# Advanced Distributed Systems - Assignment 2(GHS Algorithm)

Pratik Karia - 2019MCS2568

April 2020

## 1 Introduction to GHS[2]

- GHS (Gallager, Humblet, Spira) Algorithm is a distributed algorithm for finding the Minimum Spanning Tree of a connected graph with unique edge weights. It specifically uses a message passing mechanism between the nodes to achieve the required task.
- The GHS Algorithm works on the idea of merging fragments of already computed minimum spanning tree. Initially each node is one fragment with 1 vertex. So initially we have fragments equal to the number of nodes in the graph. Gradually the fragments combine with the help of various messages and hence the size of minimum spanning tree increases and number of fragments decreases after each such combine. This process continues until there is just a single fragment remaining.
- A sample message passing of GHS Algorithm between a node p of fragment 1 and q of fragment 2 initiated by p is as follows:

  1. Node p sends connect message to node q
  2. If level(fragment1)¡level(fragment2) then q sends initiate message with the same level. Else if status of edge pq is basic then node q pushes the connect request back to it's queue(wait). Else the condition level(fragment1)== level(fragment2) holds and q sends initiate message with level = level(fragment1)+1 and name as the edge weight.
  3. Now p sets the name,level and state same as q and sets q as it's parent.Further p sends the same initiate message which it received to all it's neighbours which are present in fragment 1. Finally if state of p is find means it has to find the minimum edge and report it so it sets a counter(rec) to 0 and finds minimum weight edge to node not present in it's fragment.
  4. Suppose minimum basic edge found by p is pr. So now p sends test message with it's level to the node r which is basically seeking out if r can combine into it's fragment. If no such edge is there then p reports it's parent that it found no such edge.
  5. Now r receives the test message from p. It checks if the fragment in which p is present has a level ¿ it's own level. If true then it waits for it's level to increase and be equal to p's level and so it pushes the message received back to it's queue(wait). Else if name of both fragments is same and the edge between them is basic means p and r belong to the same fragment and hence we reject edge pr. Now if p is not the node which r was testing when it reached p's message then it sends reject message to p else r finds it's minimum outgoing edge. But if name of fragment of p and q is different then r sends accept message to p.
  6. Now if p receives accept request from r then it fixes r as the bestNode it can report and pr as the bestWeight and it reports it to it's parent. If p received reject message then it first sets state of edge pr as reject and finds some other minimum weight edge to report.
  7. Let a be parent of p. Now if p sends report message to a and current bestWeight at parent is greater than bestweight sent by p then it changes it bestWeight and bestNode and reports to it's own parent. Else if state of a is find means it is still trying to find bestNode and so it pushes the incoming message back to queue(wait). Else if bestWeight received by p is greater than bestWeight at a then the root of the fragment is changed. But if the bestWeight and weight received both are undefined, means no minimum edge could be found and we stop the algorithm as we have found the MST.
  8. Now suppose the status of edge between a and bestNode set by p is branch then the root is changed to bestNode. Else the status is set as branch which means that edge is approved to be a part of MST and a connect request is sent to bestNode and we go back to point 1.

# 2 Implementation[1]

## 2.1 Libraries Used

- thread.h
- bits/stdc++.h
- unistd.h

## 2.2 Code Files

- **ghs.cpp -** The file consisting of the main function which includes the the main algorithm.cpp file which implements the algorithm. It basically performs tasks like reading the input file, creating nodes of the graph, spawing the threads and running them, generating logs and summary as per user's wish.
- **headers.h -** The file which contains definition of all the structures and global variables and is imported in algorithm.cpp
- **algorithm.cpp -** The file which contains the implementation of the GHS Algorithm.

