Distributed Algorithms Project – GHS Algorithm Simulator

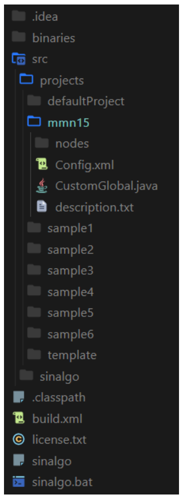
Ori Dabush

Table of contents:

Part 1 – The GHS Algorithm

How to run the code?

The code submitted is a Sinalgo project. To run it, you'll have to place the folder named \`mmn15\` and its content in the \`projects\` folder. The file hierarchy should look like this:

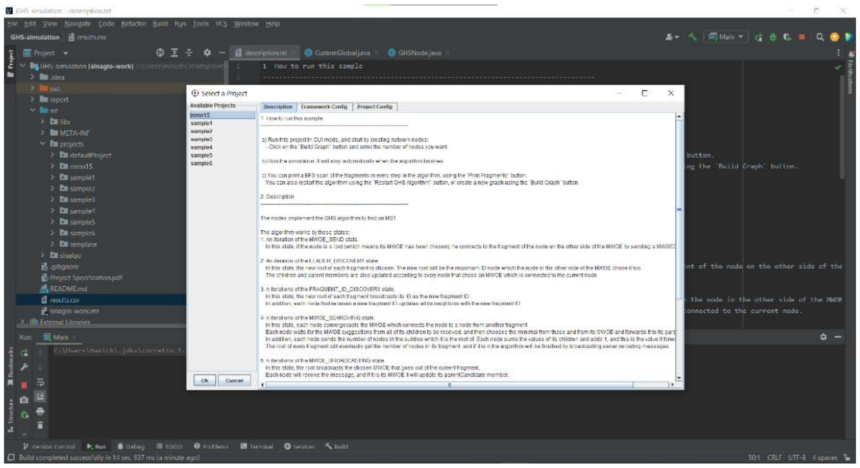


```
.idea
binaries
src
  projects
    defaultProject
    mmn15
      nodes
      Config.xml
      CustomGlobal.java
      description.txt
    sample1
    sample2
    sample3
    sample4
    sample5
    sample6
    template
  sinalgo
.classpath
build.xml
license.txt
sinalgo
sinalgo.bat
```

Now, you can run the Sinalgo simulator as they specify in their documentation. You can do it using the sinalgo.bat file, or by typing the following command:

```
java -cp binaries/bin sinalgo.Run %*
```

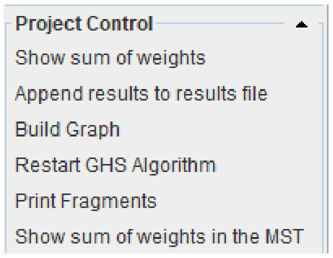
You can also run the Sinalgo simulator using the Eclipse IDE, and you can also use IntelliJ. To use IntelliJ, you need to configure it to run the Sinalgo project. I did it (and you can use my configuration instead) in the GitHub repo of the project which can be found here. You just need to clone the repo, open it using IntelliJ, and it will automatically do anything for you and you'll just need to run the program.



```
1. How to run this sample

a) Run this project in GUI mode, and start by creating network nodes:
  - Click on the 'Build Graph' button and enter the number of nodes you want

b) Run the simulation. It will stop automatically when the algorithm finishes

c) You can print a BFS scan of the fragments in every step in the algorithm, using the 'Print Fragments' button.
  You can also restart the algorithm using the 'Restart GHS Algorithm' button, or create a new graph using the 'Build Graph' button.

2. Description

The nodes implement the GHS algorithm to find an MST.

The algorithm works by these states:
1. An iteration of the MWOE_SEND state.
  In this state, if the node is a root (which means its MWOE has been chosen), he connects to the fragment of the node on the other side of the MWOE by sending a MWOE

2. An iteration of the LEADER_DISCOVERY state
  In this state, the new root of each fragment is chosen. The new root will be the maximum ID node which the node in the other side of the MWOE chose it too.
  The children and parent members are also updated according to every node that chose an MWOE which is connected to the current node.

3. n iterations of the FRAGMENT_ID_DISCOVERY state.
  In this state, the new root of each fragment broadcasts its ID as the new fragment ID.
  In addition, each node that receives a new fragment ID updates all its neighbors with the new fragment ID.

4. n iterations of the MWOE_SEARCHING state.
  In this state, each node convergecasts the MWOE which connects the node to a node from another fragment.
  Each node waits for the MWOE suggestions from all of its children to be received, and then chooses the minimal from those and from its MWOE and forwards it to its par
  In addition, each node sends the number of nodes in the subtree which it is the root of. Each node sums the values of its children and adds 1, and this is the value it forw
  The root of every fragment will eventually get the number of nodes in its fragment, and if it is n the algorithm will be finished by broadcasting server rerouting messages.

5. n iterations of the MWOE_BROADCASTING state.
  In this state, the root broadcasts the chosen MWOE that goes out of the current fragment.
  Each node will receive the message, and if it is its MWOE it will update its parentCandidate member.
```

