Assignment № 2

Akshat Khare, IIT Delhi 14/04/2020

Introduction to GHS

There exists centralized algorithms to compute minimum spanning trees, example: Kruskal, Prim's algorithm, etc. In 1983, Gallager, Humblet and Spira proposed an algorithm which computes minimum spanning tree from a distributed system perspective. MST is required in many applications involing networking overlay and other distributed applications. GHS calculates MST in at max $\mathbf{5Nlog_2N + 2E}$ messages without every node just knowing weight of edges it is connected to. It is proved that if the edge weights are unique MST is unique [1]. If the edges are not unique we can establish a trichotomy by enforcing an order based on the unique ids of the nodes across the edge.
GHS works by merging fragments till there is a single connected fragment. It is also noteworthy that once edge is designated as a *Branch* edge it stays that way till end. Hence the algorithm proceeds progressively marking required edges as Branch edges till MST is found. More information about the algorithm can be found in original paper[1]. I also referred Guy Flysher and Amir Rubinshtein's explanation[2] of GHS because original paper failed to explain many concepts in required brevity.

Details of Implementation

I used $python3$ for my implementation. I made use of $multiprocessing$ module for ipc procedures and communications. Since communication channels need to be multiple producer and single consumer (a node will receive message from it's neighbouring nodes), I made use of $queues$ instead of $pipes$ as pipes are single producer single consumer.
I made use of **blocking receive calls** as node essentially is trigerred by incoming messages and there is no background task to be done which will get affected by process going to sleep. Since number of processes (number of nodes) is at max 100. I used $mutiprocessing.Process$ class as I tested that 100 processess can be handled by Linux OS. If the number of processes would be higher I would have made use of $multiprocessing.Pool$ as then OS would not be able to handle that high number of processes.
At last the main process which parsed the input file and spawned N process sends a message to each process asking about their branch edges. Main process compiles the results and prints on the console in the format required. This additional process which was essential to assignment involved $2N$ messages and didn't affect the message complexity of $\mathbf{5Nlog_2N + 2E}$.
Packages **numpy, multiprocessing and sys** need to be installed.
The skeleton of Node class is:

Listing 1: Node Class.

```python
class Node:
    """docstring for Node"""
    def __init__(self, infoStart):
        self.uid = infoStart.uid #unique identifier
        self.edges = infoStart.edges #list of tuples of the form (nodeId, ↩
           Weight)
        self.edgeToWeight = {} #a dictionary for the fast retrieval of ↩
           edgeWeight from NodeId
        for i,j in self.edges:
            self.edgeToWeight[i] = j
        self.queues = infoStart.queues #a dictionary of the messageQueue ↩
           of the form (nodeId, queue)
        self.queue = infoStart.queue #queue of the self
        self.masterQueue = infoStart.masterQueue #queue of the master ↩
           process which spawned all processes
        self.SN = "Sleeping" #State of the Node
        self.SE = {} #Dictionary of the form (nodeId, status of edge) used↩
            to get the status of edges
        self.test_edge = None
        for i,j in self.edges:
            self.SE[i] = "Basic"
```

About Listing 1... Each node maintains its own set of variables, consisting of its state (denoted by SN and assuming possible values Sleeping, Find, and Found) and the state of the adjacent edges. The state of edge j is denoted by SE(j) and can assume the possible values Basic, Branch, and Rejected. It is possible for the edge states at the two nodes adjacent to the edge to be temporarily inconsistent[1]. Initially for each node, SN = Sleeping and SE (j) = Basic for each adjacent edge j. Each node also maintains a fragment identity FN, a level LN, and variables best-edge, best-wt, test-edge, in-branch, and find-count, all of whose initial values are immaterial.[1]

Listing 2: Node's knowledge of the graph

```python
class InfoStart:
    """docstring for InfoStart"""
    def __init__(self, uid, edges, queues, queue, masterQueue):
        self.uid = uid #Unique id
        self.edges = edges #neighbouring edge and their weights
        self.queues = queues #neighbouring edges queues
        self.queue = queue #nodes own queue
        self.masterQueue = masterQueue #master process queue
```

About Listing 2... A node is spawned with only above information.