## 2.3 Code Implementation

1. **Reading of input :** The input is read using ifstream line by line and each line is parsed using a function processLine to generate a vector of integers containing node1,node2 and weight. After that it is stored in each data structure.

```
vector<int> processLine(string s)
{
  vector<int> out;
  s.erase( std::remove_if(s.begin(), s.end(),[](char ch){
  return ch=='(' || ch ==')'; }),s.end());
  s.erase( std::remove_if(s.begin(), s.end(),[](char ch){
  return ch==','; }),s.end());
  istringstream ss(s);
  while (ss)
  {
      string word;
      ss >> word;
      if(word=="")
        continue;
      out.push_back(atoi(word.c_str()));
  }
  return out;
}
```

2. **Starting of threads :** Threads are started using the thread.h library of C++. This library was used instead of pthread and pthread has support only in linux whereas thread (which uses pthread) has wider support. The number of threads is same as the number of nodes in the graph.

```
thread t[numNodes];
for(int i=0;i<numNodes;i++)
  t[i]=thread(entryFunction,allNodes[i]);

for(int i=0;i<numNodes;i++)
  t[i].join();
```

3. **Structures and Enums :** To store messages and edges and also denote various states of edges, nodes and types of messages, few structures and enums were created in headers.h file which also contains all global variables and a global mutex.

```
enum messageType
{START,CONNECT,INITIATE,TEST,REJECT,ACCEPT,REPORT,CHANGEROOT};
```

```
enum edgeState
{BASIC,BRANCH,REJECTE};

enum nodeState
{SLEEP,FIND,FOUND};

class Node;

typedef struct edge
{
  Node* destNode;
  int weight;
  edgeState state;
}edge;

typedef struct message
{
  messageType message;
  string name;
  nodeState state;
  int arguments[2];
}message;
```

4. **Node Attributes :** Node class is declared in file algorithm.cpp with multiple attributes.

```
class Node
{
public:
  nodeState state;
  string name;
  int level,bestWeight,rec,nodeId;
  Node* parent;
  Node* bestNode;
  int testNode;
  map<int,edge*> adjacentEdges;
  queue<message> messageQueue;
  mutex shareMutex;
  int numberOfMessages;
  Node(int nodeId)
  {
    this->state=SLEEP;
    this->level=-1;
    this->name="";
    this->parent=NULL;
    this->bestWeight=INT_MAX;
    this->bestNode=NULL;
    this->rec=-1;
    this->testNode=-1;
    this->nodeId=nodeId;
    this->numberOfMessages=0;
  }
};
```

5. **Pushing Messages in the queue :** The messages are pushed in the queue using the pushMessage function. These messages are later read using pullMessage function.

```
  void pushMessage(message msg)
  {
```

```
    this->shareMutex.lock();
    this->messageQueue.push(msg);
    this->shareMutex.unlock();
  }
void pullMessage()
{
    if(!this->messageQueue.empty())
    {
      this->shareMutex.lock();
      message newMessage = this->messageQueue.front();
      this->messageQueue.pop();
      this->shareMutex.unlock();
      /*Check the message type and call the appropriate
      functions using switch case (Code not copied as a very long code)*/
    }
}
```

6. **Connect Function :** The processConnectRequest function is called on receiving a *CONNECT* request.

```
void processConnectRequest(int levelOfNode,int key)
 {

    if(levelOfNode<this->level)
    {
      this->adjacentEdges[key]->state=BRANCH;
      message messageToSend.message=INITIATE;
      messageToSend.arguments[0]=this->level;
      messageToSend.arguments[1]=this->adjacentEdges[key]->weight;
      messageToSend.name=this->name;
      messageToSend.state=this->state;
      this->adjacentEdges[key]->destNode->pushMessage(messageToSend);
    }
    else if(this->adjacentEdges[key]->state==BASIC)
    {
      message messageToSend.message=CONNECT;
      messageToSend.arguments[0]=levelOfNode;
      messageToSend.arguments[1]=this->adjacentEdges[key]->weight;
      this->pushMessage(messageToSend);
    }
    else
    {
      int temp = this->adjacentEdges[key]->weight;
      message messageToSend.message = INITIATE;
      messageToSend.arguments[0] = this->level+1;
      messageToSend.arguments[1] = this->adjacentEdges[key]->weight;
      messageToSend.name=to_string(temp);
      messageToSend.state=FIND;
      this->adjacentEdges[key]->destNode->pushMessage(messageToSend);
    }
  }
```

