分散式系統雲端與雲端應用開發實務

Distributed Minimum Spanning Tree

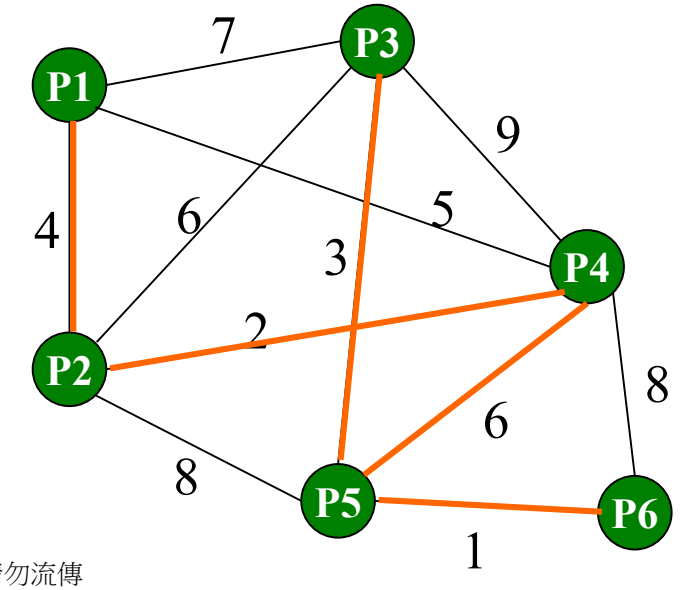
莊 裕 澤
Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

Minimum Spanning Tree (MST)

- MST: a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum total edge weight.
- How do you construct an MST in the network?
  - Sequential algorithm?
    - Kruskal's (1956), Prim's (1957)
      - Recall Dijkstra's shortest path algorithm (1959)
  - Distributed algorithm?

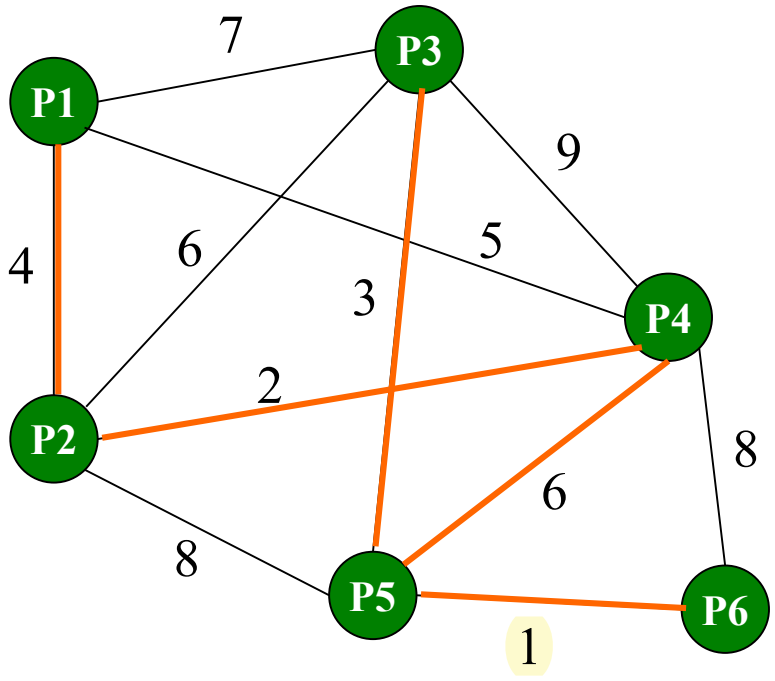
Complexity?

How to cope with failures?



Kruskal's Algorithm for MST

1. Given a connected graph G=(V, E), sort all the edges E in non-decreasing order of their weight. Initialize tree T to empty.
2. Pick the smallest edge *e* in E. If adding *e* to T does not result in a cycle, add *e* to T; otherwise discard *e*.
3. Repeat step 2. until are all vertices are in T.



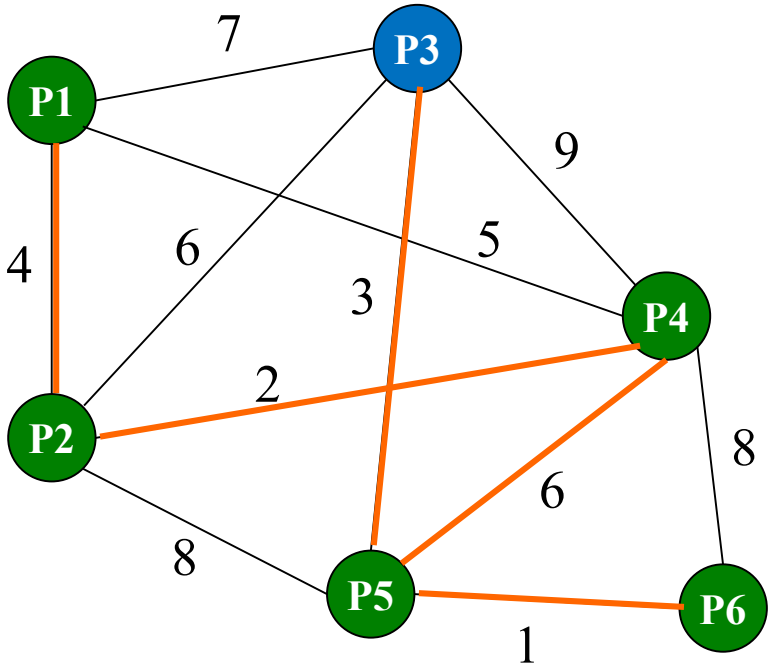
Can the idea of this algorithm be used in a distributed environment?

Why?

complexity: O(E log E)

Prim's Algorithm for MST

1. Given a connected graph G=(V, E), initialize a tree T with a single vertex, chosen arbitrarily from V.
2. Let *e* be minimum-weight edge that connects T to the vertice not yet in the tree, add *e* to T.
3. Repeat step 2 until all vertices are in the tree.

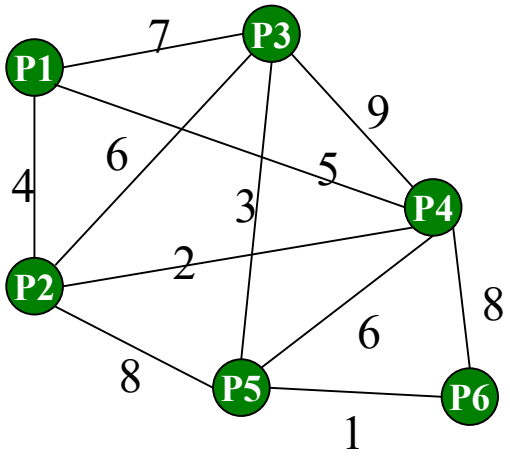


Can the idea of this algorithm be used in a distributed environment?