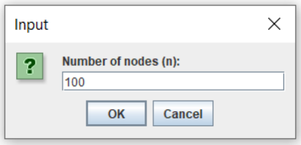
The starting screen of the Sinalgo algorithm.

To run my algorithm, you will need to choose the `mmn15` project in the `available projects` menu. After it, you'll have multiple options:



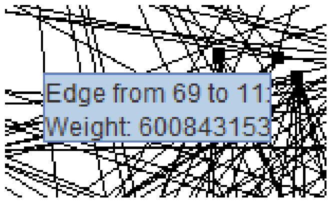
The buttons of the different options the user has.

- The `Build Graph` button – this button will build a graph as specified the assignment which is ready for the GHS algorithm. You'll need to enter a number of nodes to be in the graph, and also to choose the node which will be the server. After the graph will be built, starting the simulation will start the GHS algorithm. You can also look at the ID of every node and at the weight of every edge by simply hover the mouse over them or right-click on them and choose the info option.
  Pay attention that you can choose between running the algorithm for a given number of rounds, and you can also choose to run the algorithm until it finishes (the run forever option will stop once it finishes).



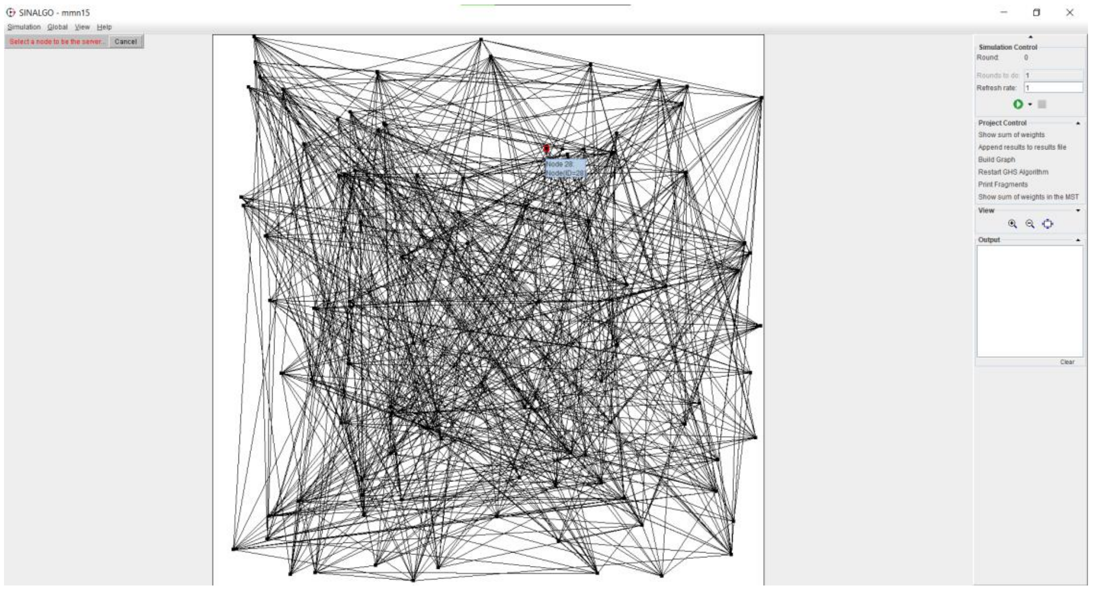
Number of nodes (n): 100

The popup where the user enters the number of nodes.



