BRIEF ALGORITHM DESCRIPTION

The algorithm proposed is a simplified version of the pioneeristic distributed algorithm GHS([1]) for computing the *minimum weight spanning tree*  over a network. It is basically the result of the application of a synchronizer ([2]) to the traditional algorithm. The resulting algorithm (called Synch\_GHS) has the same time complexity upper bound as GHS, namely , where is the number of nodes of the network. Communication complexity is worse because of synchronization messages used at every level: now it is , where is the number of nodes of the network. In the following, I describe the *network model* the algorithm works upon and I provide a practical example of execution over a simple network composed of six nodes.

I start describing the network model adopted for this algorithm, that is called the *asynchronous network model*. The *asynchronous* network is a point-to-point (store-and-forward) communication network, described by an undirected communication graph (V, E) , where the set of nodes V represents processors of the network and the set of links E represents bidirectional noninterfering communication channels operating between them. No common memory is shared by the node’s processors, and each node has a distinct identity. Each node processes messages received from its neighbors, performs local computations, and sends messages to its neighbors. All these actions are assumed to be performed in a negligible time compared to the time taken by a sent message to reach its destination. All the messages have a fixed length and may carry only a bounded amount of information. Each message sent by a node to its neighbor arrives within some finite but unpredictable time. When a node receives a message, it stores it at the end of a message queue following FIFO logic. Therefore when the node has done with the computation associated to the current message, it extracts the next message from the head of the queue and processes it. In this way messages can accumulate at a node. In addition, each edge is labeled with a positive integer representing the *cost of communication* (or *weight*) that may have a physical meaning, such as the inverse of bandwidth. When the node wants to communicate with the node , it sends a message along the path connecting to ; the *total cost* of the path is the sum of the costs of edges composing the path. A *spanning tree* is a tree built on top of a graph and that contains every vertex of the graph. A *minimum spanning tree* (MST)is a spanning tree that minimizes the total cost for any source process to communicate with all the other processes in the network. The MST is unique if all the edges of the graph have distinct weights. Actually, if the nodes in network have unique IDs (as supposed in the model adopted), it is always possible to calculate a unique MST even if there are edges with the same weights. It needs just to substitute a weight with the concatenation of the lowest ID between the nodes connected by the edge, the weight itself, and the highest ID between the IDs of the nodes connected by the edge. The resulting string of this concatenation is indeed unique.

Now, suppose we want to calculate the MST associated to a the graph represented by the following adjacency matrix:

Each row of the matrix contains the weights of the outgoing edges of a given node. For example the first row is related to the node indexed with . This node has outgoing edges having weights and leading to nodes and 5 respectively. Following the same reasoning for all the nodes, it is possible to draw the graph:

9

2

1

5

4

3

6

In the very beginning, every node is in *sleeping* state and idling, and all the edges have *state* *basic*. At the end of the algorithm all the edges must have state either *branch*, which it means that the edge belongs to MST, or *rejected* if the edge doesn’t. A node awakes upon receipt of any algorithm message from a neighbor or if it receives a signal triggered by the outer world. The algorithm starts when at least one node is awoken by an external input. In this example, let us awake the node . When node wakes up, it detects its minimum weight outgoing edge (MWOE), changes its state to *branch* and sets *fragment-level* to . Eventually it sends a *connect*  message along the MOWE (link (0-1)).

When node 1 receives the *connect* request, it wakes up and follow the same wakeup routine explained above. Both node 0 and 1 now knows that they have a common MWOE. Therefore link (0-1) is the new *core* of the fragment containing nodes {0, 1}. Both these nodes set *fragment-level* to 1 and *fragment-ID*  to 0. *Fragment-ID* is the lowest ID between the IDs of the nodes connected by the *core* edge and the node whose index is equal to *fragment-ID* is the *leader* of the cluster. The figure below shows the state of the graph after nodes 0 and 1 have formed a fragment (or *cluster)*. In figures, edges belonging to MST are represented with continuous line, whereas colors are used to indicate belongingess to a particular fragment (in the figure the fragment with ID equal to 0 is connected with red edges). Arrows, instead, represent the *connect* messages.

2

6

3

1

4

5

9

When the “connect” phase is over, the leader broadcast an *initiate* message throughout the cluster to update *fragment-ID* and *fragment-level*. When the nodes at the boundaries of the cluster receive the *initiate* message they convergecast an *ack*  message back to the leader. In this case when the node 1 receives the *initiate* message from node 0, it sends back an *ack* message indicating that the cluster 0 is formed. Node 0 then, starts the *test* phase for detecting the MWOE of the new cluster.In order to do that, it sends *test* messages to all its neighbors whose edges have not been put to reject, namely nodes 1, 5 and 2. Node 1 relays the *test* messages to all the nodes outside the cluster, namely to node 4.