Why?

GHS algorithm for Distributed MST

- Assumption:
  - asynchronous message-passing model
  - messages are delivered in FIFO order in each edge
  - each node knows its own edges and weights
  - either edge weights are distinct, or nodes have distinct IDs
    - edges of the same weight can be ordered by the ids of their adjacent nodes, e.g., [6, (P2, P3)] < [6, (P4, P5)]
  - each node can spontaneously initiate the algorithm, or be awakened by neighbors upon receipt of messages

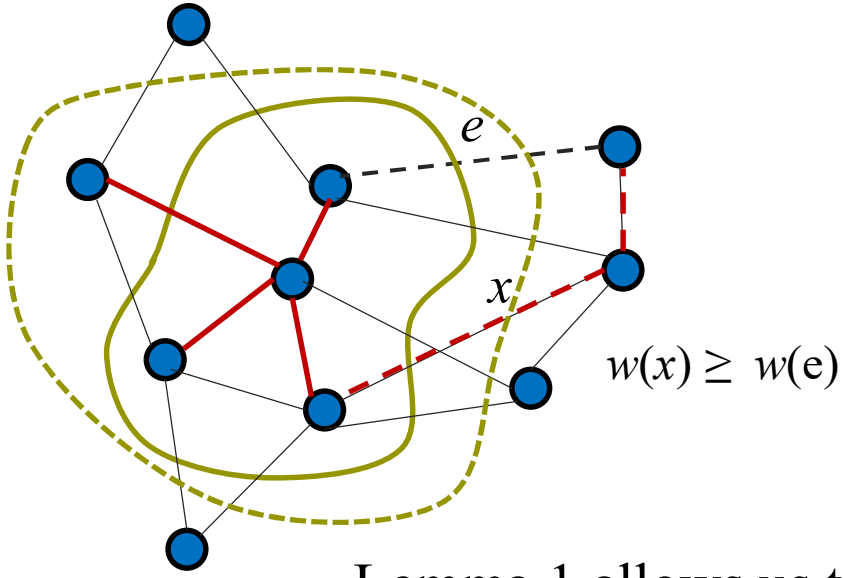


Robert G. Gallager, Pierre A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," ACM Transactions on Programming Languages and Systems, vol. 5, no. 1, pp. 66–77, Jan. 1983

An enhanced version by Guy Flysher and Amir Rubinshtein.

GHS Algorithm: Lemmas

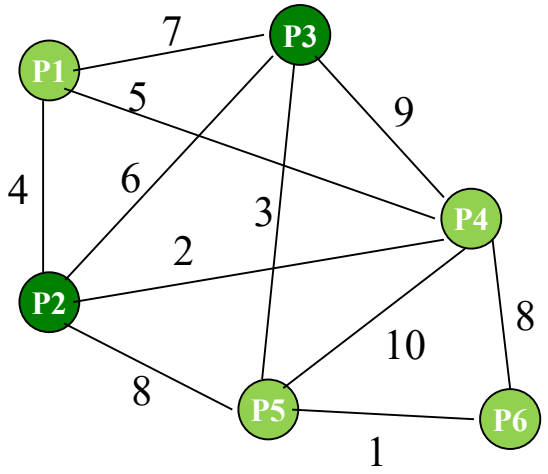
- Lemma 1: Given a fragment (subtree) of an MST, let e be a minimum-weight outgoing edge of the fragment. Then joining e and its adjacent non-fragment node to the fragment yields another fragment of an MST.
- Lemma 2: If all the edges of a connected graph have different weights, then the MST of the graph is unique.



Lemma 1 allows us to enlarge fragments in any order, and Lemma 2 assures that their union is also a fragment.

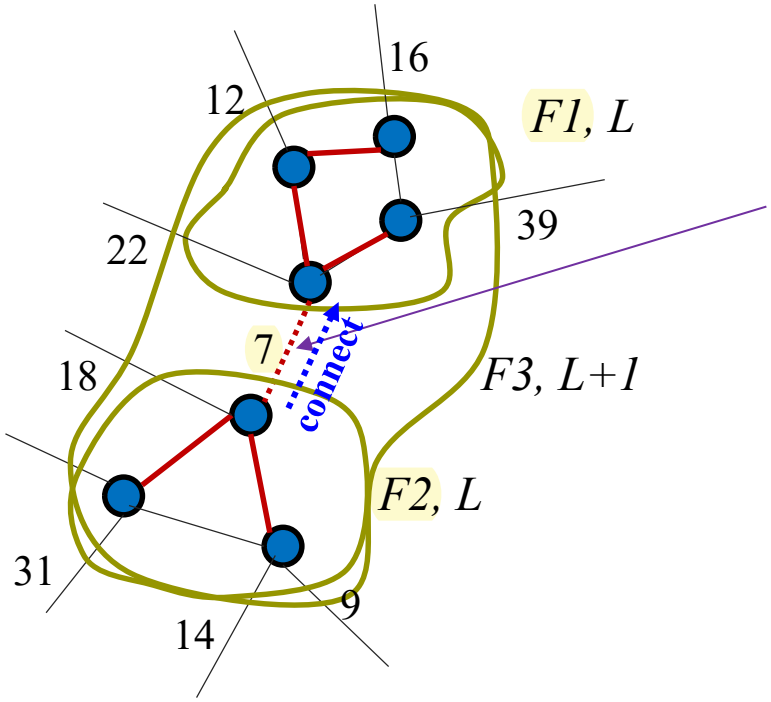
GHS Algorithm: Idea

- works in levels
  - each node is initially in level 0 with a fragment of itself
  - Members of each fragment cooperate to find the minimum-weight outgoing edge of their fragments.
  - When fragments meet, they are combined to form a larger one, and the process continues until an MST is formed.



