

Switching and Routing

IP LOOKUP B – PREFIX LENGTH ALGORITHM

Federico Broggi 876735

Andrea Ganassa 884791

Chiara Golin 877420

Daniele Uboldi 874996

2016/2017

1. Introduction

The aim of our project is to design a network composed by multiple switches each with a different subnet mask. Once designed the network and the topology we implement binary trie IP lookup algorithm and binary search on prefix length algorithm and we measured and compared the performance parameters of different algorithms.

The crucial point of our project is to implement the prefix length algorithm, hard because of some passages like the introduction of markers or an inside error of the algorithm that we will explain later.

1. What we need

The requirements that we need to do this assignment are:

* Knowing what are the Binary trie IP lookup and binary search on prefix length algorithm
* Being familiar with the language Python
* Having Ryu installed on our PC
* Being familiar with the principal elements of a network system

1. What is SDN

Software-defined networking is an emerging architecture that is dynamic, manageable, cost effective and adaptable. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. In practise, the control plane (which takes decisions on where to send the incoming packet) resides on a different system from the data plane (which forward the packets to its corresponding destination).

The Openflow protocol is a fundamental element to build SDN solutions.

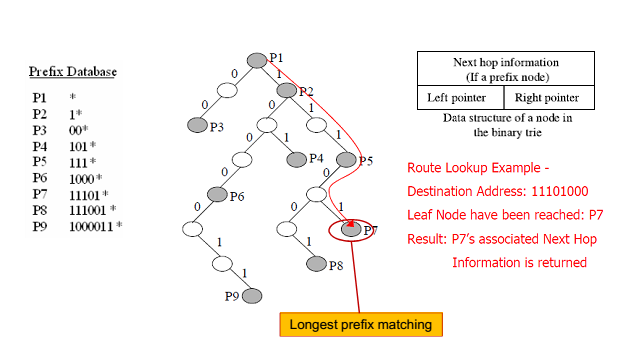
The SDN architecture, composed by a North-bound interface (between the controller and the control apps) and a South-bound interface between the controller and the network elements) is:

* **Directly programmable**: Network control is directly programmable because it is decoupled from forwarding functions.
* **Agile**: SDN helps organizations rapidly provide new applications, services, and infrastructure to quickly meet changing business goals and objectives.
* **Centrally managed**: Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
* **Programmatically configured** : SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software. These also simplify the network design because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

1. BINARY TRIE

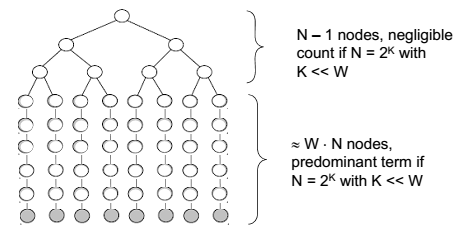
It is a multi-way tree with each node containing next-hop info and 0-1-2 pointers to child node(s).

Example with final destination 11101000\*:



Performance of this algorithm are calculated in terms of:

* + Storage complexity: O(NW)

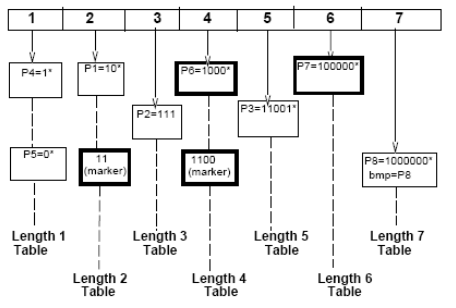


* Lookup complexity: O(W)
* Update complexity: O(W)
* N: number of prefixes
* W: number of hash tables

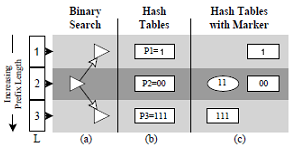
1. BINARY SEARCH ON PREFIX LENGTHS ALGORITHM

The aim is to search the longest match. The idea of the algorithm is decomposing the longest prefix matching into a series of exact matching operations, each performed on the prefixes with the same length. The exact matching is performed by hash tables.

To implement this algorithm we should add some markers, useful guide search in order to find longer prefixes. Markers include length of longest matching sub-prefix and they are used to find match after failed search for longer prefixes.



Markers lead to an error that we show in our implementation later. Here there is an example with a destination of 110. There is a marker (11) that should lead to a longer prefix, but no longer prefixes will match with this destination.



