The Gallagher-Humblet-Spira algorithm

Distributed algorithm to compute a Minimum Spanning Tree of a connected weighted undirected graph

Goal and requirements of the algorithm

- The algorithm applies on connected **weighted** graphs whose edges have all different weights
  - This entains that the MST of the graph is **unique**
- A **Minimum spanning tree** is a subset of **Graph** G, which has all the vertices covered with minimum possible number of edges and whose sum of all edges' weights is the minimum possible
- The **goal of the algorithm** is to build a Minimum Spanning Tree of the connected graph on which the algorithm is applied

Notation

- **Fragments**: a fragment is a subtree of the MST. Each fragment is characterized by:
  - **Fragment name**, that is unique to each fragment and is used, during the search for outgoing edges, to check if an edge is internal or external to the fragment
  - A **fragment level**, that is used to determine how fragments must unify (in general, the fragment with the lowest levei is unified to the other one)
- **Outgoing edge:** an edge that connects a node of a fragment F to a node of another fragment F'
  - The **minimal weight outgoing edge** of a fragment is the fragment's outgoing edge with the minimum weight. Each time, the fragment searches for this edge and, when found, sends on it a <connect> request to the other fragment.
- **Channel status**: can be
  - **Basic** if still not explored
  - **Branch** if included in the MST
  - **Reject** if exluded from the MST

Fragments combining strategies

A. If the 2 fragments F and F' have different levels L and L', then the fragment with the lowest level combines into the other and assumes the name and the level of the fragment with the highest level
B. If the 2 fragments have the same level L and, for each of them, the lowest weight ougoing edge leads to the other fragment, then the 2 fragments combine in a new big fragment whose level is L + 1, whose name is the weight of the edge that connects the 2 fragments and whose core edge is the edge that connects the 2 fragments .
C. In the other cases, the fragment receiving the *connect* request must wait:
    - If the sender's level is > receiver level, the request must wait for an increase of the receiver's level
    - If the channel is stil not explored (*basic*) by the receiver, it must wait to send a *connect* request on that channel

Overview of the algorithm

1. At the beginning, each node is in a different 1-node fragment whose level is 0. Each node selects its lowest weight outgoing edge and uses it to request merging with another fragment.
2. After a node is unified to a new fragment, if the old fragment was in the *find* state, the new nodes join the search for the lowest weight outgoing edge of the new fragment
3. Each node performs a local search of the fragment's lowest-weight outgoing edge by testing his lowest weight neighbor, if any:
   - If this neighbor is in the same fragment as the testing node, the edge is rejected and the local search continues with another edge;
   - Otherwise, the testing node updates the minimum weight found (if this edge is better) and reports to his father
4. In the report() procedure, each node awaits to receive the reports from all of its sons and to end its local search before reporting to his father

Overview of the algorithm (2)

5. Each time a node receives a report from one of his sons, it comparates the report's *bestweight* with its one: if the report one is better, it updates its *bestweight* and its *bestchannel*. In this way, the nodes keep track of the path to reach the node incident to the candidate minimum weight outgoing edge.
6. The reports move from sons to father until they reach the core nodes.
   - The core node on whose side there is the leaf, starts the changeroot() procedure
7. In the changeroot() procedure, a message moves from the core edge to the node incident to the minimum weight outgoing edge.
   - When it reaches this node, the node sends a connection request to the other fragment's node

Overview of the algorithm (3)

8. Upon the receipt of a connection request from fragment A, the other fragment (B) may:
   - Communicate to the sender's fragment (A) that it must merge to B (using Rule A), if LevB > LevA
   - Await if the channel hasn't been explored yet by fragment B or if LevA > LevB
   - Communicate to the sender's fragment (A) that it must merge with B using Rule B, hence increasing the level and using the new core edge's weight as new fragment name, if the 2 fragments are one the same level

Initialization

- Node level is initialized to 0
- The state of the channels is saved in a dictionary whose keys are the neighbors. Each element in this dictionary can assume a value of the *ChannelState* enum defined in *tools.da*.

