GHS Implementation Report

2017CS50405 Dhananjay Kajla
2017CS50421 Vijay Kumar Meena

12^{th} April 2021

Contents

1 Introduction to the GHS Algorithm

The GHS Algorithm (named after it's authors Gallager, Humblet and Spira) was first published in a 1983 paper. It gives an asynchronous distributed algorithm that runs identically on each node and in due time, each node learns which of it's incident edges belong in the minimum spanning tree.

The algorithm essentially works by creating, expanding and merging fragments of the graph. These fragments are guaranteed to be a part of the final MST of the complete graph.

Each node can be in one of 3 states : SLEEP, FIND and FOUND.

For a node, the edge between every neighbor can be in one of 3 states : BASIC, BRANCH and REJECT.

In basic state represents an edge that is yet to be tested for its presence in the MST. An edge in state branch represents an edge that is known to be in the final MST. And an edge in reject state has been rejected and cannot be a part of the MST.

At the start of the algorithm, all nodes are in state SLEEP, they are either woken up by a message, or they wake up spontaneously. For getting the best bound on messages, we shall assume that all nodes are woken up simultaneously at the start.

After that each node can either be in state FIND where it is looking for new edges for MST or it can be in state FOUND when it's passive.

Each node has a LEVEL and a NAME which is representative of it's fragment. Two fragments combine when either the node with smaller requests a connection to the larger or when two nodes with same level combine and form a node with level incremented by one. The first case is called the LT rule of merging, and the second case is the EQ rule.
explain The nodes communicate through the following types of messages :

- CONNECT : Request for merging the two nodes.
- INITIATE : Communicates new fragment level and identity to the neighbors
- TEST : Tests to check whether an edge belongs to the MST
- ACCEPT : Communicates to the neighbor that the received edge tested is accepted into the MST
- REJECT : Communicates to the neighbor that the received edge cannot possibly be in the MST
- REPORT : Reports the best weight outgoing fragment for each node So by this plot we can say that time is bounded by number of messages sent
- CHANGEROOT : The root node of the fragment is shifted to the node with the minimal outgoing edge

Under the assumption of initial awakening of all the nodes, GHS Algorithm guarantees the number of messages sent to be $< 2 \cdot E + 5 \cdot N \cdot \log_2(N)$. We shall see more about this in our experiments

We shall see the algorithm itself in next section :

2 Implementation Details

We have implemented a multi-threaded parallel implementation of the GHS Algorithm. Each node of the GHS is given a single thread in the implementation.

In this section, we shall outline the two parts of our implementation :

1. Inter-Thread Communication
2. GHS Algorithm at each node
3. Main Function that runs thread

2.1 Inter-Thread Communication

2.1.1 Messages

The messages sent by the threads(nodes) are dynamic arrays of strings.

We have created a message class to standardize all messages :

```
class Message
{
private:
  std::vector<std::string> msg; //!< message content
public:
  Message(std::vector<std::string> m) //!< Initiates a message
  {
    msg = m;
  }
  std::vector<std::string> getMessage() //!< returns the message
  {
    return msg;
  }
};

```

Listing 1: Message Class

For the ease of sending messages, we have created the a helper function msgCreater() :

```

Message *GHSNode::msgCreater(std::vector<std::string> msg)
{
    std::vector<std::string> m;
   m.push_back(std::to_string(nodeid));

    for(auto it : msg)
    {
      m.push_back(it);
    }

    Message *mg = new Message(m);
    return mg;
}

```

Listing 2: msgCreater

This dynamic array has the structure : {sender_id, Message type, Parameter1, Parameter2, ...}

Here is an example of message sending (sendMessage() will be explained later) :

```
m.push_back("initiate"); //!< Initiate()
m.push_back(std::to_string(LN)); //!< LN
m.push_back(FN); //!< FN
m.push_back(SN); //!< SN

 sendMessage(j, msgCreater(m)); //!< send Initiate(LN, FN, SN) on edge j

