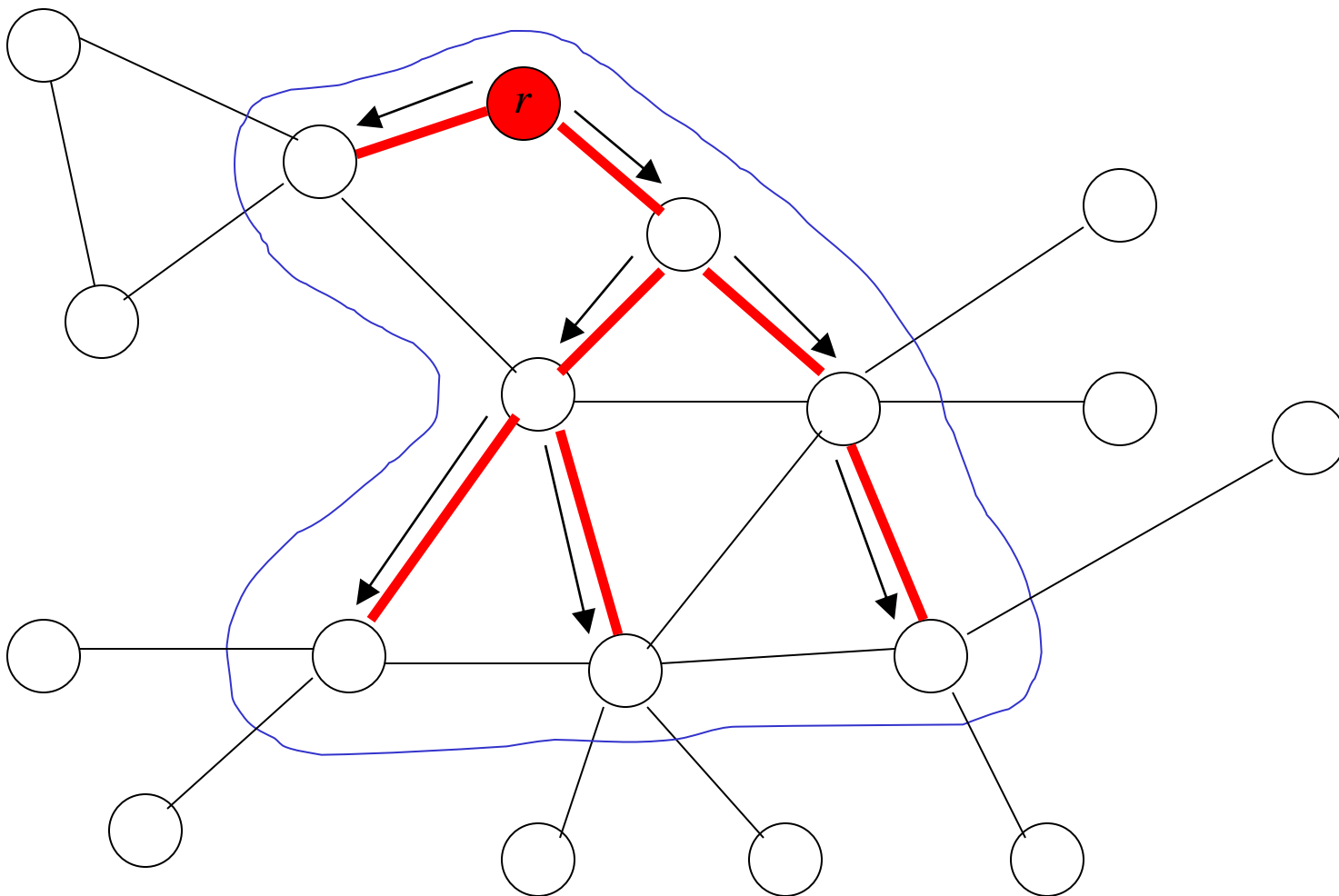
Each processor stores:

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)

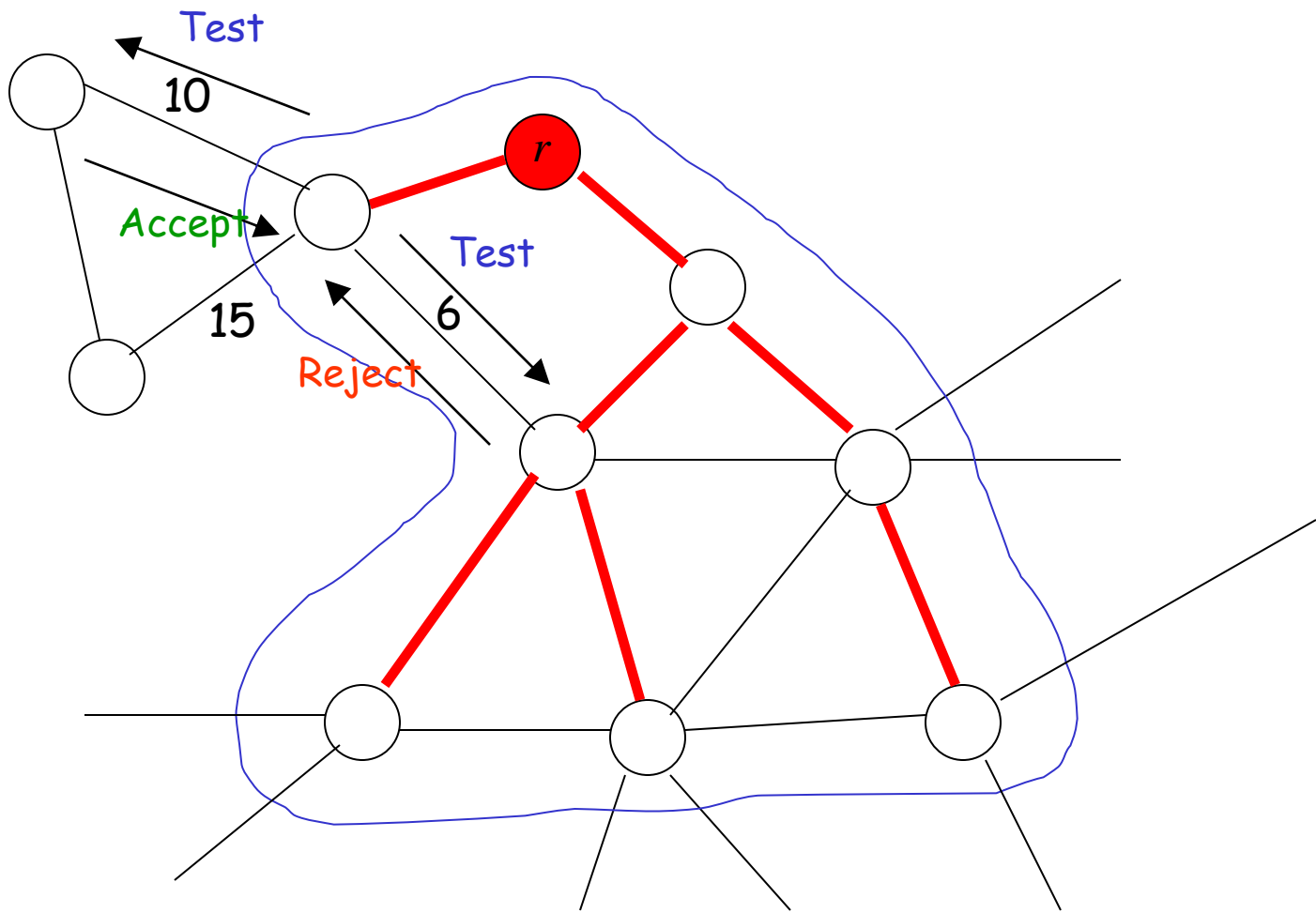
Type of messages in asynchronous Prim

1. **Search_MOE**: coordination message initiated by the root
2. **Test**: check the status of an incident **basic** edge
3. **Reject, Accept**: response to **Test**
4. **Report(weight)**: report to the parent node the weight of the MOE of the appended subfragment
5. **Add_edge**: say to the node adjacent to the fragment's MOE to add it (i.e., change its status from **basic** to **branch**)
6. **Connect**: perform the union of the found MOE to the fragment (this changes the status of the other end-node from **out** to **in**)
7. **Connected**: notify the root that connection has taken place

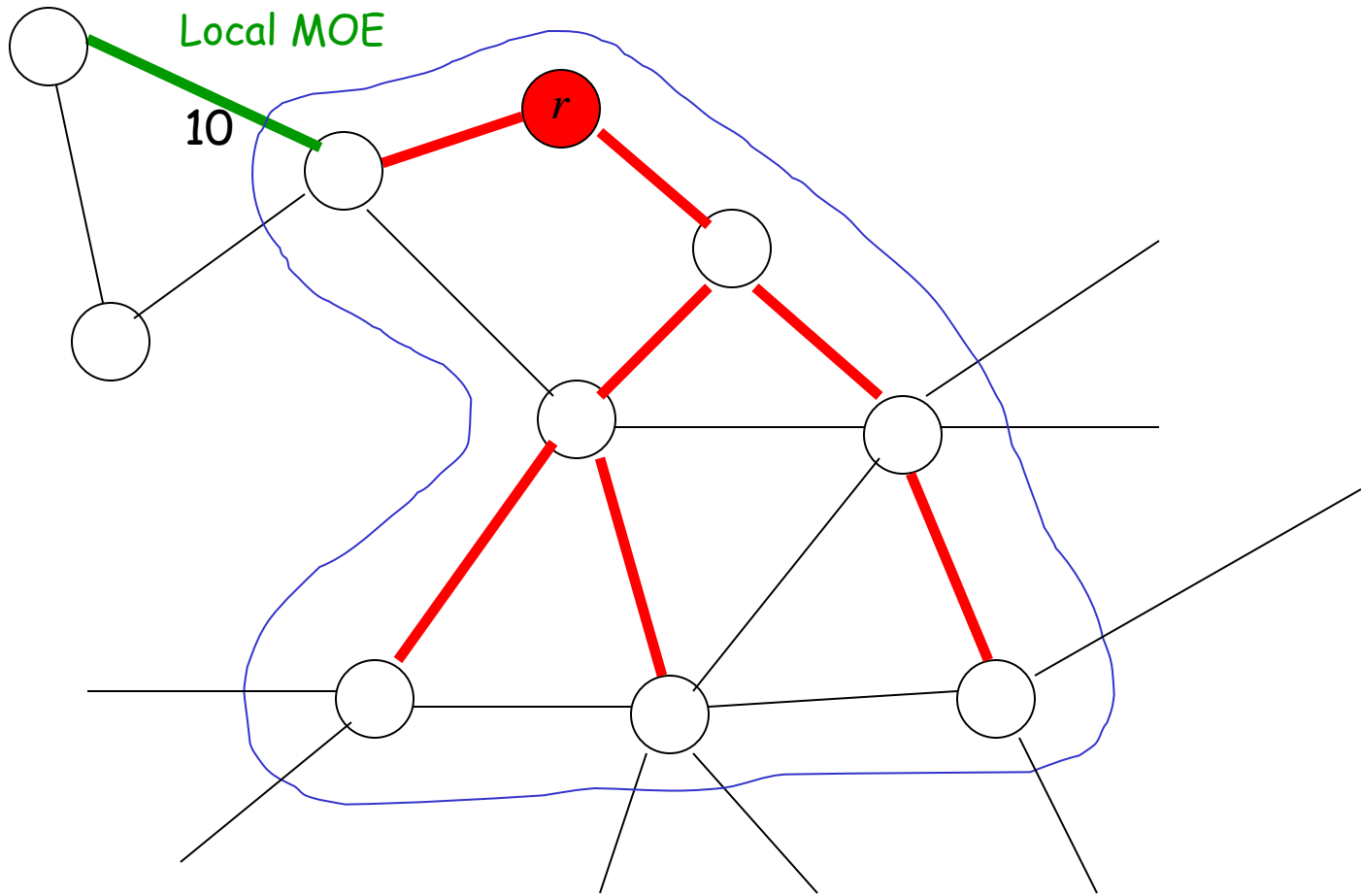
Sample execution: The root sends a **Search_MOE** message, and the message floods along the fragment, so that each node in fragment starts looking for its local MOE



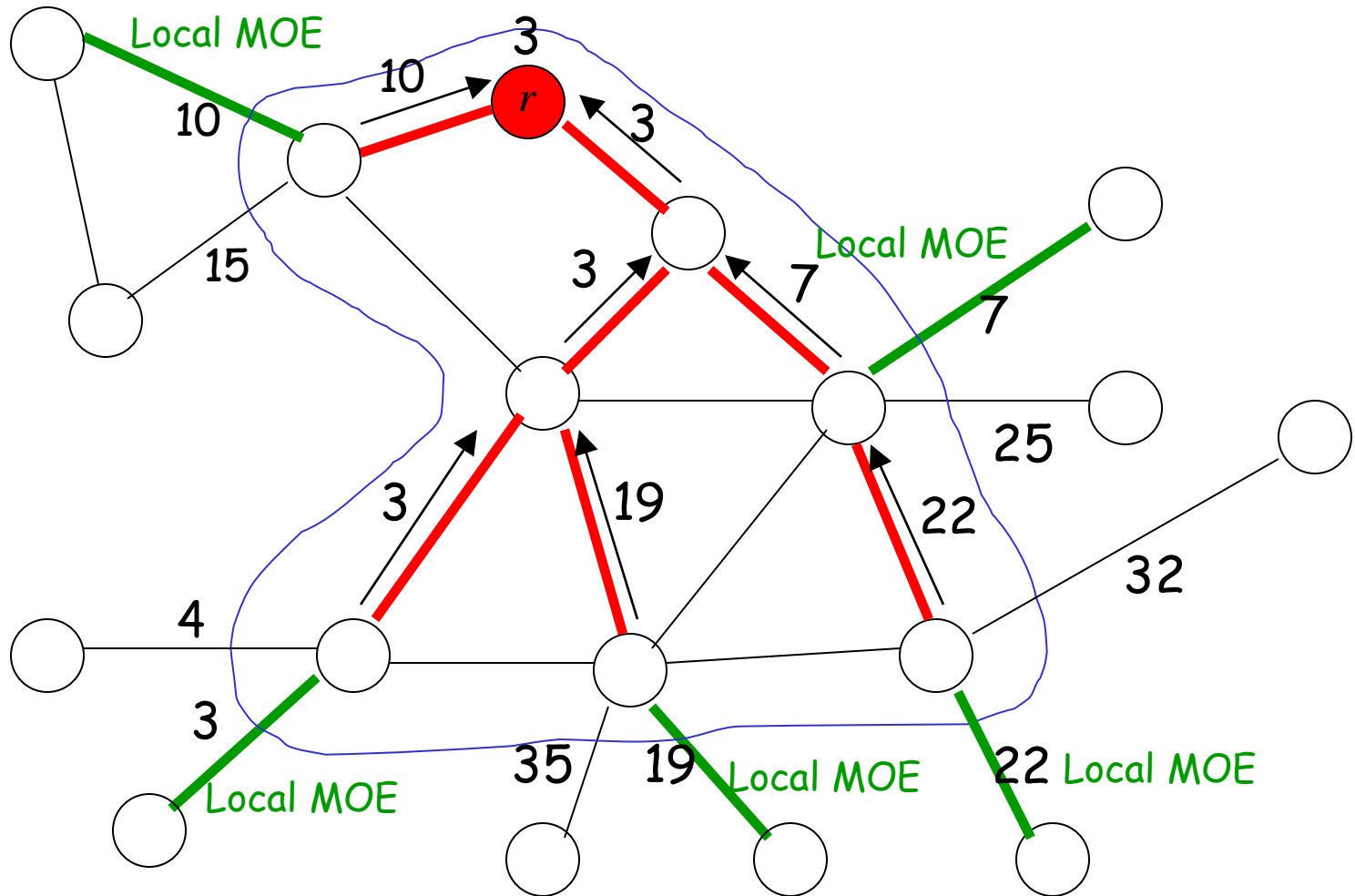
To discover its local MOE, each node sends a **Test** message over its **basic** edges in increasing order of weight, until it receives an **Accept**. Rejected tests turn to **reject** the status of the corresponding edge.



