# COL 819 : Assignment - 2
# Distributed Minimum Spanning Tree

Shubham Gupta
csz198470@cse.iitd.ac.in

April 14, 2020

## 1 Introduction

Given a Graph $G = (V, E, W)$ which is connected,weighted and un-directed, its minimum spanning tree contains all the vertices $V$ of $G$ and subset of edges $E^{'} \in E$ such that tree is connected and sum of weights of edges in spanning tree is minimum. MSTs are have multiple applications like circuit designing and package routing in a network.

In distributed setting, we are required to compute the minimum spanning tree where each process/node will have the information of only a node of graph and its corresponding edges in graph. Essentially no process know the entire topology of graph. Each node will run a same algorithm which involves sending messages to its neighbours and receiving messages from them. We will follow the algorithm proposed by Gallager, Humblet, and Spira (1983) to solve this problem in a distributed setting. Assumption followed by authors is that each edge weight is unique. It can be easily seen that why this condition is important. Let's assume a network of 3 nodes which are connected to each other with edges of same weight. Since each node has information only about its own edges and each node has same information, each node can never know that which of its two edge will be part of graph's MST.

Algorithm begins by assuming each node as a fragment. Each fragment has a name and level associated with it. Initially each fragment has $level = 0$. Now each fragment asynchronously using its internal nodes look for minimum cost outgoing edge out of it to other fragments. Once two fragments agree on common outgoing edge, they merge and form a new fragment. Nodes in old fragments, now change their fragment's name to new fragment's name. As algorithm progresses, number of fragments decreases and finally only 1 fragment remains. This fragment is the required MST. In next section, we underline the fine details of algorithm using implementation snippets.

## 2 Implementation details

Algorithm is implemented in python 3.7. Multi-processing library of python is used to spawn a process for each node. System used to implement had specification of 1.8GHz Intel core i5 , 8GB RAM and Mac OS Mojave OS. In this simulation, main program reads the graph data from input file, spawns processes which are equal to number of nodes in graph. Now main process provides each process with information of corresponding node id and adjacent edges. This implementation follow closely true distributed system where shared memory doesn't exist. Each process can only exchange information through message passing. For message passing, we utilize the Queue library provided by multiprocessing framework of python. Queue library support message passing between processes. We create a separate queue for each node and now each process can put message in other process's queue. Using this, each process can read the message from its own queue.

We have implemented the pseudo-code provided in paper Gallager et al. (1983) and slides Sarangi (2020). Edge class is implemented as follows. We provide an comparator operator for class, so its become easier for each node to sort the edges as per requirement. Initially each edge is in state of "Basic" which means that it has not decided whether its a part of the MST or not. "Branch" means its part of the MST and "Rejected" means its not a part of the MST.

```
class Edge():
   def __init__(self,src,dst,weight):
      self.src = src
      self.dst = dst
      self.weight = weight
      self.state = "Basic" ### Other can be Branch and
         ↪ Rejected

   def __str__(self):
      return str(self.src) +"->" + str(self.dst) + ":" + str
         ↪ (self.weight)
   def __gt__(self, other):
      if(self.weight>other.weight):
         return True
      else:
         return False
   def __lt__(self,other):
```

```
        if(self.weight <=other.weight):
            return True
        else:
            return False
```

Similarly each node is in initial state of ”Sleeping”. It also maintains its level, fragment name which it is part of, parent edge, best_wt etc. All the required variables can be seen in below listing.

```
class Node():
    def __init__(self,id,V,adj_nodes,debug=False):
        self.id = id
        self.V= V
        self.state = "Sleeping" ## Sleeping, Find,Found
        self.FN = -1
        self.LN = None
        self.debug = debug
        self.best_edge = None
        self.best_wt = sys.maxsize
        self.test_edge = None
        self.parent = None
        self.find_count = 0
        self.nbrs = {nbr[0]:Edge(self.id,nbr[0],nbr[1]) for
            ↪ nbr in adj_nodes}
        self.edges = list(self.nbrs.values())
        self.edges.sort()
        self.msg_count = {}
```