```

Listing 3: sendMessage Sample

2.1.2 Queues

To handle simultaneous messages, we have defined a thread safe data structure Queue built upon the std::queue provided by the Standard Template Library of C++ :

```
class Queue
{
  private:
  std::mutex mut; //!< mutex lock for thread safety
  std::queue<Message *> q; //!< Message queue
  int queueid; //!< Id of the node to which this queue belongs
  public:
  Queue()
  {
    queueid = -1; //!< Initialized as bad node
  }
```

```
  Queue(int qid)
  {
    queueid = qid; //!< ID of the queue
  }
  int getqueueid()
  {
    return queueid; //!< returns id for the node which owns this queue
  }
  void push(Message *m)
  {
    mut.lock(); //!< Puts the lock in place
    q.push(m); //!< This push operation is atomic
    mut.unlock(); //!< Unlocks the queue
  }
  Message *front()
  {
    mut.lock(); //!< Puts the lock in place
    if(q.size() == 0)
    {
      mut.unlock(); //!< Unlocks the queue
      return NULL; //!< Returns NULL if queue is empty
    }
    Message *temp = q.front(); //!< This front operation is atomic
    mut.unlock(); //!< Unlocks the queue
    return temp;
  }
  Message *pop()
  {
    mut.lock(); //!< Puts the lock in place
    int temp = q.size(); //!< Temporary variable to store queue size
    if(temp == 0)
    {
      mut.unlock(); //!< Unlocks the queue
      return NULL; //!< Returns NULL if queue is empty
    }
    Message *tmp = q.front(); //!< Temporary variable to store the front of queue
    q.pop(); //!< Pops the queue
    if((int)q.size() != temp - 1) //!< Pop should change the size of queue by 1
    {
      std::cerr << "QUEUE Error" << std::endl;
      exit(1);
    }
    mut.unlock(); //!< Unlocks the queue
    return tmp;
  }
  bool empty()
  {
    mut.lock(); //!< Puts the lock in place
```

```
      int temp = q.size(); //!< Temporary variable for queue size
      mut.unlock(); //!< Unlocks the queue
      return (temp == 0); //!< Returns if queue is empty
    }
    int getQueueSize()
    {
      mut.lock(); //!< Puts the lock in place
      int temp = q.size(); //!< Size of queue
      mut.unlock(); //!< Unlocks the queue
      return temp; //!< returns size of queue
    }
  };

```

Listing 4: Queue

Each thread(node) gets it's own Queue instance, which is instantiated in the Network class. A thread which of the queue has all the operations available to it whereas a non-owner thread can only push to the queue.

Each node (i.e. each Instance of the GHSNode class) is provided with a pointer to it's queue at it's instantiation. The pointer to other node's queue can be taken from the Network class which we describe next.

2.1.3 Network

To send and receive messages from different nodes, we have a defined an internet of sorts for the threads. This is the NETWORK class :

```
    class Network
    {
    private:
      std::unordered_map<int, Queue *> msg_queues; //!< Network queues
    public:
      Network(std::vector<int> nodes) //!< Network Constructor
      {
        for(auto it : nodes)
        {
          Queue *q = new Queue(it); //!< Initiates a new queue
          msg_queues[it] = q; //!< Initializes the hashmap
        }
      }
      Queue *getQueue(int i) //!< Get's the ith queue
      {
        if(msg_queues.find(i) == msg_queues.end())
        {
          std::cerr << "BAD network Request" << std::endl;
        }
        return msg_queues[i]; //!< Returns the pointer to ith queue
      }
    }
```

```
;
```

Listing 5: Network

The Network class is instantiated only once, and each node keeps a pointer to the single instance.

2.1.4 IsComplete

IsComplete is a class which we have defined to trigger the end of the algorithm. This is a thread join of sorts. Once this is complete we shutdown all nodes and ask them for the part of MST that they posses.

This class is instantiated only once and every node possesses a pointer to this instance so that any node can give out the halt signal.

```
struct IsComplete
{
  bool complete; //!< If set true, the threads exit
  IsComplete() //!< Constructor that sets the default value false
  {
    complete = false;
  }
};
```

Listing 6: IsComplete

2.1.5 GHS Algorithm Constructor

To show what we pass to each node initially, we shall list out the constructor for the GHSNode :

```
GHSNode::GHSNode(int nid, std::unordered_map<int, int> &neighbors, Network *net, IsComplete *iscom)
{
  this->nodeid = nid;
  this->nbd = neighbors;
  this->network = net;
  this->SN = "sleep";
  this->isc = iscom;
  this->nodequeue = net->getQueue(nid);
}
```

Listing 7: GHSNode constructor

As we can see, each node receives the pointer to it's queue and the pointer to the network along with the IsComplete instance.