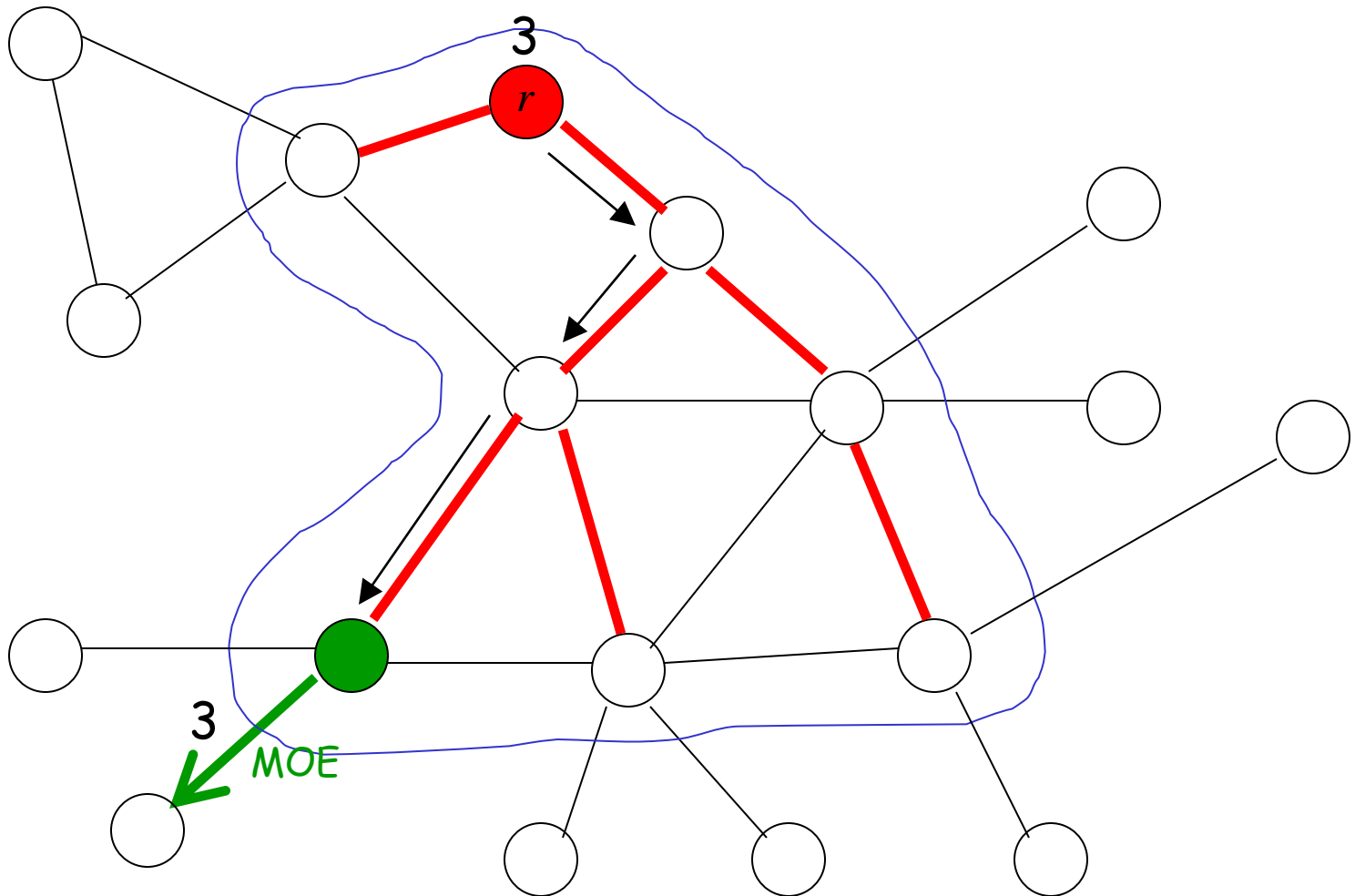
Then it knows its local MOE (notice this can be void)



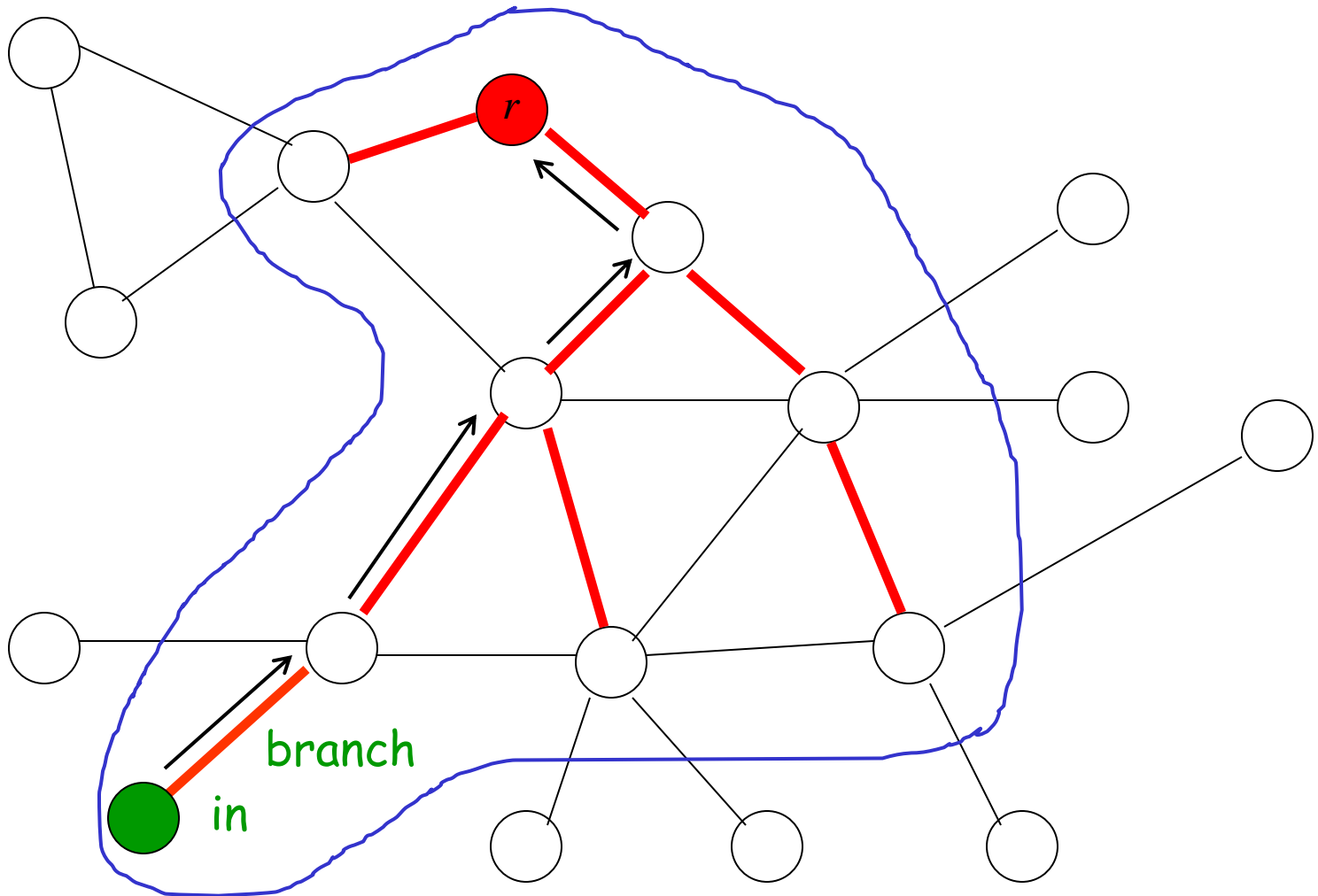
Then, if a node is a leaf, it sends a **Report** with the weight of its MOE (if any) to its parent, while if a node has children, it waits for a **Report** from each child, and then selects a global minimum between the weight of its local MOE and the weights of the reported MOEs, which will be then **reported** 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 resumes 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 iteration, 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 iteration globally circulate $O(n)$ messages. Since we have $n-1$ iterations, the claim follows.

END OF PROOF 39

Synchronous Prim

It will work in $O(n^2)$ rounds...think to it by yourself...

Kruskal's Algorithm (sequential version)

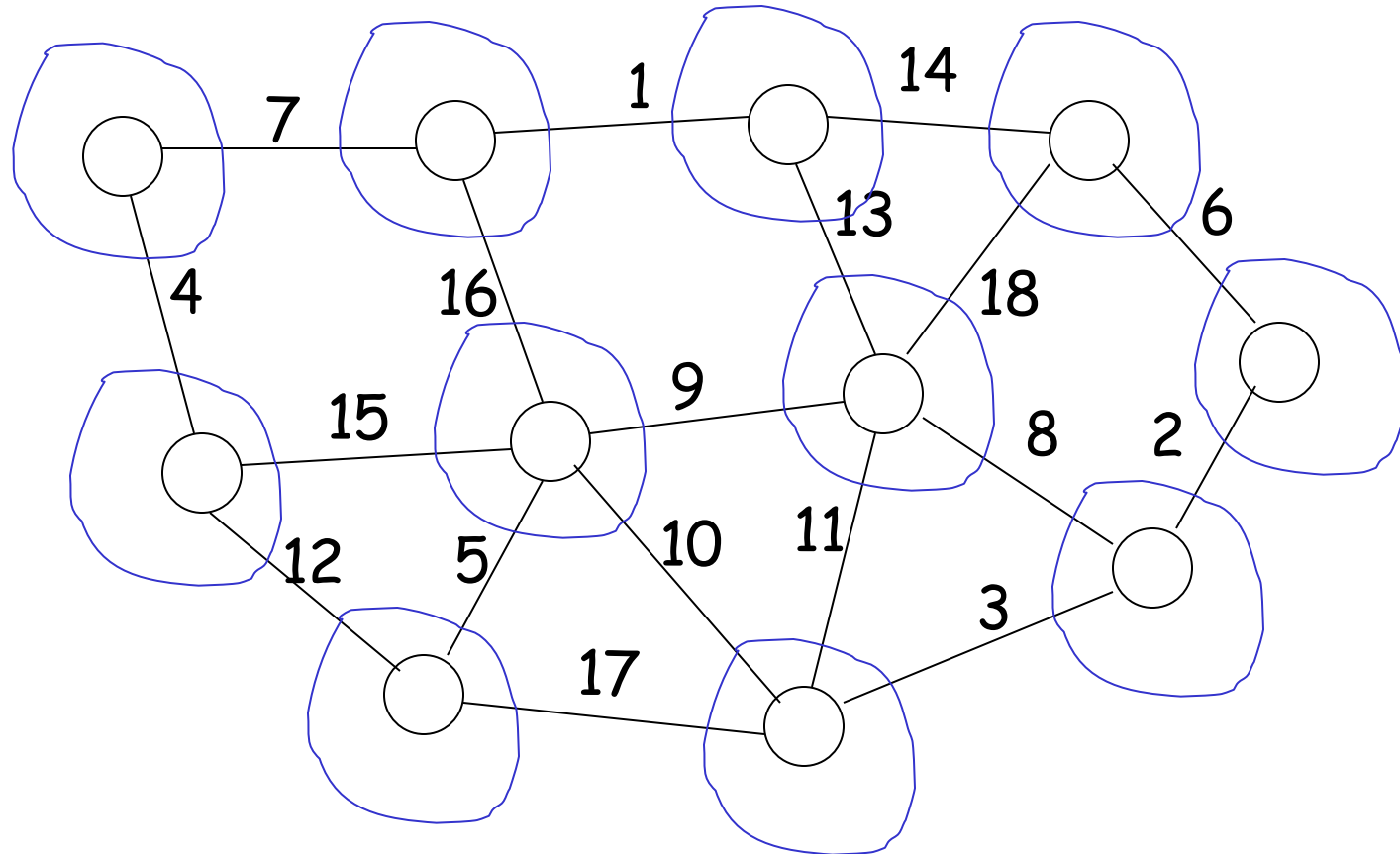
Initially, each node is a fragment

Repeat

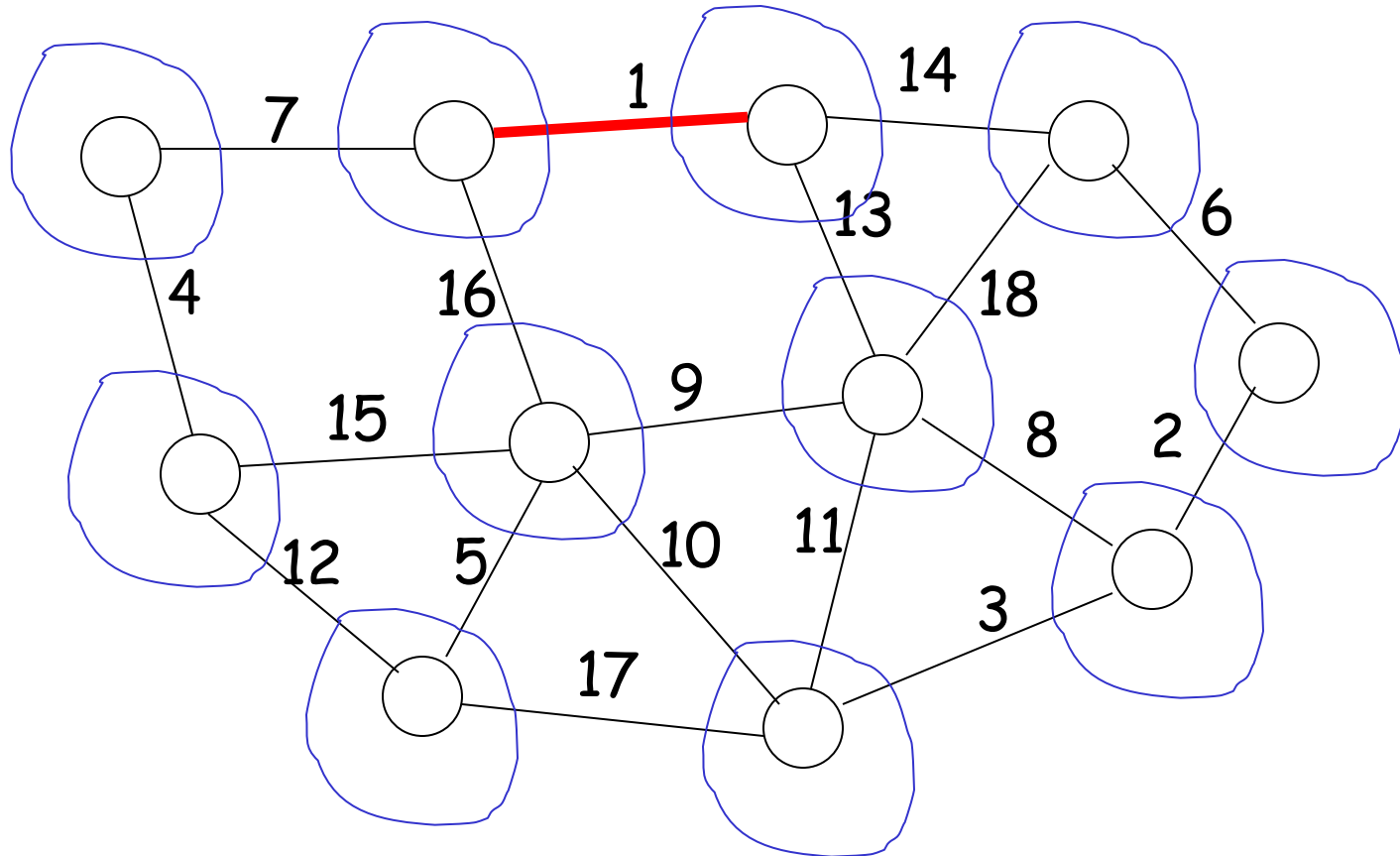
- Find the smallest MOE e of all current fragments
- Merge the two fragments adjacent to e