In this implementation, main process awakens each node from sleeping state during spawning the process itself. So when a process is spawned, it execute the following function.

```
    def wakeup(self,msg_qs):
        self.edges[0].state = "Branch"
        self.find_count = 0
        self.LN = 0
        self.state = "Found"
        connect_msg = Message("Connect",self.id,self.edges[0].
            ↪ dst,L=0)
        msg_qs[self.edges[0].dst].put(connect_msg)
```

Each node maintains its edges in increasing order of weight. So, first edge is always shortest weight edge. Node marks this edge as ”Branch” and put itself in ”Found state”. Since initially each node is fragment at level 0 itself, it sends a connect message with its level to its neighbour with lowest weight edge.

Whenever a node receives a Connect message, if incoming connect message' level is lower or equal to its own level, then it sends the initiate message to incoming edge's node otherwise it defers the response till its own fragment level become equal or higher.

```
if msg.type == "Connect":
   if msg.L < self.LN:
      self.edges[incoming_edge_index].state = "Branch"
      initiate_msg = Message("Initiate",self.id,
         ↪ incoming_node,L=self.LN,F=self.FN,S=self.state)
      msg_qs[incoming_node].put(initiate_msg)
      if self.state == "Find":
         self.find_count += 1
   elif self.edges[incoming_edge_index].state == "Basic":
      msg_qs[self.id].put(msg)
   else:
      initiate_msg = Message("Initiate",self.id,
         ↪ incoming_node,L=self.LN+1,F=self.edges[
         ↪ incoming_edge_index].weight,S="Find")
      msg_qs[incoming_node].put(initiate_msg)
```

```
elif msg.type == "Initiate":
   self.LN = msg.L
   self.FN = msg.F
   self.state = msg.S
   self.parent = incoming_node
   self.best_edge = None
   self.best_wt = sys.maxsize
   for edge in self.edges:
      if edge.dst != incoming_node and edge.state == 'Branch
         ↪ ':
         initiate_msg = Message("Initiate",self.id,edge.dst,
            ↪ L=self.LN,F=self.FN,S=self.state)
         msg_qs[edge.dst].put(initiate_msg)
```

```
            if self.state == "Find":
                self.find_count += 1
    if self.state == "Find":
        self.test(msg_qs)
```

Initiate messages basically provide new fragment's information to each internal node. Whenever two fragments connect, they broadcast initiate messages to their internal nodes, to allow them to update their information and figure out next outgoing minimum weight edge of new fragment. When a node receives initiate message, it updates its own fragment level and name to new fragment's information. It send the initiate message to its branch edges except incoming message edge. Also it increments a find_counter whenever it sends a initiate message. Whenever a internal node sends a initiate message to other node, it expects that node to report back with lowest weight outgoing edge from new fragment it could find. This node also executes the following test function to find the lowest weight outgoing edge it can find. Test function basically asks its neighbour node with lowest weight Basic edge whether if it is a part of same fragment or not. So it will wait for accept or reject from that node. If reject message is received then it will move on to the next lower weight basic state edge. Once this node receives report messages containing lowest weight outgoing edge from all the node to which it sent initiate message to, it will aggregate the results and report back to node which originally sent the initiate message to it.

```
def test(self,msg_qs):
    flag_found= False
    for edge in self.edges:
        if edge.state == "Basic" and not flag_found:
            flag_found = True
            self.test_edge = edge.dst
            test_msg = Message("Test",self.id,edge.dst,L=self.
                ↪ LN,F=self.FN)
            msg_qs[edge.dst].put(test_msg)

    if not flag_found: ### No test send, so simply report to
        ↪ parent.
        self.test_edge = None
        self.report(msg_qs)
```