Listing 3: Master spawning nodes and waking up

```python
if __name__ == '__main__':
    .....
    nodesQueues = [multiprocessing.Queue() for i in range(numNodes)] #master process has queues of all of the nodes
    masterQueue = multiprocessing.Queue() #master's own queue
    processes = []
    for i in range(numNodes):
        queuedic = {} #node's relevent queue data
        for j,k in adjacencyList[i]:
            queuedic[j] = nodesQueues[j] #only providing neighbouring edges queues
        infoStart = InfoStart(i, adjacencyList[i], queuedic, nodesQueues[i], masterQueue)
        p = multiprocessing.Process(target=nodecode, args=(infoStart,))
        p.start()
        processes.append(p)
    nodesQueues[0].put(Message("wakeup", [], -1))#waking up nodeId 0
```

About Listing 3... Main function (process) spawns node processes with relevent information embedded in $Infostart$ class. It wakes up 0^{th}node.

Listing 4: Function executed by every Node

```python
def nodecode(infoStart):
    node = Node(infoStart)
    node.receiveAndProcess()
```

About Listing 4... Node initiates its revelent information table from infostart which is a instance of $Infostart$ class. Internals of receiveAndProcess are described below. It does a blocking receive message call and processes the messages endlessly until it halts itself or receive a halting reply from master.

Listing 5: Receive Message and Process

```python
 class Node:
    ....
    def receiveAndProcess(self):
        while(True):
            message = self.queue.get() #blocking receive
            self.processMessage(message)
    def processMessage(self, message):
        typemessage = message.typemessage
        senderid = message.senderid
        metadata = message.metadata
        if DEBUG: print("Process ", self.uid, "received ", typemessage, "from " , senderid, "with metadata ", metadata)
        if typemessage=="wakeup":
            self.wakeup()
        elif typemessage=="connect":
            self.connect(metadata[0], senderid, message)
        elif typemessage=="initiate":
            self.initiate(metadata[0], metadata[1], metadata[2], senderid)
        elif typemessage=="test":
            self.testResponse(metadata[0], metadata[1], senderid, message)
        elif typemessage=="accept":
            self.accept(senderid)
        elif typemessage=="reject":
            self.reject(senderid)
        elif typemessage=="report":
            self.reportResponse(metadata[0], senderid, message)
        elif typemessage=="changeRoot":
            self.changeRootResponse()
        elif typemessage=="queryStatus":
            self.queryStatusResponse()
        else:
            if DEBUG: print("Unrecognised message")
```

About Listing 5... Handling of the messages is as described by paper[1].Message has $metadata$ which consists of extra data bundled up with message.

Listing 6: Message Class

```
class Message():
    """docstring for Message"""
    def __init__(self, typemessage, metadata, senderid):
        self.typemessage = typemessage #Type of message like wakeup, connect, etc
        self.metadata = metadata #Any extra information bundled up with data
        self.senderid = senderid #Id of whoever sent the message
```

About Listing 6... This message is passed via $multiprocessing.Queue$.

Listing 7: Core nodes inform main process that MST is computed in last elif

```
class Node:
    ....
    def reportResponse(self, weightparam, senderEdge, message):
        ....
        elif weightparam == self.bestWeight and self.bestWeight == float('↩
           inf'):
            self.masterQueue.put(Message("done", [self.SE, self.inBranch],↩
                self.uid)) #GHS is computed, inform master
```

About Listing 7... When the bestWeight received by core nodes is received and it is infinite, core nodes inform main process that MST is computed.