Until there is only one fragment left

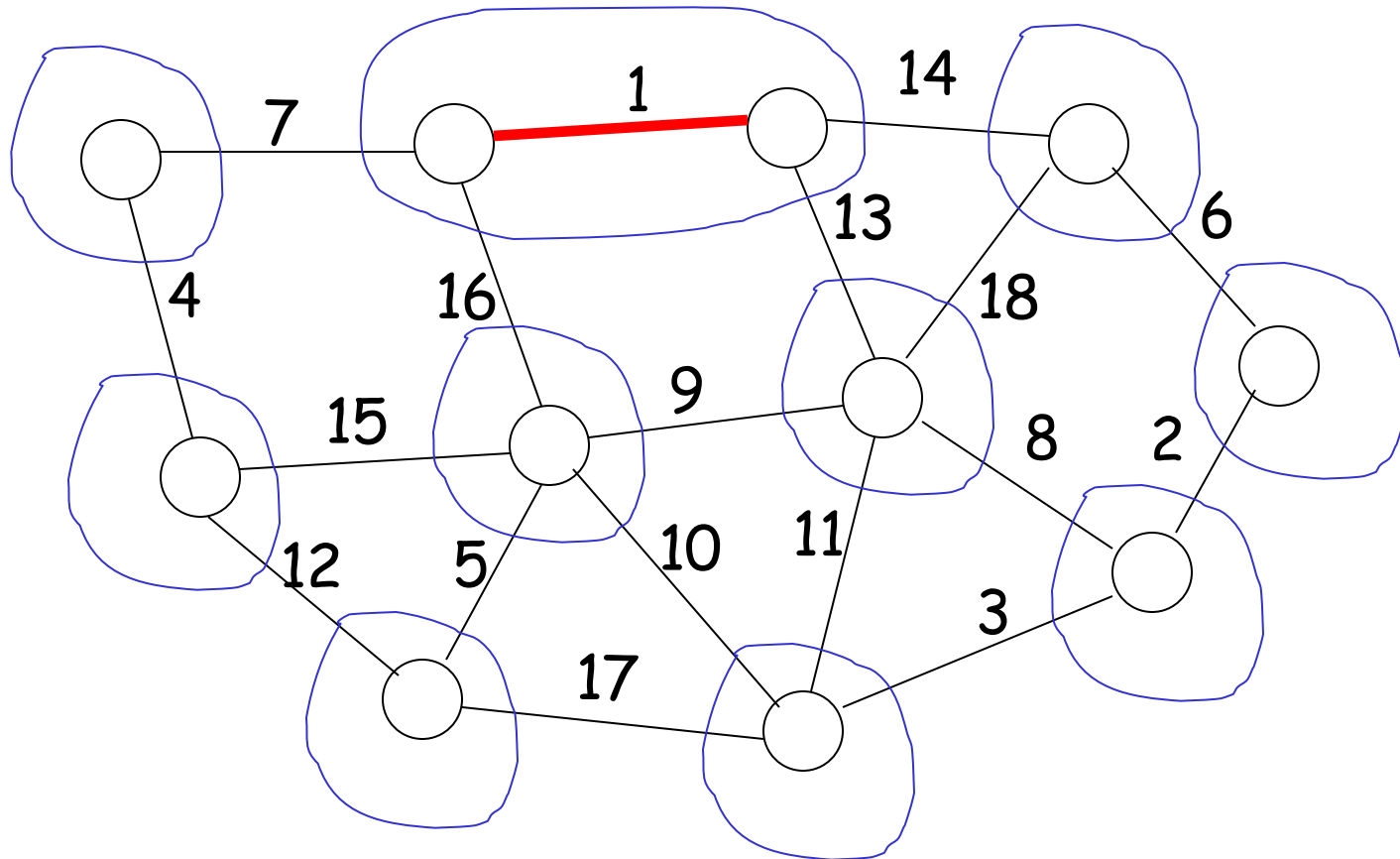
Initially, every node is a fragment



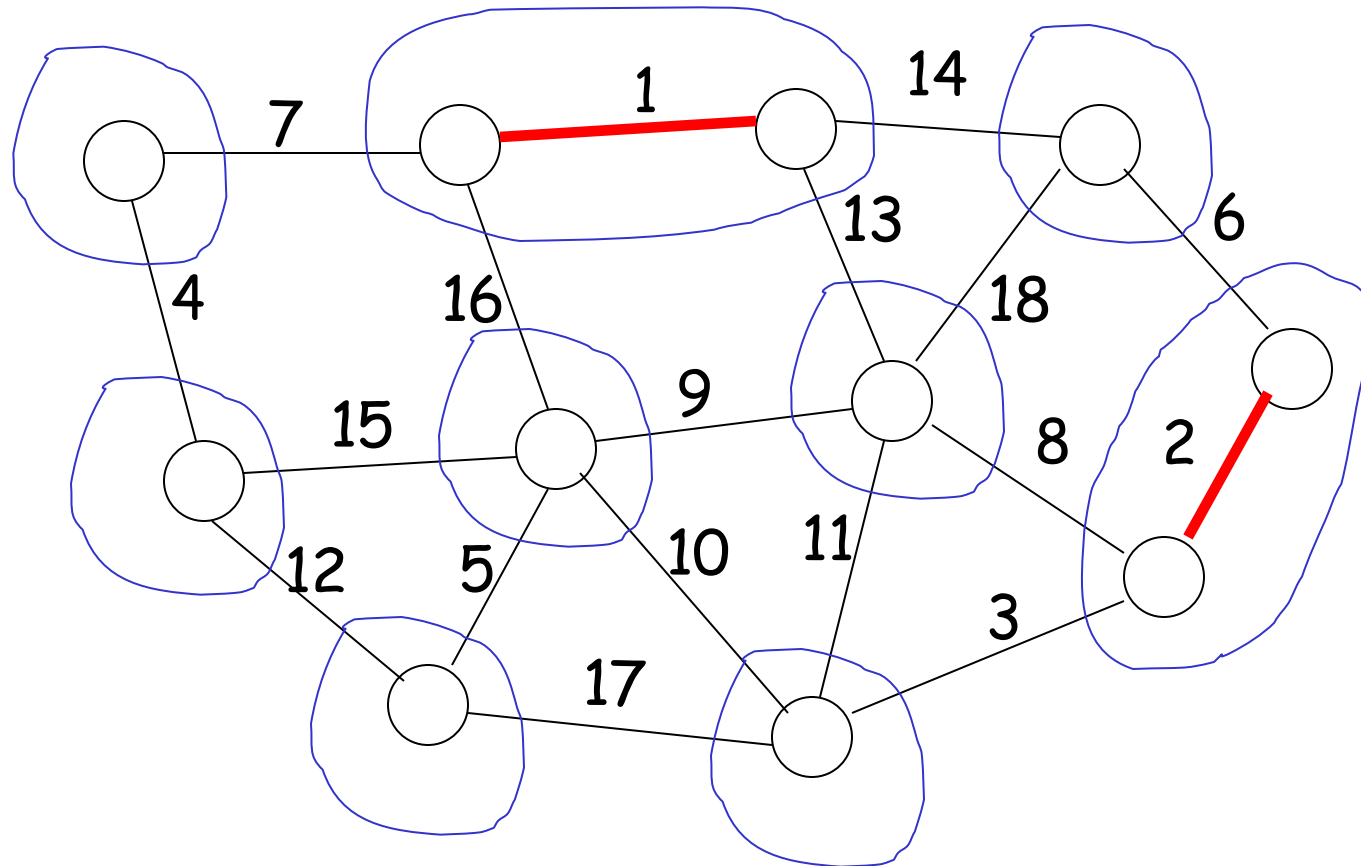
Find the smallest MOE



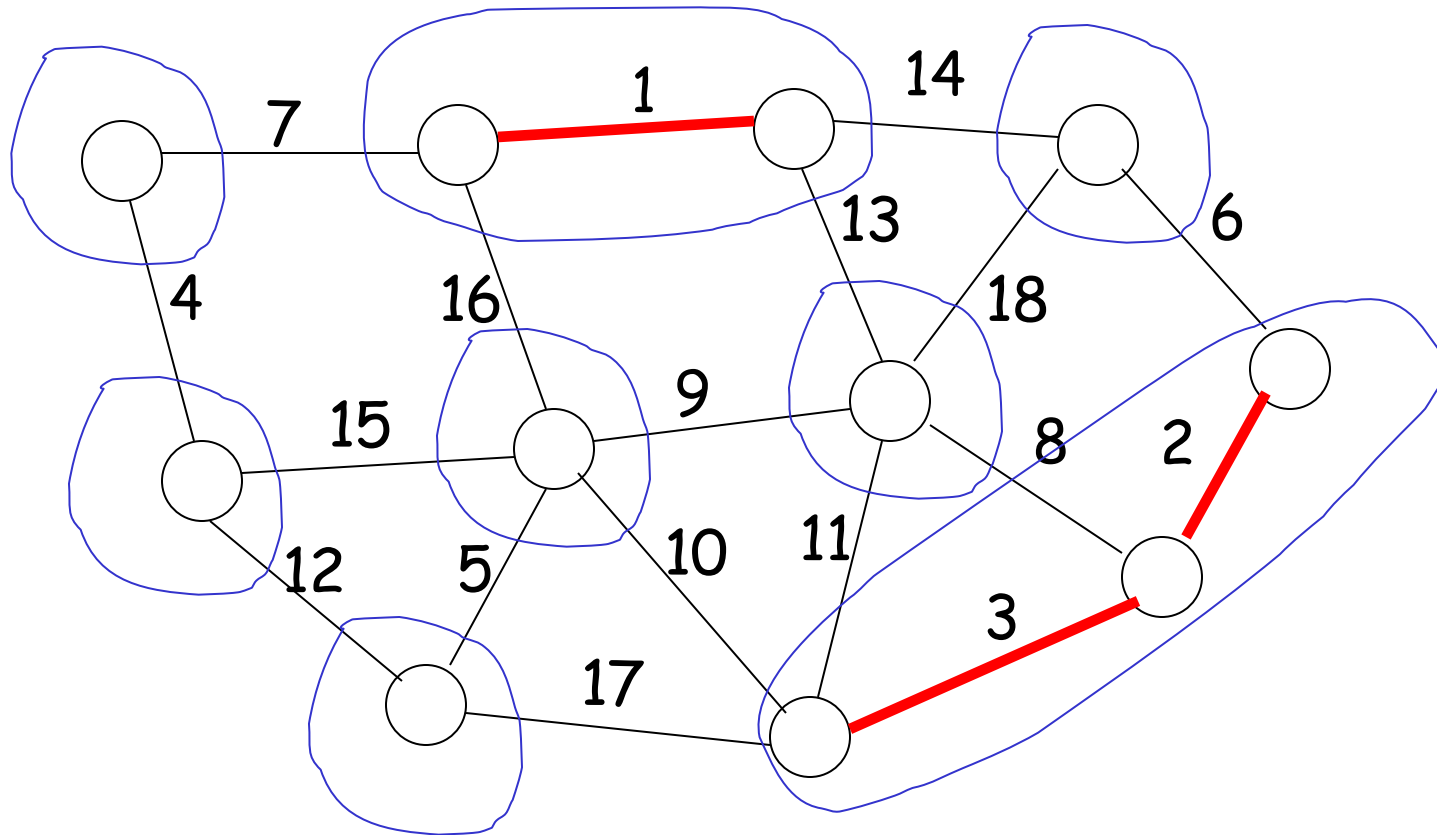
Merge the two fragments



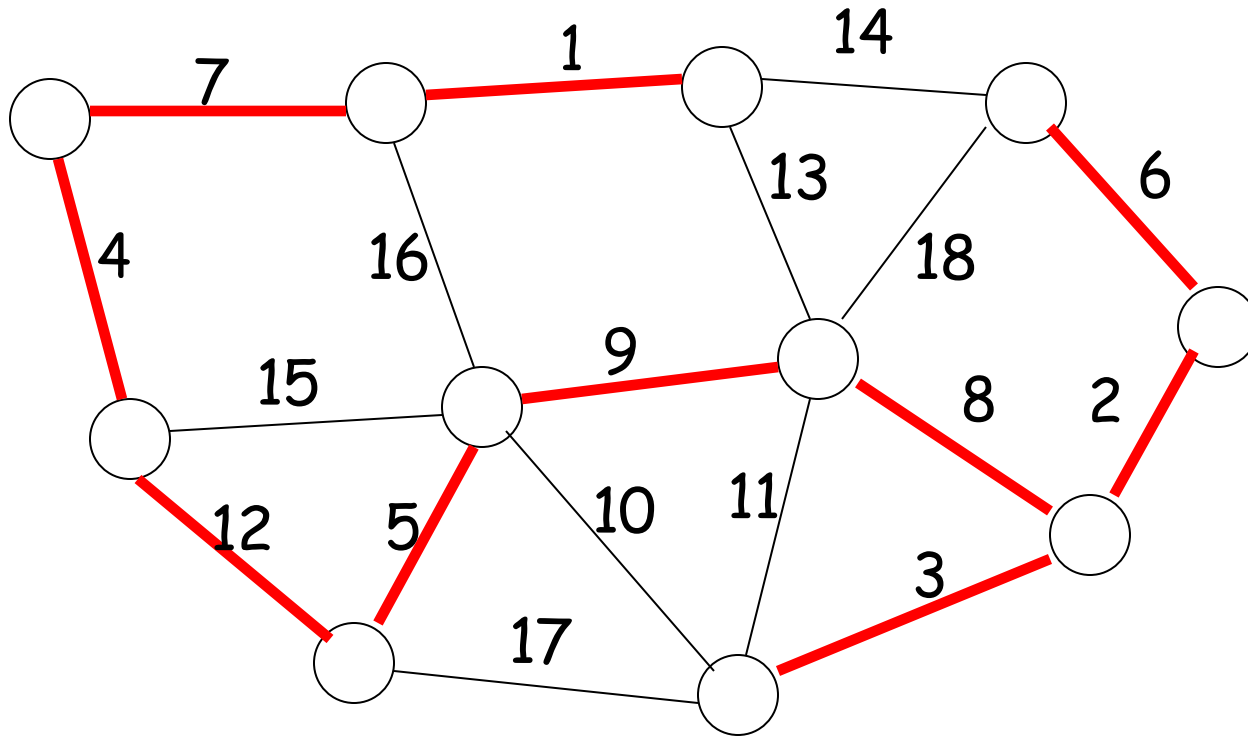
Merge the two fragments



Merge the two fragments



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 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 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 adjacent to MOE's

Until there is only one fragment left

Local description of syncr. GHS

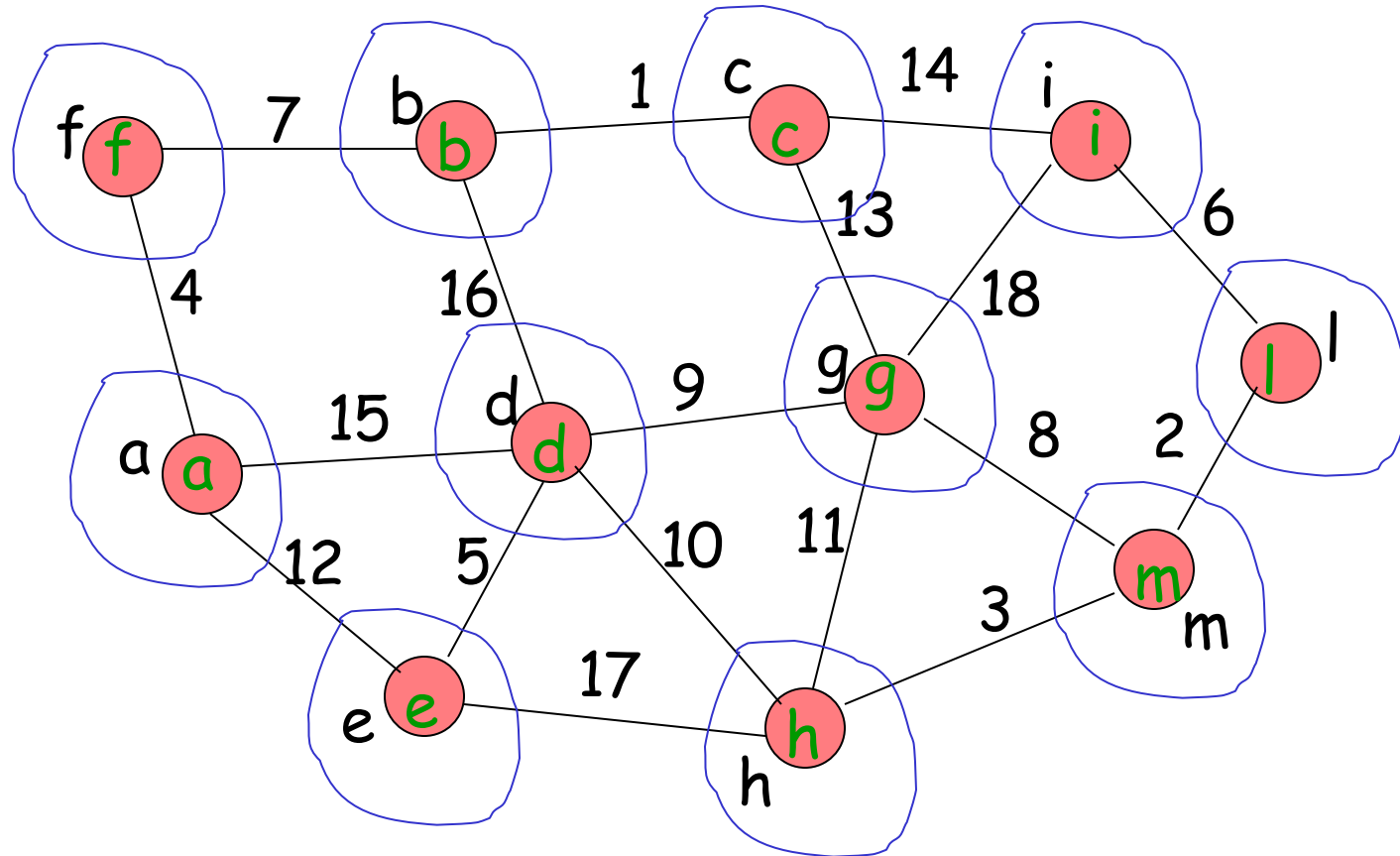
Each processor stores, besides its own ID:

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, this is equal to the node ID
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)

Type of messages

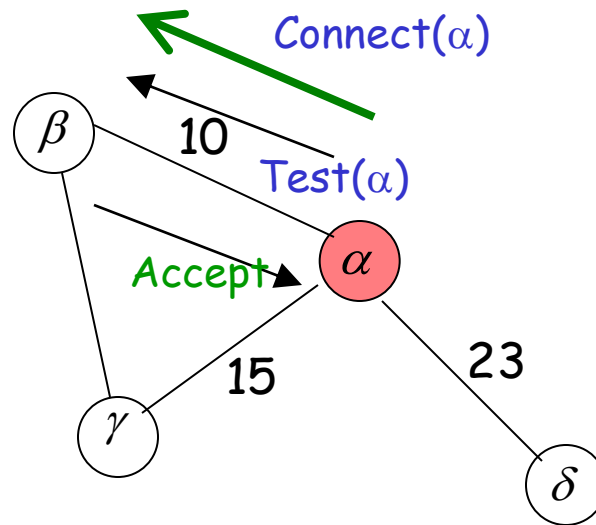
1. **Test(fragment identity)**: each node starts a phase by checking the status of an incident basic edge (if any)
2. **Reject, Accept**: response to **Test**
3. **Report(weight)**: for reporting to the parent node the MOE of the appended subfragment
4. **Merge** (this was called **Add_edge** in Prim): sent by the root to the node incident to the MOE to activate merge 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 (old) fragment identity of the new root

Phase 0: Initially, every node is a fragment...