Performance of this algorithm are calculated in terms of:

* + Storage complexity: O(N log2W)

Because only log2W markers would be probed instead of W markers, the storage complexity decreases from O(NW) to O(N log2W)

* Lookup complexity: O(log2W) Taking no account of the hash collision
* Update complexity: O(log2W)
* N: number of prefixes
* W: number of hash table

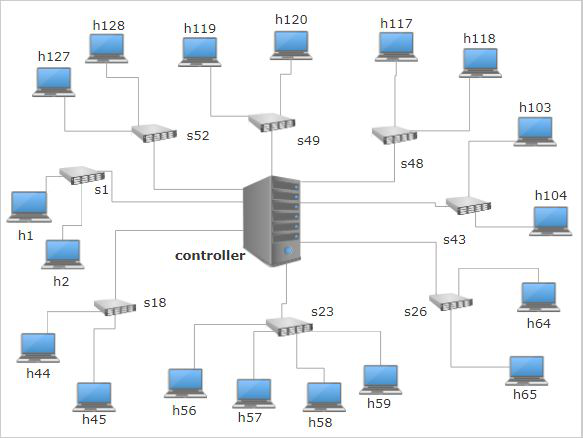
1. Implementation
   1. Network simulation

Firstly we must emulate an OpenFlow network, in fact we must implement our 2 algorithms on SDN controller. So we use Mininet and Ryu to create the controller: both support OpenFlow.

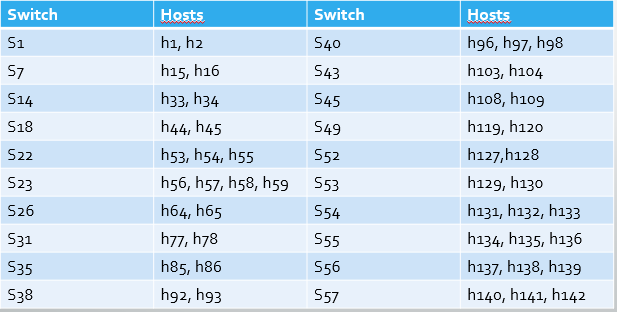
In order to show our simulation, we created three different topologies, one smaller with 8 different swiches and 18 hosts, one medium with 20 switches and 48 hosts and one larger with 35 switches ad 85 hosts. The aim of this procedure is to take over the differences in terms of:

* Minimum
* Maximum
* Medium
* RMSD
* lookup time

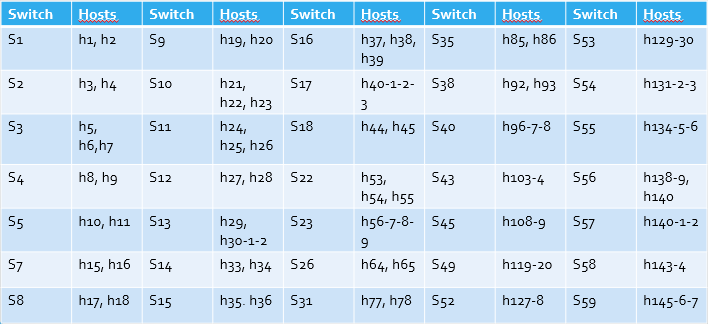
This is the smaller topology, with all the switches and their hosts connected:



This is the medium topology:

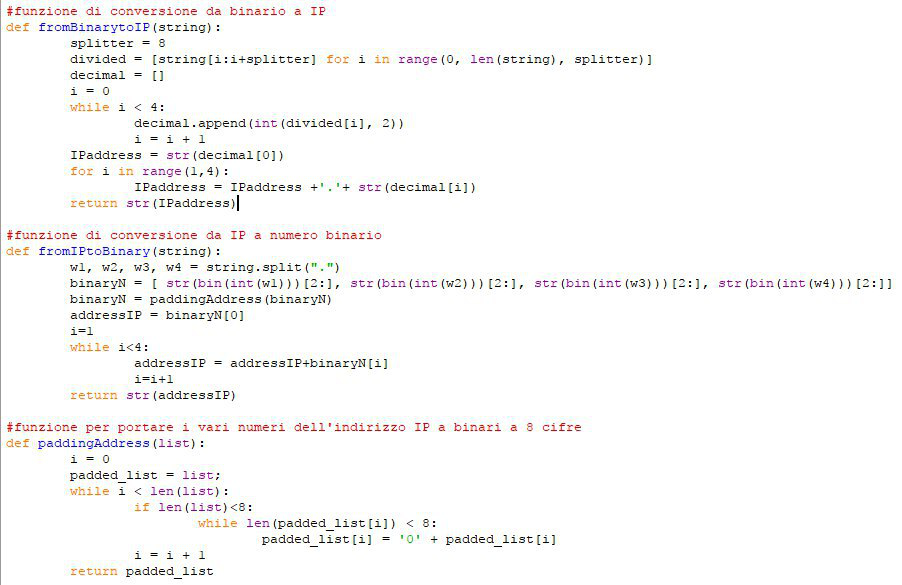


And this is the larger one with all the switches and connections:

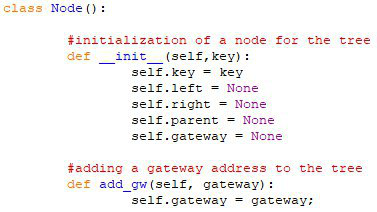


* 1. Algorithm implementation

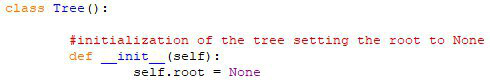
**FUNCTION TO CONVERT ADDRESS FROM IP TO BINARY AND FROM BINARY TO IP**

Thanks to padding address every transformed address become 8bit/8bit/8bit/8bit.

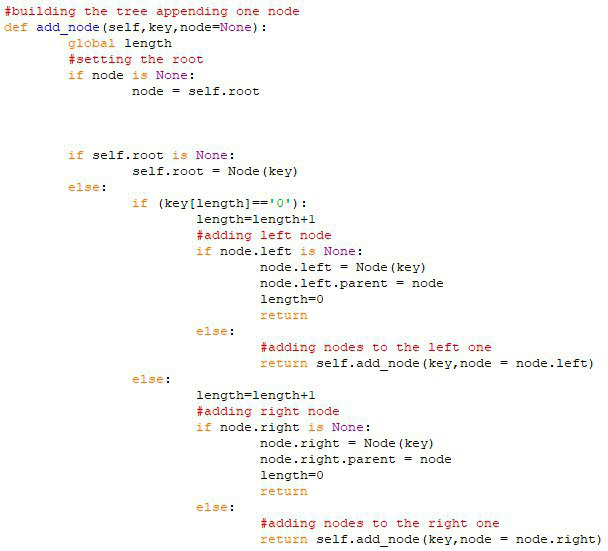
**CLASS NODE OF THE TREE AND FUNCTION TO ADD GATEWAYS**