Fragments Merge & Absorb (1)

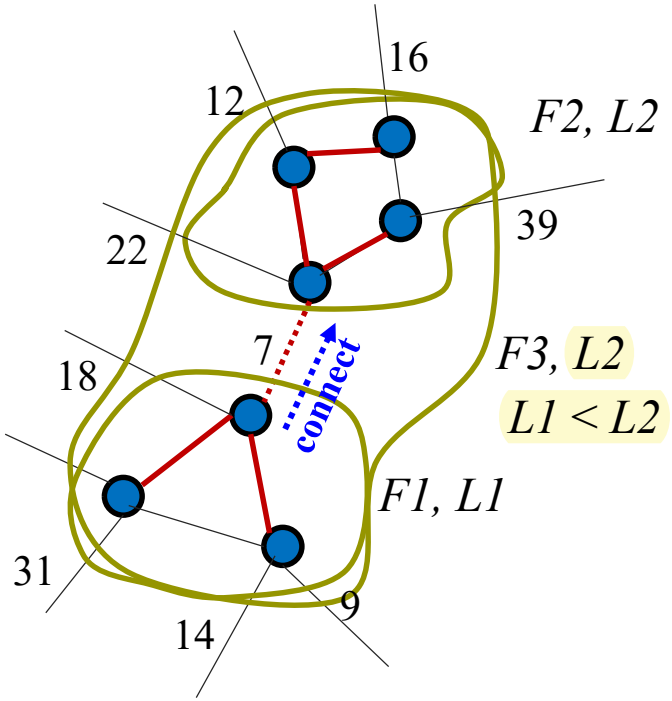
- Case 1: when two fragments *F1* and *F2* meet and they have the same level *L* and the same minimum weight outgoing edge, they combine to form a new fragment at level (*L*+1).



The connecting edge becomes the "**core**" of the new fragment. The weight of the core edge is used as the ID of the new fragment.

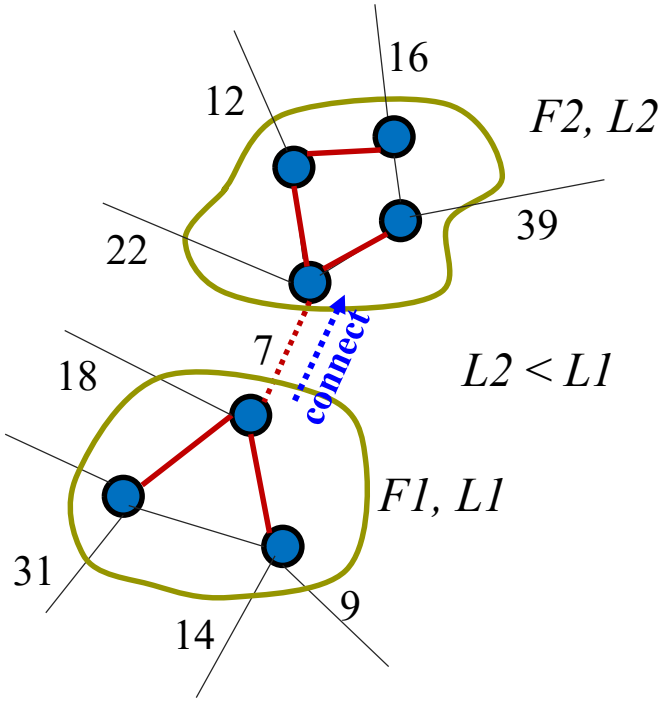
Fragments Merge & Absorb (2)

- Case 2: If fragment *F1* at level *L1* requests to connect fragment *F2* at level *L2*, where *L2* > *L1*, then *F1* is absorbed by *F2*, and the level of the new fragment remains *L2*.

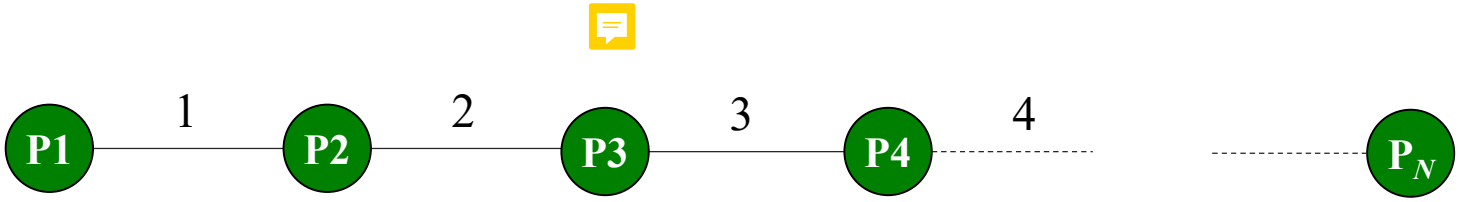


Fragments Merge & Absorb (3)

- Case 3: If fragment *F1* at level *L1* requests to connect fragment *F2* at level *L2*, where *L2* < *L1*, then the connect request is put in wait.



Why let higher level fragment wait?

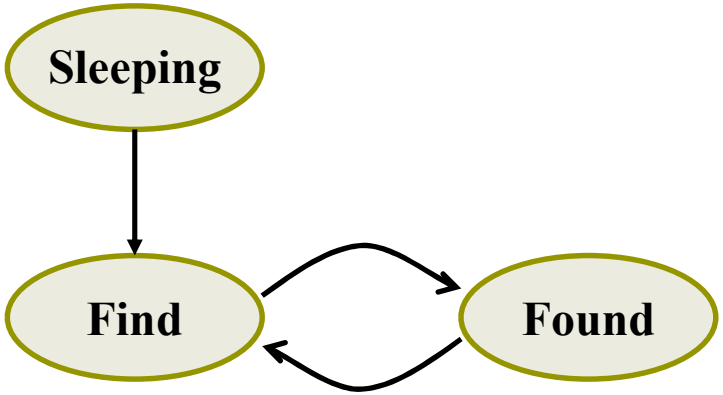


- P1 wakes up and sends a connect request to P2, they form a level-1 fragment {P1, P2}.
- P2 sends **Test**(core(P1, P2), level 1) to P3, which is waked up, and sends **Connect** to P2.
- P3 accepts P2's **Test** message. P2 forwards the **Report** message to P1.
- P1 forwards **change-core** message to P3, which will result in an **initiate** message to P3.
- P3 sends **Test**(core(P1, P2), level 1) to P4, which is waked up, and sends **Connect** to P4.
- P4 accepts P3's **Test** message. P3 forwards the **Report** message to P1.
- P1 forwards **change-core** message to P4, which will result in an **initiate** message to P4.
- P4 sends **Test**(core(P1, P2), level 1) to P5, which is waked up, and sends **Connect** to P5.
- …

message complexity: $\Theta(N^2)$

Node States

- **Sleeping:** can spontaneously wake up to initiate the algorithm, or is awakened by the receipt of any algorithm message from another node
- **Find**: looking for the minimum-weight outgoing edge
- **Found**: waiting for a response from the fragment at the other end of the selected edge



