D – GNG

A Distributed Erlnag implementaion for Growing Neural Gas

Nils Jähnig
Francesco Cervigni

Our goal is to experiment and exploit the advantages of distribution and parallelization in an algorithm like GNG(taken from "A growing Neural gas Network Learns Topologies" from Bernd Fritze).

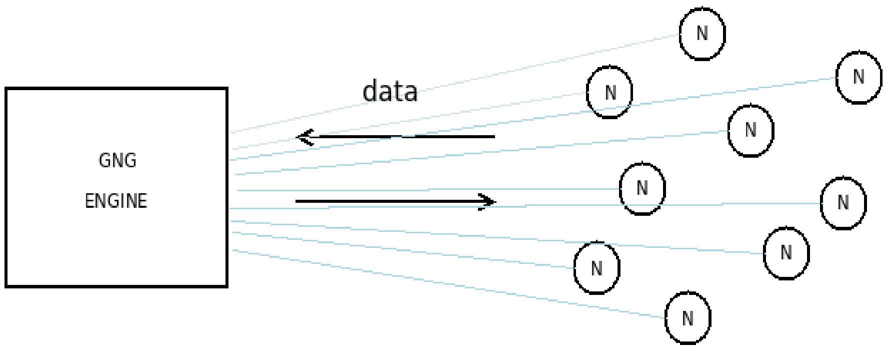
Our goal is to implement the GNG algorithm in a higly-distributed oriented way in order to experiment and exploit the advantages of parallelization and scalability offered by a language like Erlang.

How distributed ?

First Step

We firstly thought about an implementation were a node’s behaviour is performed by an autonomous process able to comunicate with other modules of the algorithm.

In this way the distribution, but in order to perform an algoritmh like GNG working on this node is necessary to continously collect informations form those, analyze it and send back actions to perform.
In order to do this is necessary to have centered data structures and a big amunt of broadcast messages coming form the algorithm module to the nodes.

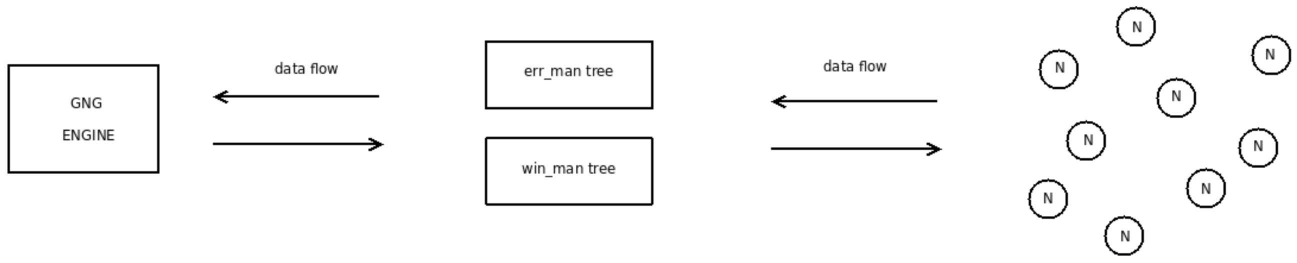


Still scalability reamins limited.
The dimension of number of nodes influences massively the general complexity, mostly in order of data storage, number of messages and because of these, performances.

Second Step

We needed to add a data structure that was able to distribute reduce the

comlexity of the global interprocess comunication suited for the tasks needed for GNG.
We designed a structure where the engine of the algoritmh comunicates collects data from the nodes space that would indipendent by the number of nodes.