  For each neighbor, the corresponding channel state is initialized to *basic*.
- Initially, each node gets a name that is its index in the general nodes vector. The nodes' names are used for printing purposes. Each node gets also a *namesDict* to associate nodes to their names.
- Each node has 3 queues for the messages that can be delayed: *connect, test* and *report* messages
- Since, at the beginning, each fragment is made by only 1 node, the node's name is used as fragment's name and the node's father is set to None

```
class Agent(process):
    def setup(neighs, weights, name, namesDict, obs):
        self.state = NodeState.sleep
        self.level = 0
        self.stach = {} #dictionary <nextNode, EdgeState>

        # queue for the delayed <connect> messages
        # (to be re-processed if the level increases)
        self.connectMessQueue = deque()

        # queue for the delayed <test> messages
        # (to be re-processed after the receipt of <initiate>)
        self.testMessQueue = deque()

        # queue for the delayed <report> messages
        # (to be re-processed if the node's state changes)
        self.reportMessQueue = deque()
        for n in neighs: self.stach[n]=EdgeState.basic
        self.father = None
        self.fragName = name
        self.name = name
        self.testch = None
        self.bestch = None
        self.rec = 0 #num of records received from sons
        self.bestwt = INF
        self.namesDict = namesDict #dict to associates nodes'  to their ID
```

First connection request

```
def run():
    # 1. INITIALIZATION
    # At the beginning, this node's fragment consists only of this node.
    # In order to expand the fragment, the node connects to another fragment on its lowest-weight outgoing edge.
    minWeightEdge = min(weights, key=weights.get)
    self.stach[minWeightEdge] = EdgeState.branch
    self.state = NodeState.found
    send(('connect',self.level), to=minWeightEdge)
    output("Node ", self.name, "sent <connect> to node ", self.namesDict[minWeightEdge])

    #await termination of the algorithm to end the process
    await(self.state == NodeState.finished)
```

- At this early stage, the fragment is made only by 1 node so it doesn't need to execute the *test()* procedure to check if an outgoing edge belongs to the same fragment or the *report()* procedure to check if other nodes in the fragment have found a better edge
- Hence, the node sends a connect request on the minimum weight outgoing edge and sets its state to 'found'.
- To avoid the termination of the *DistAlgo* process, an await() instruction is placed at the end of the run() method

Receipt of a connection request

```
def processConnectMess(Lq, q):
    #called at the reception of a <connect> message
    if Lq < self.level:
        # combine with RULE A: the <connect> message comes from a smaller fragment (lower level)
        # hence, that fragment must join this node's fragment: an <initiate> message is sent
        # and the other fragment's nodes will acquire our level, name and state
        self.stach[q] = EdgeState.branch
        send(('initiate', self.level, self.fragName, self.state), to=q)
        output("Node ", self.name, "sent <initiateA> to node ", self.namesDict[q])
        #once this fragment merges with q's one, we discard the previous <connect> msgs from the same fragment
        cleanConnectsQueue(q)
        return True #message is consumed
    elif self.stach[q]==EdgeState.basic:
        #process the message later, after the edge has been explored (this node sends <connect> to q)
        # or the level has increased
        self.connectMessQueue.append((Lq, q))
        output("Node ", self.name, "has delayed <connect> message from ", self.namesDict[q])
        return False #message is not consumed
    else:
        #RULE B: the two fragments are of the same level, they reciprocally exchange the following <initiate> message
        send(('initiate', self.level+1, self.weights[q], NodeState.find), to=q)
        output("Node ", self.name, "sent <initiateB> to node ", self.namesDict[q])
        return True

def receive(msg=('connect', Lq), from_=q):
    output("Node ", self.name, "received <connect> from node ", self.namesDict[q])
    #2. RECEIPT OF <CONNECT> MESSAGE
    processConnectMess(Lq, q)
```

When the connectsQueue is checked

- At the end of the *changeroot()* method, if executed by the node incident to the lowest weight outgoing edge, after sending the *connect* request, because one of the edges changes status from *basic* to *branch*
- When receiving a *reject* on a certain edge, to remove from the queue the *connect* messages coming from that edge
- After receiving an *initiate* message because the level of the fragment may have increased

Receipt of a test message

```
def processTestMsg(Lq, FNq, q):
    # procedure to process a <test> message
    if Lq > self.level:
        # process is delayed because this node and q could be in the same fragment,
        # but this node may still not know that
        self.testMessQueue.append((Lq,FNq,q))
        output("Node ", self.name, "has delayed <test> message from ", self.namesDict[q])
        return False #message is not consumed
    elif FNq == self.fragName:
        #received from node in the same fragment
        if self.stach[q]==EdgeState.basic: self.stach[q]=EdgeState.reject #reject edges in the same fragment
        if self.testch != q: # send <reject> only if we didn't send <test> before (avoids redundancy)
            send('reject', to=q)
            output("Node ", self.name, "sent <reject> to node ", self.namesDict[q])
        else: test() # received <test> msg (from node in same fragment) on the same channel that we were testing:
                     # continue search on other edges
        return True #message is consumed
    else:
        #q has found the best local outgoing edge
        send('accept', to=q)
        output("Node ", self.name, "sent <accept> to node ", self.namesDict[q])
        return True #message is consumed
```