Edge State

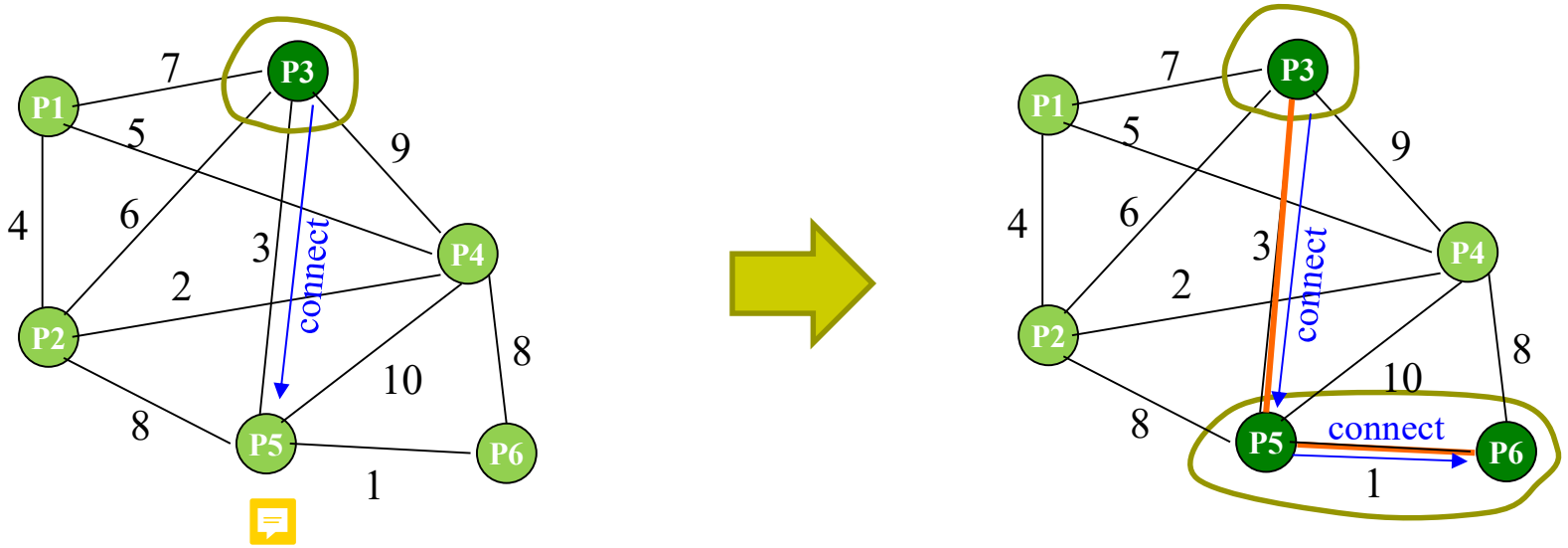
- **Branch**: the edge is a part of the MST
- **Rejected**: the edge is not part of the MST
- **Basic**: the algorithm has not yet decided whether or not the edge is part of the MST

Message Type

- **Initiate:** broadcast the id of the fragment to members and tell them to look for the minimum-weight outgoing edge of the fragment.
- **connect:** request to connect to a fragment
- **Test:** searching for minimum-weight outgoing edge
- **Reject/Accept:** replies to Test messages
- **Report(*w*):** reporting the minimum-weight outgoing edge *w* found
- **change-core** (also called "*change-root*" in the original paper): used by the core nodes to inform the new core nodes that a new fragment has been formed.

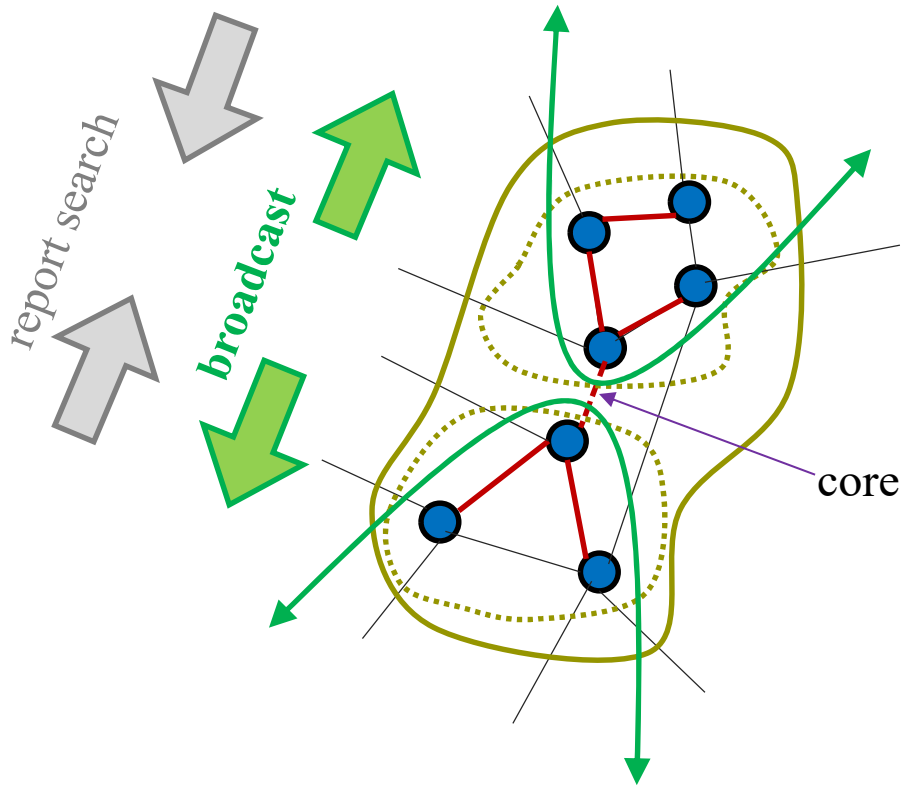
Example: From Level-0 to Level-1

For level-0 fragments, an awakened node chooses its minimum-weight incident edge, marks the edge as a branch, sends a **connect** request to the node on the other side, and waits for a reply.



P3 awakens, sends a connect request to P5, and enters state Found.
Assume P5 is asleep before receiving P3's connect request.
Then P5 is awakened by P3, and then sends a connect request to P6, which then merges with P5 to form a level-1 fragment {P5, P6} with core edge (P5, P6).

Fragments Merge

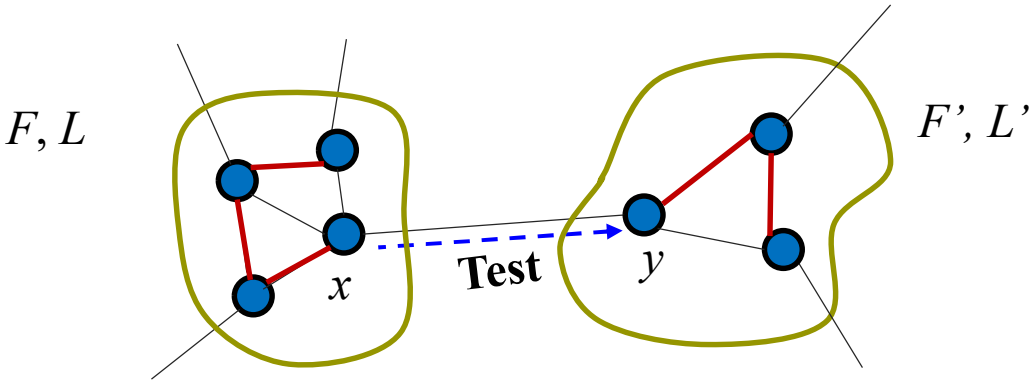


When two fragments merge, the two nodes adjacent to the core broadcast an **Initiate (*L*, *ID*)** message carrying the new fragment level *L* and identity *ID* to the rest of the nodes in the fragment (messages are sent via the branch edges).