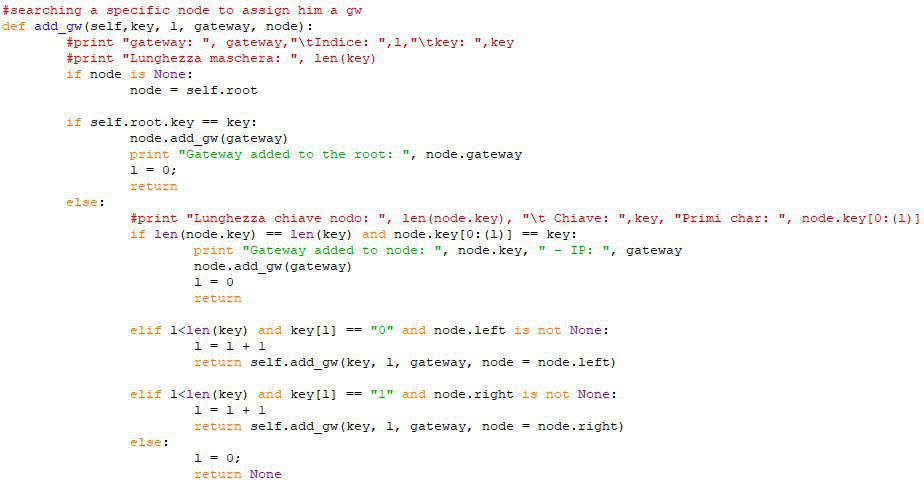


**CLASS TREE AND “CONSTRUCTION OF THE TREE”**



**FUNCTION ADDING NODES**



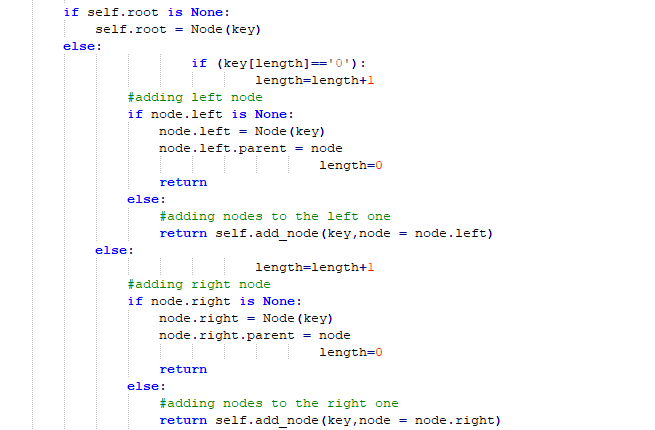
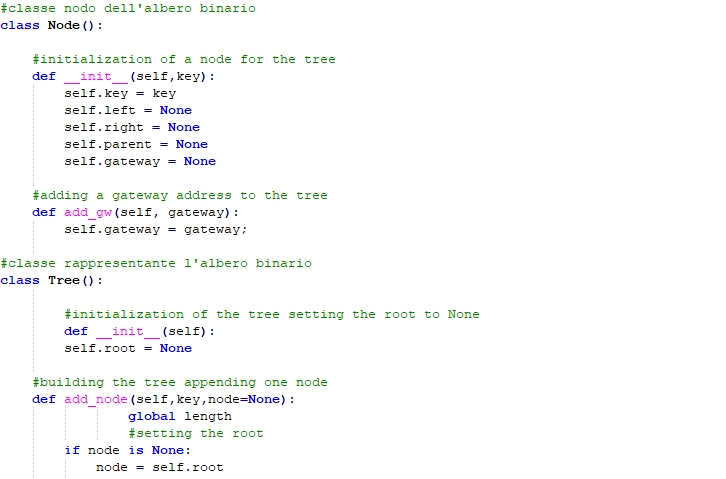
**FUNCTION ASSIGNING A GATEWAY TO A SPECIFIC NODE**

**BINARY ALGORITHM**

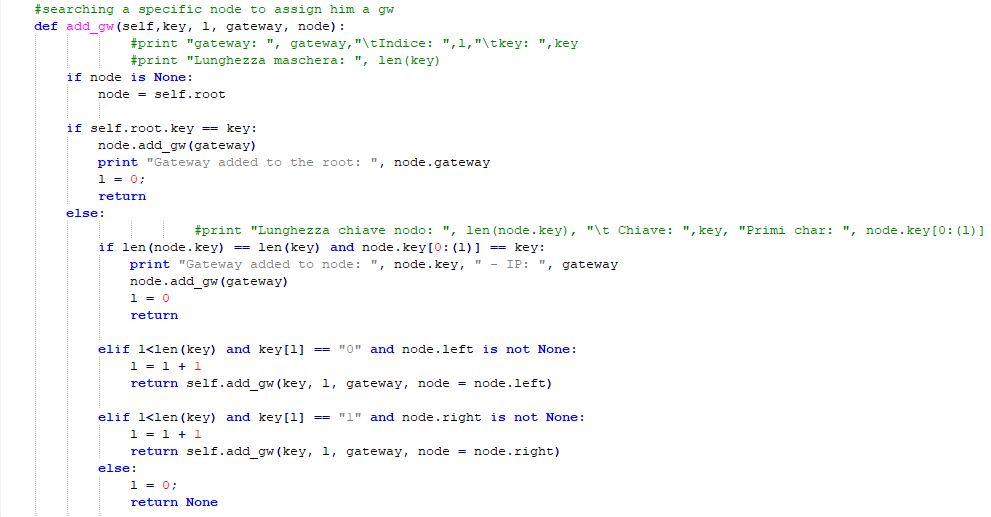
With the aim of implementing the algorithm of IP lookup through a binary trie, we built a network with a series of nodes distinguished by different prefixes and some gateway, in which an IP address could reach easily his corresponding longest prefix match.

In the first class we created the foundations to build a tree: we inizializated the nodes and the gateway address of the tree. In particular we choose, in this case, to implement a series of nodes and one gateway for each level of the tree.

In the second class we started building the tree. We inizializated the root, firstly setting to a null value. Then we started adding a nodes appending it to the root: if the value of the first corresponding bit of the prefix was 0 the node was appending to the left branch and the root became the father of this node. If the corrisponding bit was 1 the node was appending to the right branch. This process was recursive: from father to son we built a tree with some levels corresponding to the lenght of the prefix of each nodes.

