Instituto Tecnológico de Aeronáutica

CES-27: Processamento Distribuído

Distributed Minimum Spanning Tree

Teachers:
Celso M Hirata
Juliana de M Bezerra

Group:
Carlos M B da Silva
Eduardo A M Barbosa*
Igor Bragaia

December 4, 2019

Contents

1 Introduction

This project was made as a final exam for a distributed processing subject by three fourth year computer engineering students at Instituto Tecnológico de Aeronáutica (ITA). During the graduation course, two of the algorithms that we usually learn are the Prim's and Kruskal's minimum spanning tree (MST) creation algorithms. Curious for a distributed implementation to solve this problem, the group searched and found the algorithm of Gallager, Humblet and Spira (GHS algorithm), one of the best-known algorithms in distributed computing theory. In the search, it wasn't found any Golang implementation of the algorithm, motivating the group to do so.

The project deals with the implementation of a Golang script that constructs the MST in an asynchronous message-passing model, know as distributed minimum spanning tree (DMST), using the GHS algorithm availabe at Gallager et al. [1983]. The communication model was made using remote procedure calls (RPC) and an animation generation was made with a Python script using the graphviz library from the GHS algorithm output.

2 Background

The main concepts that serves as basis for this work are the minimum spanning tree and the remote procedure calls.

2.1 Minimum Spanning Tree

A minimum spanning tree is a sub-graph from an edge-weighted undirected graph that contains all the vertices and connects them with the minimum cost. A MST have many utilities such as network design (telephone, electrical, hydraulic, TV cable, computer, road) and approximation algorithms for NP-hard problems, Wikipedia [a].

2.2 Remote Procedure Calls

A remote procedure call is when one environment, normally a computer program, makes a procedure run in another environment, usually a different computer in the same network. The communication is made with a client–server interaction typically using request–response message-passing system, Wikipedia [b]. For this work, we use only requests, as we are not taking into account any type of failures and we don't need any data consistency.

3 Related work

This project uses one main article, Gallager et al. [1983], as a source of content to the algorithm and uses a second one, Flysher and Rubinshtein, that better describes the implementation of the algorithm.

One important note is that the GHS algorithm is made only for a graph with distinct edge weights. The implementation made by this project take this as a requirement, but you can make it accept any graph if, instead of using only the weight, you use a set with the weight and the nodes and use a rule to choose the minimum weight edge, like the edge with the minimum nodes identification number.

The GHS algorithm was taken as the state of art for the distributed MST problem, it constructs an MST in $O(n \log n)$ rounds exchanging a total of $O(m + n \log n)$ messages, where n and m denote the number of nodes and the number of edges of the network, respectively. Today, we got new deterministic and randomized algorithms that are able to do it with $\tilde{O}(D + \sqrt{n})$ rounds and $\tilde{O}(m)$, where D is the hop-diameter of the graph, Pandurangan et al. [2018].

4 Requirements

The requirements of the projects are:

- Create a distributed graph where each node is a different system and the edges of the graph holds up the connection address to the other node
- Handle the communication between the nodes through the edges with a RPC message-passing model
- Generate an output with an animation of all the steps that the program took to generate the MST
- Generate the MST

5 Architecture

The distributed architecture of the project consists of different systems in the same network representing each node of the graph and the communication between each system is made with RPC where the edges of the graph are the connection addresses. For better visualizing, we have as example on the following figure a three nodes graph with all nodes connected.

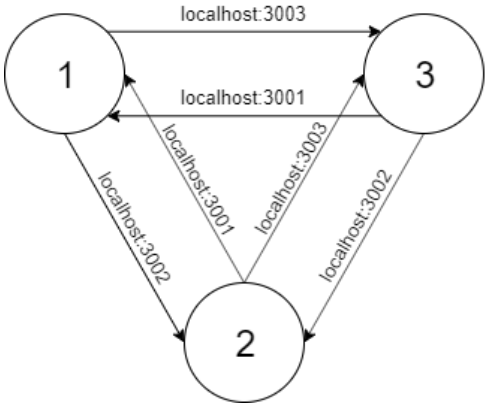

Figure 1: Three nodes graph architecture example

In the figure we can see that we have two communications between each pair of nodes. This doesn't mean that our graph is oriented. That's because each node have his own edge since one system doesn't know other system's content, they only know their individual content and their neighbors.

6 Development

The development can be divided in three main sections: RPC communication, GHS algorithm and graph animation.