The messages also tell the nodes in the fragment cooperate to find the minimum-weight outgoing edge of the new fragment.

Then, the searched information is reported back to the core nodes for the next round of merger.

Find Minimum-weight Outgoing Edge



Edges states:

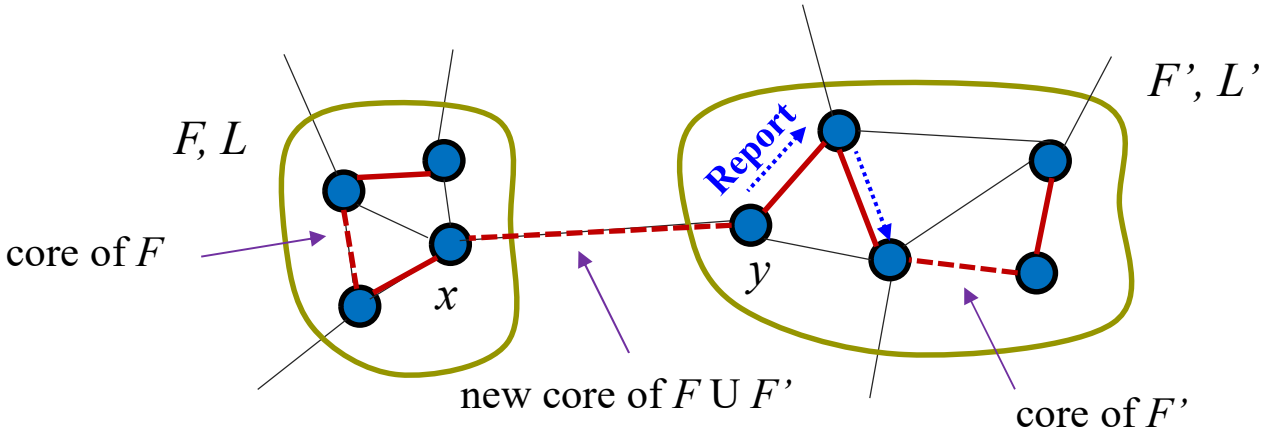
- **Branch**: part of the MST
- **Rejected**: not part of the MST
- **Basic**: not yet decided

Each node, after receiving an **Initiate** message with the argument Find, starts to find its minimum-weight outgoing edge. It picks the minimum weight Basic edge and sends a **Test (*L*, *ID*)** message, carrying the level ***L*** and fragment identity ***ID*** as arguments, to the adjacent node.

Test message is answered with **Accept**, **Reject** or delayed, depending on their fragment ids and levels:

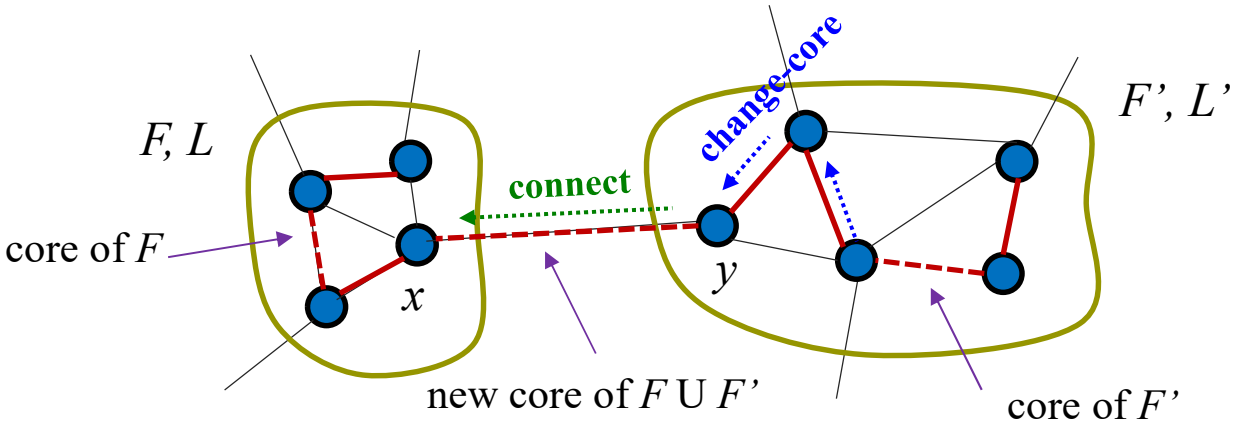
- **Reject**: both x and y are in the same fragment (edge(x,y) is marked rejected)
- **Accept**: x and y are in different fragments and $L \leq L'$(edge(x,y) is marked branch)
- delayed: $L > L'$until the situation has changed

Report minimum outgoing edge found



After a node has found its minimum weight outgoing edge (or found none), it waits for a **Report** message from all the nodes it flooded the initiate message to, and then sends back a report message (containing the minimum weight outgoing edge through its branch) back to the node from which it received the initiate message.
This process continues until the report messages reach the core nodes.

Change Core

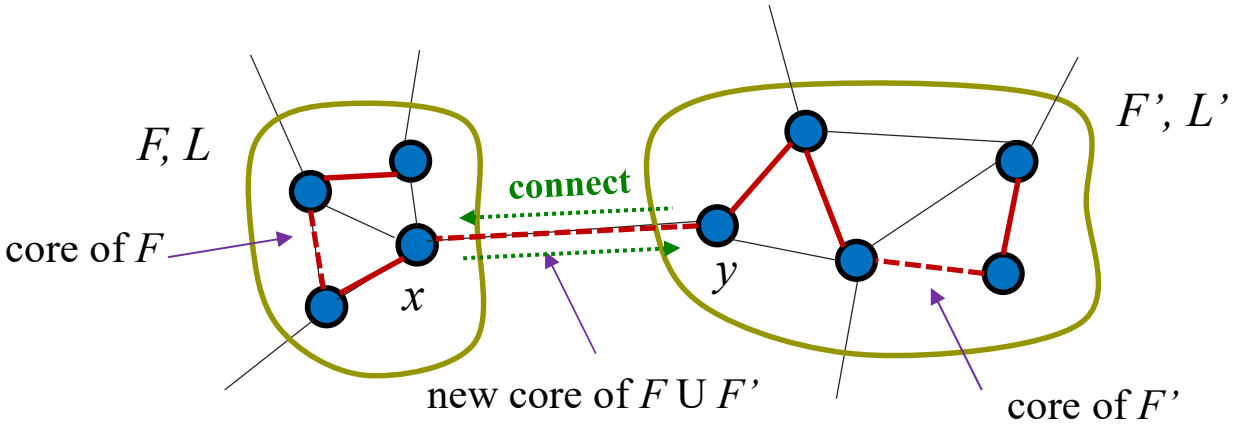


After both core nodes know through which the minimum outgoing edge can be reached, the core node which is "closer" to the minimum outgoing edge sends a **change-core** message along the report message path to the node (*y* in the figure) having the minimum outgoing edge.