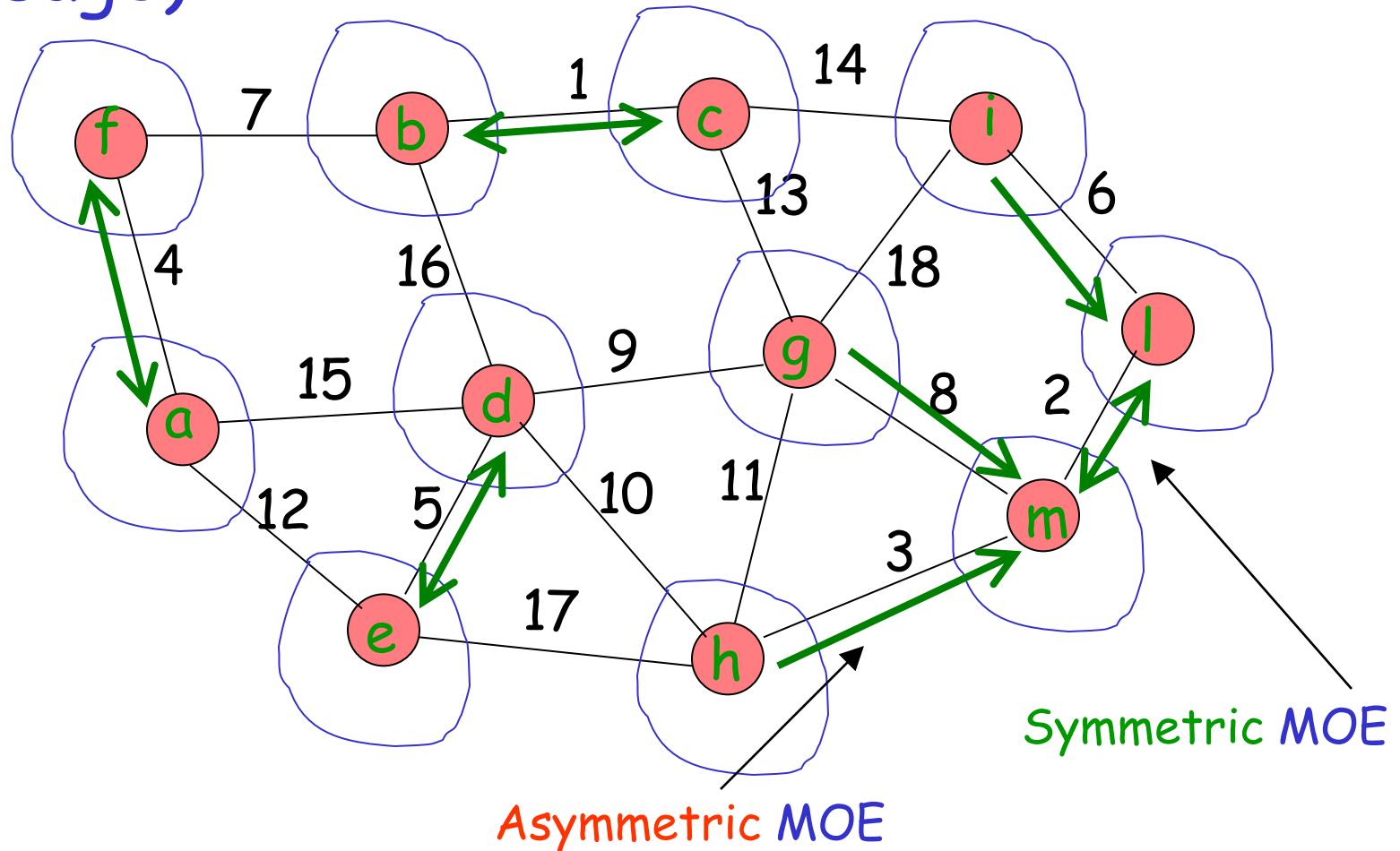


... and every node is the **root** of a fragment, and the **fragment identity** (reported within the nodes) is the node ID (reported nearby the nodes), and so **non-anonymity** is required

In this very first phase, to discover its own MOE, a node sends a **Test** message containing its fragment identity over its **basic** edges of minimum weight, and it will certainly receives an **Accept**; then, it will send a **Connect** message containing its fragment identity

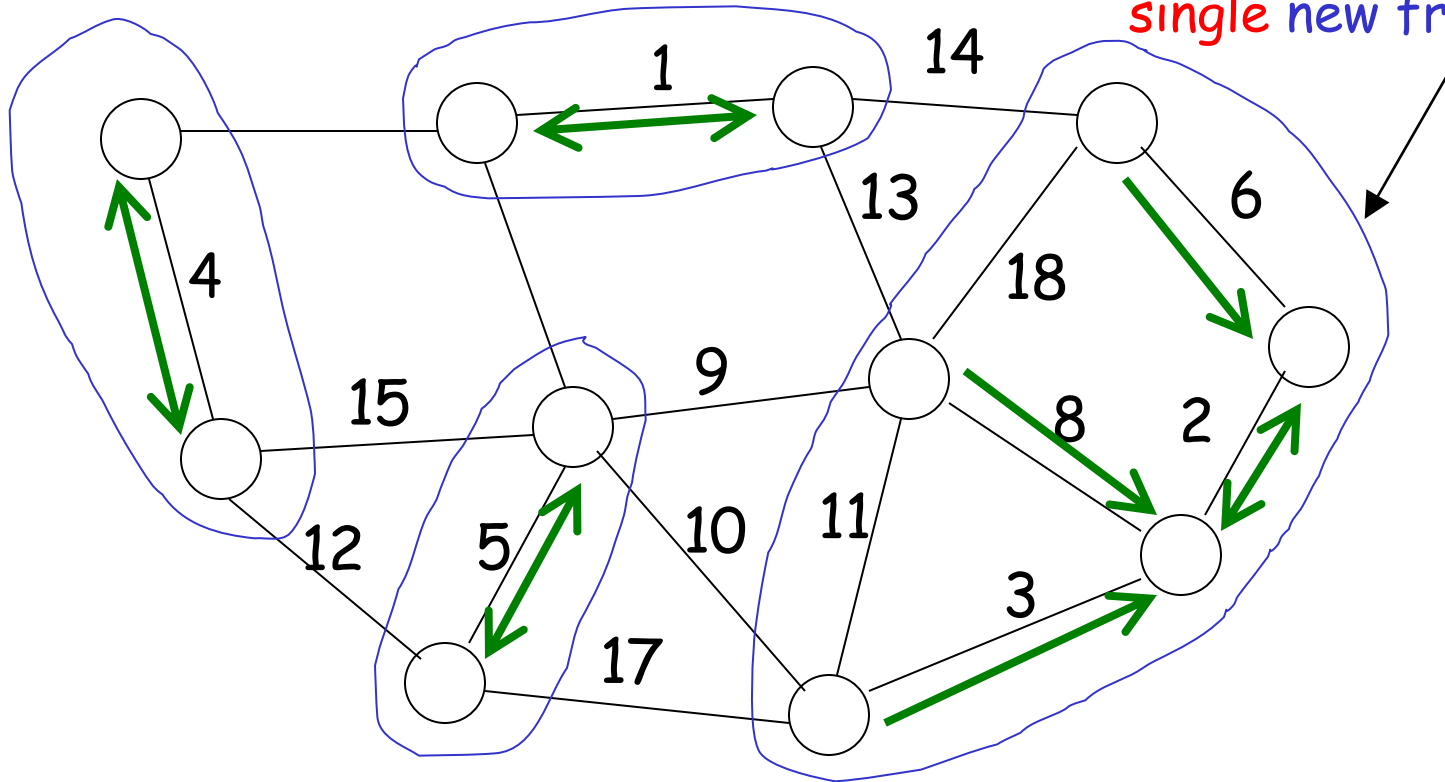


Phase 1: In our example, each node finds its MOE and sends a **Connect** message (arrows denote the direction of the **Connect** message)



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

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



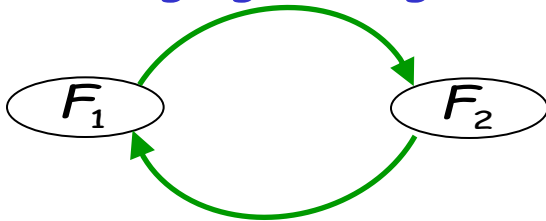
Question: How do we select the **new roots**?

Selecting a **new root**: a useful property

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

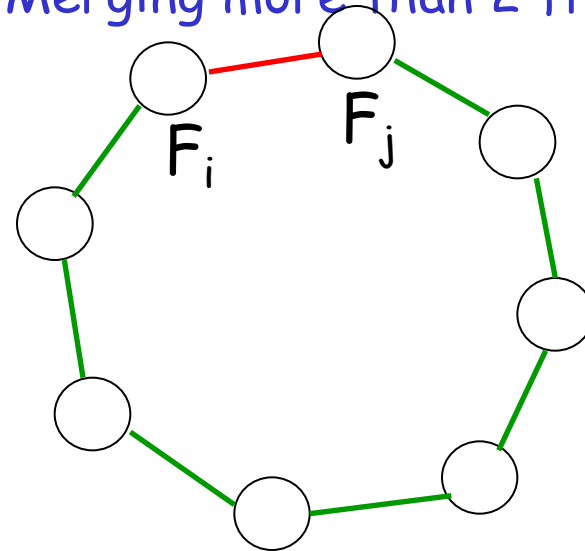
Proof: Recall that each merging fragment has exactly 1 MOE. Observe that since edge weights are distinct, then no cycles are created during merging. Indeed, for the sake of contradiction, assume this is false. We can have two cases:

Merging 2 fragments



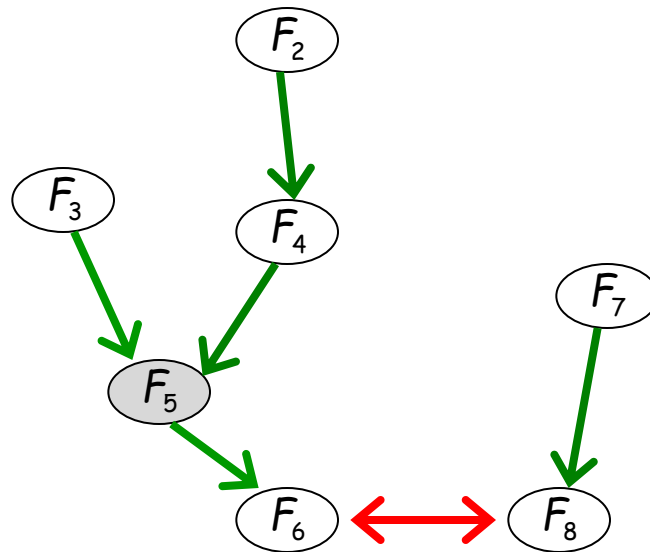
Impossible: either F_1 or F_2 is choosing a wrong MOE!

Merging more than 2 fragments



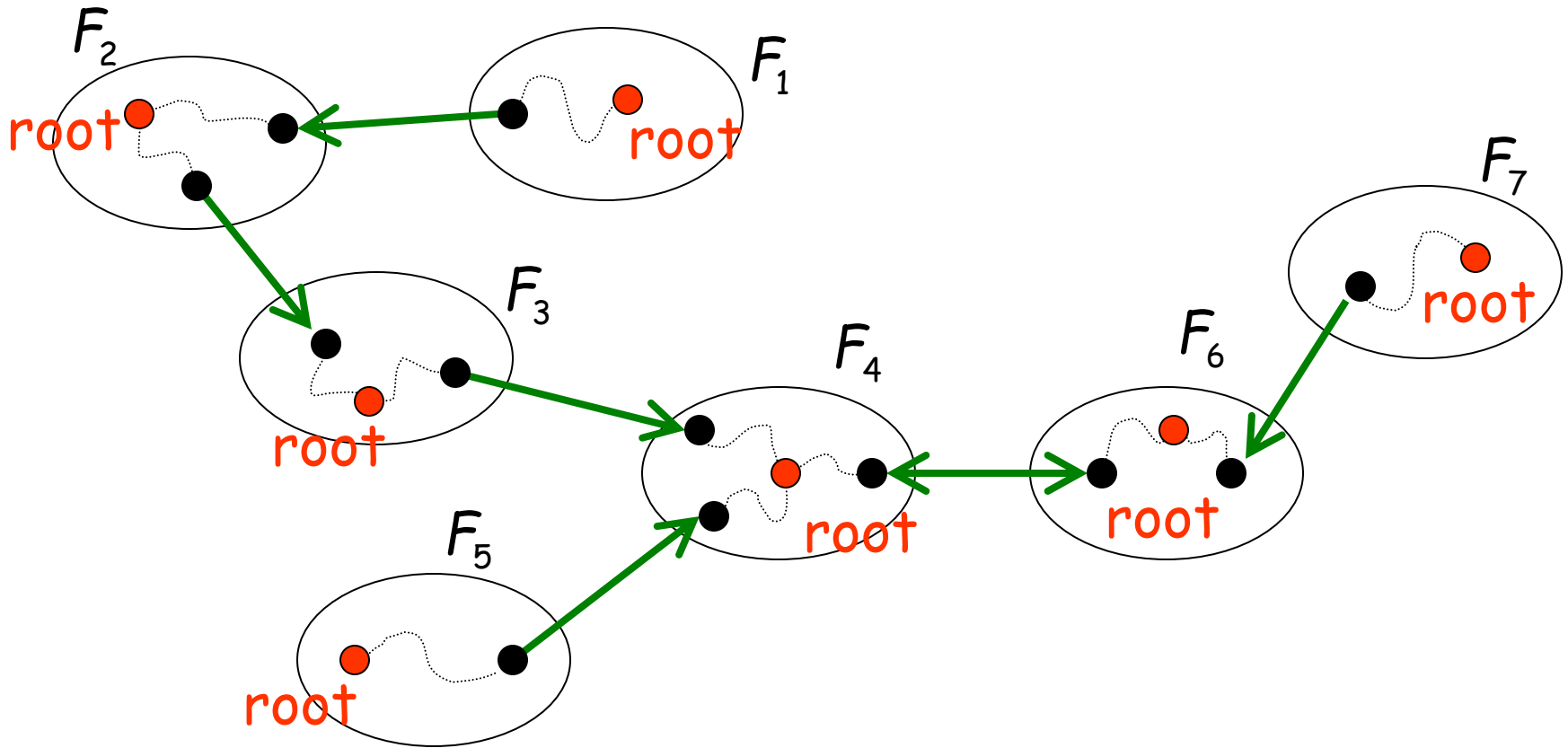
Impossible: let (F_i, F_j) be the (only) max-weight edge on the cycle; then either F_i or F_j is choosing a wrong MOE!

(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.