6.1 RPC communication

The RPC communication is made with a structure of a server for each node. The server is created when the node executes it's initialization method. The server have two methods, startListening and acceptConnections, the second one is called inside the first one, that is called when the node starts the loop that will handle the communication.

The RPC call is a method for the node that first dial to an RPC server and after it do the call. For the node's methods we use another method (sendMessage) that calls the RPC call method (CallHost)

6.2 GHS pseudo-algorithm

This is the Algorithm implemented, extracted from Gallager et al. [1983], that is executed at each node. It's divided in twelve procedures.

```
(1) Response to spontaneous awakening (can occur only at a node in the sleeping state)
        execute procedure wakeup

(2) procedure wakeup
        begin let m be adjacent edge of minimum weight;
            SE(m) ←Branch;
            LN ←0;
            SN ←Found;
            Find−count ←0;
            send Connect(O) on edge m
        end

(3) Response to receipt of Connect(L) on edge j
        begin if SN = Sleeping then execute procedure wakeup;
            if L < LN then
                begin SE(j) ←Branch;
                    send Initiate(LN, FN, SN) on edge j;
                    if SN = Find then
                        find−count ←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

(4) Response to receipt of Initiate (L, F, S) on edge j
        begin LN ←L; FN ←F; SN ←S; in−branch ←j;
            best−edge ←nil; best−wt ←∞;
            for all i ≠j such that SE(i) = Branch do
                begin send Initiate(L, F, S) on edge i;
                    if S = Find then find−count ←find−count + 1
                end;
            if S = Find then execute procedure test
        end

(5) 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

(6) Response to receipt of 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 ≠FN then send Accept on edge j
        else
            begin if SE(j) = Basic then SE(j) ←Rejected;
                if test−edge ≠j then send Reject on edge j
                else execute procedure test
            end
    end

(7) Response to receipt of Accept on edge j
    begin test−edge ←nil;
        if w(j) < best−wt then
            begin best−edge ←j; best−wt ←w(j) end;
        execute procedure report
    end

(8) Response to receipt of Reject on edge j
    begin if SE(j) = Basic then SE(j) ←Rejected;
        execute procedure test
    end

(9) procedure report
    if find−count = 0 and test−edge = nil then
    begin SN ←Found;
        send Report(best−wt) on in−branch
    end

(10) Response to receipt of Report(w) on edgej
    if j ≠in−branch then
        begin find−count ←find−count − 1
            if w < best−wt then begin best−wt ←w; best−edge ←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 = ∞then halt

(11) 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

(12) Response to receipt of Change−root
    execute procedure change−root
```

6.3 Graph animation

The graph animation was made using the graphviz python library. All procedures were configured to print to a file the logs with a timestamp of each procedure the node executed. Using the timestamp as a reference, the script first joins all nodes logs into an unique log. After that, the script reads each line of the log updating the state of each graph element and saving an image. Finally, it generates a gif with all the images that it got.

7 Prototype

In order to run the project, clone the code from this GitHub repository. To start a process that can make RPC calls to another specific process, you must run the following command line inside run folder

Listing 1: command line example

```
go run run.go −id [id] −nid [nid] −nwt [nwt]
```

Where [id] is an integer as "1"; [nid] is the joint id list of other neighbor processes from which this one can make RPC calls using comma separator as "2,3"; [nwt] is the joint weight list of other neighbor processes illustrative RPC calls weight. This weight list represents graph edges used for the minimum spanning tree calculation.

There is a real example about how to run the project. Let's suppose you want to run the GHS algorithm for the following graph

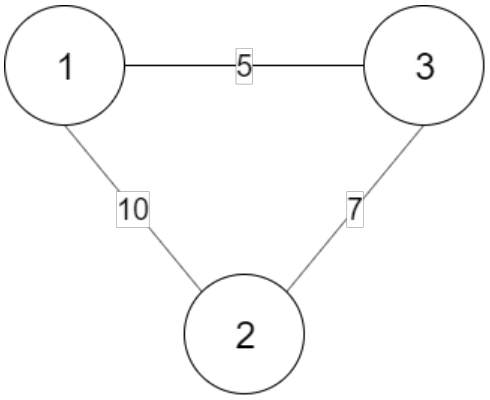

Figure 2: Graph example

Then you should start three different processes in different terminals

Listing 2: first process

```
go run run.go -id 1 -nid 2,3 -nwt 10,5
```

Listing 3: second process

```
go run run.go -id 2 -nid 1,3 -nwt 10,7
```

Listing 4: third process

```
go run run.go -id 3 -nid 1,2 -nwt 5,7
```

Note that the only process that awakes as soon as it starts it is the process with id 1 which starts the GHS algorithms when it awakes. On this scenario, you must let the process with id 1 to be the last one started to make sure that all other processes have already started previously. As a further improvement for the project, the awakening process can be flexible according to user interests and input.