Edge from 69 to 11
Weight: 600843153

Visualization of an edge weight – will popup when the cursor will hover over an edge.

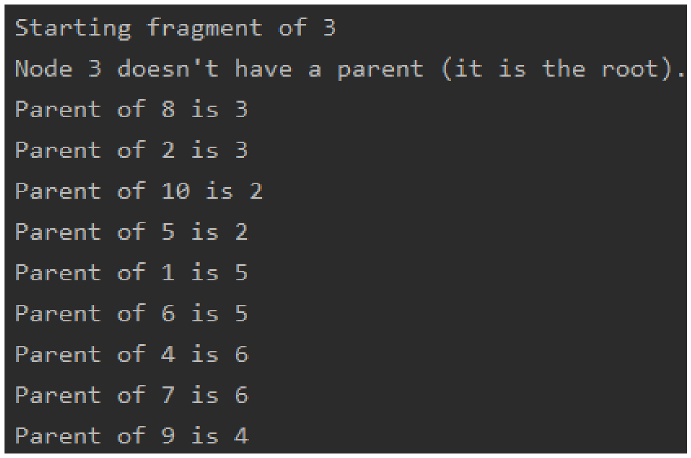


Select a node to be the server...

Round: 0
Rounds to do: 1
Refresh rate: 1

Choosing the server node

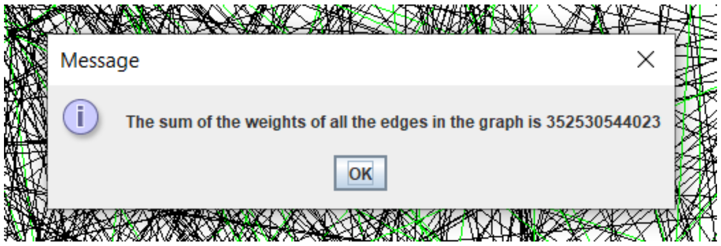
- The `Restart GHS Algorithm` - this button will reset the graph parameters and will restart the GHS algorithm run.
- The `Print Fragments` button – this button will print a BFS scan of the current fragments in the graph.



```
Starting fragment of 3
Node 3 doesn't have a parent (it is the root).
Parent of 8 is 3
Parent of 2 is 3
Parent of 10 is 2
Parent of 5 is 2
Parent of 1 is 5
Parent of 6 is 5
Parent of 4 is 6
Parent of 7 is 6
Parent of 9 is 4
```

Example of the output of `Print Fragments` with n=10 after finding the MST.

- The `Show sum of weights` button – this button will show the sum of the weights of the edges in the graph.



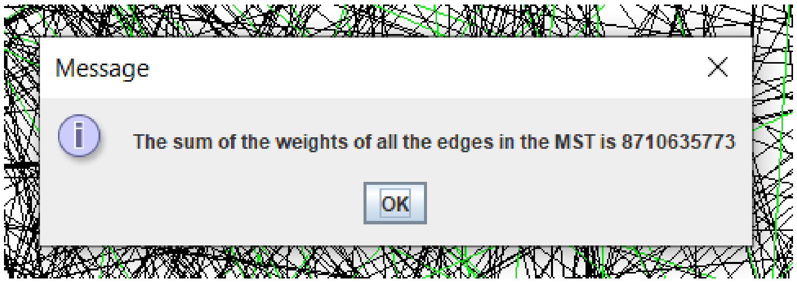
Message

The sum of the weights of all the edges in the graph is 352530544023

OK

Example of the output of `Show sum of weights`.

- The `Show sum of weights in MST` button – this button will show the sum of the weights of the edges in the built MST. This will work only after the algorithm finishes.



Message

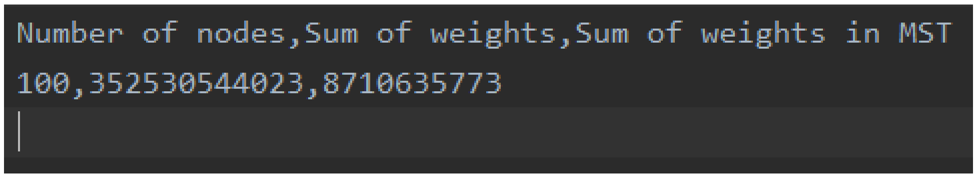
The sum of the weights of all the edges in the MST is 8710635773

OK

Example of the output of `Show sum of weights in MST`.