2.1.6 Sending and Receiving messages

Now to summarize the thread messaging, let us give the code snippets for the sendMessage() and receiveMessage() methods :

```
void GHSNode::sendMessage(int dest, Message *m)
{
  Queue *q = network->getQueue(dest);
  q->push(m);
}

```

Listing 8: sendMessage

```
bool GHSNode::recieveMessage()
{
  msg = nodequeue->pop();
  return (msg != NULL);
}

```

Listing 9: receiveMessage

This completes our summary of thread communication

2.2 GHS Algorithm implementation at each node

Each GHSNode has the following state variables :

Variable Name	**Variable Title**	**Range of the variable**
NodeId	Id of the node	$\mathbb{N}$
SN	State of the node	[”sleep” , ”find” , ”found”]
FN	Name of the node	-
LN	Level of the node	$\mathbb{N}$
nbd	Adjacency list	$\{key : value\}$ key $\in \mathbb{N}$, value $\in \mathbb{N}$
SE	State of edges	[”baisc”, ”branch”, ”reject”]
best-edge	Currently the minimum weight outgoing edge	$\mathbb{N}$
best-weight	Current best outgoing weight	$\mathbb{N}$
test-edge	Edge currently being tested	$\mathbb{N}$
in-branch	Parent node of sorts	$\mathbb{N}$
find-count	number of MST Edges found	$\mathbb{N}$

The following types of messages can be sent :

Message Type	Description	Parameters
Connect	Request for merging the two nodes	Level
Initiate	Communicates new fragment level and identity to the neighbors	Level, Name, State
Test	Tests to check whether an edge belongs to the MST	Level, Name
Accept	Received test edge is accepted into the MST	-
Reject	Received test edge cannot possibly be in the MST	-
Report	Reports the best weight outgoing fragment for each node	best weight
ChangeRoot	Start Changing root of fragment	-

We will now list the 12 Algorithms as mentioned in the original GHS paper and our implementation for the same.

These are long pseudocodes and codes. If required one can skip to the next subsection on thread_runner.

2.2.1 Algorithm 1

Spontaneous awakening at each node at the starting instant. RUN() is the function that the thread starts.

```
void GHSNode::run()
{
    wakeup();
    runner();
}

```

Listing 10: run

2.2.2 Algorithm 2

Pseudocode :

wakeup()
  begin
  let m be adjacent edge of minimum weight
  $SE(m) \leftarrow Branch$;
  $LN \leftarrow 0$;
  $SN \leftarrow Found$;
  $Find - count \leftarrow 0$;
  send $Connect(0)$ on edge m

Implementation :

```
void GHSNode::wakeup()
{
    if(nbd.empty())
    {
        isc->complete = true;
    }
```

```
    for(auto it : nbd)
    {
      SE[it.first] = "basic";
      basic.insert(std::make_pair(it.second, it.first));
    }

    int m = findMinEdge(); // let m be adjacent edge of minimum weight;
    changestat(m, "branch"); //!< SE(m) <- Branch
    LN = 0; //!< LN <- 0
    SN = "found"; //!< SN <- Found
    find_count = 0; //!< Find-count <- 0
    std::vector<std::string> st;
    st.push_back("connect");
    st.push_back(std::to_string(LN));
    sendMessage(m, msgCreater(st)); //!< send Connect(0) on edge m
  }

```

Listing 11: wakeup

2.2.3 Algorithm 3

Pseudocode :

Response to $Connect(L)$ on edge j

begin if $SN = Sleeping$ **then execute procedure** wakeup;
    if $L < LN$
    then begin $SE(j) \leftarrow Branch$;
        send $Initiate(LN, FN, SN)$ on edge j
        if $SN = Find$ **then**
          $find\_count \leftarrow find\_count + 1$
      end
    else if $SE(j) = Basic$
      then place received message on end of queue
      else send $Initiate(LN + 1, w(j), Find)$ on edge j
end

Implementation :

```
    void GHSNode::handleConnect()
    {
      std::vector<std::string> mg = msg->getMessage();
      int j = std::stoi(mg[0]); //!< edge j
      int L = std::stoi(mg[2]); //!< L
      if(SN == "sleep")      //!< if SN = Sleeping
      {
```

```
      wakeup(); //!< execute procedure wakeup
    }
    if(L < LN) //!< If L < LN
    {
      changestat(j, "branch");

      std::vector<std::string> m;
      m.push_back("initiate"); //!< Initiate()
      m.push_back(std::to_string(LN)); //!< LN
      m.push_back(FN); //!< FN
      m.push_back(SN); //!< SN

      sendMessage(j, msgCreater(m)); //!< send Initiate(LN, FN, SN) on edge j

      if(SN == "find") //!< if SN = Find then
      {
        find_count++; //!< find-count <- find-count +1
      }

      free(msg);
      msg = NULL;
    }
    else if(SE[j] == "basic") //!< else if SE(j) = Basic
    {
      sendMessage(nodeid, msg); //!< place receieved message on end of queue
    }
    else
    {
      std::vector<std::string> st;
      st.push_back("initiate"); //!< Initiate()
      st.push_back(std::to_string(LN+1)); //!< LN +1
      st.push_back(std::to_string(nbd[j]));//!< w(j)
      st.push_back("find");//!< Find
      sendMessage(j, msgCreater(st)); //! send Initiate(LN+1, w(j), Find) on edge j
      free(msg);
      msg = NULL;
    }
  }

