COL 819 Assignment 2

Dipika Tanwar (2020MCS2456)
Hemant Dhankhar (2020MCS2458)

April 13, 2021

1 Introduction to GHS algorithm

A minimum spanning tree is a tree in which all vertices are connected and weight of the tree is minimum.
If all the nodes and edges are known beforehand to a single system beforehand then traditional algorithms like Prim's algorithm and Kruskal's algorithm exist by which we can find the minimum spanning tree, but the task to find the minimum spanning tree becomes tricky once the system gets distributed as in distributed systems there is no single entity or system which knows about the complete system, so that it can apply traditional minimum spanning tree algorithms like Prim's or Kruskal's. In distributed system Gallager, Humblet and Spira (or GHS) algorithm comes to our rescue. GHS algorithm is an asynchronous algorithm which uses message passing to form the minimum spanning tree.We have implemented the GHS algorithm here which finds the minimum spanning tree in a distributed fashion. We have used threads to simulate the distributed environment.

2 Details of Implementation

For implementing GHS we have used Python 3 programming language. We have tested the code on a machine with Intel i5 processor with 4GB of RAM running on Linux(Ubuntu) OS. For easy debugging and comprehension of code we have divided the code into three files :

- main.py
- Node.py
- constants.py

Out of the above files main.py files serves as a vessel for the final integration while constants.py contains the variables related to the GHS algorithm. The heart of the code where the main part of the implementation lies is present in the Node.py file, following is the content of the crucial part of Node class:

Node.py

```
class Node():

    def recv_test( self , data):
        q, leveld , named = data[mauve'mauvesendermauve'],data[
            mauvemauve'mauvemauvemauvelevelmauve
            mauvemauve'mauve ], data[mauvemauve'mauvemauvename
            mauve']
        if leveld > self. level :
            sendData = data.copy()
            sendData[mauve'mauverepeatmauve'] = True
            sleep (0.01)
            self .sendMessage(MSG.TEST,self,data.copy())
        elif self .name == named:
            if self .status [q.nodeId] == BASIC:
                self .status [q.nodeId] = REJECT
            if q != self .testNode:
                sendData = {}
                sendData[mauve'mauvesendermauve'] = self
                self .sendMessage(MSG.REJECT, q,sendData)
            else:
                self .findMin()
        else:
            sendData = {}
            sendData[mauve'mauvesendermauve'] = self
            self .sendMessage(MSG.ACCEPT, q , sendData)

    def recv_accept( self ,data):
        q = data[mauve'mauvesendermauvemauve'mauve]
        self .testNode = None
        wpq = self .nbrDict[q.nodeId]
        if wpq < self.bestWt:
            self .bestWt = wpq
            self .bestNode = q
        self .report ()
```

```
def recv_reject(self,data):
    q = data[mauve'mauvesendermauvemauve'mauve]
    if self.status[q.nodeId] == BASIC:
        self.status[q.nodeId] = REJECT
    self.findMin()

def recv_report(self,data):
    q, w = data[mauve'mauvesendermauve'], data[
        mauvemauve'mauvemauvebestWtmauve']
    if q!= self.parent:
        if w < self.bestWt:
            self.bestWt = w
            self.bestNode = q
        self.rec += 1
        self.report()
    else:
        if self.state == FIND:
            # sleep(0.1)
            sendData = data.copy()
            sendData[mauve'mauverepeatmauve'] = True
            sleep(0.01)
            self.sendMessage(MSG.REPORT,self,sendData)
        elif w > self.bestWt:
            self.changeRoot()
        elif w == MAX and self.bestWt == MAX:
            self.stopAll()

def recv_connect(self, data):
    q, L = data[mauve'mauvesendermauvemauve'mauve], data[
        mauvemauve'mauvemauvemauvelevelmauve
        mauvemauve'mauve]
    if L < self.level:
        self.status[q.nodeId] = BRANCH
        sendData = {}
        sendData[mauve'mauvesendermauve'], sendData[mauve'
            mauvemauvelevelmauvemauvemauve'mauve], sendData[
            mauve'mauvenamemauve'], sendData[mauve'mauvestate
            mauvemauve'mauve] = self, self.level, self.name, self.
```

```
                state
            self.sendMessage(MSG.INITIATE, q, sendData)
        elif self.status[q.nodeId] == BASIC:
            sendData = data.copy()
            sendData[mauve'mauverepeatmauve'] = True
            sleep(0.01)
            self.sendMessage(MSG.CONNECT, self, data.copy())
        else:
            sendData = {}
            sendData[mauve'mauvesendermauve'], sendData[mauve'
                mauvemauvelevelmauvemauvemauve'mauve], sendData[
                mauve'mauvenamemauve'], sendData[mauve'mauvestate
                mauvemauve'mauve] = self, self.level + 1, self.nbrDict[
                q.nodeId], FIND
            self.sendMessage(MSG.INITIATE, q, sendData)

    def recv_initialte (self, data):
        q, leveld, named, stated = data[mauve'mauvesendermauve'],
            data[mauvemauve'mauvemauvemauvelevelmauve
            mauvemauve'mauve], data[mauvemauve'mauvemauvename
            mauve'], data[mauvemauve'mauvemauvemauvestatemauve
            mauvemauve'mauve]
        self.parent, self.level, self.name, self.state = q, leveld,
            named, stated
        self.bestNode, self.testNode, self.bestWt= None, None, MAX
        self.status[q.nodeId] = BRANCH

        for r, _ in self.nbrList:
            if self.status[r] == BRANCH and r != q.nodeId:
                sendData = {}
                sendData[mauve'mauvesendermauve'], sendData[mauve'
                    mauvemauvelevelmauvemauvemauve'mauve],
                    sendData[mauve'mauvenamemauve'], sendData[mauve'
                    mauvestatemauvemauve'mauve] = self, leveld,
                    named,stated
                self.sendMessage(MSG.INITIATE,nodeInfo[r],sendData)
        if stated == FIND:
            self.rec = 0
```

```
            self.findMin()

    def recv_changeRoot(self):
        self.changeRoot()


    def initialize(self):
        self.status = {}
        for n, _ in self.nbrList: self.status[n] = BASIC

        self.parent = None
        bNbr, _ = self.nbrList[0]
        self.name,self.status[bNbr],self.level, self.state, self.rec = 0,
            BRANCH,0,FOUND,0

        bNbr = nodeInfo[bNbr]
        sendData = {}
        sendData[mauve'mauvesendermauve'] = self
        sendData[mauve'mauvemauvelevelmauvemauvemauve'mauve] =
            self.level
        self.sendMessage(MSG.CONNECT, bNbr, sendData)

    def findMin(self):
        bNbr = None
        for n, _ in self.nbrList:
            if self.status[n] == BASIC:
                bNbr = n
                break
        if bNbr != None:
            self.testNode = nodeInfo[bNbr]
            sendData = {}
            sendData[mauve'mauvesendermauve'] = self
            sendData[mauve'mauvemauvelevelmauvemauvemauve'mauve]
                = self.level
            sendData[mauve'mauvenamemauve'] = self.name
            self.sendMessage(MSG.TEST,self.testNode, sendData)
        else:
            self.testNode = None
```

```
            self.report()

    def report(self):
        count = 0
        for n,_ in self.nbrList:
            if self.status[n] == BRANCH and n != self.parent.nodeId:
                count += 1
        if(count == self.rec and self.testNode == None):
            self.state = FOUND
            sendData = {}
            sendData[mauve'mauvesendermauve'], sendData[mauve'
                mauvebestWtmauve'] = self, self.bestWt
            self.sendMessage(MSG.REPORT, self.parent, sendData)

    def changeRoot(self):
        if self.status[self.bestNode.nodeId] == BRANCH:
            sendData = {}
            sendData[mauve'mauvesendermauve'] = self
            self.sendMessage(MSG.CHNAGE_ROOT, self.bestNode,
                sendData)
        else:
            self.status[self.bestNode.nodeId] = BRANCH
            sendData = {}
            sendData[mauve'mauvesendermauve'], sendData[mauve'
                mauvemauvelevelmauvemauvemauve'mauve] = self, self
                .level
            self.sendMessage(MSG.CONNECT, self.bestNode,sendData)
```