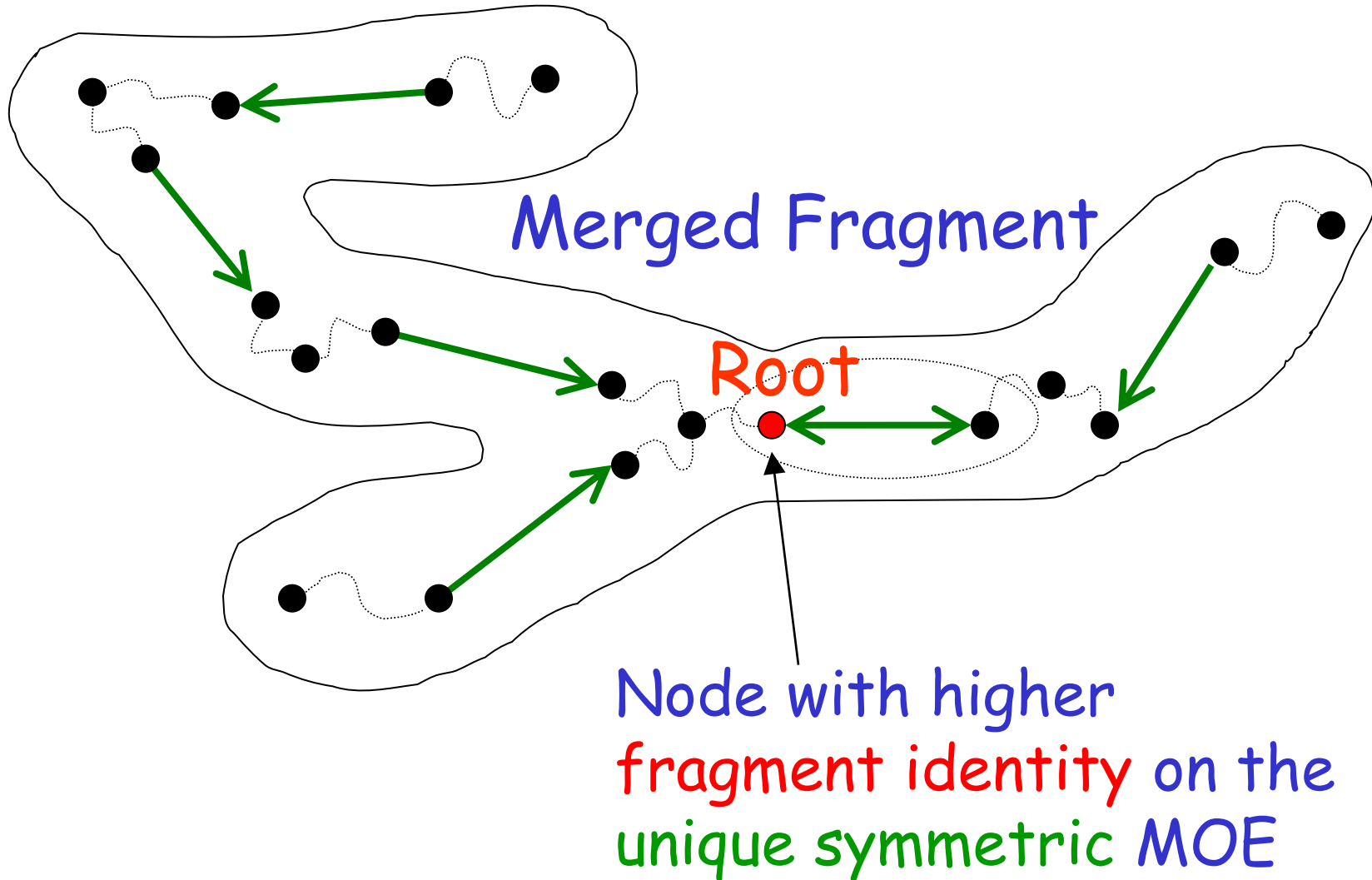


QED

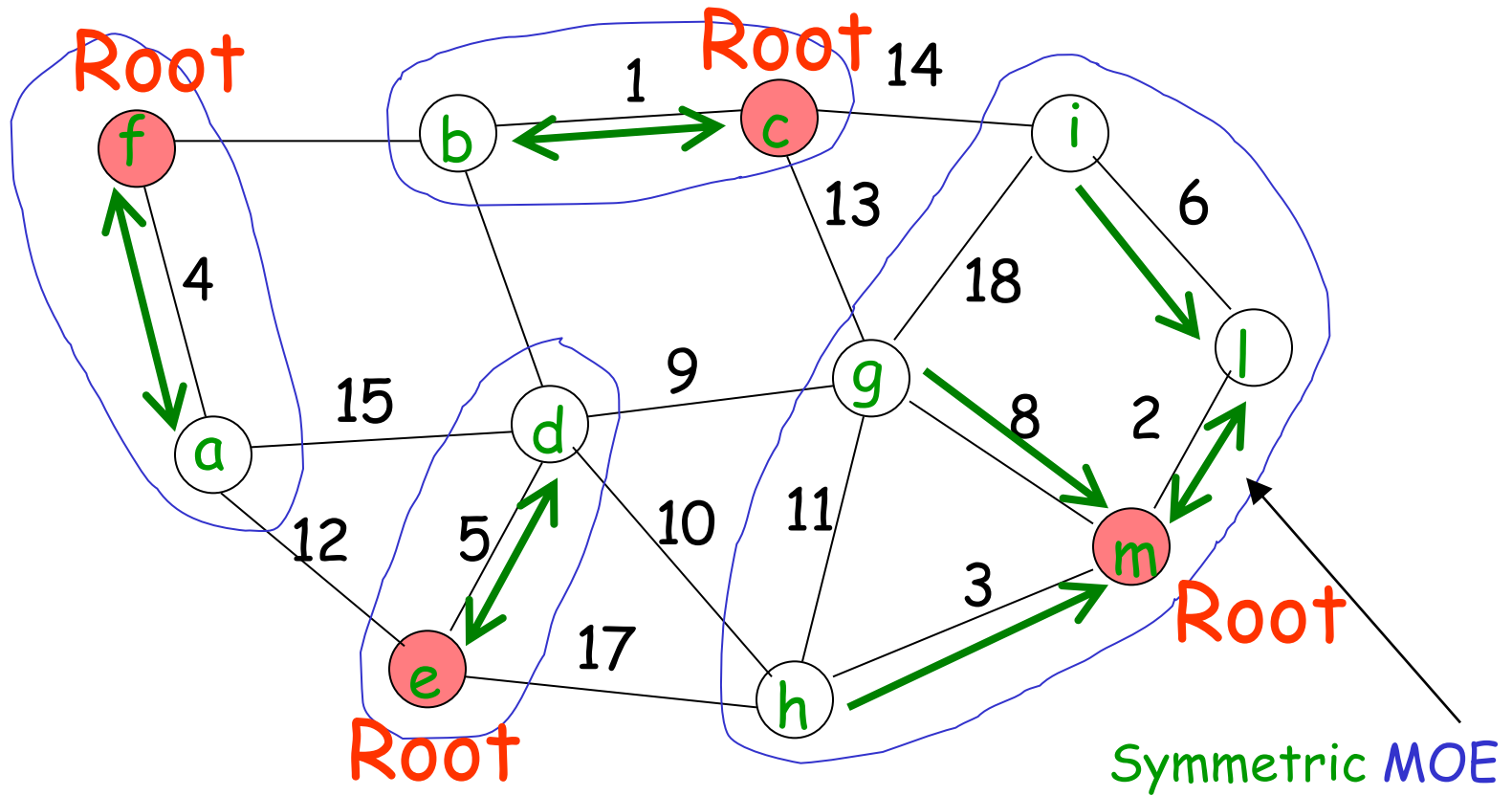
Rule for selecting a **new root** in a fragment



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

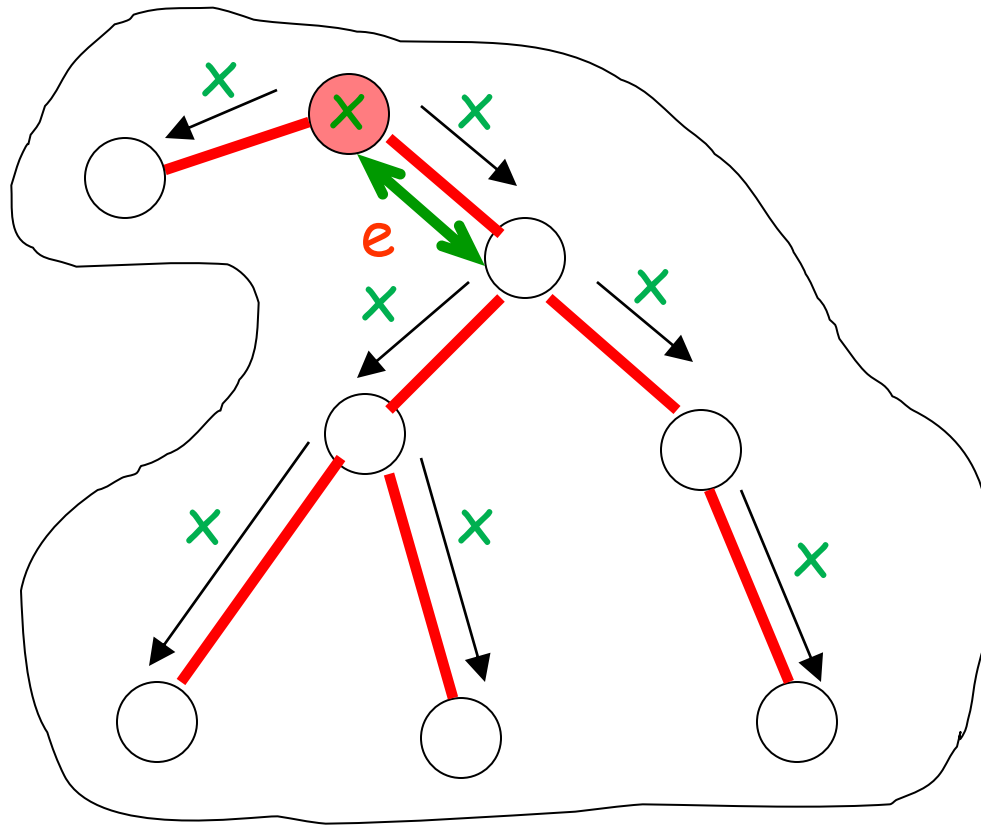


In our example, after the merges, each fragment has its **new root**



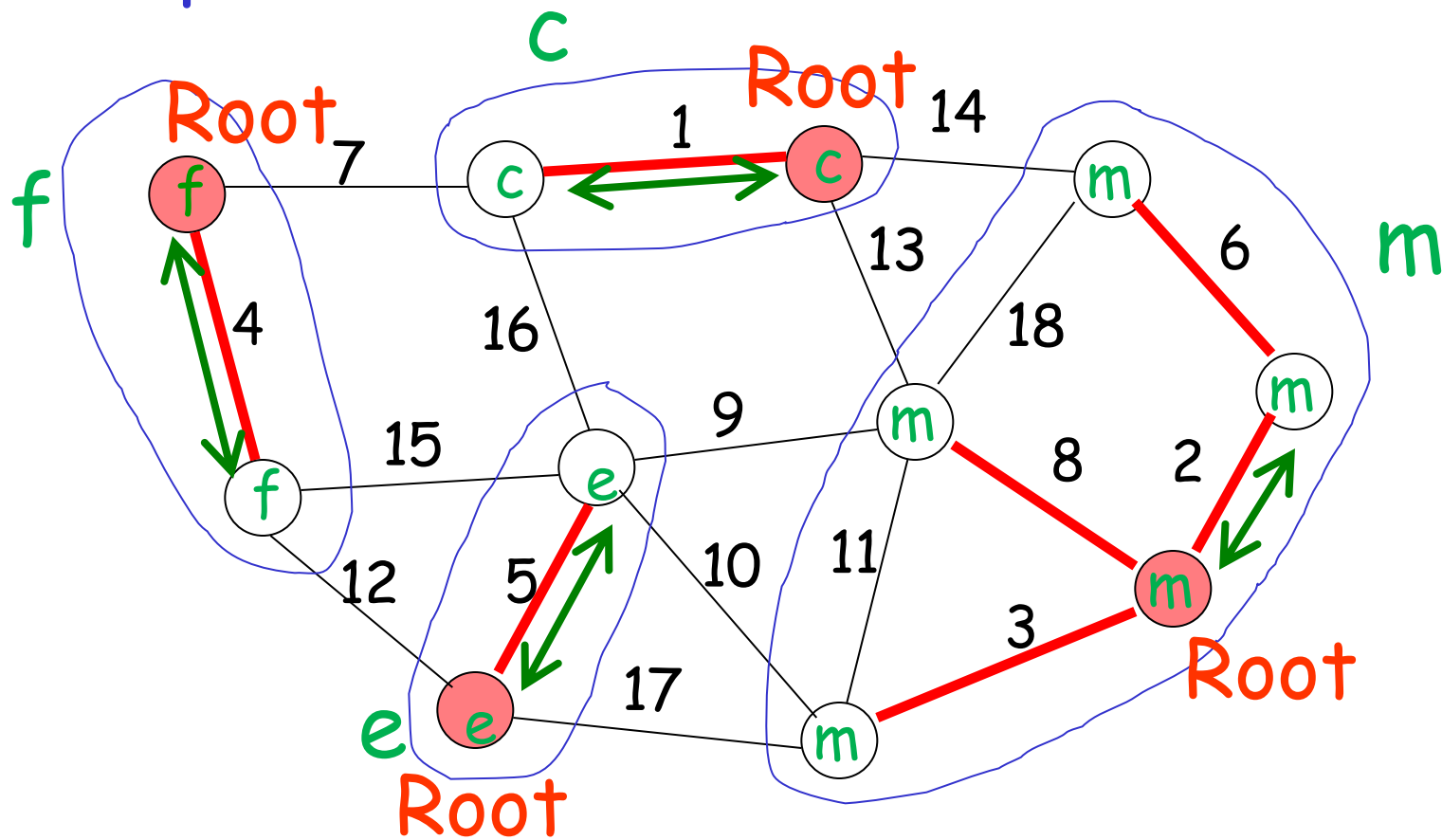
After merging has taken place, the new root broadcasts **New_fragment(x)** to all the nodes in the new fragment, where x is the previous fragment identity of the new root, and once this notification is completed a **new phase** starts

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



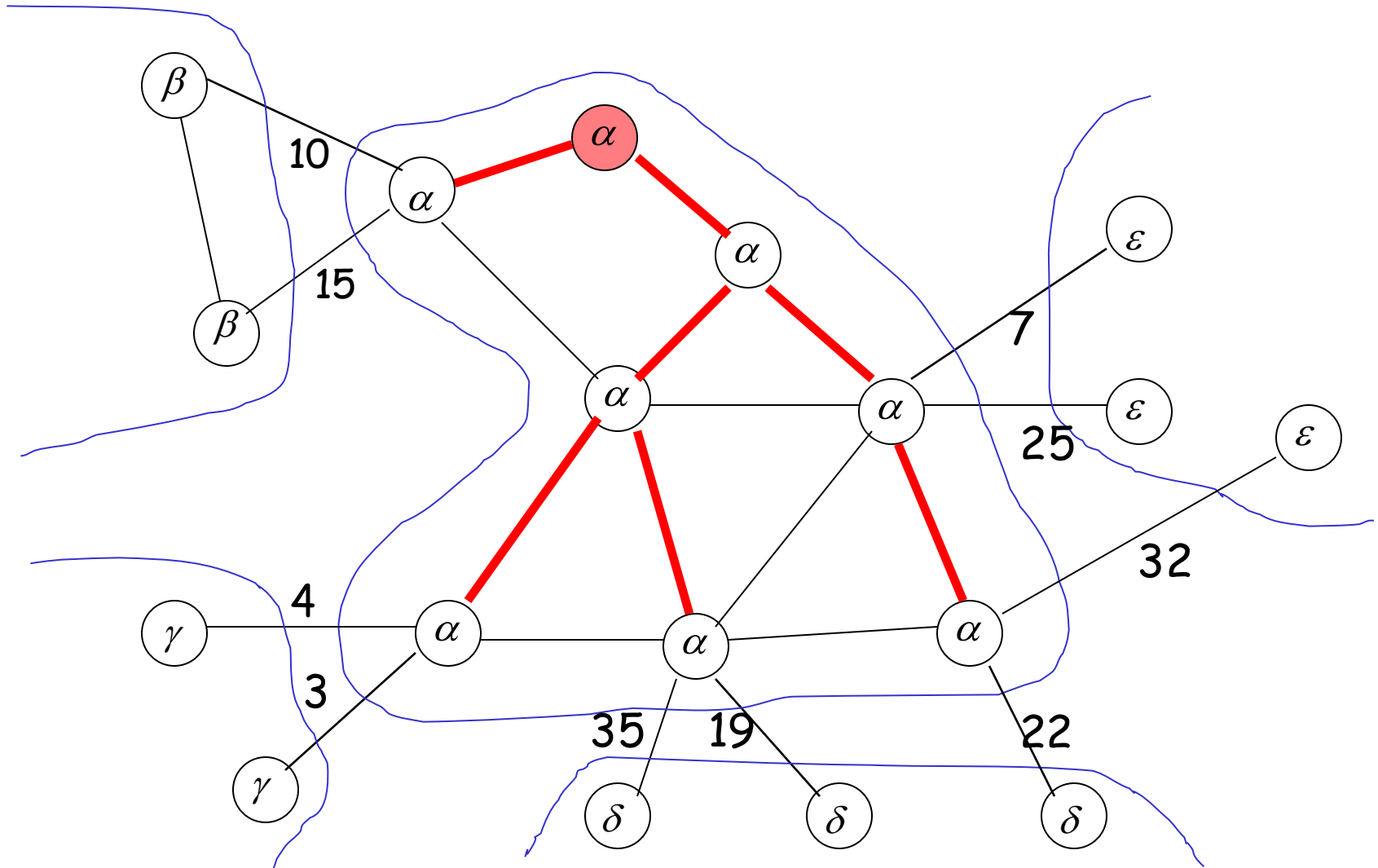
x is the identity of the new fragment

In our example, at the end of phase 1 each fragment has its new fragment identity, and a new phase can start