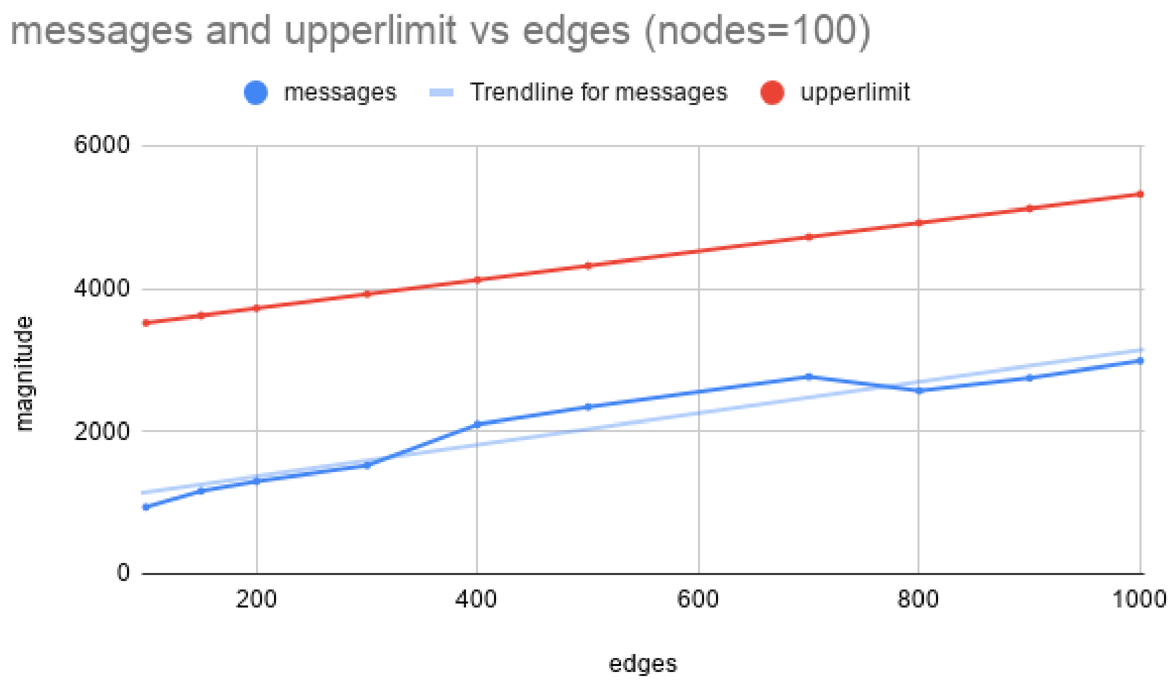
Listing 8: Main process gathers MST information

```
if __name__ == '__main__'::
    ....
    recvmessage = masterQueue.get()
    if recvmessage.typemessage=="done":
        for i in range(numNodes):
            nodesQueues[i].put(Message("queryStatus",[], -1)) #get Branch ↩
               edges information
class Node:
    ....
    def queryStatusResponse(self):
        self.masterQueue.put(Message("queryAnswer", [self.SE, self.↩
           inBranch], self.uid))
        sys.exit() #MST computed and sent information to master, time to ↩
           exit
```

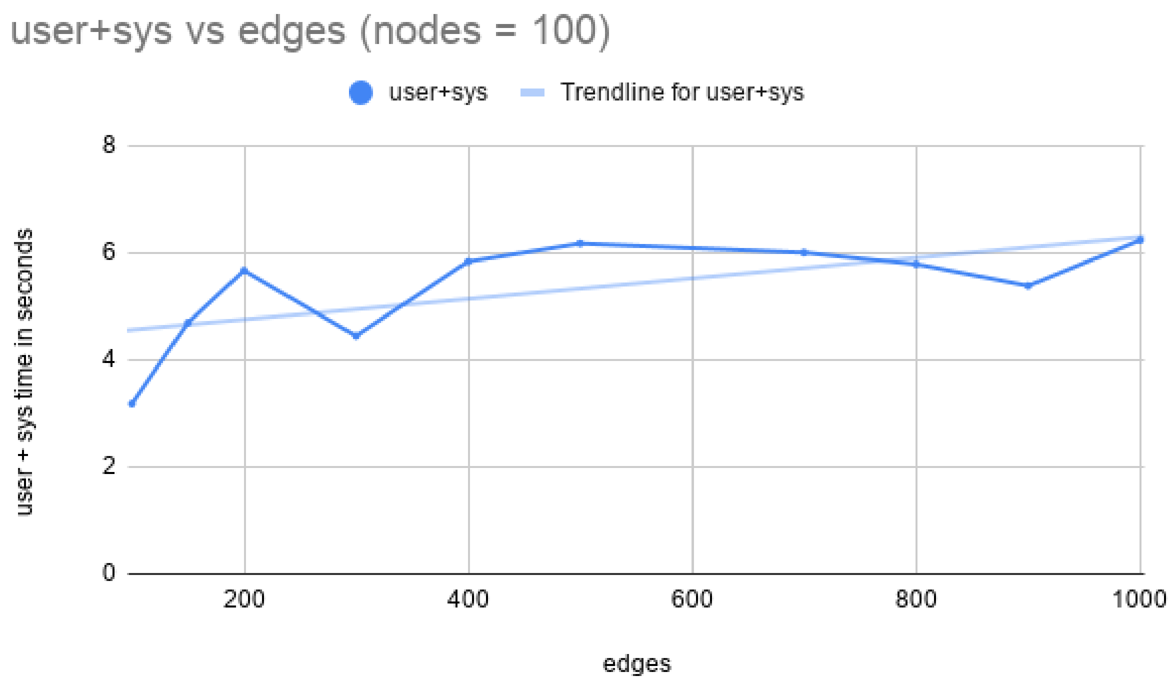
About Listing 8... Main process gathers information about branch edges and prints the output according to specification.