3 Plots for the complexity analysis

We have attached four graphs, apart from the two graphs asked in the report we have done more complexity analysis upto 100 intervals to clearly depict the 2E + 5logN messsage complexity of the algorithm. It can be clearly observed that our implementation runs with the intended complexity of 2E + 5logN as seen in the graphs below.

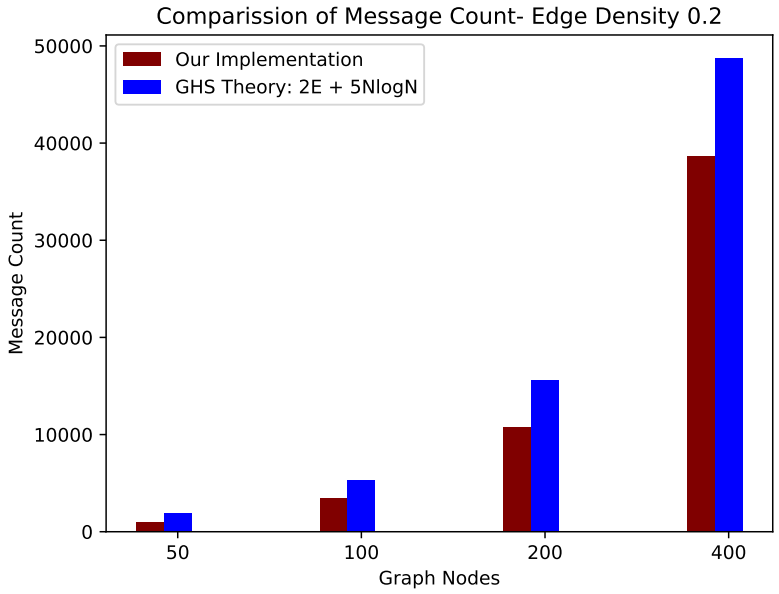

Figure 1: graph for number of messages vs graph nodes for sparse graphs.