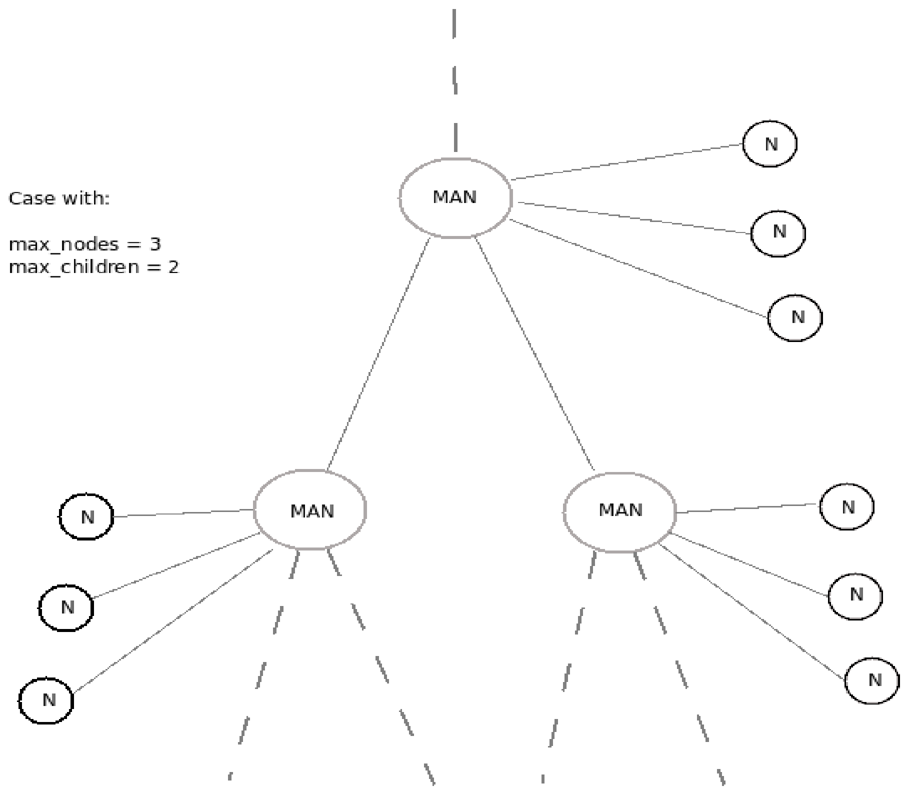


Tasks like electing the leader or look for the maximum error are now performed though a tree of processes called Manager.

A Manager has:
- A list of Nodes
- A list of SubManagers

The max_lenght of these lists is defined by parameters of D-GNG and can be choosed at the start calling.

A manager tree can be represented in like this :



Using this structure and doing as possible to limitate the message number and the blocking receive statements we limited the complexity in term of messages to be LINEAR respect to the maximum number of nodes per Manager.

Gng can be decomposed in 3 main task:

- Winner election
- Neighbourhood updates
- Error analyzing and network Growth

We used two different and parallel trees to reach the nodes:

- A tree called win_man, performs the winner election.
- A tree called err_man, performances the max_error decision and controls the tree growth

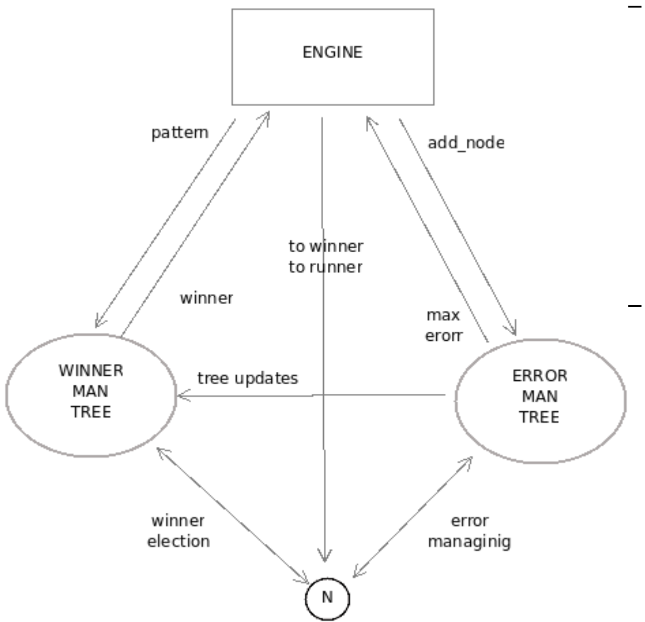
IMPORTANT:
Those two trees must always have the same network representation.
Every manager in a tree has a “twin” in the corrispondent position in the other graph.
Every modification of the tree like adding or removing a node is performed by the err_tree’s managers and sudden comunicates to the win_man to do the same.
We choosed err_man for this task because the win_manager has to spread informations **and** wait for answers, while the err_manager usually just spreads error updates down the tree. This makes him idle when waiting for the win_manager.