When a node receives a test message, if incoming message node's fragment level is higher than own fragment level, then it defers the reply. Otherwise If it is equal,

then Reject message is sent since both node belongs to same fragment level otherwise Accept is sent.

```
elif msg.type == "Test":
   if msg.L > self.LN: ### put to wait
      msg_qs[self.id].put(msg)
      time.sleep(0.001)
      #time.sleep(2)
   elif msg.F != self.FN: ## different fragment so accept
      accept_msg = Message("Accept",self.id,incoming_node)
      msg_qs[incoming_node].put(accept_msg)
   else: ### in same fragment
      if self.edges[incoming_edge_index].state == "Basic":
         self.edges[incoming_edge_index].state = "Rejected"
      if incoming_node != self.test_edge:
         reject_msg = Message("Reject",self.id,incoming_node
            ↪ )
         msg_qs[incoming_node].put(reject_msg)
      else: #### received test from edge where test was sent
         ↪  so now do test on another edge
         self.test(msg_qs)
```

When a node receives Accept message, it updates its own knowledge about the least weight outgoing edge out of its fragment and executes report. Report function basically checks whether it has received replies from all node it sent initiates to and sends the weight of least weight outgoing edge to parent node( node from which it received initiate message). If it sends a report back to parent, it mark itself in Found state thus declaring that in current level iteration, its work is done. Similarly if it receives Reject message then it calls the test function to ping lowest basic edge out of remaining edges.

```
elif msg.type == "Accept":
   self.test_edge= None
   if self.edges[incoming_edge_index].weight < self.best_wt:
      self.best_edge = incoming_node
      self.best_wt = self.edges[incoming_edge_index].weight
   self.report(msg_qs)
elif msg.type == "Reject":
   if self.edges[incoming_edge_index].state == "Basic":
      self.edges[incoming_edge_index].state = "Rejected"
```

```
    self.test(msg_qs)
```

```
def report(self,msg_qs):
    if self.find_count ==0 and self.test_edge is None:
        self.state = 'Found'
        report_msg = Message("Report",self.id,self.parent,S=0,
            ↪ F=0,L=0,W=self.best_wt)
        msg_qs[self.parent].put(report_msg)
```

When a node receives a Report message except from its parent, it compares the report message's weight with its own best weight, update the incoming message edge as best edge and send the updated weight to its parent. But if a node receives a Report from its parent, then that means they were the earlier core nodes of the fragment. So, change_root procedure is invoked which basically broadcasts Change_root message along the best_edge path. Once it reaches to node with lowest outgoing weight edge, then this sends Connect message to the node on the other end of this edge. This edge and these 2 nodes become core edge/node of combined new fragment. Now further Initiate messages are broadcast to internal nodes to figure out new least weight outgoing edge.

```
elif msg.type == "Report":
    if incoming_node != self.parent:
        self.find_count = self.find_count -1
        if msg.W < self.best_wt:
            self.best_wt = msg.W
            self.best_edge = incoming_node
        self.report(msg_qs)
    else:
        if self.state =="Find":
            msg_qs[self.id].put(msg)
            time.sleep(0.0001)
        elif msg.W > self.best_wt: ### core edges and need to
            ↪ switch path
            self.change_root(msg_qs)
        elif msg.W == sys.maxsize and self.best_wt == sys.
            ↪ maxsize:
            done = True
            outputq.put([self.id,self.parent,self.msg_count,
                ↪ self.LN])
            if self.debug:
```

```
            print("Nothing to do for , ", self.id , " ",self
                ↪ .parent, " best edge", self.best_edge)
elif msg.type == "Change_root":
    self.change_root(msg_qs)
```

```
def change_root(self,msg_qs):
    best_edge_index = self.findEdgeIndexUsingNodeId(self.
        ↪ best_edge,"dst")
    if self.edges[best_edge_index].state == "Branch":
        change_root_msg = Message("Change_root",self.id,self.
            ↪ best_edge)
        msg_qs[self.best_edge].put(change_root_msg)
    else:
        self.edges[best_edge_index].state = 'Branch'
        connect_msg= Message("Connect",self.id,self.best_edge,
            ↪ L=self.LN)
        msg_qs[self.best_edge].put(connect_msg)
```

Once leaf nodes of fragment are unable to find a new outgoing least weight edge, that signals the completion of procedure. In our implementation though, we have used a 5 seconds wait at each node for new message to figure out whether algorithm has finished executing or not.