```

Listing 12: handleConnect()

2.2.4 Algorithm 4

Pseudocode :

Response to $Initiate(L, F, S)$ on edge j
begin $LN \leftarrow L; FN \leftarrow F; SN \leftarrow S; in\_branch \leftarrow j;$
$best\_edge \leftarrow nil; best\_wt \leftarrow \infty;$
for all $i \neq j$ such that $SE(i) = Branch$
do begin send $Initiate(L, F, S)$ on edge i;
if $S = Find$ **then** $find\_count \leftarrow find\_count + 1;$
end
if $S = Find$ **then execute procedure** *test*
end

Implementation :

```
void GHSNode::handleInitiate()
{
  std::vector<std::string> mg = msg->getMessage();
  int j = stoi(mg[0]); //!< edge j
  int L = stoi(mg[2]); //!< L
  std::string F = mg[3]; //!< F
  std::string S = mg[4]; //!< S
  LN = L; //!< LN <- L
  FN = F; //!< FN <- F
  SN = S; //!< SN <- S
  in_branch = j; //!< in-branch <- j
  best_edge = -1; //!< best-edge <- nil
  best_weight = INF; //!< best-wt <- infinity
  for(auto it : branch) //!< forall
  {
    if(it.second == j)
    {
      continue;
    }//!< i != j and SE(i) = branch
    std::vector<std::string> st;
    st.push_back("initiate"); //!< Initiate()
    st.push_back(std::to_string(L)); //!< L
    st.push_back(F); //!< F
    st.push_back(S); //!< S
    sendMessage(it.second, msgCreater(st)); //!< send Initiate(L, F, S) on edge i
    if(S == "find") //!< if S = Find
    {
      find_count++; //!< find-count <- find-count + 1
    }
  }
  if(S == "find") //!< if S = Find
```

```
    {
      test(); //!< execute procedure test
    }
    free(msg);
    msg = NULL;
  }

```

Listing 13: handleInitiate()

2.2.5 Algorithm 5

Pseudocode :

procedure *test*
  if there are adjacent edges in the state *Basic*
    then begin *test_edge* ← the minimum-weight adjacent edge in state *Basic*;
        send $Test(LN, FN)$ on *test_edge*;
      end
  else begin *test_edge* ← *nil*; **execute procedure** report **end**

Implementation :

```
void GHSNode::test()
{
  test_edge = findMinEdge(); //!< test-edge <- nil if no basic edges
  //!< test-edge <- the minimum-weight adjacent edge in state Basic
  if(test_edge != -1) //!< If there are adjacent edges in the state Basic
  {
    std::vector<std::string> st;
    st.push_back("test"); //!< Test()
    st.push_back(std::to_string(LN)); //!< LN
    st.push_back(FN); //!< FN
    sendMessage(test_edge, msgCreater(st)); //!< send Test(LN, FN) on test-edge
  }
  else //!< Else test-edge <- nil (by default)
  {
    report(); //!< execute procedure report()
  }
}

```

Listing 14: test()

2.2.6 Algorithm 6

Pseudocode :

Response to $Test(L, F)$ on edge j

begin if $SN = Sleeping$ **then execute procedure** wakeup;
  if $L > LN$ **then** place received message on end of queue;
   else if $F \neq FN$ **then send** *Accept* on edge j ;
    else begin if $SE(j) = Basic$ **then** $SE(j) \leftarrow Rejected$;
     if $test\_edge \neq j$ **then send** *Reject* on edge j
      else execute procedure *test*
     end
end

