Gallager Humblet Spira Algorithm

(Advanced Distributed Systems)

by

Shruti Goel, Ruchir Dhiman

(2016MCS2657, 2016MCS2685)

MASTER OF TECHNOLOGY

in

Computer Science & Engineering

Introduction

The Gallager Humblet Spira (GHS) algorithm is a distributed algorithm used to create a minimum spanning tree in an environment where nodes only communicate with each other via message passing and are unaware of the entire network.
One application of a distributed spanning tree algorithm is to create a tree which, then, can be used for purposes such as broadcasting. This minimum spanning tree can consequently reduce message passing costs in such scenarios by reducing the total number of messages needed to spread an update to all other nodes in the network. Creation of a tree is also necessary if we want to use other distributed-tree-algorithms like leader election etc.
The GHS uses some basic properties of a minimum spanning tree. A fragment is a sub-tree of the MST and an outgoing edge from a fragment has one end-point in the fragment and the other in another fragment. Now, if an outgoing edge 'e' is the least weight outgoing edge for a fragment F, then $F \cup e$ is also a fragment. The GHS algorithm uses this information to generate a MST in a distributed manner. All nodes start as fragments containing a single node. Gradually the nodes fuse together and form larger fragments; this is done by identifying the least weight outgoing edge for each fragment. The nodes in each fragment co-operate to locate the least weight outgoing edge by running a distributed algorithm. Finally, when only one fragment remains, we have are MST.
This is facilitated by providing each fragment with its own unique name and a level. On the combination of two fragments all the nodes, in these two fragments, are given a new name and they all acquire one level. A fragment F_1 can connect to another fragment F_2, if level(F_1) $\leq$ level(F_2). The level of the new fragment is chosen as the greater of the two original levels. If the levels of both fragments is the same, then, the new level is, simply, old-level + 1.
All of the above is accomplished by message passing between nodes; furthermore, the nodes can only talk to their neighbours. This means that they aren't aware of the entire network. So, the formation of the MST is entirely accomplished in a distributed manner.

Design

Here, each node in the network is being simulated by one thread, where each thread is locally maintaining the necessary information required by the algorithm. Every node has a unique node Id, and for each node, the variable *name* stores the fragment name, *level* stores the fragment level. The information about neighbors and edges is stored using three arrays *peerList, weightList, and status.* The boolean flag *stop*, which is initially false, is made true when the termination conditioned is reached.

Message Passing

Communication between nodes is essential for any distributed network, and in real life networks, this is handled by the underlying networks on which the system is set-up. Here, however, there is no such network, and all communication between the nodes is being done by using message queues. We have used a BlockingQueue which is a part of *java.util.concurrent library.* All accesses and functions of BlockingQueue class are inherently synchronized.
Each thread has its own BlockingQueue to store the messages it has received. The node, simply, waits for a message to arrive, and on its arrival, adds the received message to the queue. Processing of messages is done by a function described below.
The messages we send are an object of class M_Message. The type of message sent is determined by a field. The processing of the message is done based on what this field contains.

Network Setup

The program starts in the main *sub_main* class. The main function reads the input file and adds the unique node Ids to a set (to avoid repetition); the size of the set, then, gives the number of nodes (N).We, then, create an adjacency matrix of size $N\ x\ N$, and assign the edge weights in the respective position.
Once the adjacency matrix has been set-up, main calls the *add_node* function, which creates a new node by providing the following information: the node ID and the edge information. The edge statuses are set as *BASIC* by default by the constructor. Finally, the *add_node* calls the *Thread.start* for the node to start the

execution of the node thread.

Function: initialization

The threads initially wait for sometime, so as to ensure that all other nodes in the system are created and can be contacted. Then, the nodes call the *initialization* function, which performs the first phase of the algorithm which is finding the least weight edge (by calling function *getMinWt*) it has in its edge list and sending a connect message to the respective peer.
The processing of all messages is done in the *processMessage* function described below.

Function: findMin

The findMin function is the function nodes locally call to find the node with the least weight outgoing edge. It initially calls *getMinWt* to obtain the least weight outgoing edge with *BASIC* status. The node, then, sends a *TEST* message to the respective peer.

Function: reportMethod

The reportMethod function is used by a node to check the number of *REPORT* messages it has received from its child nodes (the nodes it is connected to via branch edges). This is necessary because the node with the least weight outgoing edge may be the last one to send the *REPORT* message;therefore, waiting for all nodes to report is essential.
This method also sets the bestnode and bestWt received out of all the best weights it has received or found itself.

Function: changeRoot

This function is used to change the root node for the entire fragment to the node which has the least weight outgoing edge. Also, this function is also used to send the *CONNECT* message to the closest peer so as to begin the connection of the

fragments.