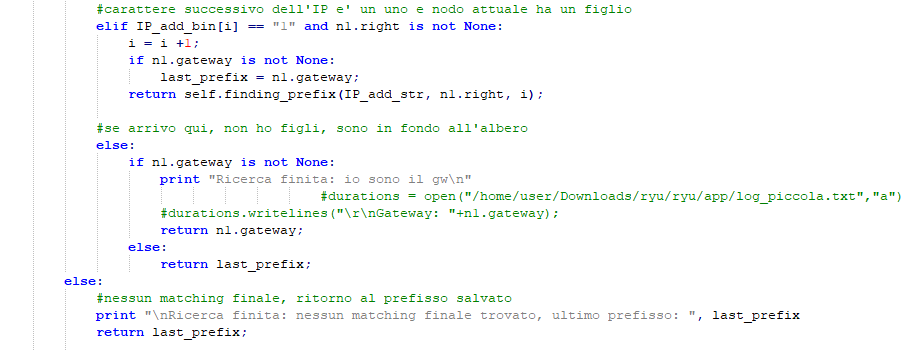
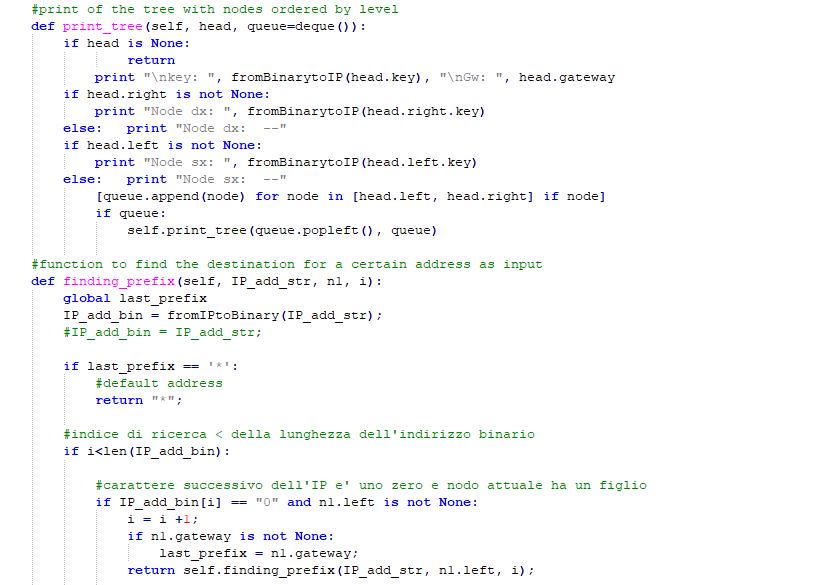


Then we had the problem of assigning a gateway to a node. Using the function *add\_gw* we added some gateway (one for each level) to some nodes: analyzing bit by bit the search passed though the father to the son until it reached his node destination and corresponding. Also this procedure was recursive. All the gateway added had also IP addresses corresponding.

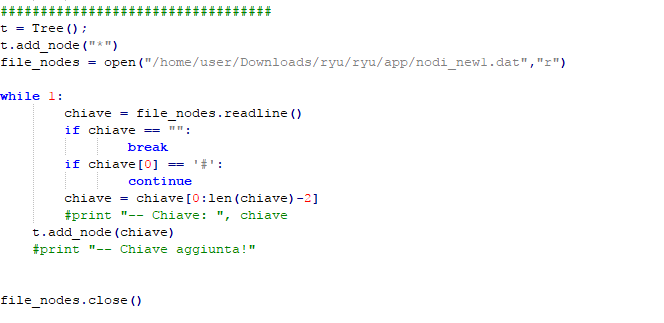


After the function written with the aim of printing the tree ordered with the right levels, we implemented the IP lookup and the binary search. The address must controlled his prefix bit by bit (of course with the constraint that the search must continue until the end of the lenght of the corresponding address) and, recursively, if the branch choosen from his corresponding bit existed (for example if the bit was 0 and the nodes in which it was arrived had a son on his left) it reached the node son and the search continued. If son didn’t exist the search was finished, we were at the end of the tree and the gateway corresponding was the last visited and saved.