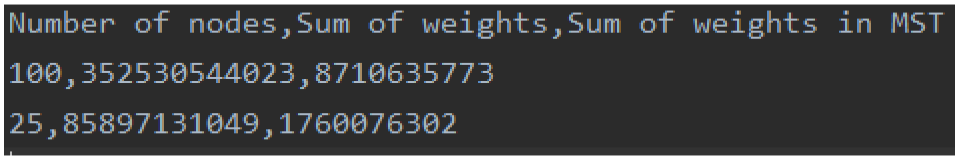
- The `Append results to results file` button – this button will append the number of nodes, the sum of the weights in the graph and the sum of the weights in the

MST to the results file name `results.csv`. If this file doesn't exist, it will create it with the appropriate headings.



```
Number of nodes,Sum of weights,Sum of weights in MST
100,352530544023,8710635773
```

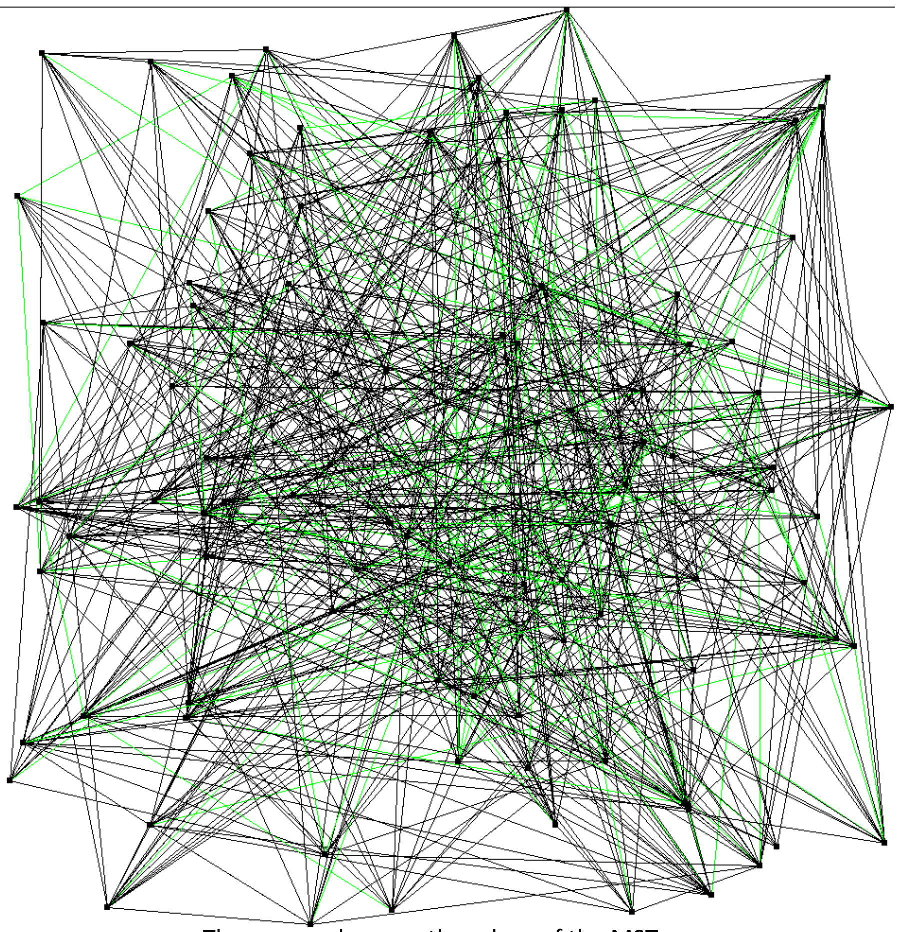
The content of results.csv on the first press on `Append results to results file`.



```
Number of nodes,Sum of weights,Sum of weights in MST
100,352530544023,8710635773
25,85897131049,1760076302
```

The content of results.csv after another press on `Append results to results file` with different results.