7. **Initiate Function :** On receiving initiate message in the message queue, processInitiateRequest function is called.

```
  void processInitiateRequest(int levelOfNode,
  nodeState stateOfNode, int key, string nameOfFragment)
  {
    this->level = levelOfNode;
    this->state = stateOfNode;
```

```
    this->name = nameOfFragment;
    this->parent = this->adjacentEdges[key]->destNode;
    this->bestNode = NULL;
    this->bestWeight = INT_MAX;
    for(auto i:this->adjacentEdges)
      if(i.second->state==BRANCH && i.first!=this->parent->nodeId)
      {
        message messageToSend.message = INITIATE;
        messageToSend.arguments[0] = this->level;
        messageToSend.arguments[1] = i.second->weight;
        messageToSend.name=this->name;
        messageToSend.state=this->state;
        this->adjacentEdges[i.first]->destNode->pushMessage(messageToSend);
      }
    if(this->state==FIND)
    {
      this->rec=0;
      testPhase();
    }
  }
```

8. **Test Function :** On receiving Test message, processTestMessage code is executed as follows:

```
void processTestRequest(int levelOfNode,string nameOfFragment, int key)
 {
   if(levelOfNode>this->level)
   {
       //Add message back to queue
   }
   else if(nameOfFragment.compare(this->name)==0)
   {

     if(this->adjacentEdges[key]->state==BASIC)
       this->adjacentEdges[key]->state=REJECTE;
     if(this->testNode!=this->adjacentEdges[key]->destNode->nodeId)
     {
       //Send reject message to this->adjacentEdges[key]->destNode
     }
     else
       testPhase();
   }
   else
   {
     //Send reject message to this->adjacentEdges[key]->destNode
   }
 }
```

9. **Accept and Reject Function :** On receiving accept/reject request, processAcceptRequest and processRejectRequest are executed respectively.

```
void processAcceptRequest(int key)
{
  this->testNode=-1;
  if(this->adjacentEdges[key]->weight<this->bestWeight)
  {
    this->bestWeight=this->adjacentEdges[key]->weight;
    this->bestNode=this->adjacentEdges[key]->destNode;
  }
  report();
```

```
}

void processRejectRequest(int key)
{
  if(this->adjacentEdges[key]->state==BASIC)
    this->adjacentEdges[key]->state=REJECTE;
  testPhase();
}
```

10. **Report Function :** On receiving Report message, processReportRequest is executed.

```
void processReportRequest(int bestWt, int key)
{
  if(this->adjacentEdges[key]->destNode->nodeId!=this->parent->nodeId)
  {
    if(bestWt<this->bestWeight)
    {
      this->bestWeight=bestWt;
      this->bestNode=this->adjacentEdges[key]->destNode;
    }
    this->rec=this->rec+1;
    report();
  }
  else
  {
    if(this->state==FIND)
    {
      //Push message back into your own queue(Wait)
    }
    else if(bestWt>this->bestWeight)
      changeRoot();
    else if(this->bestWeight==INT_MAX && bestWt==INT_MAX)
      stopFlag=1; //Program Terminates
  }
}
```

11. **Changeroot Function :** On receiving Changeroot message, processChangeRoot function is called which calls changeRoot function.

```
void changeRoot()
{
  if(this->adjacentEdges[this->bestNode->nodeId]->state==BRANCH)
  {
    //Send Changeroot request to this->bestNode
  }
  else
  {
    //Edge gets approved to be a part of MST
    this->adjacentEdges[this->bestNode->nodeId]->state=BRANCH;
    //Send Connect request to this->bestNode

    //Adding Edges to MST
    addEdgeToMSTLock.lock();
    int ind = this->bestNode->nodeId;
    int index=ind;
    string outputString="";
    outputString+="("+to_string(this->nodeId)+","
    +to_string(this->adjacentEdges[index]->destNode->nodeId)+","
    +to_string(this->adjacentEdges[index]->weight)+")";
```

```
            mstEdges.push_back(outputString);
            addEdgeToMSTLock.unlock();
        }
    }
```

12. **Postprocessing and Displaying MST :** Finally the output from algorithm is postprocessed and converted to required format of output and displayed.

```
void postProcessing()
{
  map<int,pair<int,int>> mst;
  for(int j=0;j<mstEdges.size();j++)
  {
    vector<int> temp = processLine(mstEdges[j]);
    for(auto i:allEdges)
      if(i.first==temp[2])
        mst[i.first]=i.second;
  }

  for(auto i:mst)
    cout<<"("<<i.second.first<<",␣"<<i.second.second
    <<",␣"<<i.first<<")"<<endl;
}
```