Running time analysis

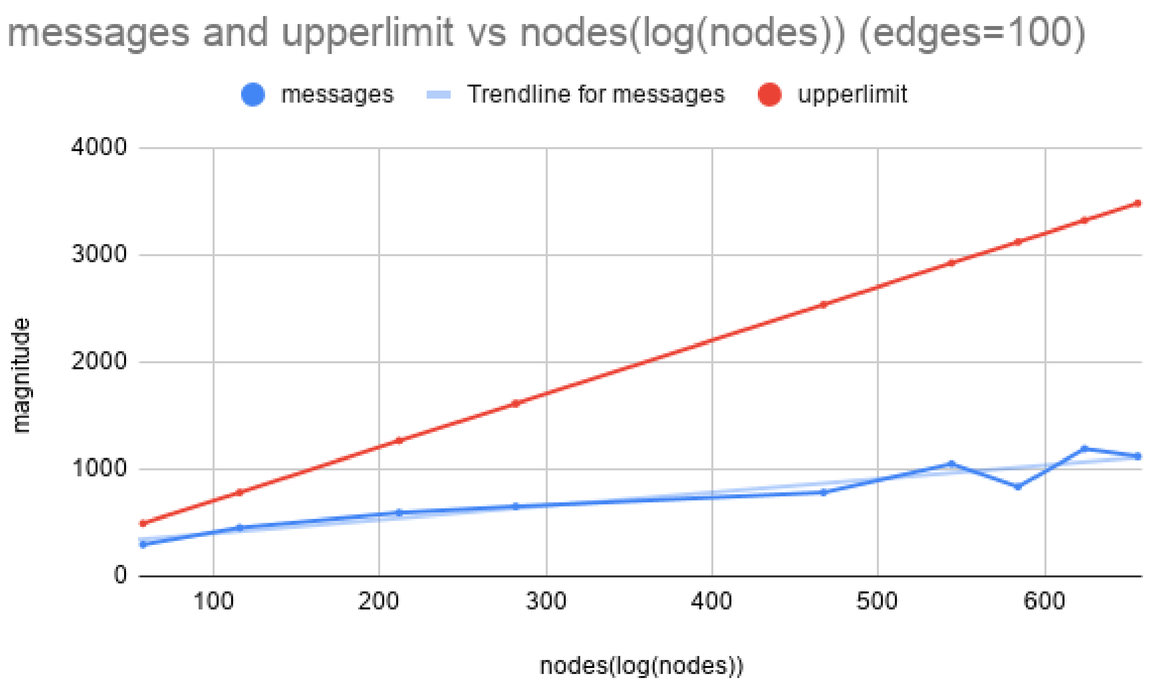
Upper limit of messages is $\mathbf{5N\log_2N + 2E}$.



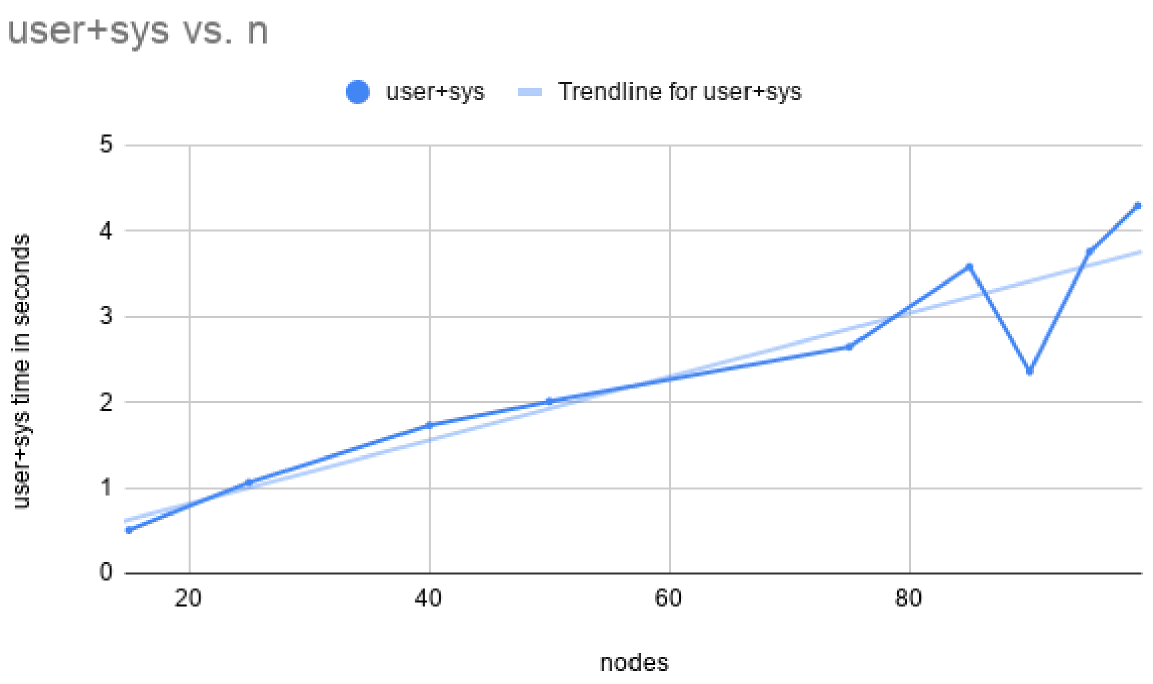
Keeping nodes constant we see that number of messages indeed increase linearly.