This is done by message passing.

So , generally the interactions in D-GNG are :

- win_man collects distances by every node and computes the winner and runner in a number of step corrispondent to the depth of the tree, and sends it to the Engine.
- The Engines comunicates directly with the Winner and the Runner via PID for the edge updates.
- err_man updates the errors, and can return the node with max error when requested. It also performs the node adding, that let the tree grow.

- A schema:



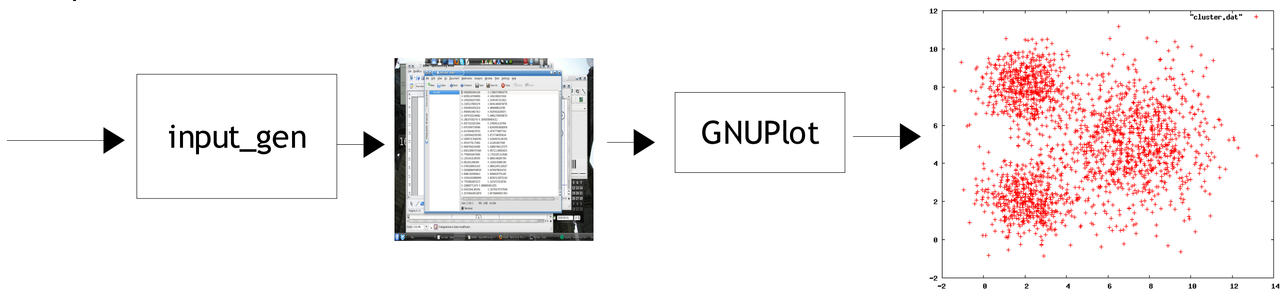
I/O

The data is read by a file, it is dimension indipendent, both integer and float values are recognized.
Every column represents a pattern, every column is a dimensional coordinate:
3-D example:

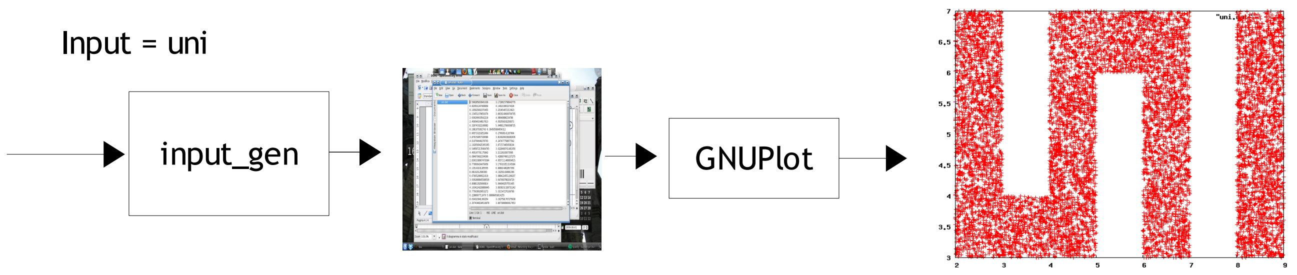
```
#X    Y     Z
1.1   43    5.3
2.52  5.23  3
4     3.1   4
```

We generated the input file developing a program called input_gen.erl.
In order to visualize the data we used GNUPlot, performing a script to control the output. (A script plots continously data to visualize the input position)