Implementation :

```
void GHSNode::handleTest()
{
  std::vector<std::string> mg = msg->getMessage();
  int j = std::stoi(mg[0]); //!< edge j
  if(SN == "sleep") //!< If SN = Sleeping
  {
    wakeup(); //!< execute procedure wakeup
  }
  int L = std::stoi(mg[2]); //!< L
  std::string F = mg[3]; //!< F
  if(L > LN) //!< L > LN
  {
    sendMessage(nodeid, msg); //!< place recieved message on end of queue
  }
  else if(F != FN) //!< F != FN
  {
    std::vector<std::string> st;
    st.push_back("accept"); //!< Accept()
    sendMessage(j,msgCreater(st)); //!< Send Accept on edge j
    free(msg);
    msg = NULL;
  }
  else
  {
    if(SE[j] == "basic") //!< if SE(j) = Basic
    {
      changestat(j, "reject");
    }
    if(test_edge != j) //!< test-edge != j
    {
      std::vector<std::string> st;
      st.push_back("reject"); //!< Reject()
      sendMessage(j, msgCreater(st)); //!< send Reject on edge j
    }
    else
```

```
        {
            test(); //!< execute procedure test
        }
        free(msg);
        msg = NULL;
    }
}

```

Listing 15: handleTest()

2.2.7 Algorithm 7

Pseudocode :

Response to *Accept* on edge j

begin $test\_edge \leftarrow nil$;

if $w(j) < best\_wt$

then begin $best\_edge \leftarrow j$; $best\_wt \leftarrow w(j)$ **end**

execute procedure *report*

end

Implementation :

```
void GHSNode::handleAccept()
{
    std::vector<std::string> mg = msg->getMessage();
    int j = std::stoi(mg[0]); //!< edge j
    test_edge = -1; //!< test-edge <- nil
    if(nbd[j] < best_weight) //!< w(j) < best-wt
    {
        best_edge = j; //!< best-edge <- j
        best_weight = nbd[j]; //!< best-wt <- w(j)
    }
    free(msg);
    msg = NULL;
    report(); //!< execute procedure report()
}

```

Listing 16: handleAccept()

2.2.8 Algorithm 8

Pseudocode :

> Response to *Reject* on edge j
> **begin if** $SE(j) = Basic$ **then** $SE(j) \leftarrow Rejected$;
> **execute procedure** test
> **end**

Implementation :

```
void GHSNode::handleReject()
{
    std::vector<std::string> mg = msg->getMessage();
    int j = std::stoi(mg[0]); //!< edge j
    if(SE[j] == "basic") //!< if SE(j) = Basic
    {
        changestat(j, "reject");
    }
    free(msg);
    msg = NULL;
    test(); //!< execute procedure test()
}

```

Listing 17: handleReject()

2.2.9 Algorithm 9

Pseudocode :

> **procedure** report
> **if** $find\_count = 0$ and $test\_edge = nil$;
> **then begin** $SN \leftarrow Found$;
> **send** $Report(best\_wt)$ on in_branch;
> **end**

Implementation :

```
void GHSNode::report()
{
    if(find_count == 0 && test_edge == -1) //!< if find-count = 0 and test-edge
 = nil
    {
        SN = "found"; //!< SN <- Found
        std::vector<std::string> st;
```

```
            st.push_back("report"); //!< Report()
            st.push_back(std::to_string(best_weight)); //!< best-wt
            sendMessage(in_branch,msgCreater(st)); //!< send Report(best-wt) on in-branch
        }
    }

```

Listing 18: report()

2.2.10 Algorithm 10

Pseudocode :

Response to *Report*(w) on edge j
 if $j \neq in\_branch$
  then begin $find\_count \leftarrow find\_count - 1$
   if $w < best\_wt$ **then begin** $best\_wt \leftarrow w$; $best\_edge \leftarrow j$ **end**
   execute procedure report
  end
 else if $SN = Find$ **then** place received message on end of queue
  else if $w > best\_wt$
   then execute procedure *change_root*
   else if $w = best\_wt = \infty$ **then halt**

Implementation :

```
    void GHSNode::handleReport()
    {
        std::vector<std::string> mg = msg->getMessage();
        int j = std::stoi(mg[0]); //!< edge j
        int w = std::stoi(mg[2]); //!< w
        if(j != in_branch) //!< if j != in-branch
        {
            find_count--; //!< find-count <- find-count - 1
            if(w < best_weight) //!< w < best-wt
            {
                best_weight = w; //!< best-wt <- w
                best_edge = j; //!< best-edge <- j
            }
            free(msg);
            msg = NULL;
            report(); //!< execute procedure report()
        }
        else if(SN == "find") //!< SN = Find
        {
            sendMessage(nodeid, msg); //!< place recieved message on end of queue
        }
```

```
        else if(w > best_weight) //!< w > best-wt
        {
          free(msg);
          msg = NULL;
          changeRoot(); //!< execute procedure change-root()
        }
        else if(w == best_weight && w == INF) //!< w = best-wt = infinity
        {
          isc->complete = true; //!< halt
          free(msg);
          msg = NULL;
        }
      }

```

Listing 19: handleReport()

Note that at halt step, we set the isc pointer to true, this tells the main thread that the nodes have found the MST.