After a while, algorithm message-passing will be finished and a MST will arise. All messages sent or received from a specific process can be seen at each terminal log. Furthermore, each time a message is exchanged, the process state as well as its edges state are printed to a process log file at folder /logs. By looking at these files you can understand MST fragment progress for a specific process.

Furthermore, once all distributed log files have been generated, you can run Python script run/gen graph.py that will merge all log files and create a gif animation that gives an overview about the distributed algorithm progress over time.

8 Install directives

You must be able to run a Golang file. Furthermore, only install the required Python3 libraries graphviz and imageio.

9 Evaluation

In order to evaluate the algorithm progress and correctness, each time a message is exchanged, we print on the terminal a message specifying the what was received or sent. Also, we created a go routine that writes the process and its edges state to a log file each time a message is exchanged. Thus, to run the project and see results, we have to start a process in each terminal to come up with the graph that will run the GHS. Regarding this, we took a small example and analyzed the results.

Let the example be the one explained at Prototype section. We got three different log files and terminal logs describing message exchanged for process 1, 2 and 3.

Note that there is two types of log entry

Listing 5: node (process) log entry

```
TIME >> timestamp >> NODE >> [id] [node state]
```

Listing 6: edge log entry

```
TIME >> timestamp >> EDGE >> [id from] [id to] [weight] [edge state]
```

Thus, the complete log for process 1, 2 and 3 including message log and state log are

Listing 7: process 1 message log

```
user:~/go/src/dmst/run: go run run.go -id 1 -nid 2,3 -nwt 10,5
2019/12/03 23:01:26 [SERVER] Listening on 'localhost:3001'
2019/12/03 23:01:26 Starting Handler
2019/12/03 23:01:26 [SERVER] Accepting connections.
START LOG GO ROUTINE
2019/12/03 23:01:26 [NODE 1] Connect message received from node 3
2019/12/03 23:01:26 [NODE 1] Initiate message received from node 3
2019/12/03 23:01:26 [NODE 1] Reject message received from node 2
2019/12/03 23:01:26 [NODE 1] Test message received from node 2
```

Listing 8: process 1 state log file

```
TIME >> 1575284188810656900 >> NODE >> 1 Found
TIME >> 1575284188810656900 >> EDGE >> 1 2 10 Basic
TIME >> 1575284188810656900 >> EDGE >> 1 3 5 Branch
TIME >> 1575284189414675800 >> NODE >> 1 Find
TIME >> 1575284189414675800 >> EDGE >> 1 2 10 Basic
TIME >> 1575284189414675800 >> EDGE >> 1 3 5 Branch
TIME >> 1575284190819083900 >> NODE >> 1 Find
TIME >> 1575284190819083900 >> EDGE >> 1 2 10 Rejected
TIME >> 1575284190819083900 >> EDGE >> 1 3 5 Branch
TIME >> 1575284190819083900 >> NODE >> 1 Found
TIME >> 1575284190819083900 >> EDGE >> 1 2 10 Rejected
TIME >> 1575284190819083900 >> EDGE >> 1 3 5 Branch
```

Listing 9: process 2 message log

```
user:~/go/src/dmst/run: go run run.go -id 2 -nid 1,3 -nwt 10,7
[SERVER] Listening on 'localhost:3002'
2019/12/03 23:01:21 Starting Handler
2019/12/03 23:01:21 [SERVER] Accepting connections.
START LOG GO ROUTINE
2019/12/03 23:01:26 [NODE 2] Test message received from node 1
2019/12/03 23:01:26 [NODE 2] Test message received from node 3
2019/12/03 23:01:26 [NODE 2] Initiate message received from node 3
2019/12/03 23:01:26 [NODE 2] Test message received from node 1
2019/12/03 23:01:26 [NODE 2] Test message received from node 3
```