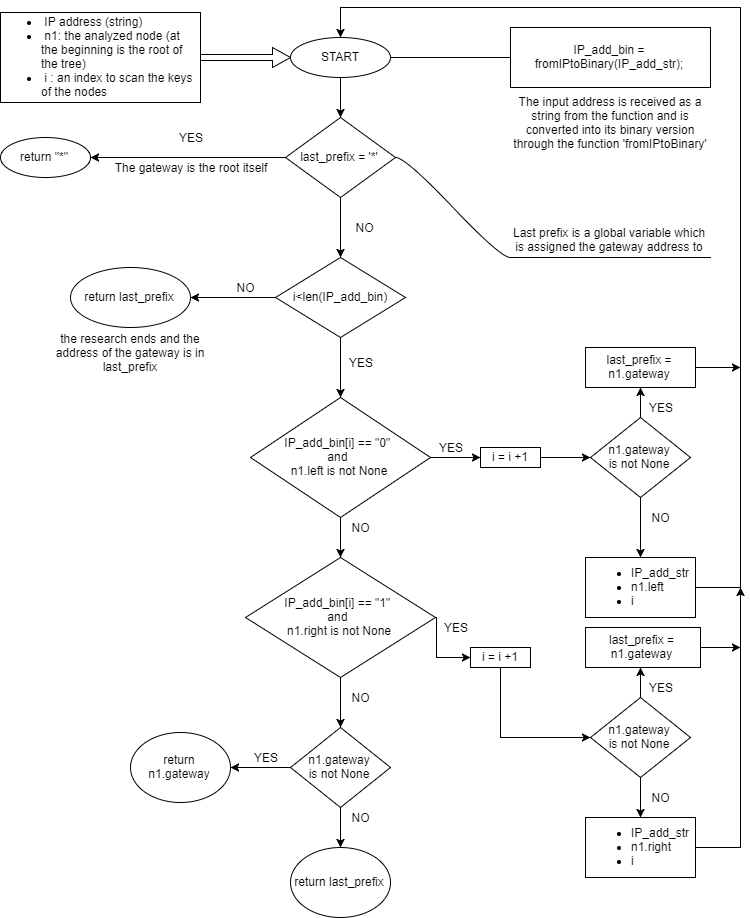
With the function “*finding\_prefix*” we choose an address to show that our search worked.



And the final function to write on file:



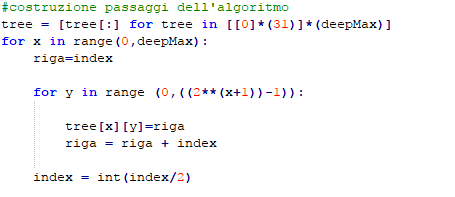
And this is the flow chart of binary trie algorithm:



**BINARY SEARCH ON PREFIX LENGTHS ALGORITHM**

This is the prefix length algorithm.

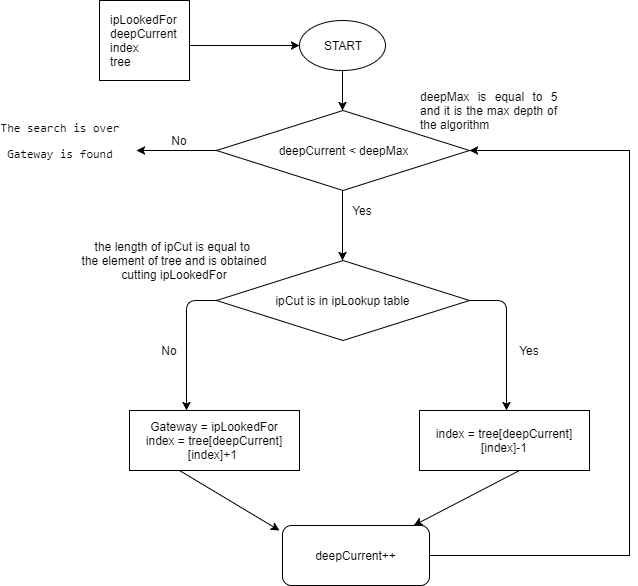
Firstly we built the series of tables corresponding to the passages of the algorithm.



Then we started searching a value: if a prefix was found, in the next row we started looking at the right side of the series of tables; if not, we started searching at the left one.



And this is the flow chart of prefix length algorithm:

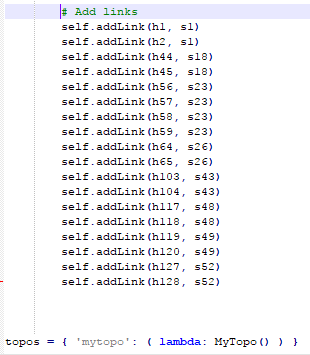
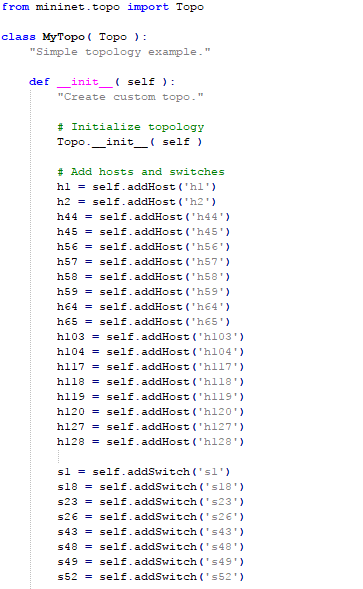


In order to implement the various topologies, we had to modify three types of file:

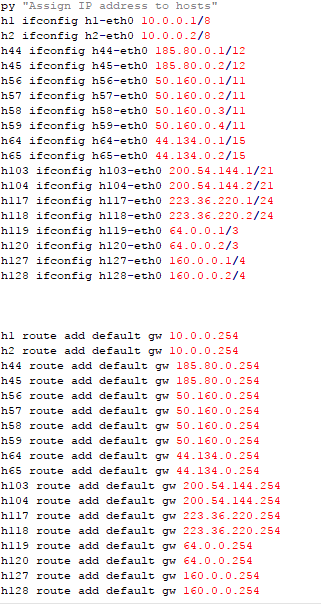
* The topogy
* The file config
* The ipLookup file.

We present here an example of what we modified, in the case of the smaller topology.

The topology, in which we initialized the switches, the hosts and how they were connected.

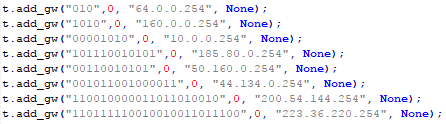


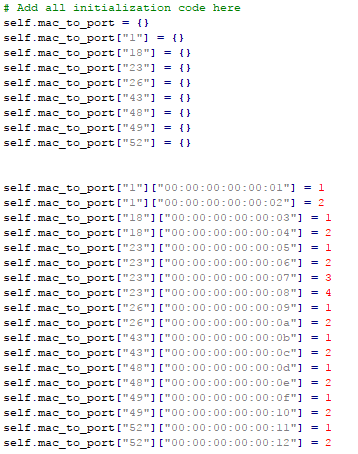
The config, in which we configured the network, with all the address associated to the switches.

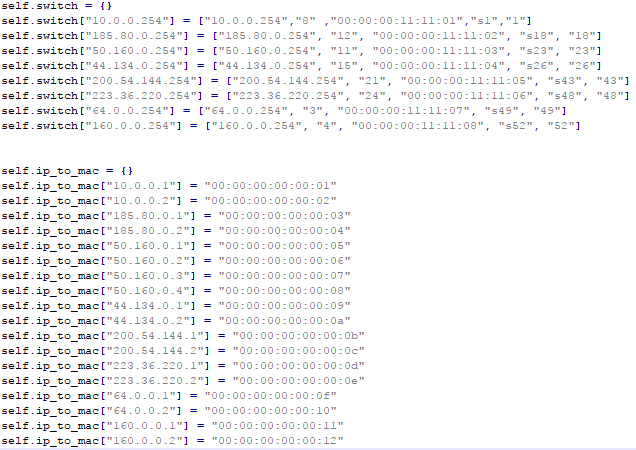


Finally the IpLookup file, that contains also the code of the algorithm.

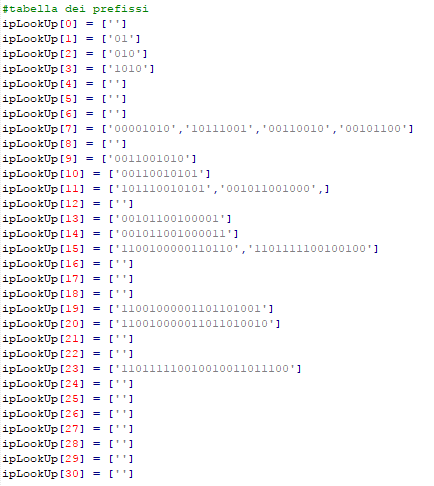
First the binary trie, in which we added the gateway, we initialized all the ports connected to the switches.



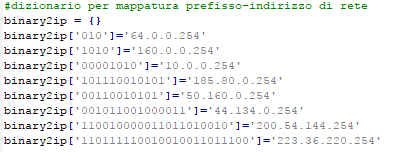




And here there is the prefix algorithm: we created the prefix table, the marker table (empty in this case) and the mapping between prefix and address.



C:\Users\polo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Cattura27.png



* 1. How to run the code
     1. File

To implement the simulation we need to have these files:

- *config\_xxx.py*: [mininet] this file contains hosts with their IP address and subnet mask. It also adds a default route to a gateway that connect the host to its switch.

- *topology\_xxx.py*: [mininet] this file configures the net of the simulation. It adds hosts, switches and links that connect hosts to their switches.

- *ip-lookup-xxx-xxx.py*: [controller] this file contains the controller of the net. It has the lookup algorithms, in our case we implement Longest prefix match and Binary Tree. It also contains the configuration of the IP address of hosts and switches, MAC address and relative port.