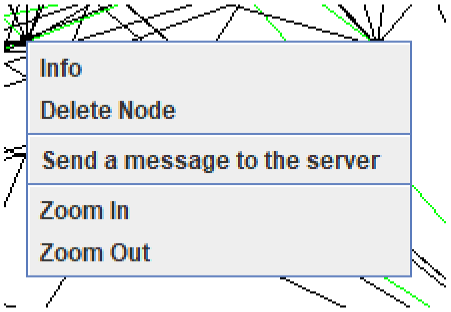
After the algorithm finishes, you'll see the edges of the MST colored in green.



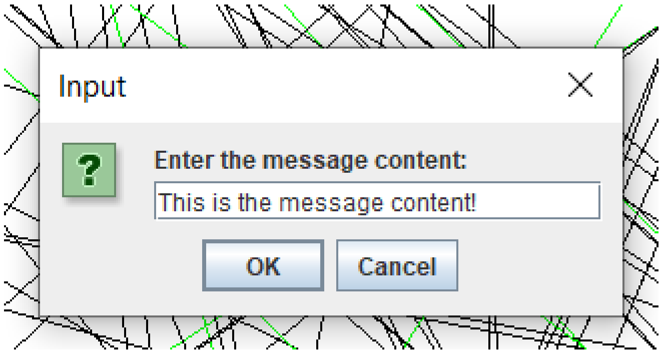
The green edges are the edges of the MST.

Now, an MST which its root is the server node was created, and you can start sending messages to the server.

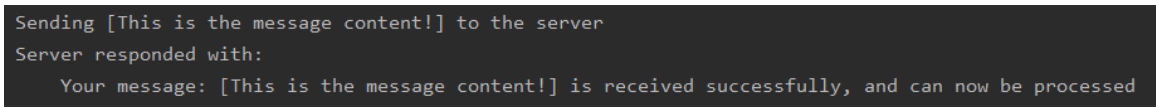
To send a message to the server, you'll need to right-click a node and choose the `Send a message to the server` option. Then, you'll need to enter a message. Now the message will be sent on the MST (you can see the edges which will transfer a message after a specific round colored in red). When the response from the server will return to the node the message was sent from, the response will be printed to the console.



The menu of a specific node, containing the `Send a message to the server` option.



The popup where the user enters the message to send to the server.



```
Sending [This is the message content!] to the server
Server responded with:
    Your message: [This is the message content!] is received successfully, and can now be processed
```

The output when the message is sent and when the message is received back.

The design of the code

- The CustomGlobal class is responsible for all the menu buttons.
- The node logic is implemented in the GHSNode class.
- The edges are implemented in the WeightedEdge class.
- There are also different types of messages which I mentioned earlier when describing the states.

Building the graph

The graph is build using the uniform distribution object of Sinalgo. The nodes' location is randomly selected using this distribution.

The weights of the edges are also randomly selected using this distribution.

The uniform distribution helps to spread the nodes all over the screen, so they won't be concentrated in one place.

The GHSNode class

This class is the biggest class, which is responsible for the algorithm implementation. The logic of the algorithm is implemented as a state machine – each phase of the algorithm has a state, and the state of the nodes is changed accordingly.

The different states:

1. The MWOE_SEND state. In this state, if the node is a root (which means its MWOE has been chosen), he connects to the fragment of the node on the other side of the MWOE by sending a MWOEChoiceMessage.
   This state takes one round.
2. The LEADER_DISCOVERY state. In this state, the new root of each fragment is chosen. The new root will be the maximum ID node which the node in the other side of the MWOE chose it too. The children and parent members are also updated according to every node that chose an MWOE which is connected to the current node.
   This state takes one round.
3. The FRAGMENT_ID_DISCOVERY state. In this state, the new root of each fragment broadcasts its ID as the new fragment ID. In addition, each node that receives a new fragment ID updates all its neighbors with the new fragment ID.
   This state takes n rounds.
4. The MWOE_SEARCHING state. In this state, each node convergecasts the MWOE which connects the node to a node from another fragment. Each node waits for the MWOE suggestions from all its children to be received, and then chooses the minimal from those and from its MWOE and forwards it to its parent.
   In addition, each node sends the number of nodes in the subtree which it is the

root of. Each node sums the values of its children and adds 1, and this is the value it forwards its parent. The root of every fragment will eventually get the number of nodes in its fragment, and if it is n the algorithm will be finished by broadcasting server rerouting messages.
This state takes n rounds.