Receipts of test-probes awakes nodes 4 and 5, they do their own wakeup algorithm mentioned earlier forming another MST fragment with *fragment-level 1* and *fragment-ID* 4 containing nodes {4,5}, while during the whole process queuing the test-probes to be processed after the wakeup phase is over. Fragments 4 and 5 (fragments are denoted by their IDs) were formed by a *merge* process. This process occurs when two clusters (a node can be considered a cluster formed by just one element with *fragment-level* 0 and *fragment-ID* equal to the ID of the node) send *connect* message across the same MWOE. When this happens the two merging clusters have the same *fragment-level* (the proof the this always happens is beyond the scope of this brief note) and the resulting cluster takes the *fragment-level* of the two former cluster increased by one.

2

6

3

1

4

5

9

Now they can both process the pending test-probes: having seen that the *fragment-ID*  are different and levels are equals, they both send back *report* messages containing the weight of the edge. In present example node 5 sends back to node 0 a *report* message containing weight 3 and node 4 sends back to node 1 a message containing weight 9. After fragment 1 is formed nodes 4 and 5, in turn, start sending test-probes (*test* messages) outside the fragment towards nodes 0, 1 and 3. Nodes 1 and 0 reply with the same *report* message previously sent to them. When node 3 receives the *test* message from node 4, it could already have been awoken by node 2. We shouldn’t forget, indeed, that node 0 had sent a *test* probe to node 2 that, waking up, had sent a *connect* message to node 3. If node 3 is awoken by the *test* message sent by node 4, it sends back a *connect* message to node 4 and it is therefore absorbed into fragment 4. An *absorb* operation occurs when a cluster (or a node, that can be considered a cluster as previously seen) sends a *connect* request to a cluster with higher *fragment-ID.* When this happens the cluster changes *fragment-level* and *fragment­-ID* with those of the cluster with higher *fragment-ID*. Eventually when the *connect* message from node 2 arrives, node 2 is absorbed into fragment 4 as well.

On the other hand, if node 3 is already awake when receives the *test* probe from node 4, it puts the message at the end of the queue message. When node 3 is eventually absorbed into fragment 4 (this happens since node 3, waking up, had sent a *connect* request to node 4) the *test* message is processed and relayed to node 2. When node 3 becomes part of fragment 4 ,increasing its *fragment-level* by one, it can finally process the *connect* request from node 2, since when a node receives a *connect* request from a node with the same *fragment-level* but on an edge that it is not the MWOE, that message is put at the end of the message queue. When the *connect* request from node 2 is processed, node 2 can finally be absorbed into cluster 4. The final situation is summed up in the figure below.

2

6

3

1

4

5

9

In the present situation we have two clusters with *fragment-ID* equals to one. Node 2, now part of cluster 4, eventually processes two *test* messages: one from node 0 and the other one from node 3. As reply to node 0, node 2 sends a *report* message containing the weight 6. The *test* message from 3 is instead relayed to node 0, which, in turn, sends back to node 2 a *report* message containing the weight 6 as well. All the *report* messages are convergecast to the cluster leaders, namely nodes 0 and 4 that are now able to spot their MWOE, which for both is the edge connecting node 0 and 5. At this point, cluster 4 needs to send a *connect* message along the edge (5-4) and cluster 0 to send a *connect* message along (4-5). In order to do that the cluster leader sends a *changeroot* message towards the nodes having among their outgoing edges the spotted MWOE (nodes 5 for cluster 5 and 0 for cluster 0, in this latter case node 0 doesn’t need to send any *changeroot* message). Eventually the two clusters merged together to form a new cluster with *fragment-level* 2 and *fragment-ID* 0.

3

1

4

5

2

6

9

Once the new cluster 0 is formed, the leader (node 0) sends *test* probes to node 1 and node 2. Node 1 relays the *test* message to node 4, node 2 sends a *reject* message back to node 0. Both node 2 and 0, then, put the edge connecting them to *rejected*. The same procedure is repeated for nodes 4 and 1. Finally, the leader node, having received no *report*  message, broadcasts a *finished* message over the MST. When a node receives the *finished* message the algorithm (at the node) terminates. This brief explanation, indeed, is far from being exhaustive, for more detailed information I recommed to read the comments included in file Node.cpp and, above all, the original paper about GHS ([1]) and the superb book of Nancy Lynch([3]).

[1] Gallager, R.G., Humblet, P.A. and Spira, P.M.: A distributed algorithm for minimum weight spanning trees. ACM Transaction on Programming Languages and Systems, 55: 66-77, 1983.

[2] Boruch Awerbach. Complexity of nerwork synchronization. Journal of the ACM. 35(4): 845-875, October 1988.

[3] Lynch, N.: Distributed Algorithms. 1996 Morgan Kaufmann Publishers, Inc.