When node *y* receives the **change-core** message, it sends a **connect(*L'*)** message to *x* notifying that fragment *F'* whishes to merge/be absorbed by fragment *F*.

Note that it suffices to pass the change-core messages through the edge(*x*, *y*) report path. Nodes in the path change the "edge direction" so that future messages to the fragment's core will lead them to the new core.

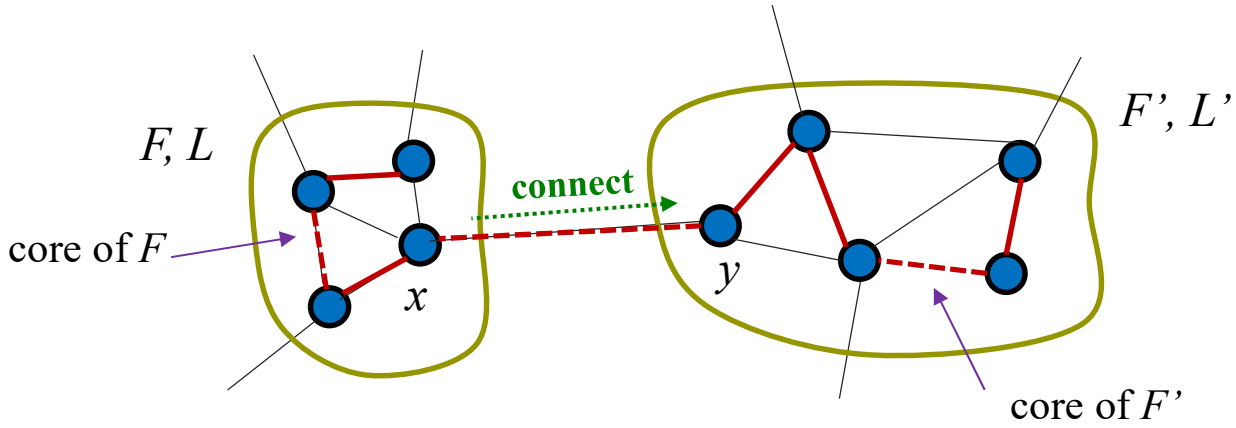
Connecting fragments (1)



If two fragments at level L have the same minimum outgoing edge, then each sends the message **connect**(L) over this edge, causing the edge to become the core of a level (L+1) fragment. The new core nodes issue new initiate messages to start search of minimum outgoing edge of the new fragment.

➔ level L fragments contain at least 2^L nodes, and so max. fragment level is $\log_2 N$

Connecting fragments (2)

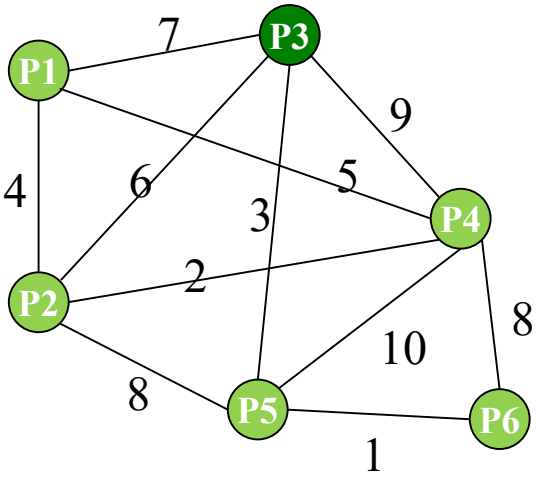


If $L < L'$, node y immediately sends back a message **Initiate** (L', F', y's state) to x, effectively causing F to be absorbed by F'.

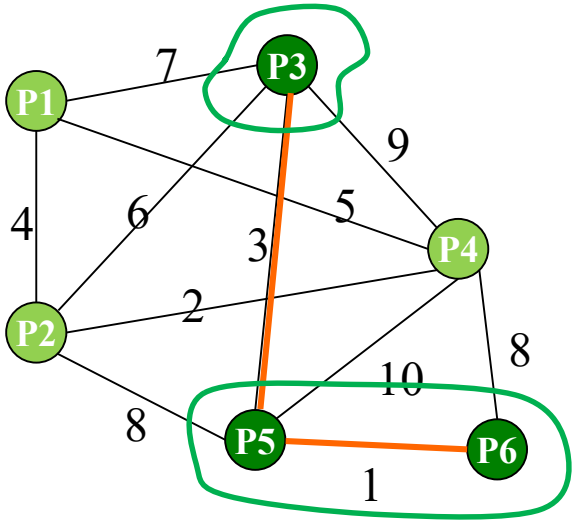
The initiate message is flooded back to x's fragment F.

Whether or not the members of F participate in finding the minimum-weight outgoing edge from the enlarged fragment $F \cup F'$ depends on y's state:

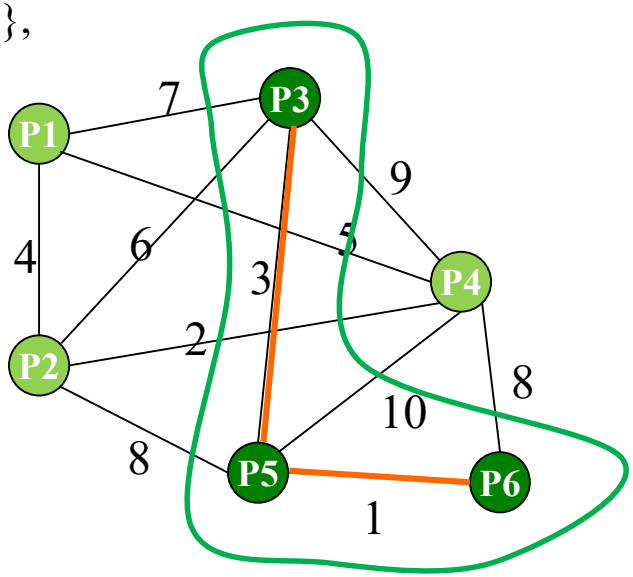
If y is in state **Find**, they join the search; otherwise (state **Found**), y has already reported its minimum-weight outgoing edge, and so they need not join the search (joining won't change the reported result, why?)