# 3 Running Time Analysis

## 3.1 Time and Message Complexity

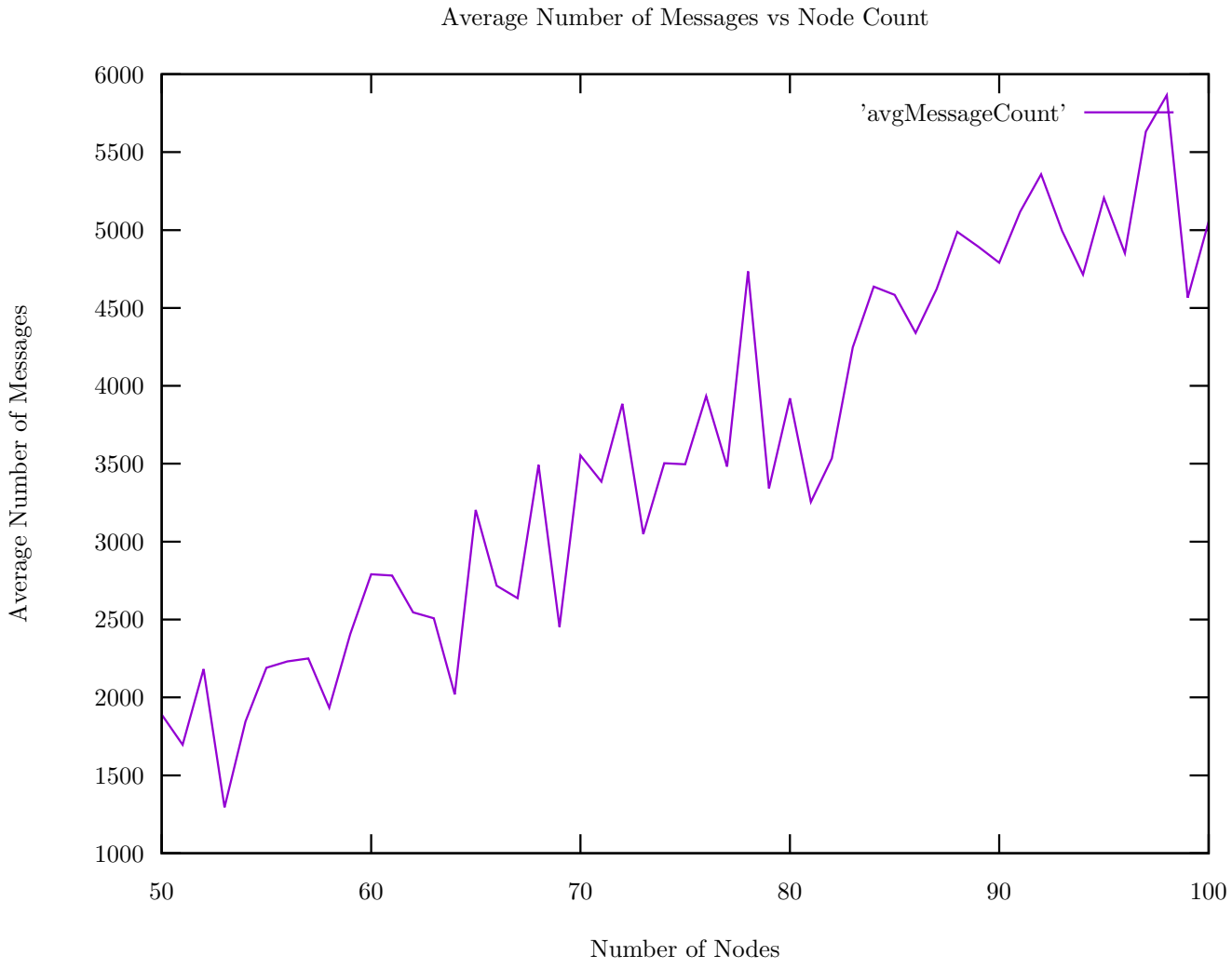
**Note :** N = Number of Nodes, E = Number of Edges

- **Time Complexity :** O(N*log(N))
- **Message Complexity :** O(2*E + 5*N*log(N))

## 3.2 Sample Input Provided

- **Number of Nodes -** 100
- **Number of Edges -** 150
- **Running Time of the program -** 4.3 seconds
- **Number of messages exchanged by the nodes -** 1505 (Varies every time program is executed)