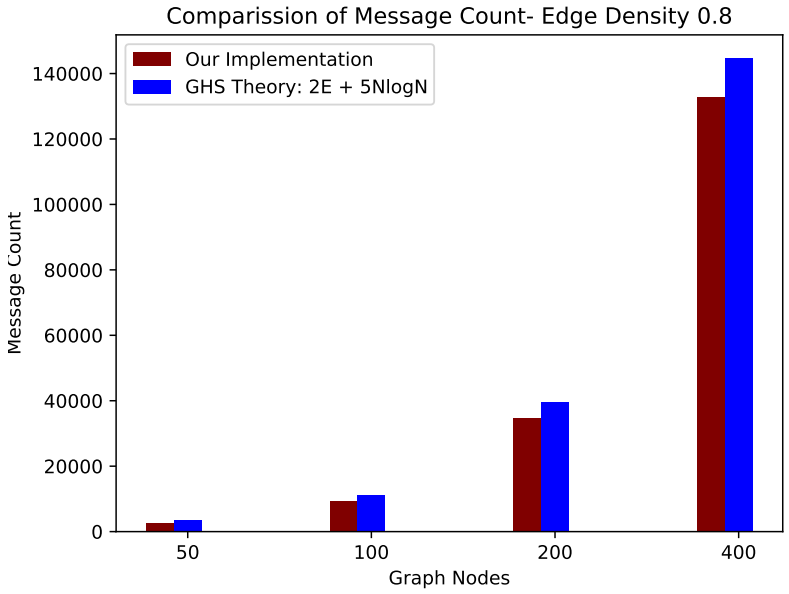

Figure 2: graph for number of messages vs graph nodes for dense graphs.

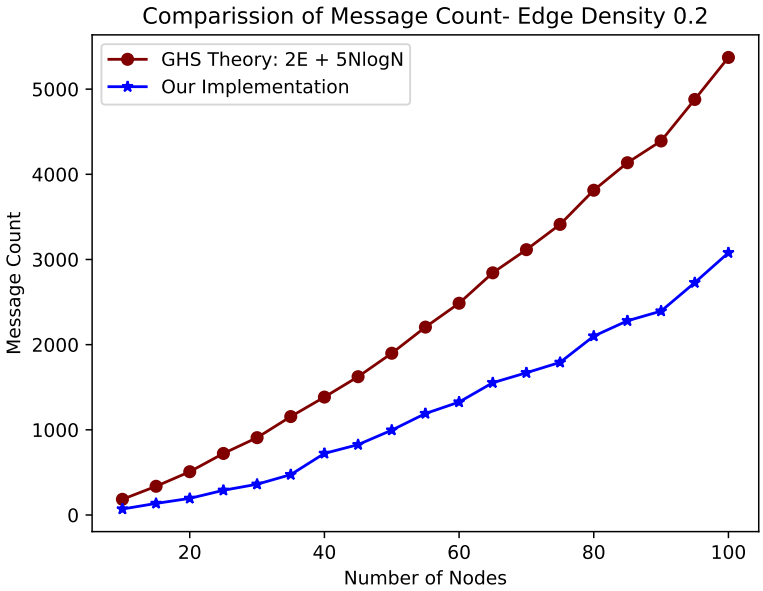

Figure 3: graph for number of messages vs graph nodes for sparse graphs.

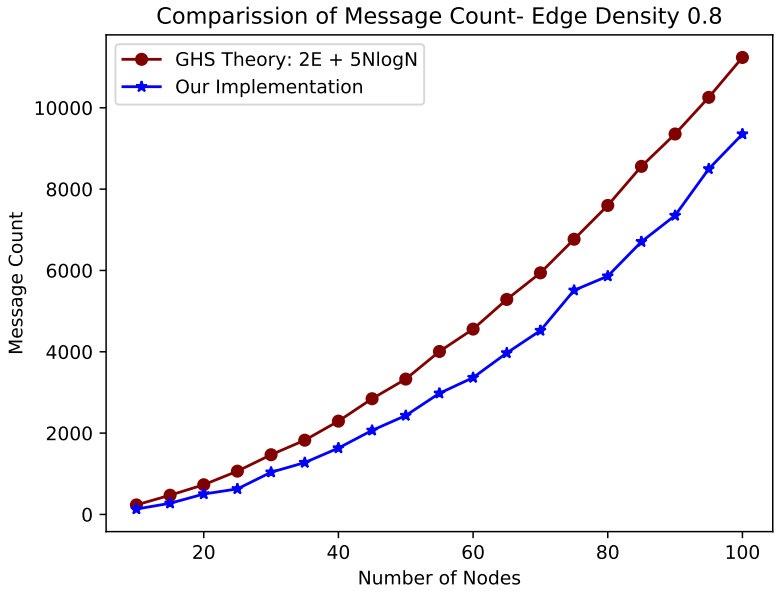

Figure 4: graph for number of messages vs graph nodes for dense graphs.