2.2.11 Algorithm 11

Pseudocode :

> **procedure** *change_root*
>   **if** *SE*(*best_edge*) = *Branch*
>   **then send** *Change_root* on *best_edge*
>   **else begin send** *Connect*(*LN*) on *best_edge*;
>   *SE*(*best_edge*) ← *Branch*
>   **end**

Implementation :

```
    void GHSNode::changeRoot()
    {
      if(SE[best_edge] == "branch") //!< SE(best-edge) = Branch
      {
        std::vector<std::string> st;
        st.push_back("changeroot"); //!< Change-root()
        sendMessage(best_edge, msgCreater(st)); //!< send Change-root() on best-edge
      }
      else
      {
        std::vector<std::string> st;
        st.push_back("connect"); //!< Connect()
        st.push_back(std::to_string(LN)); //!< LN
        sendMessage(best_edge, msgCreater(st)); //!< send Connect(LN) on best-edge
        changestat(best_edge, "branch");
```

```
        }
    }

```

Listing 20: changeRoot()

2.2.12 Algorithm 12

Pseudocode :

> Response to *Change_root*
> **execute procedure** *change_root*

Implementation :

```
    void GHSNode::handleChangeroot()
    {
        free(msg);
        msg = NULL;
        changeRoot(); //execute procedure change-root()
    }

```

Listing 21: handleChangeroot()

2.3 Main Function/ Thread Runner

In this section we provide the snippet for our thread_runner funtion which initiates all the GHS Nodes and runs them.

```
    Network *network = new Network(no); //!< Networks for the threads
    IsComplete *isc = new IsComplete(); //!< Flag to check completion
    TotMessage *tot = new TotMessage(); //!< Total message counter

    std::vector<pthread_t> threads(n); //!< Vector of threads
    std::vector<GHSNode *> nodes; //!< Vector of all GHSNodes

    int i = 0;
    for(auto it : adj_list)
    {
        GHSNode *temp = new GHSNode(it.first, it.second, network, isc, tot); //!<
  Create new GHSNode
        nodes.push_back(temp);

        int errcode = pthread_create(&(threads[i]), NULL, run_thread, (void *)temp);
    //!< Start the thread, if errcode != 0 then thread creation was not successful

        if(errcode != 0)
```

```
    {
      std::cerr << "Thread Creation at index : " << i << " Failed.\n Exiting....
  " << std::endl;
      exit(49);
    }
    i++;
  }

  while(!(isc->complete))
  {
    //threads still running
    continue;
  }
  //GHS Complete


```

Listing 22: thread_runner Sample

Here each thread calls run_thread function which is a trivial function which fires up the node :

```
  void* run_thread(void * node)
  {
    ((GHSNode *)node)->run();
    pthread_exit(NULL);
  }

```

Listing 23: run_thread()

For each node, we store classify edges into three sets : basic, branch and reject -

```
  std::set<std::pair<int, int> > basic, branch, reject; //!< Set containing (
 edge_weight, id) of edges in basic, branch and reject state respectively

```

Listing 24: branch classification

At the end of the algorithm, we collect the branch edges of all nodes via :

```
std::set<std::tuple <int, int, int > > ans; //!< MST Edges
for(auto it : nodes)
{
  std::vector<int> v = it-> getMSTEdges();
  for(auto jt : v)
  {
    ans.insert(std::make_tuple(jt, mp[jt].first, mp[jt].second));
  }
}
return ans;

```

Listing 25: MST Collection

Here jt is the weight of the edge, and mp[jt] represents the nodes adjacent to it. And getMSTEdges() is as follows :

```
std::vector<int> GHSNode::getMSTEdges()
{
  std::vector<int> v;
  for(auto it : branch)
  {
    v.push_back(it.first);
  }
  return v;
}

```

Listing 26: getMSTEdges()

Further documentation for the code can be generated with doxygen as the code has been commented extensively with doxygen styled comments.