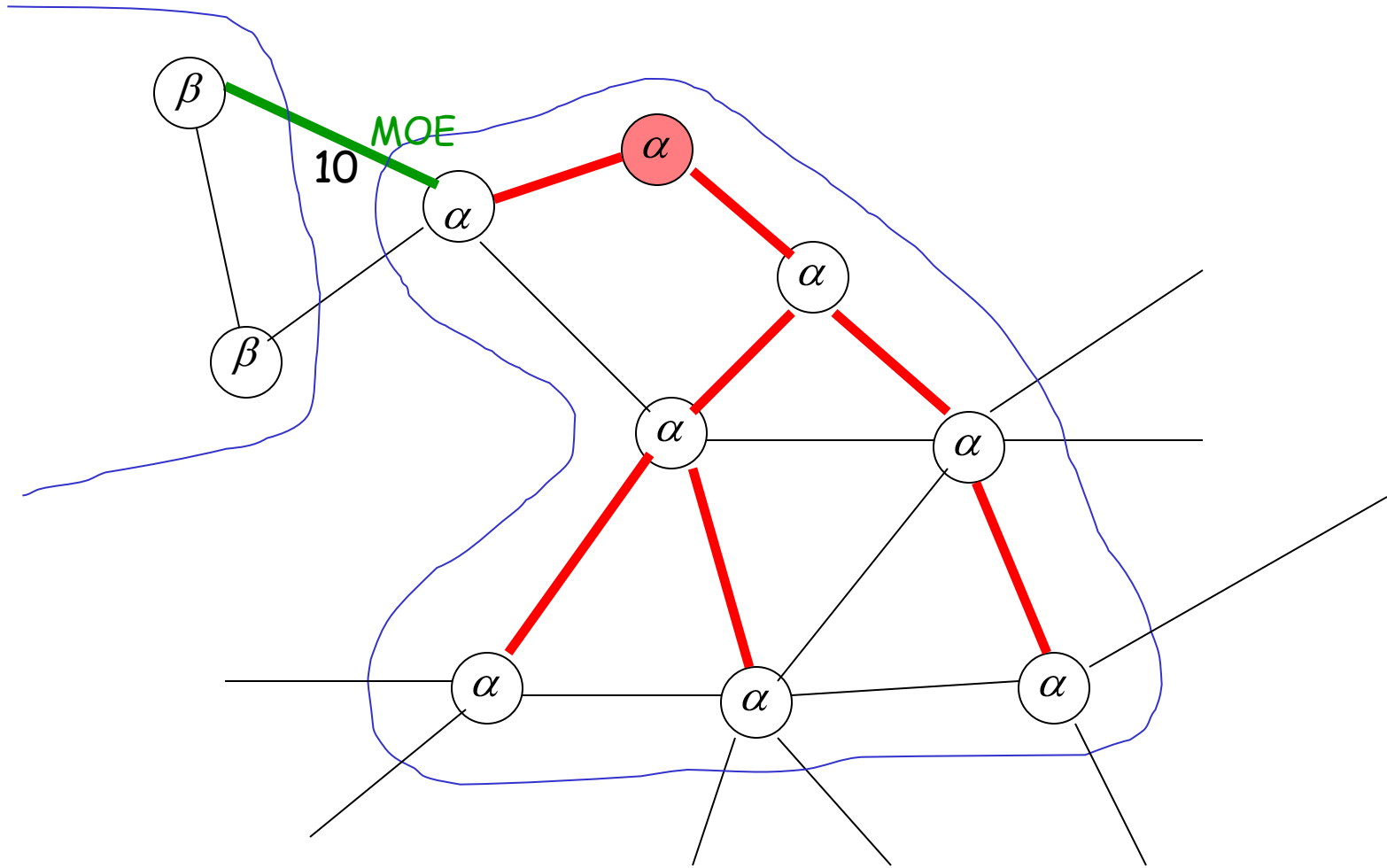


End of phase 1

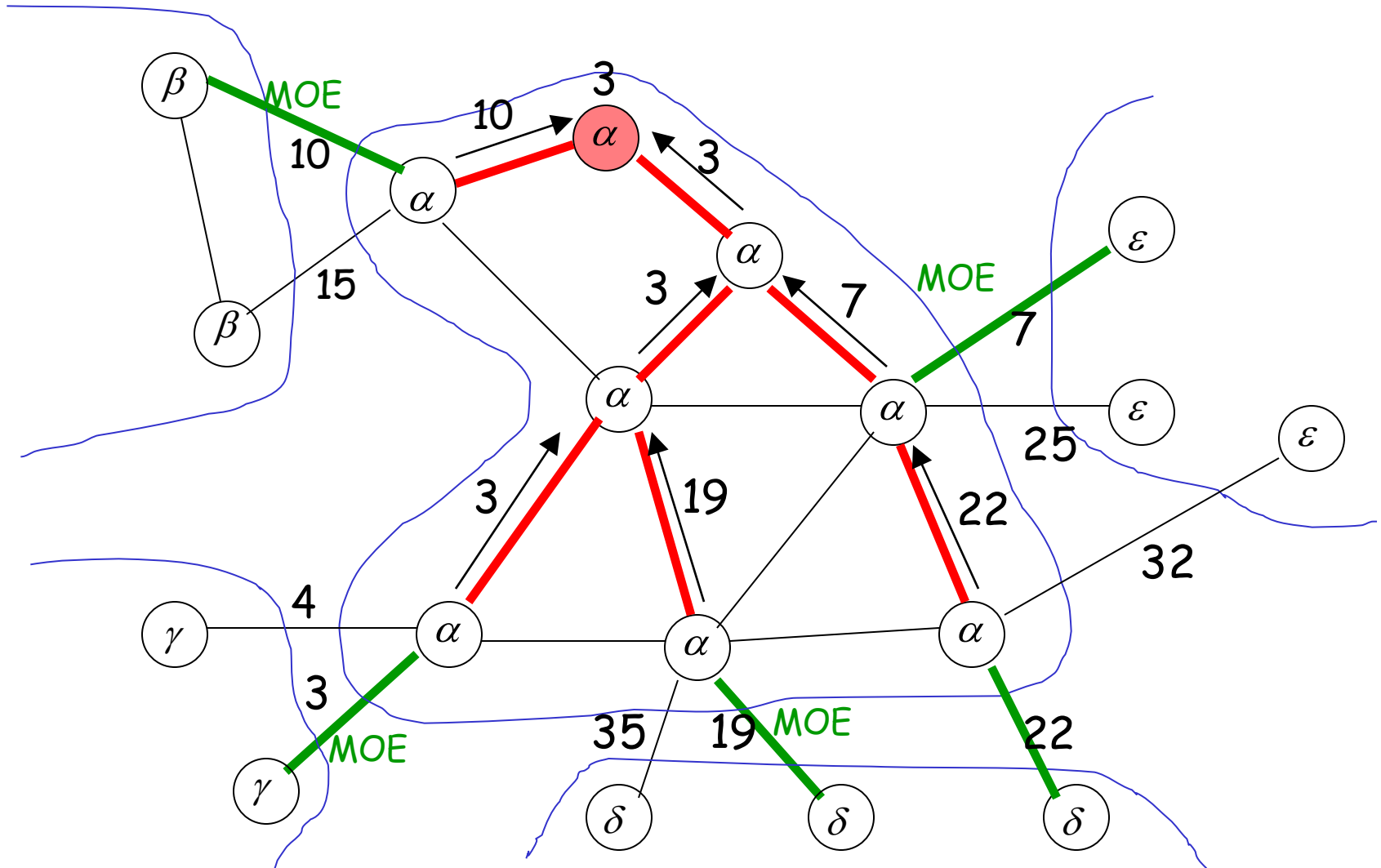
At the beginning of each new phase each node in a fragment starts finding its local MOE



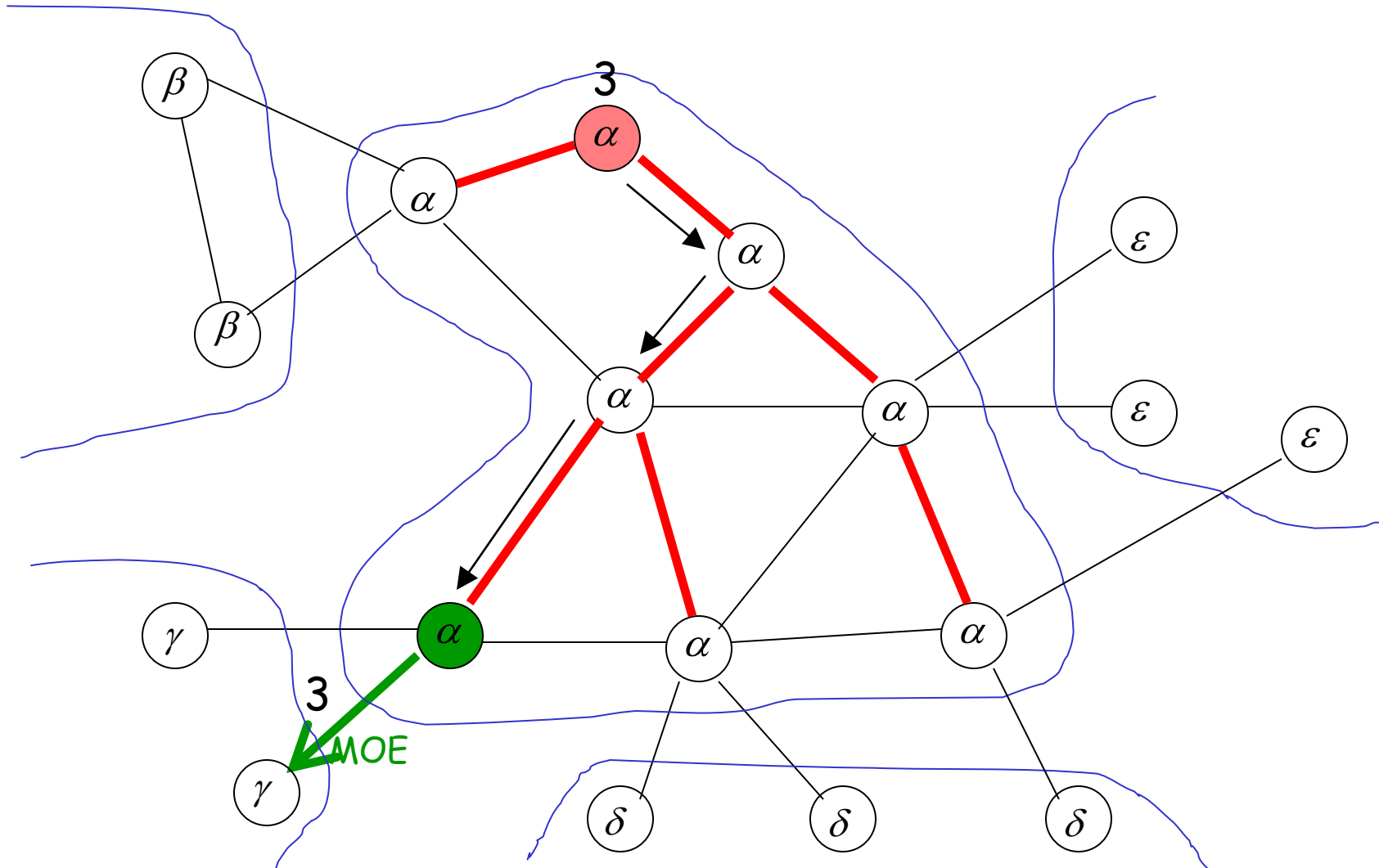
Then it knows its local MOE (notice this can be void)



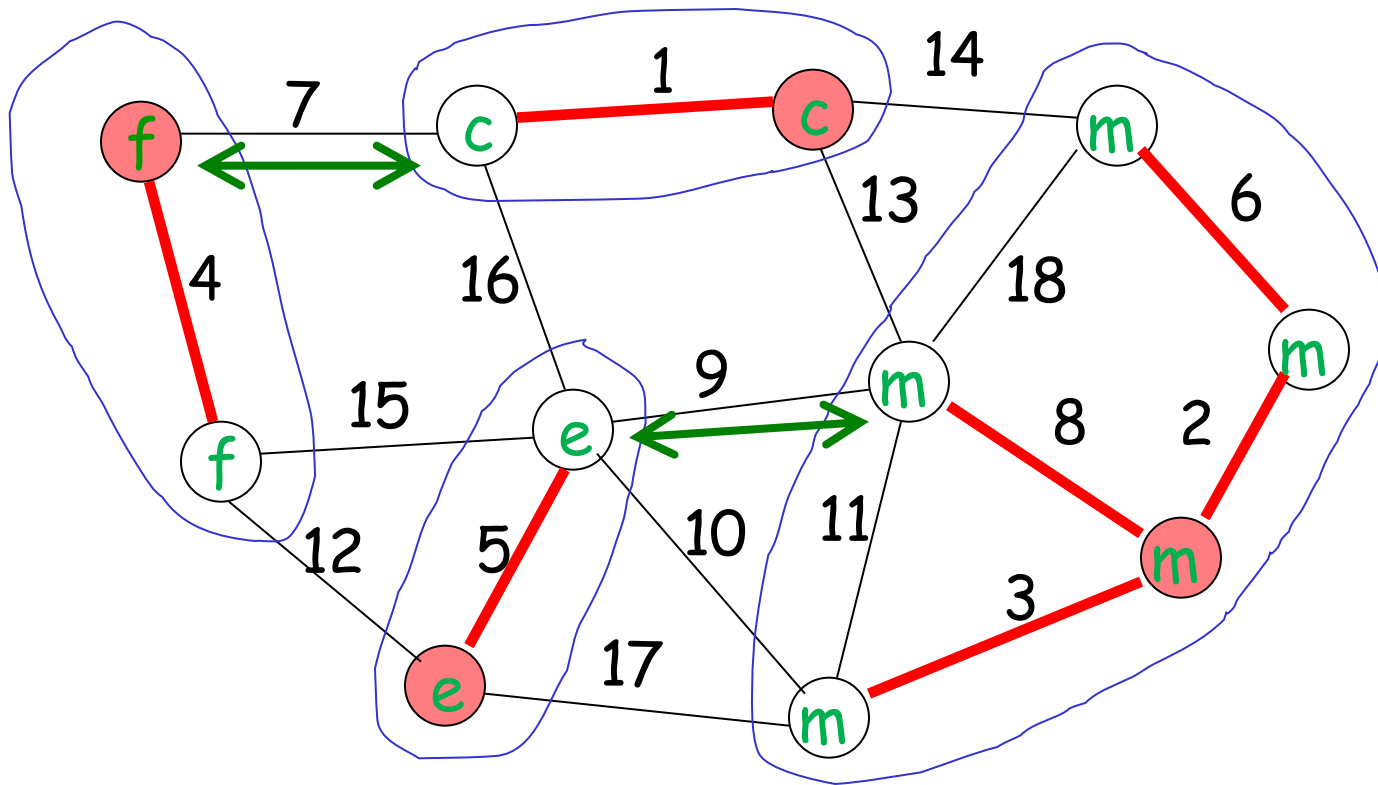
After receiving the **Report** from each child, a node sends its own **Report** to its parent with the MOE of the appended subfragment (the global minimum survives in propagation)



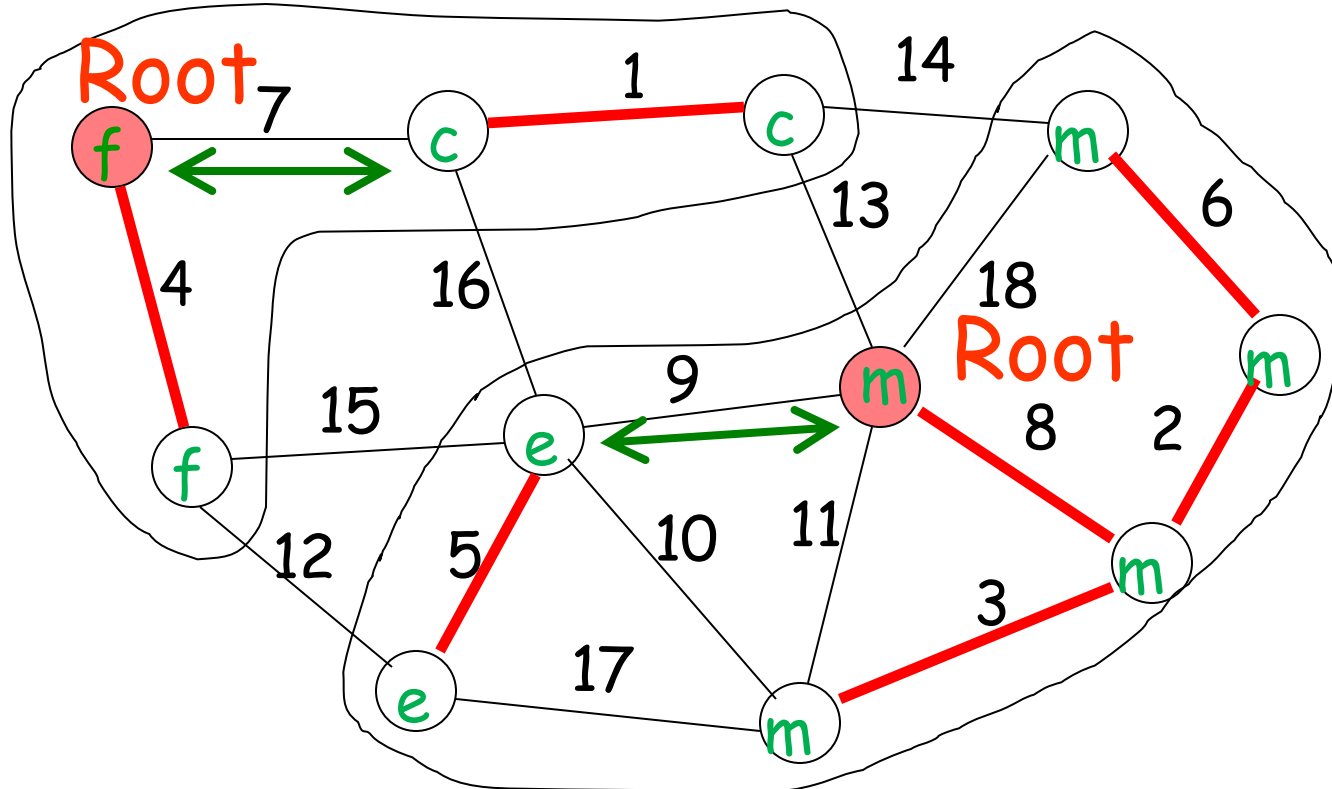
After receiving the **Report** from each child, the root selects the minimum MOE and sends along the **appropriate path** a **Merge** message, which will become a **Connect(α)** message at the proper node (which possibly becomes a **new root**)



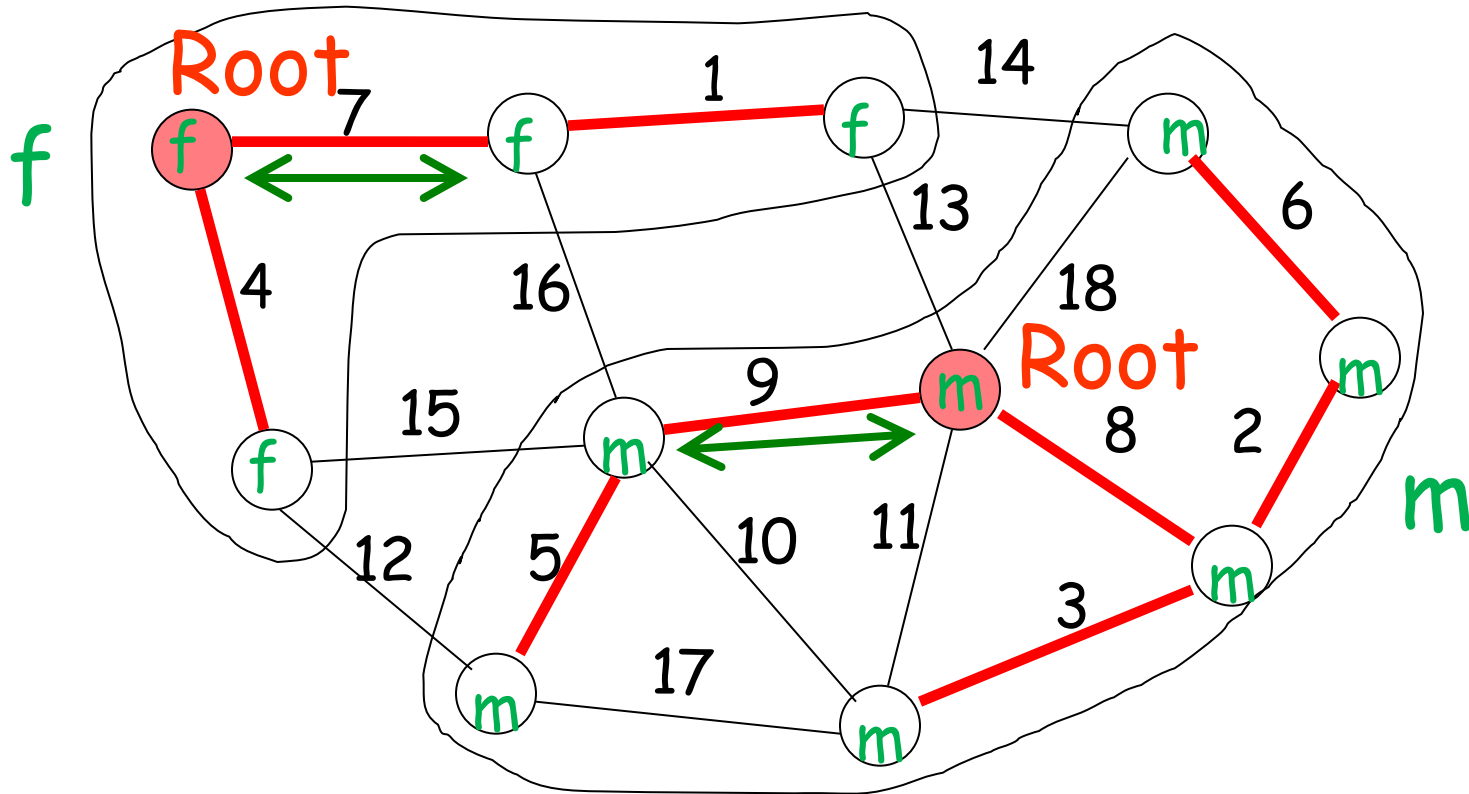
Phase 2 of our example: After receiving the new fragment identity at the end of the previous phase, find again the MOE for each fragment