### 3.3 Average Number of messages exchanged



### 3.4 Some Other Test Cases

I had executed my program on **complete graph** with number of vertices ranging from 50 to 100. The unique weights were assigned randomly. The execution time and the number of messages exchanged are as follows:

| Nodes | Edges | Time | Messages |
|---|---|---|---|
| 50 | 1225 | 3.03 | 2950 |
| 51 | 1275 | 3.55 | 3046 |
| 52 | 1326 | 4.04 | 3143 |
| 53 | 1378 | 5.31 | 3241 |
| 54 | 1431 | 4.93 | 3393 |
| 55 | 1485 | 5.52 | 3507 |
| 56 | 1540 | 4.73 | 3653 |
| 57 | 1596 | 5.69 | 3688 |
| 58 | 1653 | 5.16 | 3886 |
| 59 | 1711 | 4.50 | 4028 |
| 60 | 1770 | 5.20 | 4112 |
| 61 | 1830 | 5.23 | 4259 |
| 62 | 1891 | 5.29 | 4413 |
| 63 | 1953 | 4.60 | 4536 |
| 64 | 2016 | 6.27 | 4686 |
| 65 | 2080 | 5.20 | 4739 |
| 66 | 2145 | 5.90 | 5213 |

| Nodes | Edges | Time | Messages |
|---|---|---|---|
| 67 | 2211 | 6.32 | 5067 |
| 68 | 2278 | 5.28 | 5238 |
| 69 | 2346 | 6.63 | 5380 |
| 70 | 2415 | 6.21 | 5510 |
| 71 | 2485 | 6.74 | 5904 |
| 72 | 2556 | 6.15 | 5797 |
| 73 | 2628 | 5.47 | 5936 |
| 74 | 2701 | 5.80 | 6111 |
| 75 | 2775 | 7.28 | 6619 |
| 76 | 2850 | 6.32 | 6431 |
| 77 | 2926 | 7.48 | 6576 |
| 78 | 3003 | 6.96 | 6757 |
| 79 | 3081 | 6.38 | 6940 |
| 80 | 3160 | 7.87 | 7119 |
| 81 | 3240 | 6.70 | 7577 |
| 82 | 3321 | 7.26 | 7460 |
| 83 | 3403 | 7.91 | 7606 |

| Nodes | Edges | Time | Messages |
|---|---|---|---|
| 84 | 3486 | 7.18 | 7792 |
| 85 | 3570 | 7.29 | 7915 |
| 86 | 3655 | 7.46 | 8130 |
| 87 | 3741 | 7.88 | 8323 |
| 88 | 3828 | 7.62 | 8445 |
| 89 | 3916 | 8.21 | 9046 |
| 90 | 4005 | 8.13 | 9229 |
| 91 | 4095 | 8.38 | 9094 |
| 92 | 4186 | 8.47 | 9276 |
| 93 | 4278 | 8.71 | 9798 |
| 94 | 4371 | 8.27 | 9638 |
| 95 | 4465 | 8.69 | 9865 |
| 96 | 4560 | 8.76 | 10040 |
| 97 | 4656 | 9.26 | 10594 |
| 98 | 4753 | 9.5 | 10363 |
| 99 | 4851 | 8.64 | 10697 |
| 100 | 4950 | 10.90 | 10798 |

# 4 How to run the code

## 4.1 Run Code

**Note**: C++ 11 is required for the code to run.
Suppose code is in the folder 2019MCS2568.

```
$ cd 2019MCS2568
$ make
$ ./findMST <input file name>
```

**Note**: Output will be displayed in the desired format in the terminal.
There are multiple logging and summary options also provided in the code. To know more about it execute the following command:

```
$ ./findMST --help
```

## 4.2 Machine Configuration of Code Development

| Property | Configuration |
|---|---|
| Operating System | Ubuntu 18.04 |
| RAM | 12 GB |
| Number of CPU Cores | 4 |
| GCC version | 7.5.0 |

# References

[1] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[2] Artem Mazeev, Alexander Semenov, and Alexey Simonov. A distributed parallel algorithm for minimum spanning tree problem. *CoRR*, abs/1610.04660, 2016.