Listing 10: process 2 state log file

```
TIME >> 1575284184929434200 >> NODE >> 2 Sleeping
TIME >> 1575284184929434200 >> EDGE >> 2 1 10 Basic
TIME >> 1575284184929434200 >> EDGE >> 2 3 7 Basic
TIME >> 1575284189816275300 >> NODE >> 2 Found
TIME >> 1575284189816275300 >> EDGE >> 2 3 7 Branch
TIME >> 1575284189816275300 >> EDGE >> 2 1 10 Basic
TIME >> 1575284190016881800 >> NODE >> 2 Found
TIME >> 1575284190016881800 >> EDGE >> 2 1 10 Basic
TIME >> 1575284190016881800 >> EDGE >> 2 3 7 Branch
TIME >> 1575284190518012100 >> NODE >> 2 Find
TIME >> 1575284190518012100 >> EDGE >> 2 1 10 Rejected
TIME >> 1575284190518012100 >> EDGE >> 2 3 7 Branch
```

Listing 11: process 3 message log

```
user:~/go/src/dmst/run: go run run.go -id 3 -nid 1,2 -nwt 5,7
[SERVER] Listening on 'localhost:3003'
2019/12/03 23:01:23 Starting Handler
START LOG GO ROUTINE
2019/12/03 23:01:23 [SERVER] Accepting connections.
2019/12/03 23:01:26 [NODE 3] Connect message received from node 1
2019/12/03 23:01:26 [NODE 3] Initiate message received from node 1
2019/12/03 23:01:26 [NODE 3] Connect message received from node 2
```

Listing 12: process 3 state log file

```
TIME >> 1575284185351078200 >> NODE >> 3 Sleeping
TIME >> 1575284185351078200 >> EDGE >> 3 1 5 Basic
TIME >> 1575284185351078200 >> EDGE >> 3 2 7 Basic
TIME >> 1575284189112611000 >> NODE >> 3 Found
TIME >> 1575284189112611000 >> EDGE >> 3 1 5 Branch
TIME >> 1575284189112611000 >> EDGE >> 3 2 7 Basic
TIME >> 1575284189715745200 >> NODE >> 3 Find
TIME >> 1575284189715745200 >> EDGE >> 3 1 5 Branch
TIME >> 1575284189715745200 >> EDGE >> 3 2 7 Basic
TIME >> 1575284190017881200 >> NODE >> 3 Find
TIME >> 1575284190017881200 >> EDGE >> 3 1 5 Branch
TIME >> 1575284190017881200 >> EDGE >> 3 2 7 Branch
```

Finally, from above logs we can identify correct edge classification at the end of each log for this test case. Note that there is Branch tag for edges that belong to the MST and Reject tag for edges that do not belong to the MST. Through this we have accomplished the minimum spanning tree using a distributed approach. Each process knows only its edge, there is not an entity that knows the whole network topology.

10 Conclusions

The group put themselves into a challenge of implementing a distributed version of the well-known minimum spanning tree problem. This project had several different challenging sides, such as how to design a Golang project from scratch, how to implement RPC calls, how to deal with generic message passing and how to implement correct GHS algorithm protocols. Afterall, we had plenty difficulties understanding references related to GHS algorithm mainly because there are not too many clear available references.

Further than the GHS implementation itself, the group faced the challenge of implementing a graph algorithm in Python that read all output files from GHS, parsed it correctly in order to come up with visualizable graphs that gives a sense about GHS correcteness and progress.

Finally, we trust that the project was quite enlighten not only regarding how to deal with GHS algorithm itself but also about how to design a distributed processing software from scratch. On previous labs developed on CES-27 course, we had the opportunity to partially complete professor codes but this time we had to implement all from scratch.

On this scenario, we thank professor Hirata and Juliana to allow us to chose a project of our interest where we could apply distributed processing knowledge developed over this semester.

11 Group organization

The GHS algorithm discribed at original article "A distributed algorithm for minimumweight spanning trees" contains 12 algorithm-specific procedures that can be executed at each process. On this scenario, team has equally splitted work among the following Golang tasks: project setup, RPC calls based on RPC calls from previous Raft proposed project, GHS algorithm-specific procedures development and the Python task of GHS outcome visualization through animated gifs. All group members worked at the same time developing a unified codebase using Git for code management.

References

G. Flysher and A. Rubinshtein. A distributed algorithm for minimum weight spanning trees. URL `https://webcourse.cs.technion.ac.il/236357/Spring2005/ho/WCFiles/MST.pdf`.

R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.

G. Pandurangan, P. Robinson, M. Scquizzato, et al. The distributed minimum spanning tree problem. *Bulletin of EATCS*, 2(125), 2018.

Wikipedia. Minimum spanning tree, a. URL `https://en.wikipedia.org/wiki/Minimum_spanning_tree`.

Wikipedia. Remote procedure call, b. URL https://en.wikipedia.org/wiki/Remote_procedure_call.