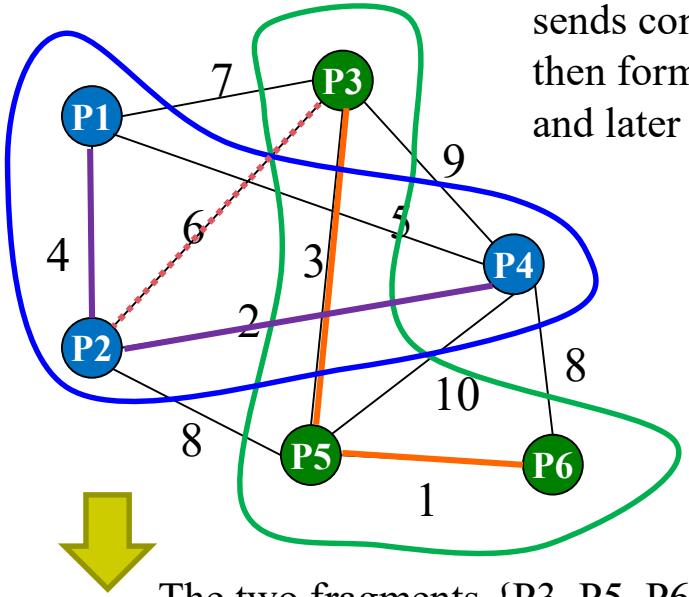
P3 wakes up, sends connect request to P5, which is waked up to send a connect request to P6, and then forms a fragment {P5, P6} of level 1.



P3 absorbed into {P5, P6}



P3 wakes up P2, which then sends connect request to P4 and then forms a fragment {P2, P4}, and later on absorbs P1.



The two fragments {P3, P5, P6} and {P1, P2, P4}merge on edge(P2, P3)

Full GHS Algorithm (1)

The Algorithm (As Executed at Each Node)

(1) Response to spontaneous awakening (can occur only at a node in the sleeping state)

```
execute procedure wakeup
```

(2) **procedure** *wakeup*

```
begin let m be adjacent edge of minimum weight;
        SE(m) ← Branch;          // State of Edge
        LN ← 0;          // Level of Node
        SN ← Found;      // State of Node
        Find-count ← 0;
        send Connect(0) on edge m
end
```

(3) Response to receipt of *Connect*(*L*) on edge *j*

```
begin if SN = Sleeping then execute procedure wakeup;
        if L < LN
          then begin SE(j) ← Branch;
                     send Initiate(LN, FN, SN) on edge j;   // FN: Fragment ID
                     if SN = Find then
                       find-count ← find-count + 1
               end
          else if SE(j) = Basic
                 then place received message on end of queue
                 else send Initiate(LN + 1, w(j), Find) on edge j
end
```

Full GHS Algorithm (2)

```
(4) Response to receipt of Initiate(L, F, S) on edge j
    begin LN ← L; FN ← F; SN ← S; in-branch ← j;
          best-edge ← nil; best-wt ← ∞;
          for all i ≠ j such that SE(i) = Branch
            do begin send Initiate(L, F, S) on edge i;
                     if S = Find then find-count ← find-count + 1
               end;
          if S = Find then execute procedure test
    end
(5) procedure test
    if there are adjacent edges in the state Basic
      then begin test-edge ← the minimum-weight adjacent edge in state Basic;
                 send Test(LN, FN) on test-edge
           end
      else begin test-edge ← nil; execute procedure report end
(6) Response to receipt of Test(L, F) on edge j
    begin if SN = Sleeping then execute procedure wakeup;
          if L > LN then place received message on end of queue
            else if F ≠ FN then send Accept on edge j
                   else begin if SE(j) = Basic then SE(j) ← Rejected;
                              if test-edge ≠ j then send Reject on edge j
                                else execute procedure test
                        end
    end
```

Full GHS Algorithm (3)

```
(7) Response to receipt of Accept on edge j
      begin test-edge ← nil;
            if w(j) < best-wt
              then begin best-edge ← j; best-wt ← w(j) end;
            execute procedure report
      end
(8) Response to receipt of Reject on edge j
      begin if SE(j) = Basic then SE(j) ← Rejected;
            execute procedure test
      end
(9) procedure report
      if find-count = 0 and test-edge = nil
        then begin SN ← Found;
                   send Report(best-wt) on in-branch
             end
```

Full GHS Algorithm (4)

(10) Response to receipt of *Report*(*w*) on edge *j*

```
if j ≠ in-branch
  then begin find-count ← find-count − 1
             if w < best-wt then begin best-wt ← w; best-edge ← j end;
             execute procedure report
       end
  else if SN = Find then place received message on end of queue
    else if w > best-wt
           then execute procedure change-root
           else if w = best-wt = ∞ then halt
```

(11) **procedure** *change-root*

```
if SE(best-edge) = Branch
  then send Change-root on best-edge

else begin send Connect(LN) on best-edge;
           SE(best-edge) ← Branch
     end
```

called “change-core” in the paper

(12) Response to receipt of *Change-root*

```
execute procedure change-root
```

Proof of Correctness

- How do you prove the correctness of the GHS algorithm?
- For sequential algorithms, how do you prove their correctness?

Safety vs. Liveness Properties

- A ***safety*** property describes a property that always holds; sometimes we put it in this way "*nothing 'bad' will happen*".
- A ***liveness*** property describes a property that will eventually hold; sometimes we put it in this way "*something 'good' will eventually happen*".

What are the safety & liveness properties of the MST problem?

Could the GHS algorithm results in deadlock?