When the testQueue is checked

- After receiving an *initiate* message because the level of the fragment may have increased (it could be the case that the fragment has unified with the fragment that sent the *test* request)

```
def receive(msg=('initiate', Lq, FNq, Sq), from_=q):
    #3. RECEIPT OF <INITIATE> MESSAGE: union of 2 fragments
    output("Node ", self.name, "received <initiate> from node ", self.namesDict[q])
    self.level = Lq
    self.fragName = FNq
    self.state= Sq
    self.father = q
    self.bestch = None
    self.bestwt = INF
    dests = [n for n in neighs if (n != q and self.stach[n]==EdgeState.branch)]
    send(('initiate', Lq, FNq, Sq), to=dests) #send to entire subtree
    output("Node ", self.name, "sent <initiateA> to neighbors ", names(dests))
    if self.state == NodeState.find:
        # new fragment gets involded in the lowest-weight outgoing edge search
        self.rec = 0
        test()
    #### when the level of the node changes, re-process delayed messages
    checkAllQueues()
```

The *test()* procedure

- The test() procedure is used to perform the local search of the minimum weight outgoing edge: each candidate edge has to be tested to check whether it belongs to the same fragment
- It uses an helper method, called *findMinBasicEdge()*, to find the minimum weight outgoing edge; if all edges have been explored, this function returns *None* and the *report()* method without improving the *bestwt*.

```
def test():
    #4. TEST PROCEDURE: local search for the lowest-weight outgoing edge
    self.testch = findMinBasicEdge()
    if self.testch is None:
        # if there aren't any other edges to explore
        report()
    else:
        # else, explore best edge
        send(('test', self.level, self.fragName), to=self.testch)
        output("Node ", self.name, "sent <test> to node ", self.namesDict[self.testch])
```

```
def findMinBasicEdge():
    # returns the basic edge with the minimum weight. Used in test() procedure
    minw = INF
    candidate = None
    for destNode, stc in self.stach.items():
        if stc==EdgeState.basic and self.weights[destNode] < minw:
            minw = self.weights[destNode]
            candidate = destNode
    return candidate
```

Accept and reject

ACCEPT:

```
def receive(msg=('accept'), from_=q):
    #6. Answer to <test> message, q is in another fragment: we found the local best edge.
    output("Node ", self.name, "received <accept> from node ", self.namesDict[q])
    self.testch = None
    if self.weights[q] < self.bestwt:
        self.bestwt = self.weights[q]
        self.bestch = q
    report()
```

REJECT:

```
def receive(msg=('reject'), from_=q):
    #7 Answer to <test> message, q is in the same fragment.
    output("Node ", self.name, "received <reject> from node ", self.namesDict[q])
    if self.stach[q]==EdgeState.basic:
        self.stach[q]=EdgeState.reject #exclude edge from SPT
        checkConnectsQueue() #<connect> msgs coming from the rejected edge will be removed
    test() #continue search
```

Receipt of an *initiate* message

```
def receive(msg=('initiate', Lq, FNq, Sq), from_=q):
    #3. RECEIPT OF <INITIATE> MESSAGE: union of 2 fragments
    output("Node ", self.name, "received <initiate> from node ", self.namesDict[q])
    self.level = Lq
    self.fragName = FNq
    self.state= Sq
    self.father = q
    self.bestch = None
    self.bestwt = INF
    dests = [n for n in neighs if (n != q and self.stach[n]==EdgeState.branch)]
    send(('initiate', Lq, FNq, Sq), to=dests) #send to entire subtree
    output("Node ", self.name, "sent <initiateA> to neighbors ", names(dests))
    if self.state == NodeState.find:
        # new fragment gets involved in the lowest-weight outgoing edge search
        self.rec = 0
        test()
    #### when the level of the node changes, re-process delayed messages
    checkAllQueues()
```