Input = cluster

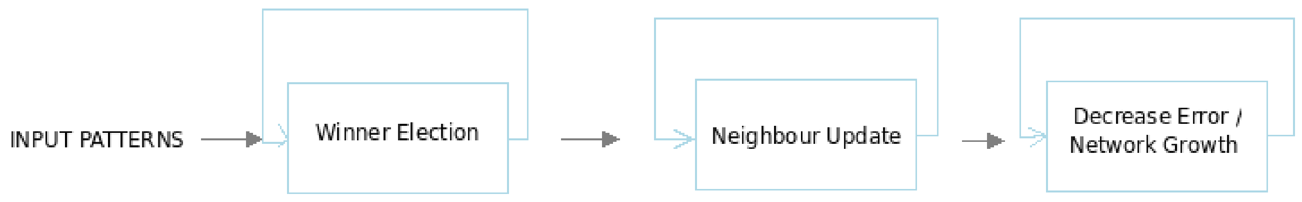


Input = uni



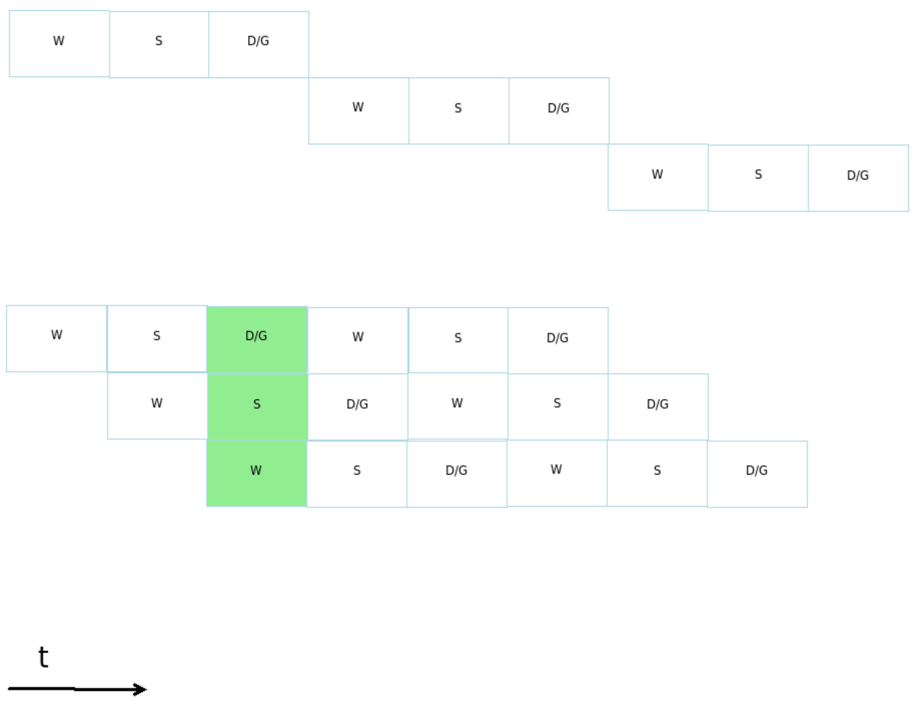
Regarding the output, will be possible to see graphically the nodes' position on the input space.
Nodes continously write their position in a file.
A bash script every second makes GNUPlot plotting the positions on a graph.

Decomposing the algorithm in 3 indipendent modules:



Imagining the algorithm as a factory chain, once the input is fed and the winner election is computed, the win_node is not used any more.
So, when the winner and runner are sent to the next tasks, the Winner Election module can accept another input and perform a new election and so on.

In these way we try to reduce as possible tha waiting time, improving parallelism.
Our implementation resembles a 3 stages pipeline, as we can show :



The code implentation is giving very promising results, but is still in debugging phase, so we can’t provide yet comparisons with other implementations or benchmark results.