3 Complexity Analysis and Experiments

3.1 Experiment data

Firstly we present the results of the experiments we ran:

Sparse Graph Experiment(P=0.2)

Nodes	**Edges**	**Number of Messages**
20	39	198
50	252	1016
100	966	3381
150	2248	6646
200	3954	9954
250	6255	16049
300	8888	22054
350	12269	30942
400	15969	37643

Dense Graph Experiment(P=0.8)

Nodes	**Edges**	**Number of Messages**
20	148	505
50	964	2436
100	4011	9049
150	9012	19525
200	15819	34471
250	24826	53218
300	35860	76051
350	48923	102848
400	63838	133375

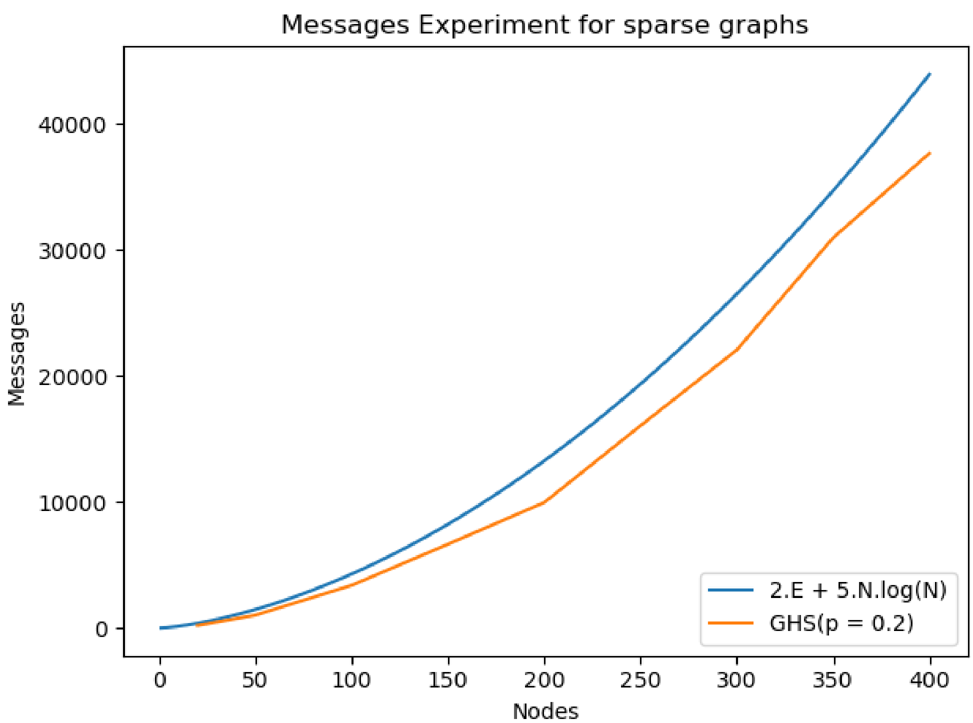
We also took samples of running time for our code for complete graphs. However analysis of this is much more complicated as it involves major multi-threading overhead. Therefore we merely present the data here :

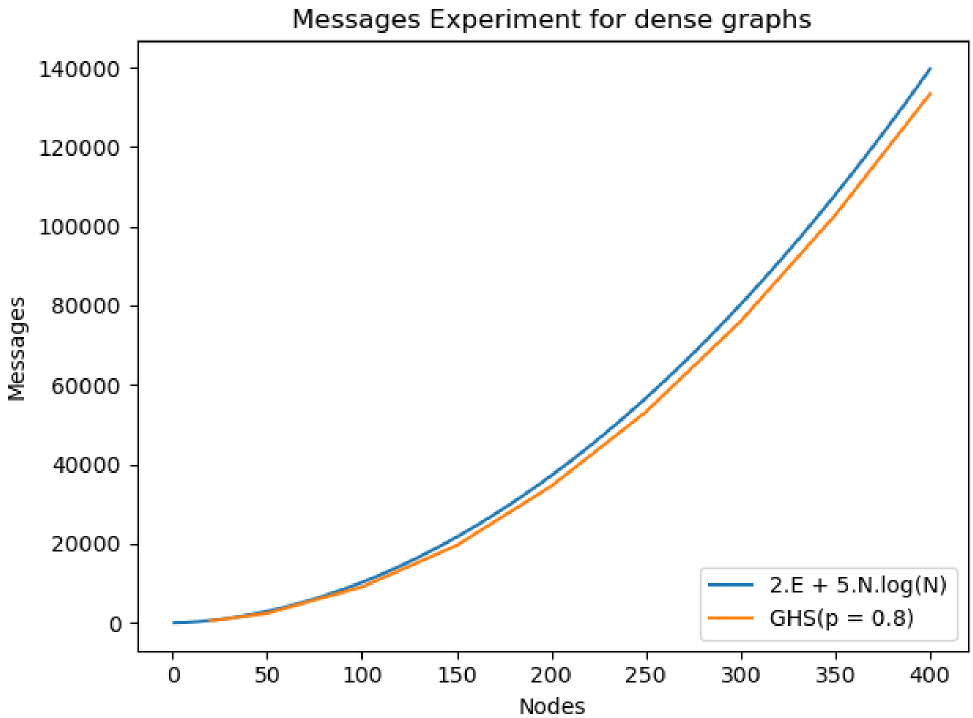
Runtime Experiment on Complete Graphs(P=1)