- If the node sending the initiate has not yet sent its report, then its state must be Find, and this causes fragment F to join the search (The node will not send report until it also receives a report message from the joining fragment).
- If the node sending the initiate had already sent its report, then its state and the parameter it sends must be Found. Since it had already sent its report, the edge on which the initiate was sent was not its minimum outgoing edge. Therefore, fragment F does not need to join the search. Indeed, notice that it's not possible that fragment F has an outgoing edge with a lower weight than the minimum outgoing edge of F' (the edge connecting F to F'), since then F would have sent connect over this edge.
- The message is forwarded to the entire subtree to inform the other nodes o the fragment
- Since the fragment's level is changed, the queue are checked

Receipt of *report* message

IF RECEIVED FROM SON:

```
def receive(msg=('report', w), from_=q):
    #9. Receive report
    output("Node ", self.name, "received <report> from node ", self.namesDict[q])
    if q != self.father:
        #reply at <initiate> message
        if(w < self.bestwt):
            self.bestwt = w
            self.bestch = q
        self.rec += 1
        report()
    else:
        #pq is the core edge
        processReportFromFather(w)
```

IF RECEIVED FROM FATHER:

```
def processReportFromFather(w):
    # the only nodes that receive reports from their fathers are the core nodes
    # (each one is father of the other one)
    if self.state == NodeState.find:
        # this node is still waiting for its sons' reports: delay the message
        self.reportMessQueue.append(w)
        output("Node ", self.name, "has delayed <report> message ")
        return False #message not processed
    elif w > self.bestwt:
        # this is the core node on the side of the leaf incident to the
        # lowest-weight outgoing edge: start changeroot()
        changeroot()
        return True #message is processed
    elif (w == self.bestwt and w == INF):
        # the fragment's search of the lowest weight outgoing edge has terminated without
        # finding any other possible edge: algorithm has terminated
        output("FINISHED")
        send('complete', to=obs) #inform the observer process (only to plot the SPT)
        return True
```

ReportsQueue is checked at the end of the report() method because the node state may change to *found*.

The *report* procedure

```
def report():
    #8. REPORT TO FATHER THAT A LOWEST WEIGHT OUTGOING EDGE HAS BEEN FOUND
    output("Node ", self.name, " called report() procedure")
    if(self.rec == self.countBranchEdges() and (self.testch is None)):
        # report only if I have received from all my sons and if i have finished my local search
        self.state = NodeState.found
        send(('report', self.bestwt), to=self.father)
        output("Node ", self.name, "sent <report> to node ", self.namesDict[self.father], " (father)")
        checkReportsQueue() #since state has changes to 'found', process the delayed reports
```

- Called by the *test()* procedure when there are no edges to explore
- Called upon the receipt of *accept* to inform the other nodes that a candidate for the minimun weight outgoing edge has been found
- Called each time a report is received from a son

Changeroot method

```
def changeroot():
    #10. Procedure called when the fragment has found its lowest-weight outgoing edge and the <report> edge has reached
    # the core node on whose side this edge is located. The procedure sends a <changeroot> message that is forwarded
    # through the framment's nodes on their respective 'bestch' towards the node incident to the lowest-weight outgoing edge.
    if self.stach[self.bestch] == EdgeState.branch:
        send('changeroot', to=self.bestch)
        output("Node ", self.name, "sent <changeroot> to node ", self.namesDict[self.bestch])
    else:
        # when the <changeroot> message reaches the node incident to the lowest-weight outgoing edge, this node
        # sends a <connect> on the edge to start unifying with the new fragment.
        # The nodes on which the unification is performed may become the new core nodes and the roots
        # of their respective subtrees.
        send (('connect', self.level), to=self.bestch)
        self.stach[self.bestch] = EdgeState.branch
        output("Node ", self.name, "sent <connect> to node ", self.namesDict[self.bestch], " in changeroot process.")
        checkConnectsQueue()
```

```
def receive(msg=('changeroot'), from_=q):
    # Process <changeroot> message
    output("Node ", self.name, "received <changeroot> from node ", self.namesDict[q])
    changeroot()
```

Observer node

- The **Observer** node is an additional node that is in charge of collecting the final MST computed by the algorithm and plotting it
- This node is initialized with a list of all the agents
- The node initializes a graph with all the nodes but without any edge
- When the algorithm terminates, the final core nodes detect termination and send a <*complete*> message to the Observer
- When the observer receives this message, it sends a <*obsQuery*> message to query the agents.
- The agents respond to the observer's query with a <*answer*> message with the state of the channels
- When the **Observer** receives an <*answer*> message, it analyzes it and adds the BRANCH edges to the graph, if not already present
- When it has received an answer from all the agents, it plots the final MST