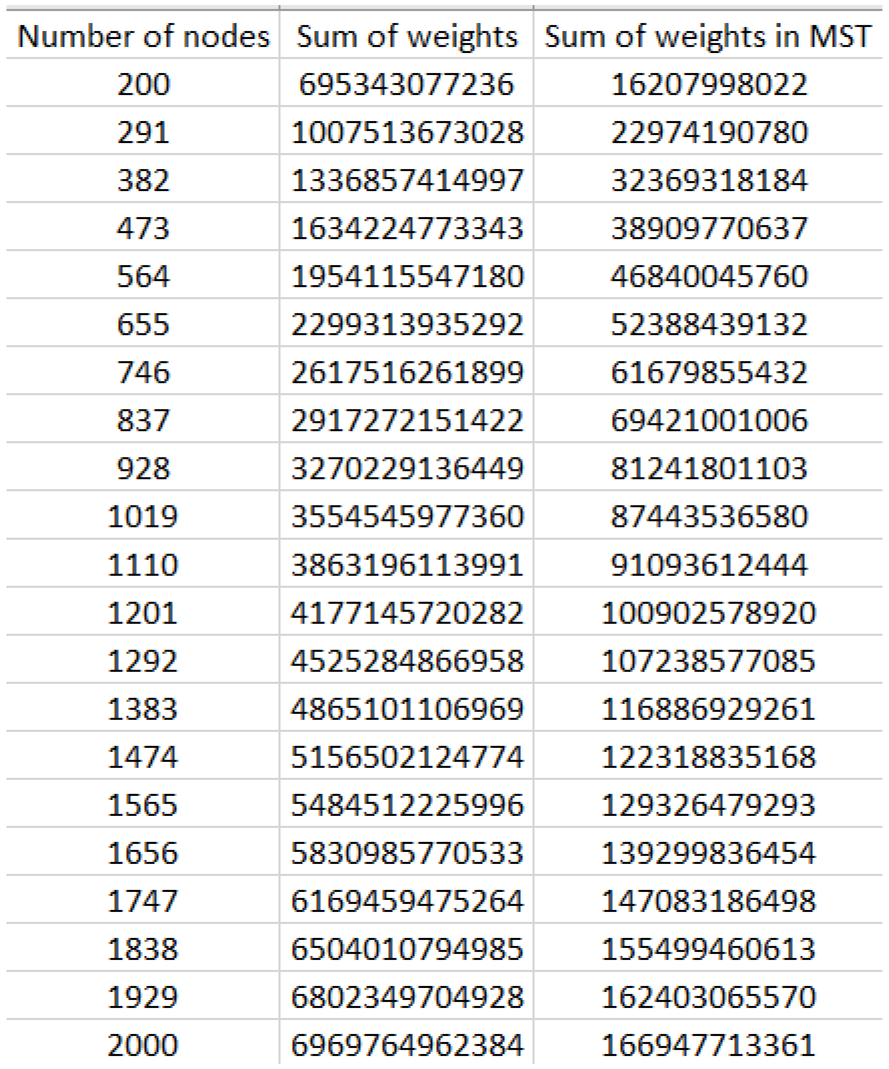
5. The MWOE_BROADCASTING state. In this state, the root broadcasts the chosen MWOE that goes out of the current fragment. Each node will receive the message, and if it is its MWOE it will update its parentCandidate member.
This state takes n rounds.
6. The NEW_ROOT_BROADCASTING state. In this state, the node which the fragment's MWOE is connected to will become the new root of the fragment. It will send a FlipEdgeDirectionMessage to its parent, and each node that receives this message will change its parent to be the sender and forward it to its old parent. This will continue all the way up to the old root of the fragment.
This state takes n rounds.
7. The SERVER_REROUTING state. In this state, the server node will send its parent a FlipEdgeDirectionMessage, which will be forwarded to the root. Every node in the way will change its parent to be the sender. This is done to make the server be the root of the MST.
This state takes 2n rounds.
8. The FINISHED state. In this state, the MST is already built. Nodes in this state will listen for messages, and if it is a server request or response messages, they will forward it to the correct node.
9. The SENDING_MESSAGE_TO_SERVER state. In this state, the node will initially send a server request to its parent, and will wait until a response will be received. Once a response is received, the state will be changed to FINISHED.