# 3 Running Time Analysis

First, Since each edge can be rejected once, at most $2E$ reject messages can be sent. At each level, each node can receive at most 1 Initiate message and 1 Accept message. Also, each node can send at-most 1 connect, report/change-root and successful test message. Since there are at-most $\log(\mathrm{N})$ levels, total message complexity is

$$O(\mathrm{N}\log(\mathrm{N}) + E)$$

.

In given sample input which consist of 100 nodes, and 150 edges, table 1 provides the distribution of message exchanged.

Similar to Message complexity, time complexity will also be $O(\mathrm{N}\log(\mathrm{N}))$. At my system with 2 cores, execution time is approx. 8.62 seconds.

We also run the implementation with Graph size of 10,20, ... 100 where number of edges is $2 * number of nodes$. Following plots shows the number of messages and

| Message Type | Number of messages passed |
|---|---|
| Connect | 13803 |
| Initiate | 400 |
| Test | 6517 |
| Report | 6943 |
| Reject | 58 |
| Accept | 190 |
| Change_root | 22 |
| Total message passed | 27933 |

Table 1: Message distribution for Graph of 100 Nodes and 150 Edges

time taken against graph size.

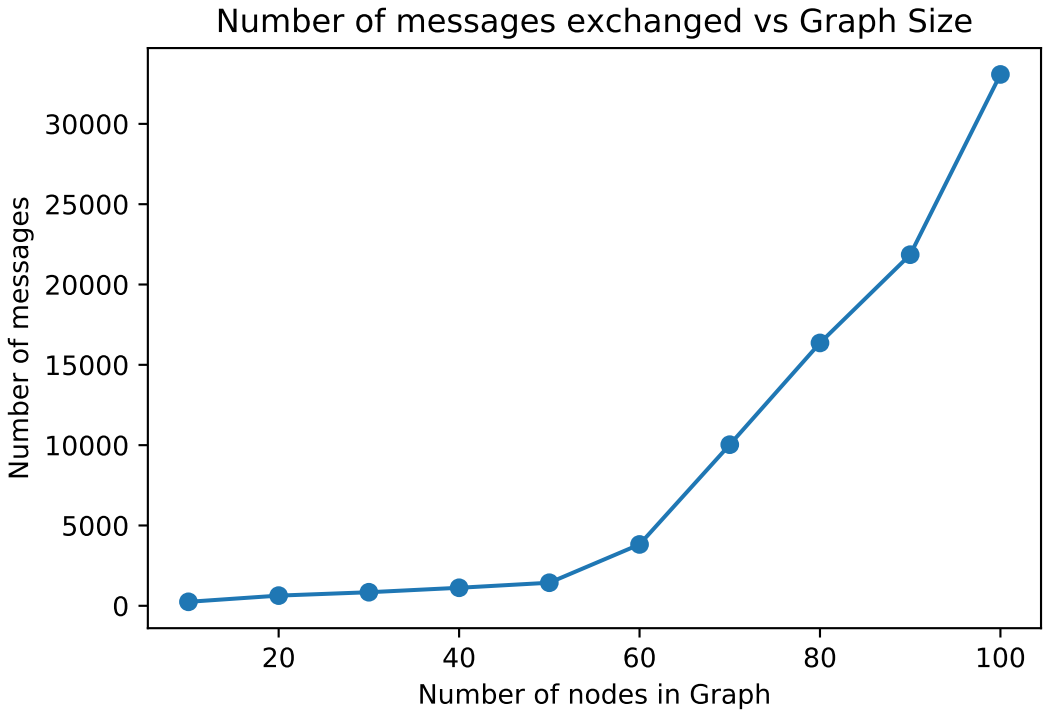


Figure 1: Number of message exchanged vs Graph size

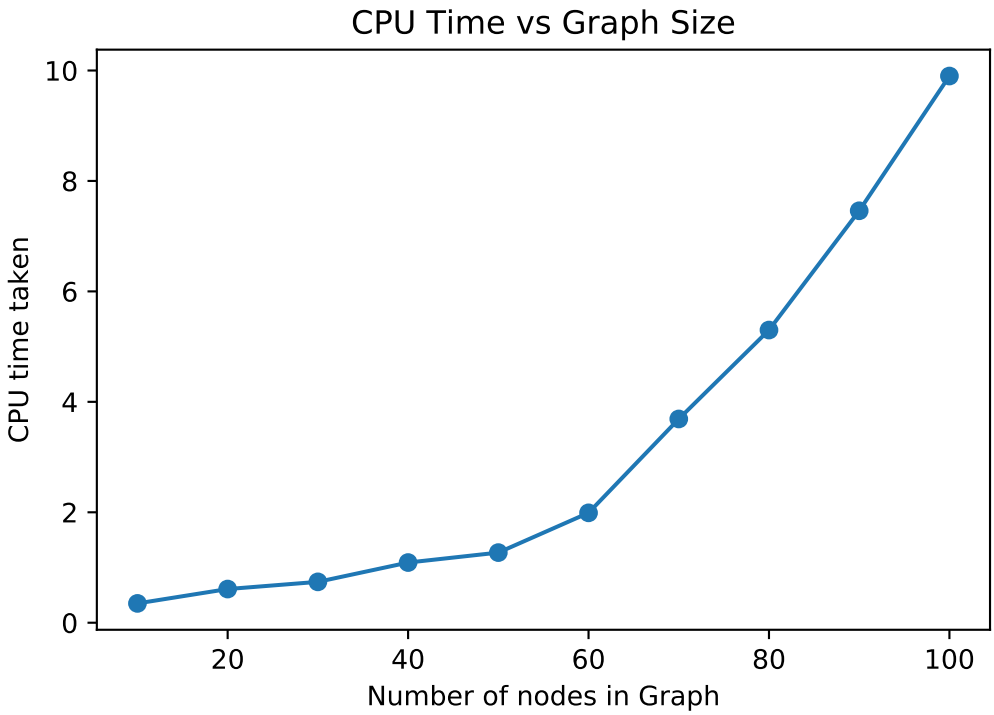


Figure 2: CPU Time taken(seconds) vs Graph Size

# 4 How to run the code

Following implementation is tested on Macos, and initially I was receiving a "Too Many files open" error on spawning 100 process for sample input. Then I had to use following commands on shell with sudo to get rid of the error.

```
sysctl -w kern.maxfiles=20480
sysctl -w kern.maxfilesperproc=18000
ulimit -S -n 10000
```

Program can be run using following command, please note that path is specific for my system. Please change appropriately according to your system.

```
python3 GHS_jupyter.py inp
for example-
/Users/shubham/anaconda3/bin/python GHS_jupyter.py
   ↪ test_cases/connected_100_999
```

# References

Gallager, R. G., Humblet, P. A., & Spira, P. M. (1983, January). A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, *5*(1), 66–77. Retrieved from https://doi.org/10.1145/357195.357200 doi: 10.1145/357195.357200

Sarangi, S. R. (2020). *Assorted algorithms: Minimum spanning trees.* Retrieved from http://www.cse.iitd.ac.in/~srsarangi/courses/2020/col_819_2020/docs/mst.pdf