Phase 2: Merge the fragments and select a new root

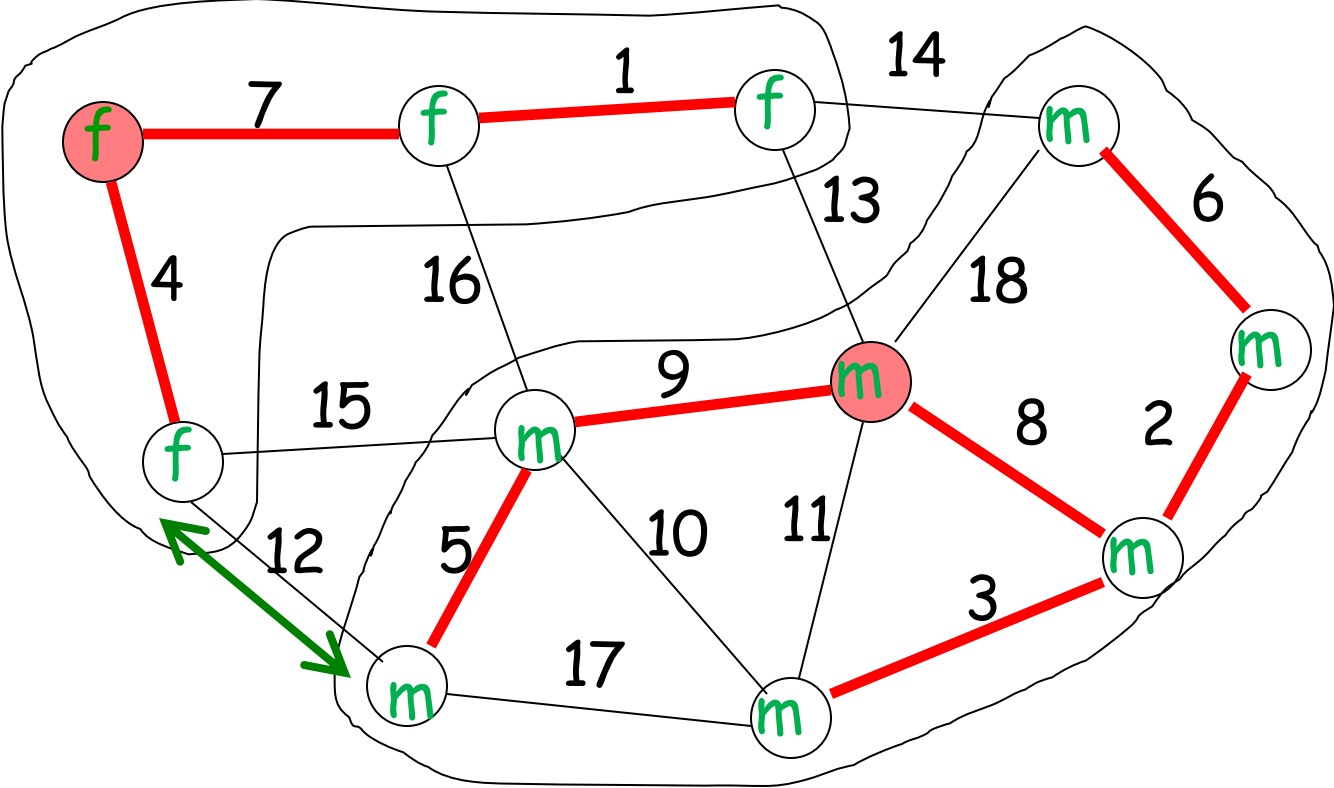


Broadcast the new fragment identity to all the nodes in the new fragment, and so at the end of phase 2 each node knows its own new fragment identity.

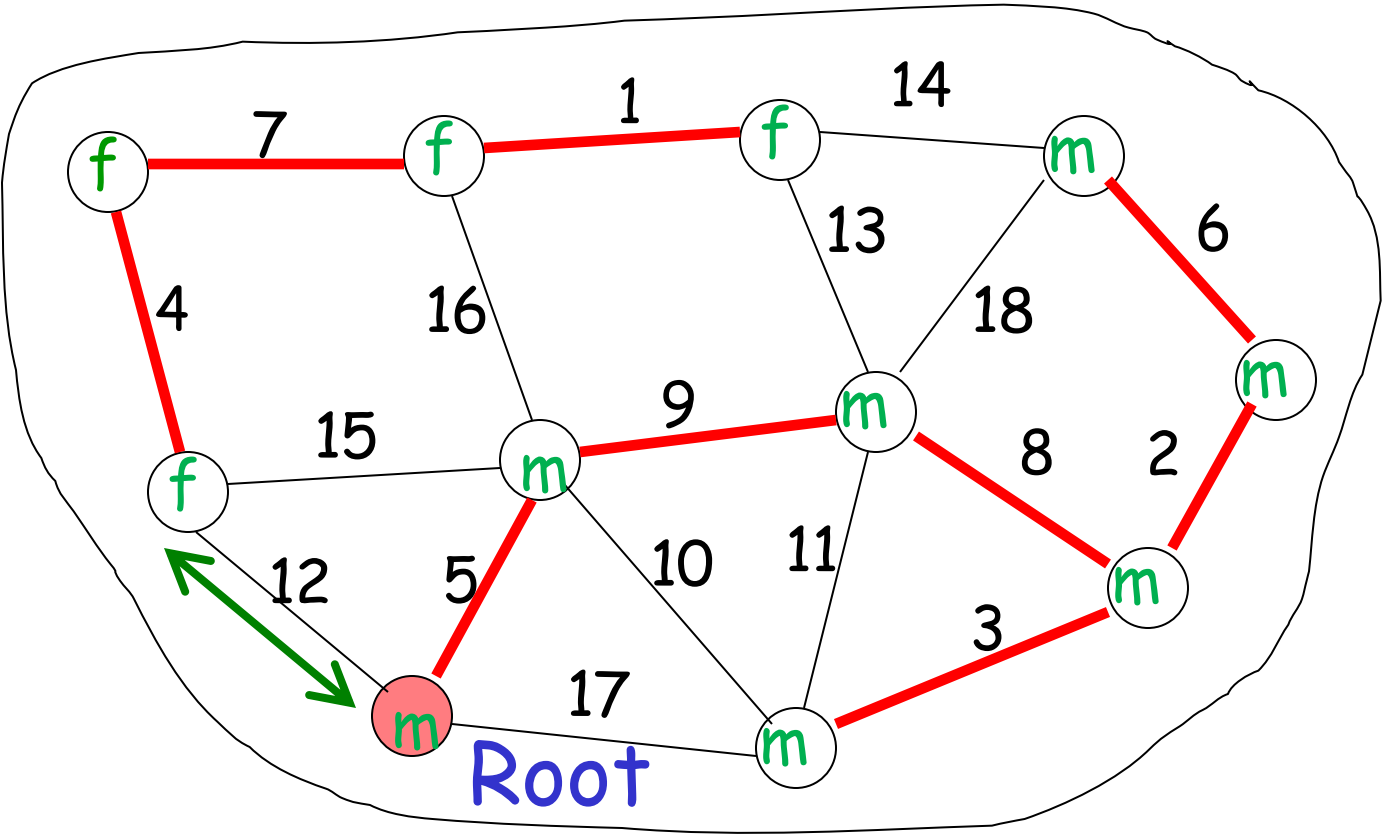


End of phase 2

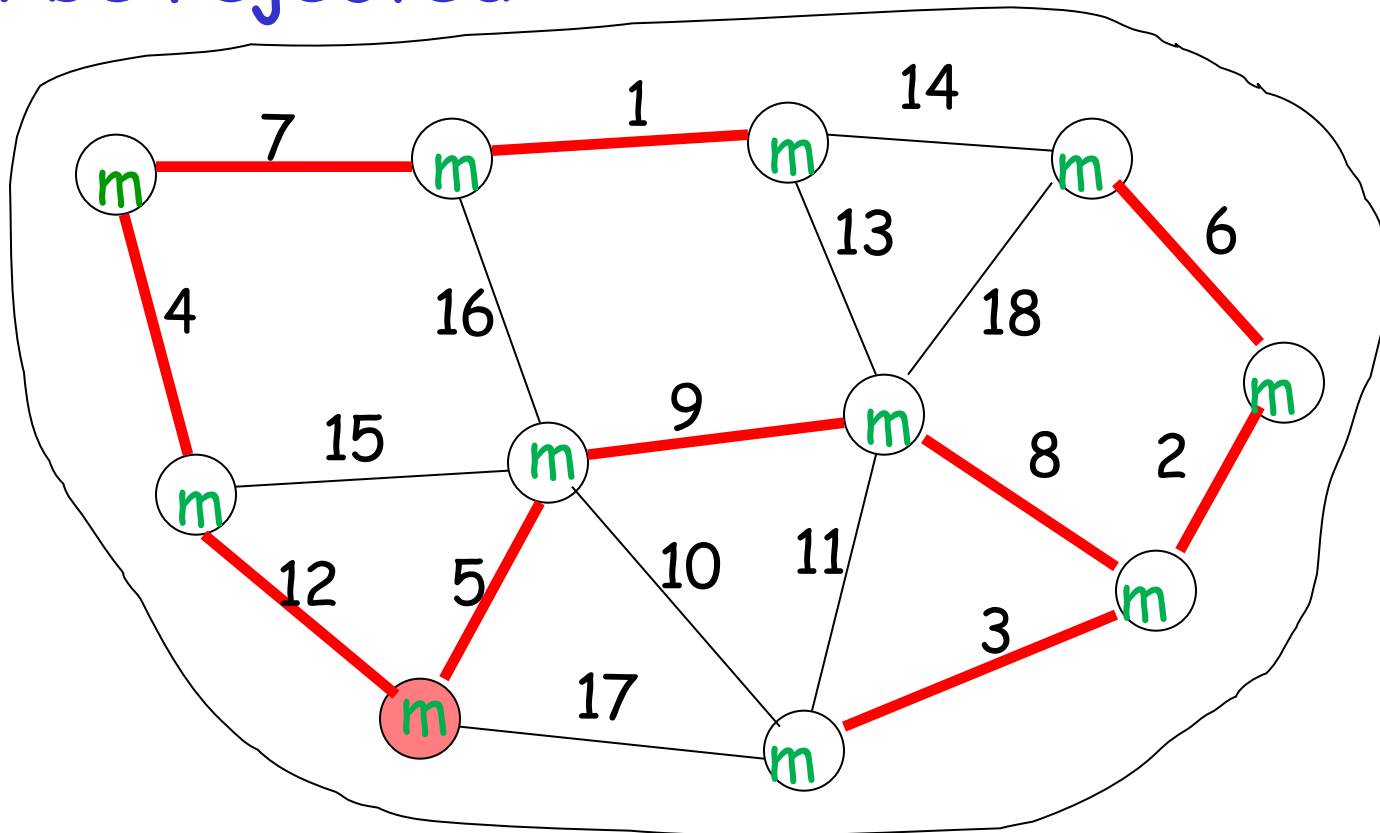
Phase 3: Find the MOE for each fragment



Phase 3: Merge the fragments



At the beginning of Phase 4, every node will start searching its MOE, but all the edges will be rejected



⇒ the source node realizes this is the FINAL MST

Synchronicity

- 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 \Rightarrow fragments and phases need to be "synchronized"

Phases' synchronization

- First of all, observe that each node has at most $n-1$ incident edges, and so the **local MOE selection** requires at most n rounds \Rightarrow we assign exactly n rounds to the local MOE selection (this means that if a node discovers a local MOE in less than n rounds, it will wait in any case till n 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 $3n$ of a phase
 - Finally, **exactly** at round $3n+1$ 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 $4n+1$ 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

⋮

⋮

i

2^i

Algorithm Time Complexity (# rounds)

Maximum possible fragment size $2^i = n$

Number of nodes

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



$O(\log n)$

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

$4n + 1$ 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**:

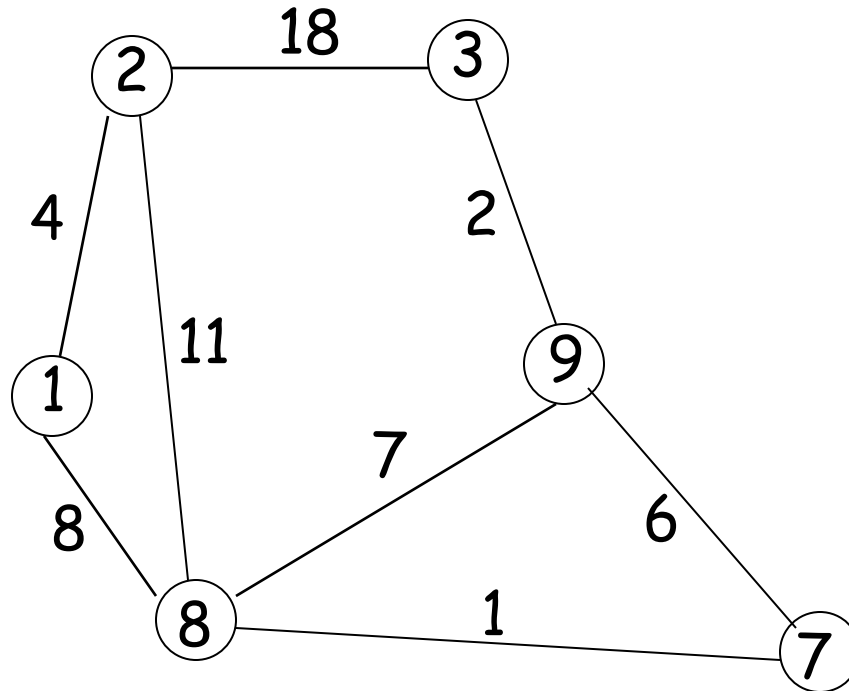
which means that globally circulate $O(n)$ messages in each phase (and in particular, on the branch edges of a fragment, circulate a constant number of messages in each phase).

Since we have at most $\log n$ phases, the claim follows.

END OF PROOF 80

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 properties of the merging fragments, as we will describe soon:
absorption and **join**

Local description of asynchr. GHS

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

1. The **Fragment identity** is given by a pair:

(**ID** of the **root node**, **level** of the fragment)

where 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 level of a fragment **increases** and depends on the level of the fragments which created it, as we will see.

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**}

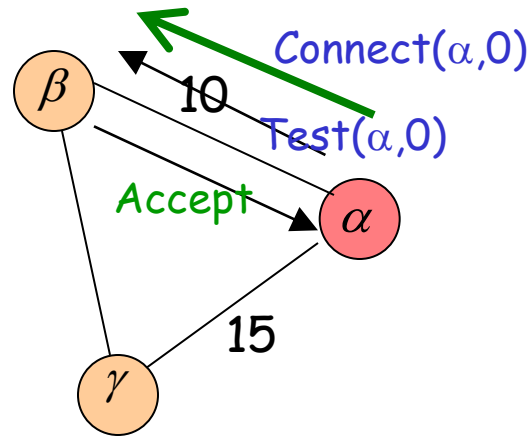
Type of messages

Similar to the synchronous case, but now:

1. **New_fragment(id,level,status)**: coordination message sent just after a merge
2. **Test(id,level)**: to test an edge; when a node is testing an edge, it is in a **finding** state
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; as we will see, this message will only travel from a fragment of level **L** to a fragment of level **$L' \geq L$**

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** (we will see this is a crucial point)



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

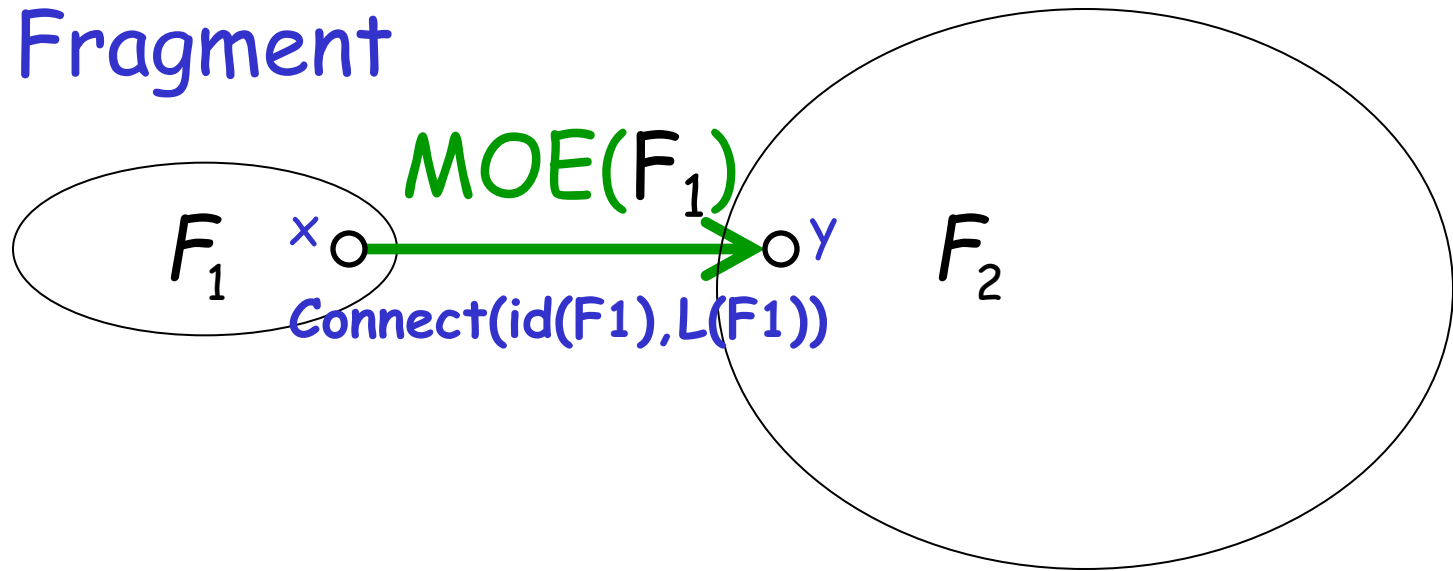
Actions taken by a node of level $d > 0$

- 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**(root node ID, level) over its **basic** edges in increasing order of weight, until it will receive an **Accept**.
- As we will see, if the level of the node receiving a **Test** message is larger than that of the querying node, the reply will be delayed; however, once that a node receives an **Accept** (or 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**(root node ID, level) message at the proper node.
- Once again, depending on the fragment identity of the other endnode, some kind of merge will take place, as we will see on the next slide.