This design allowed me to easily debug the algorithm in case there are problems. Each state is well defined, and I could easily locate the state which the problem occurred at. In addition, this design allowed me to easily add new states, for example when I wanted to add the SENDING_MESSAGE_TO_SERVER state.

A design of a state machine is also good because it helps understanding the distributed algorithm better.

Results of the algorithm

As requested in the assignment, the table containing the results of the algorithm on 20 different graphs with 200-2000 nodes in them:



Number of nodes	Sum of weights	Sum of weights in MST
200	695343077236	16207998022
291	1007513673028	22974190780
382	1336857414997	32369318184
473	1634224773343	38909770637
564	1954115547180	46840045760
655	2299313935292	52388439132
746	2617516261899	61679855432
837	2917272151422	69421001006
928	3270229136449	81241801103
1019	3554545977360	87443536580
1110	3863196113991	91093612444
1201	4177145720282	100902578920
1292	4525284866958	107238577085
1383	4865101106969	116886929261
1474	5156502124774	122318835168
1565	5484512225996	129326479293
1656	5830985770533	139299836454
1747	6169459475264	147083186498
1838	6504010794985	155499460613
1929	6802349704928	162403065570
2000	6969764962384	166947713361

You can also view these results in the big_results.csv file.

Part 2 – SINR model

The algorithm

The SINR (signal interference noise ratio) model is a more realistic distributed model. In this model messages can be lost due to other nodes transmissions interrupting them.

As shown in Goussevskaia, Moscibroda and Wattenhofer 2008, if each node in the SINR model network transmits its messages with probability $\frac{\varepsilon}{\Delta}$, its message will be received successfully by all its neighbors with high probability in $O(\Delta \log n)$ rounds.

That means, that we can change our algorithm to fit the SINR model by simulating every round of it with $O(\Delta \log n)$ rounds of the algorithm I mentioned before, which in a node transmit its messages with probability $\frac{\varepsilon}{\Delta}$.

This method will make the runtime of our algorithm frow from $O(n \log n)$ to $O(\Delta n \log^2 n)$.

In order to implement it in the code, we will simply extend each state to be $c\Delta \log n$ times longer, and each time we send a message we will do it with a probability of $\frac{\varepsilon}{\Delta}$. We can easily achieve that by overriding the send method of the Node class in the GHSNode class and implement it that way.

Improving the algorithm

$O(\Delta n \log^2 n)$ is pretty bad, and we can achieve better runtime than that. To do that, we can use the method I mentioned before to find a $O(\Delta)$ coloring which in nodes in the same color are in the geometric distance of $\frac{R'}{\varepsilon'}$ at least, for small enough ε' to make the messages transmissions successful. This can be done in $O(\Delta + \log^* n)$ rounds in the regular model, but in the SINR model using the method I suggested before it will take $O(\Delta^2 \log n + \Delta \log n \log^* n)$.

Now, in every round Only colors from the same color will transmit their messages. This will improve the simulation from $O(\Delta \log n)$ rounds in the SINR model per 1 round of the regular model to $O(\Delta)$ rounds in the SINR model per 1 round of the regular model. This will allow us to simulate the GHS algorithm in a runtime of $O(\Delta n \log n)$, and the total runtime will be $O(\Delta^2 \log n + \Delta \log n \log^* n + \Delta n \log n) = O(\Delta n \log n)$, which is better then $O(\Delta n \log^2 n)$.

Implementing the first part will require me to implement a coloring algorithm that colors the graph with $O(\Delta)$ colors so every 2 nodes with the same color are in the geometric distance of $\frac{R'}{\varepsilon'}$ at least, for small enough ε' to make the messages transmissions successful. After implementing that algorithm, running it with the method I mentioned before will make it work in the SINR model with high probability.

Now I need to use the coloring to simulate the GHS algorithm. To do it, in every round I, only nodes with color that is equal to $i \ mod \ k$, where k is the number of colors ($k \in O(\Delta)$), will transmit their messages. This can also be achieved by overriding the send method of the Node class in the GHSNode class.