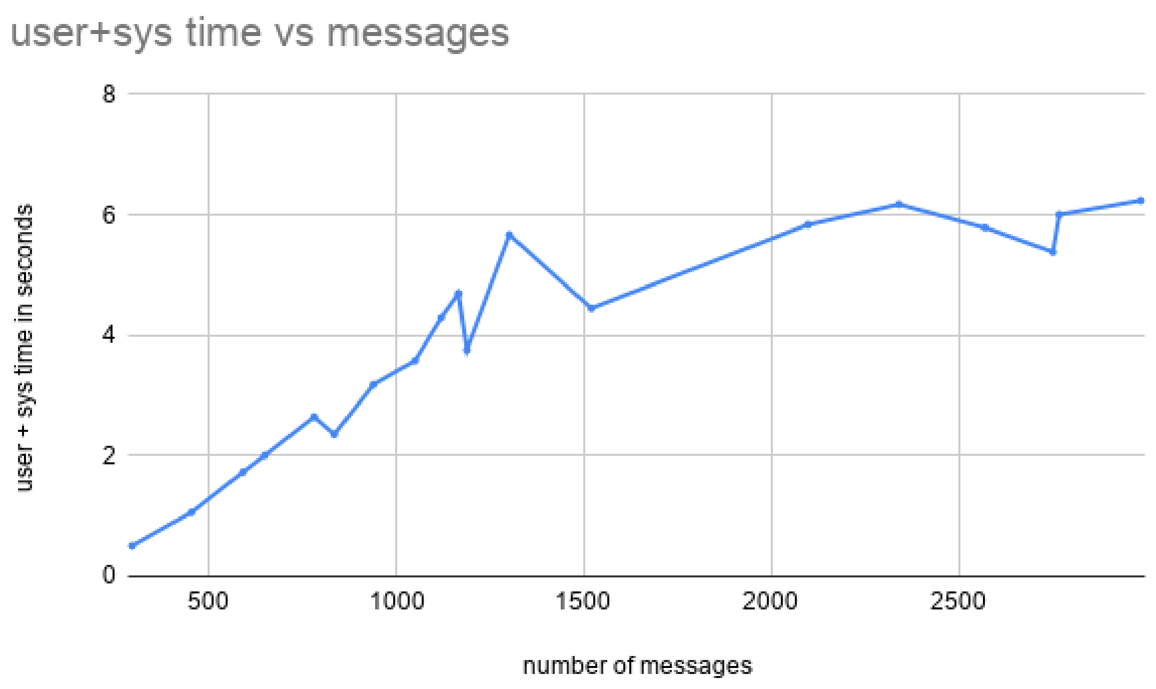
Time does not exactly follow a linear trend. This might be because of my code being executed on virtual machine (virtualbox)



Number of messages is indeed linear with $5N\log_2 N$



Time does not exactly follow a linear trend. This might be because of my code being executed on virtual machine



Time does not exactly follow a linear trend. This might be because of my code being executed on virtual machine

How to run the code

Listing 9: Execute this on console

```bash
#! /bin/bash
pip install multiprocess numpy
python main.py [name_of_input_file]
```

About Listing 9... My code requires $multiprocessing$ and $numpy$ packages to work. Please install them and then simply run $main.py$ with inputfile. The output in desired format will be printed in console.

References

[1] R. G. Gallager, P. A. Humblet and P. M. Spira. *A distributed algorithm for minimum weight spanning trees*. Massachusetts Institute of Technology, 1983.

[2] Guy Flysher and Amir Rubinshtein adaptation. *A distributed algorithm for minimum weight spanning trees*.