Merge as an Absorption

Fragment

Fragment

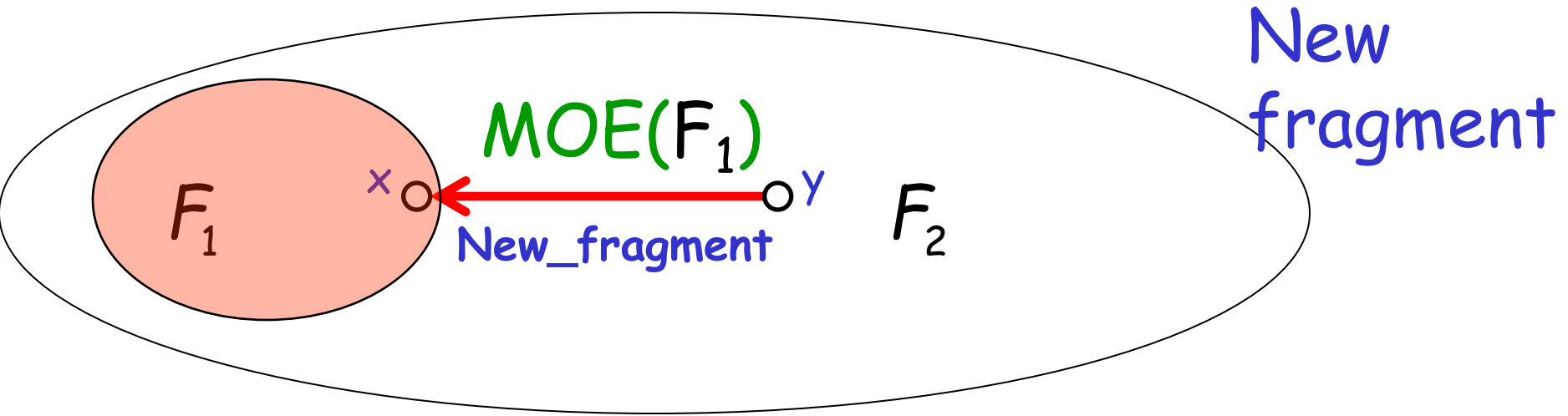


If $L(F_1) < L(F_2)$, then F_2 absorbs F_1

If $L(F_1) = L(F_2)$, then F_2 will "freeze" the message

Case $L(F_1) > L(F_2)$ 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: if $L(F_1) < L(F_2)$, then a "new" fragment is created with the **same identity** as F_2



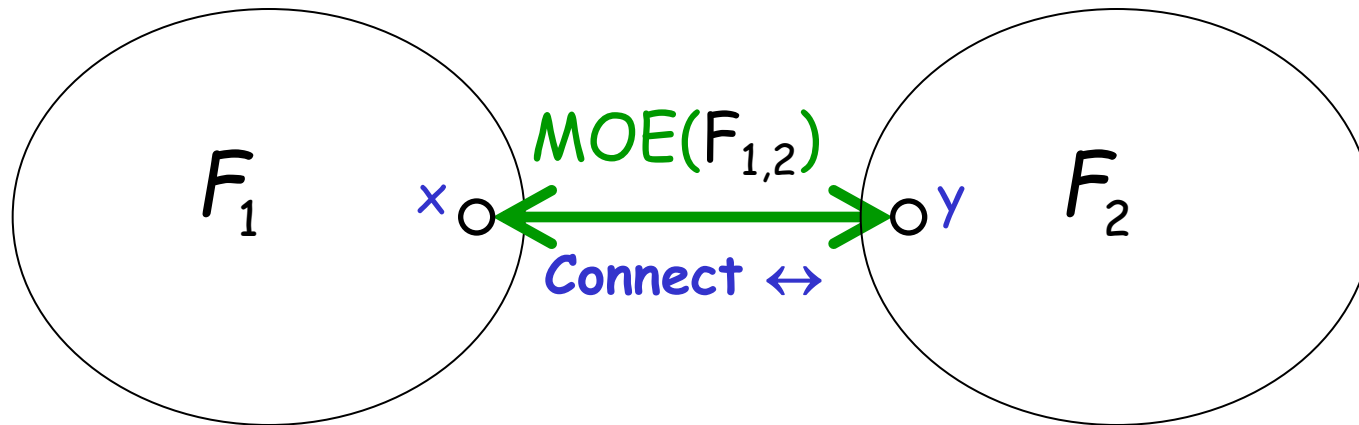
Actually, a `New_fragment(ID(F_2), $L(F_2)$, status(y))` message is **broadcasted** to nodes of F_1 by the node y of F_2 on which the merge took place

(cost of merging, in terms of number of messages, proportional to the size of F_1)

Merge as a Join

Fragment

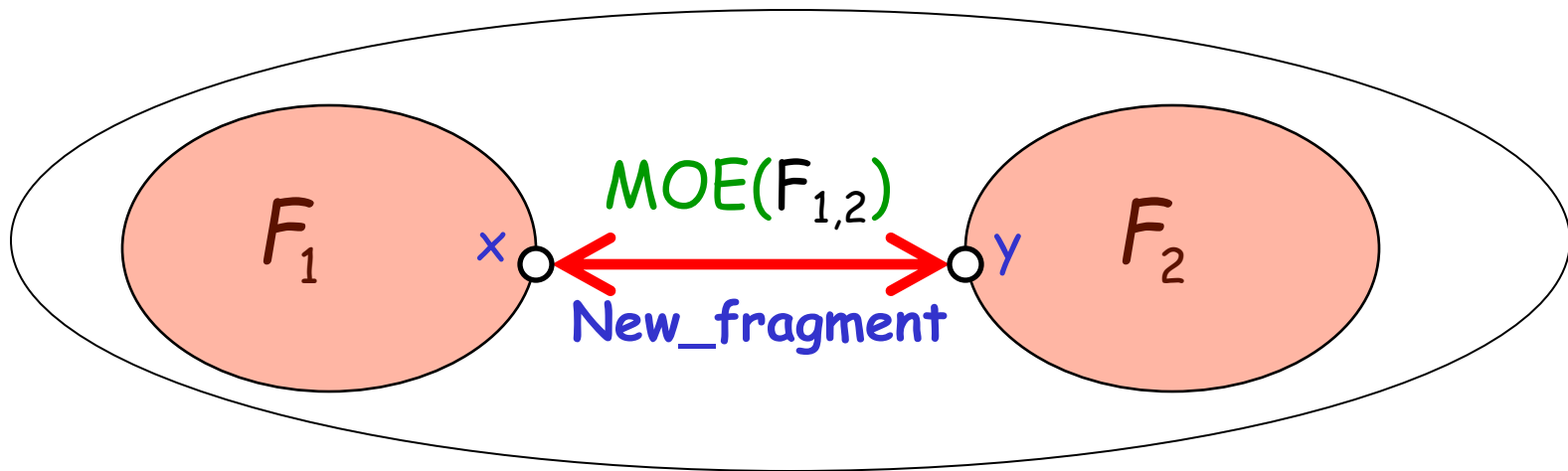
Fragment



A symmetric MOE will generate a bidirectional **Connect** message on the same edge. This can happen iff $L(F_1)=L(F_2)$, as sketched before, and in this case, F_1 **joins** with F_2

Notice that the system is asynchronous, and so the **Connect** message on the two directions may be not simultaneous (differently from sync GHS) ⁹⁰

The combined level is $L(F) = L(F_{1,2}) + 1$
New fragment

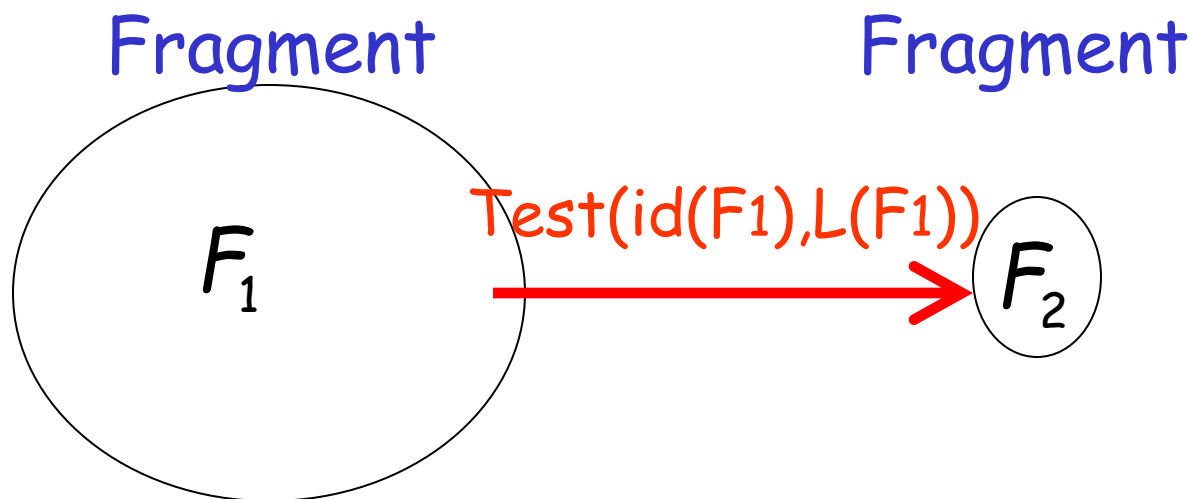


and a **New_fragment**($\max(\text{ID}(F_1), \text{ID}(F_2)), L(F_{1,2}) + 1, \text{finding}$) message is **broadcasted** to all nodes of F_1 and F_2 by the **new root**, i.e., the node with **max ID field** in the fragment identity between x and y

(cost of merging, in terms of number of messages, is proportional to the size of $F_1 \cup F_2$)

Remark: by definition, 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). How do we control this?

⇒ this is obtained by letting F_2 not replying to **Test** messages from F_1 , in case $L(F_1) > L(F_2)$, until $L(F_1) = L(F_2)$ (this prevents F_1 to find its MOE, i.e., to ask a connection to F_2 , while $L(F_1) \leq L(F_2)$)



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. **Synchronization**: 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? (YES indeed)
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?

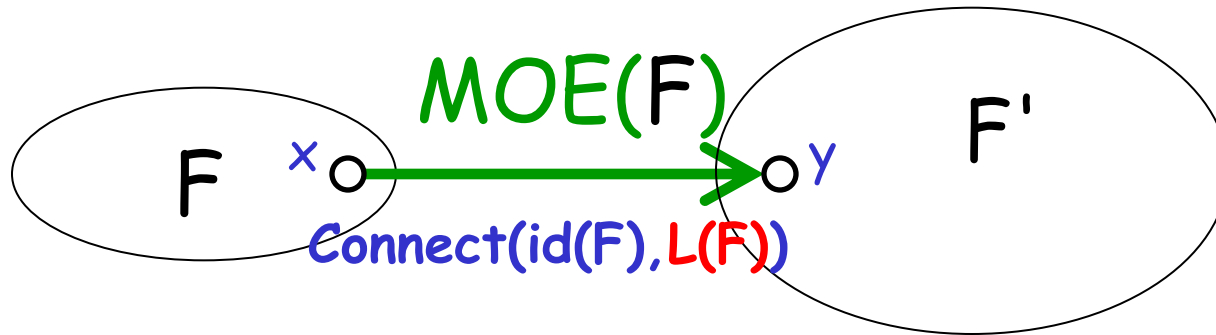
We will show formally only **termination**, while we only sketch the proof for **point 3** (point 2 is indeed very complicated and we omit it)

1. Termination of asynchronous GHS

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 $L' \geq L$ 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 \Rightarrow this creates a new configuration, onto which the argument of the proof is applied recursively \Rightarrow 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 of asynchronous GHS (2)



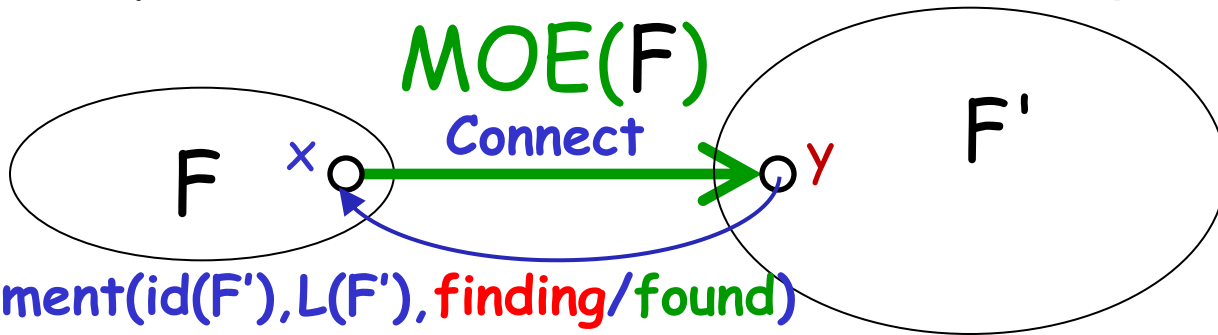
1. $L(F') > L(F)$: in this case F' absorbs F ;
2. $L(F') = L(F)$: in this case, since (x, y) is a **lightest MOE**, then it is also the MOE of F' (recall that edge weights are distinct) and F' cannot be locked (similarly to F); then, a join between F and F' takes place. **END OF PROOF**

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

3. Absorption while F' is searching for a MOE



$\text{New_Fragment}(\text{id}(F'), L(F'), \text{finding/found})$

1. Transmitted status is **finding**: in this case, nodes in F start searching for their local MOE, and node y will wait a **Report** from x before reporting to its parent in F' . Apparently, it is possible that a node u in F , after getting the new identity, tests a node v in 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 F do not participate to the selection of the MOE for $F \cup F'$, and then it seems that edges outgoing from 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 F will be heavier than (x, y) , no any of them can be the MOE of F' , and so correctness is guaranteed.

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

END OF PROOF

\Rightarrow 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, namely $O(m)$ 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 98

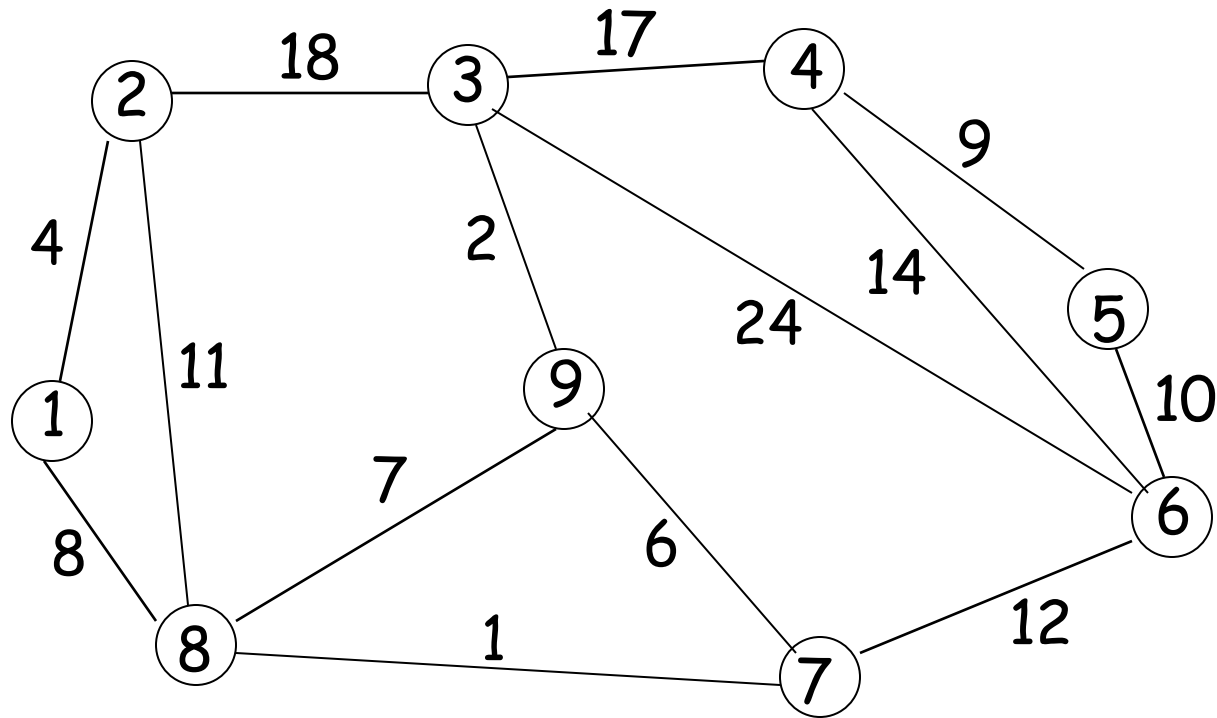
Summary of results for distributed MST

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)