Message Complexity

- Max. total number of messages exchanged by the nodes:
  - $2E + 5N \log_2 N$
    - max. fragment level: $\log_2 N$

- Message Size:
  - edge weight + Level + message types

  => $\log_2 w_{max} + \log_2 N + 3$

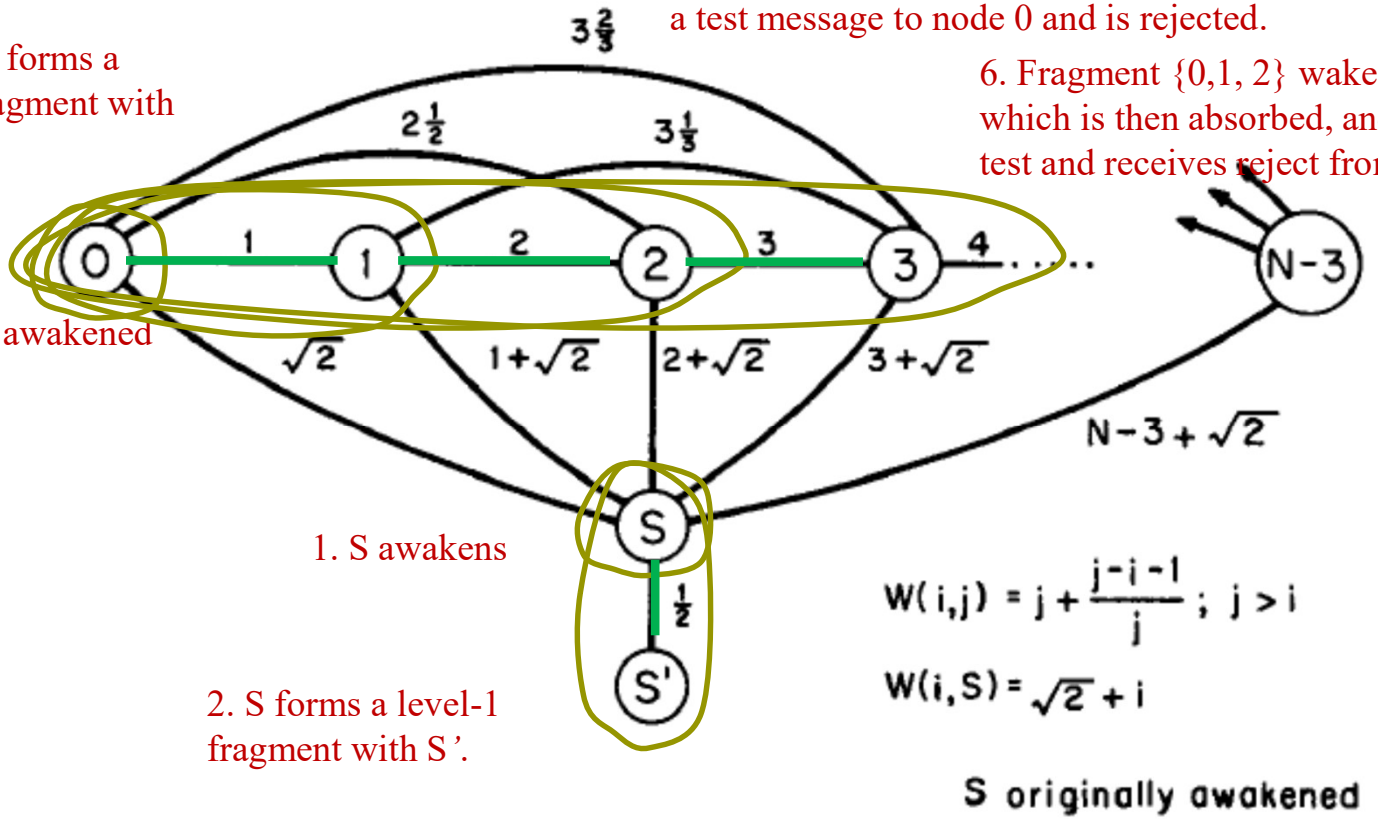
Time Complexity

5. Fragment {0,1} wakes up node 2, which is then absorbed, and later on sends a test message to node 0 and is rejected.

4. Node 0 forms a level-1 fragment with node 1.

6. Fragment {0,1, 2} wakes up node 3, which is then absorbed, and later on sends test and receives reject from 1 & 0.

3. Node 0 is awakened by {S, S'}



1. S awakens

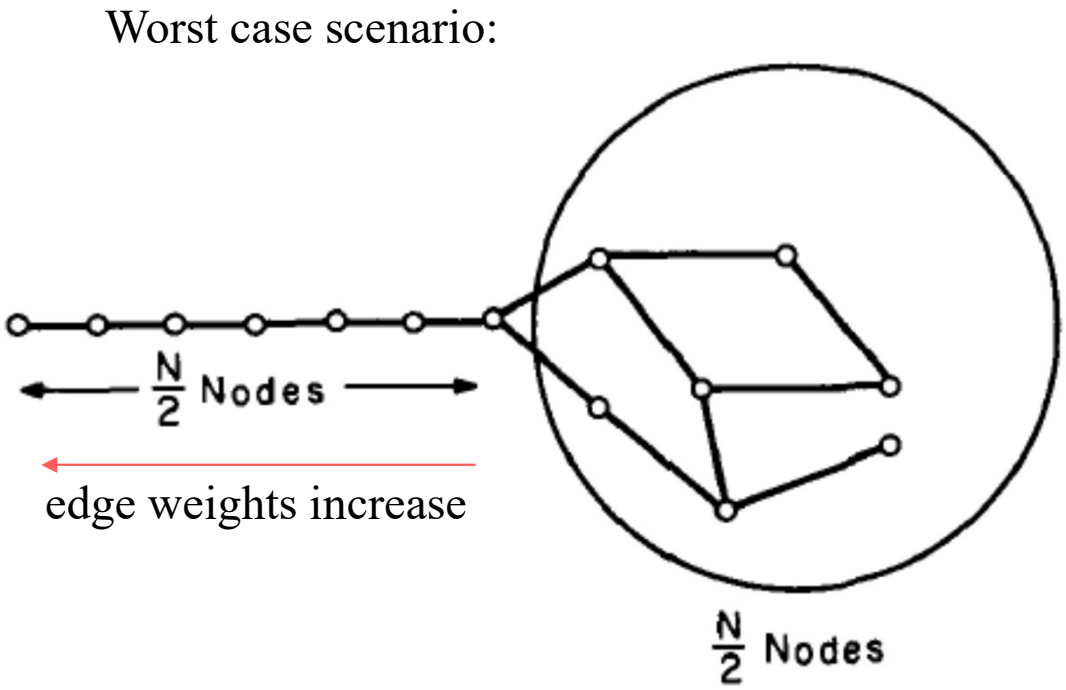
2. S forms a level-1 fragment with S'.

Fig. 2. Example when algorithm requires $N(N-1)$ message time units.

Each node j sequentially sends test and receives reject from every $i \leq j-2$ before $j' > j$ is awakened, and therefore the time complexity is $\Theta(N^2)$.

Time Complexity (2)

- If all nodes are awakened initially, then the time complexity can be reduced to $O(N \log N)$.
  - see paper for proof.



Impossibility Result

- If the network has neither distinct edge weights nor distinct node identities, then there exists no distributed for finding an MST with a bounded number of messages