Nodes	**Running time**
5	0.000503
10	0.242961
20	0.770877
50	3.93151
100	11.9101
150	23.478
200	38.7509
250	54.9079
300	78.804
350	95.1269
400	128.236

3.2 Complexity Analysis

Plots for the results on number of messages as given in the previous section are as follows :





In these plots

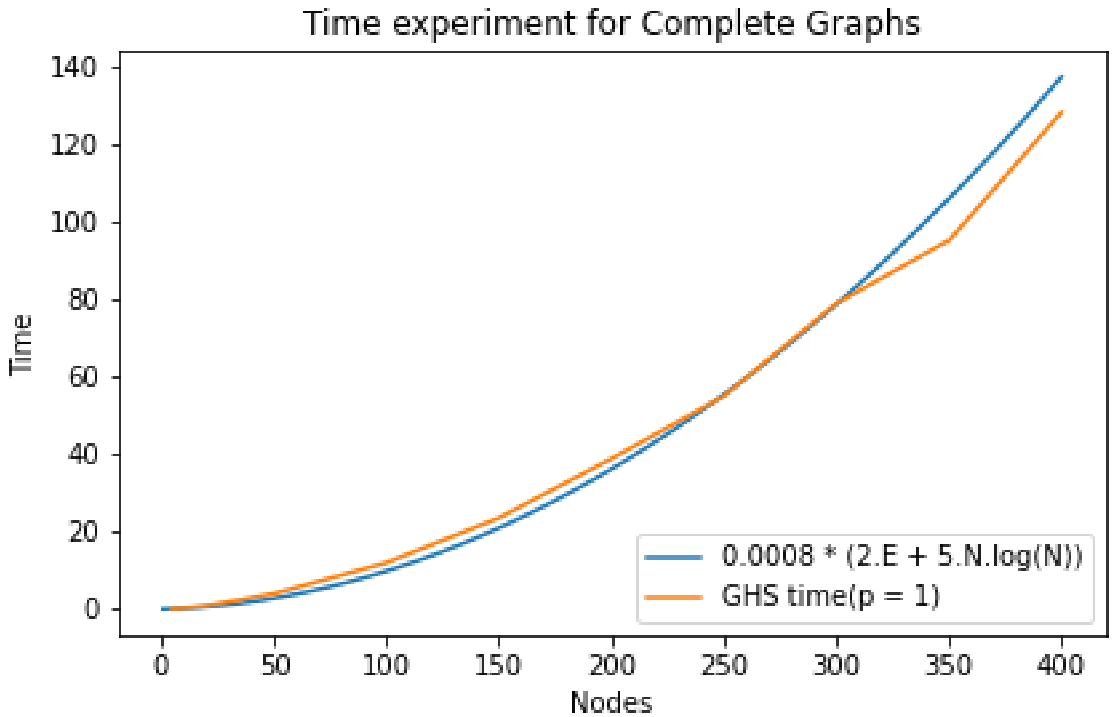
- The blue line represents the function : $2 \cdot E + 5 \cdot N \cdot \log_2(N)$ with N = number of nodes, E = number of Edges.
- Here roughly $E = \binom{N}{2} \cdot P$, where P is the probability that there is an edge between any two given nodes.
- The orange line represents the number of messages sent by all the nodes in our implementation.

Analysis :

- Clearly the orange line lies below the blue line for both, spare and dense graphs.
- As probability increases, the gap between the two lines decreases.
- The blue line is therefore a tight upper bound on the total number of messages in GHS Algorithm.
- Therefore the total number of messages sent in the GHS Algorithm grows as $2 \cdot E + 5 \cdot N \cdot \log_2(N)$

Running Time :

- Plot for running time is shown on the next page.
- Running time seems to increase in proportion to the number of messages sent.
- Analysis of this runtime would need to account for various multi-threading overheads, mutex locks, cpu thread capacity, etc..
- However in a very broad sense, as seen by the plot, communication between nodes seems to be a bottleneck for our implementation.


Time experiment for Complete Graphs
Time
Nodes
0.0008 * (2.E + 5.N.log(N))
GHS time(p = 1)
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350
400