Function: processMessage

This function is used to actually process all the messages received by a node. The function simply calls a helper function based on the type of the message received (the functions have the same name as the message type). The odd one out, here, is the processing of *CHANGEROOT* message; this involves calling the changeRoot function already described.

Helper Functions

This section includes a brief description of the helper functions we have used.

- ***getMinWt***: This function simply finds the least weight outgoing edge for the node which has the status *BASIC*.
- ***connect***: This is the helper function used to handle the receipt of the connect message. It begins the connection of the fragments by following the EQ or LE rules of fragment joining.
  This is the function which sends the initiate message to the joining nodes to ensure all nodes in the network have the same name and level.
- ***test***: This function handles the *TEST* message and checks if the edge in question is eligible for being used to connect with another fragment or not. This is done by sending a *ACCEPT* or *REJECT* messages to the sender. Based on the outcome both nodes change the edge status to either *BRANCH* or *REJECT*.
- ***report***: This function is used to handle the *REPORT* messages received from the child nodes. Based on the information received it updates its best weight (variable: bestWt) and best node (bestNode) and reports the new information to its parent (by calling report).
- ***initiate***: This is the function which causes the changes in name and level when two nodes join. All this does is update the name and level to the ones received, and then, tells its neighbors to do the same.

- ***accept***: The *ACCEPT* message is received in response to the *TEST* message sent to another node. It indicates that the edge between the two nodes is the next eligible edge for connection. The node updates its bestnode and bestWt accordingly. Finally, it call the *reportMethod* internally.
- ***reject***: This function simply sets the status of the edge in question to *REJECT*. It then calls findMin to find the next best edge it can use for connecting to its remaining peers. Like *ACCEPT* this message is received in response to *TEST*.

Example

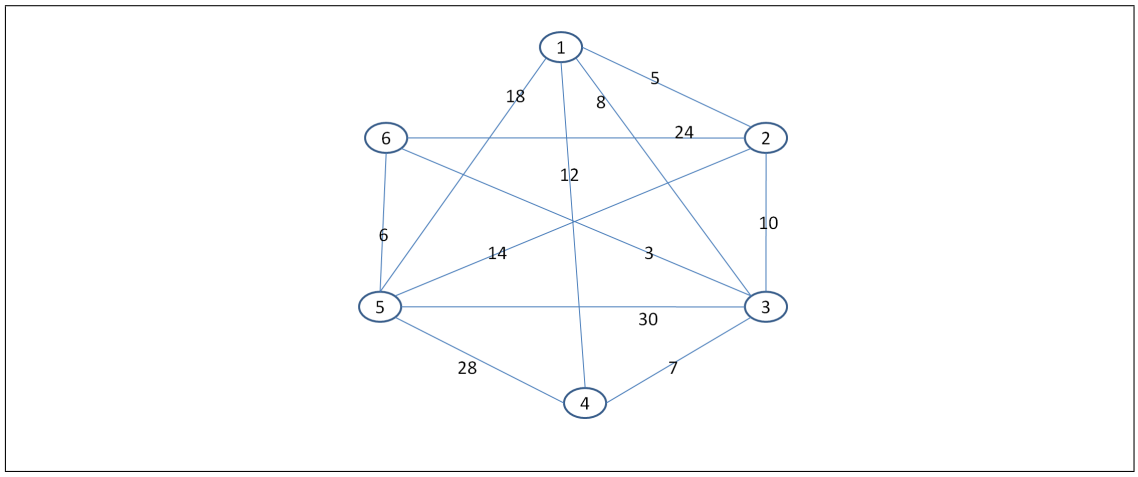

Figure 1: Initial Network

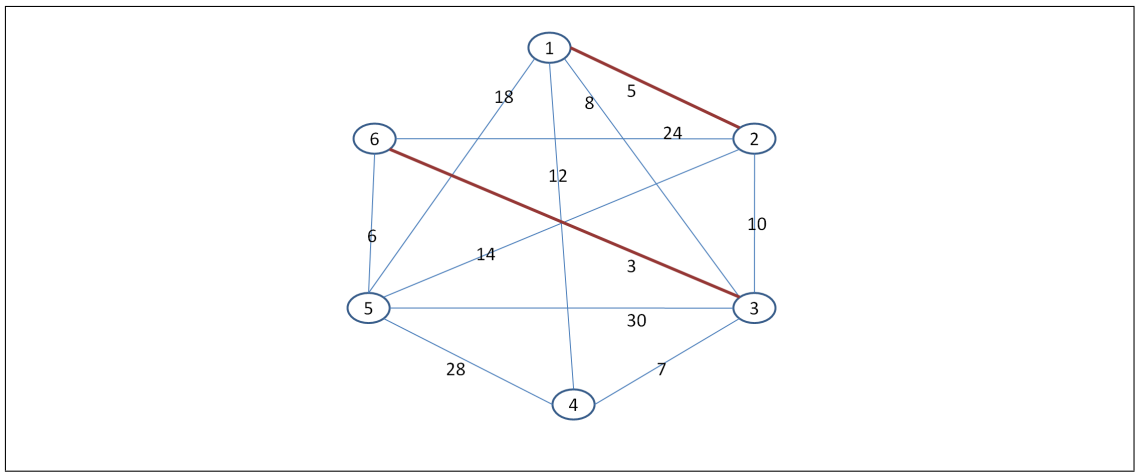

Figure 2: After Initialization Phase

After the *CONNECT* message is sent in the initialization phase, only edges 1-2 and 3-6 are set to *BRANCH* at both nodes. The *CONNECT* message through

nodes 4 and 5 are still to be processed on nodes 3 and 6 respectively.

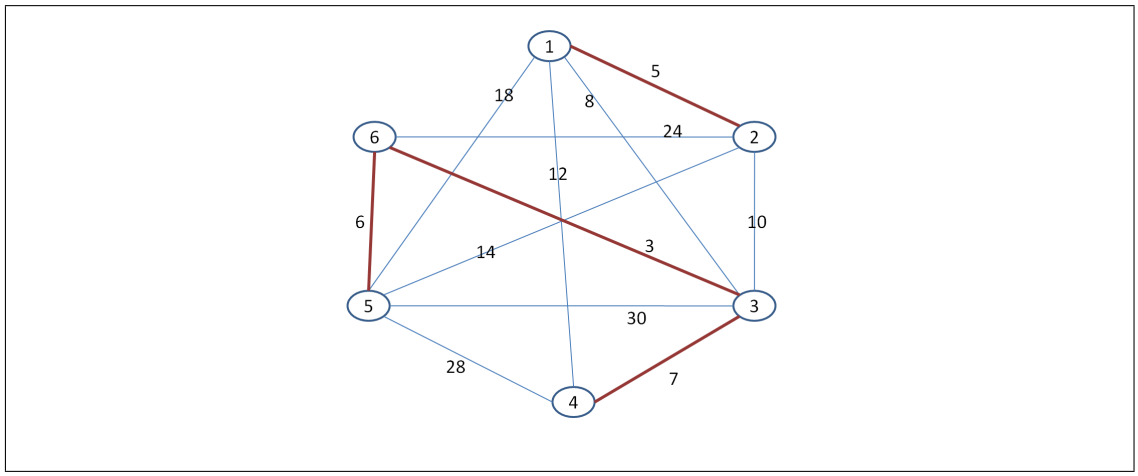

Figure 3: After *CONNECT* message of 4 and 5 are processed

Later due to level difference, the *CONNECT* message of node 4 and 5 are processed and edges 5-6 and 3-4 are set to *BRANCH* at both nodes.

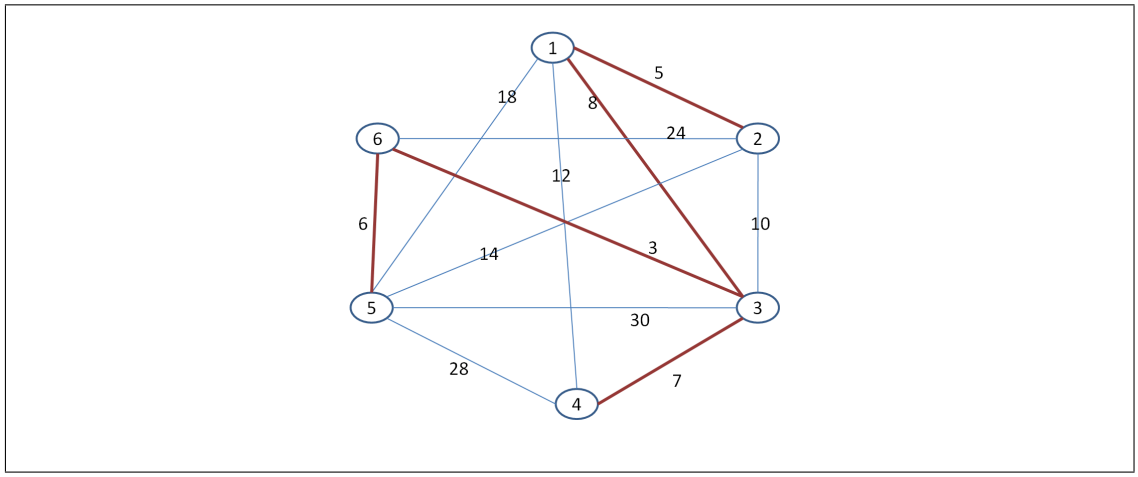

Figure 4: Final MST

The two fragments are to be connected by finding the min weight node; edge 1-3 is the min weight node. Finally, node 1 and node 3 are chosen as the core nodes of 2 fragments, and egde 1-3 is set to *BRANCH* at both nodes.