- nodi\_xxx.dat: [controller]: this file contains the information of the support tree for the Binary algorithm.

To Run and test the simulation in the virtual machine we:

* Open a terminal to run mininet with these commands:
  + cd Downloads/ryu/ryu/app
  + sudo mn --custom *topology\_xxx.py* --topo mytopo --mac --controller remote -- pre config\_xxx.py
* Open another terminal to run the controller with these commands:
  + cd Downloads/ryu
  + sudo PYTHONPATH=. ./bin/ryu-manager ryu/app/ip-lookup-xxx-xxx.py

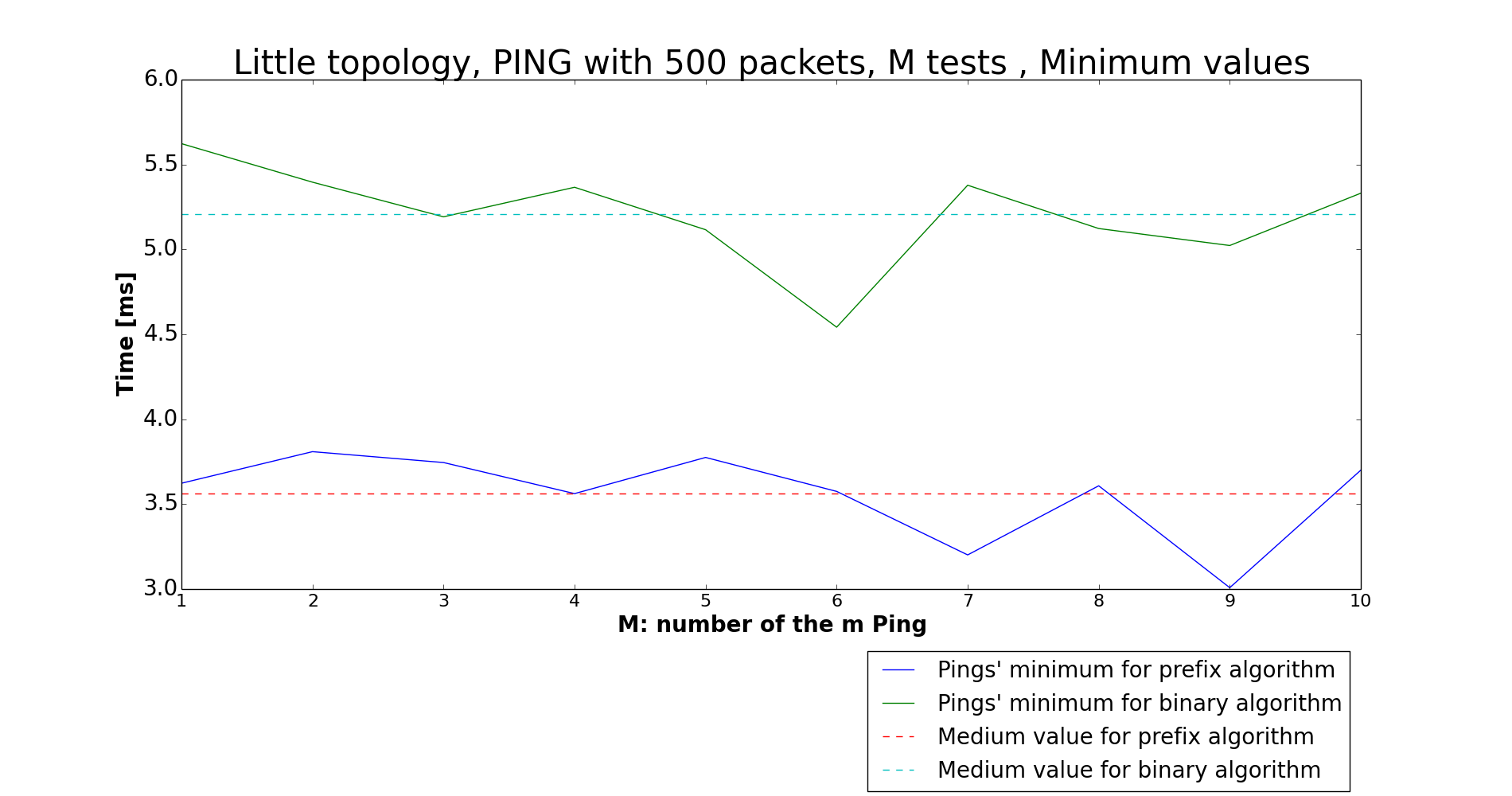
p.s. All this file must be in this folder **Downloads/ryu/ryu/app**

1. Results

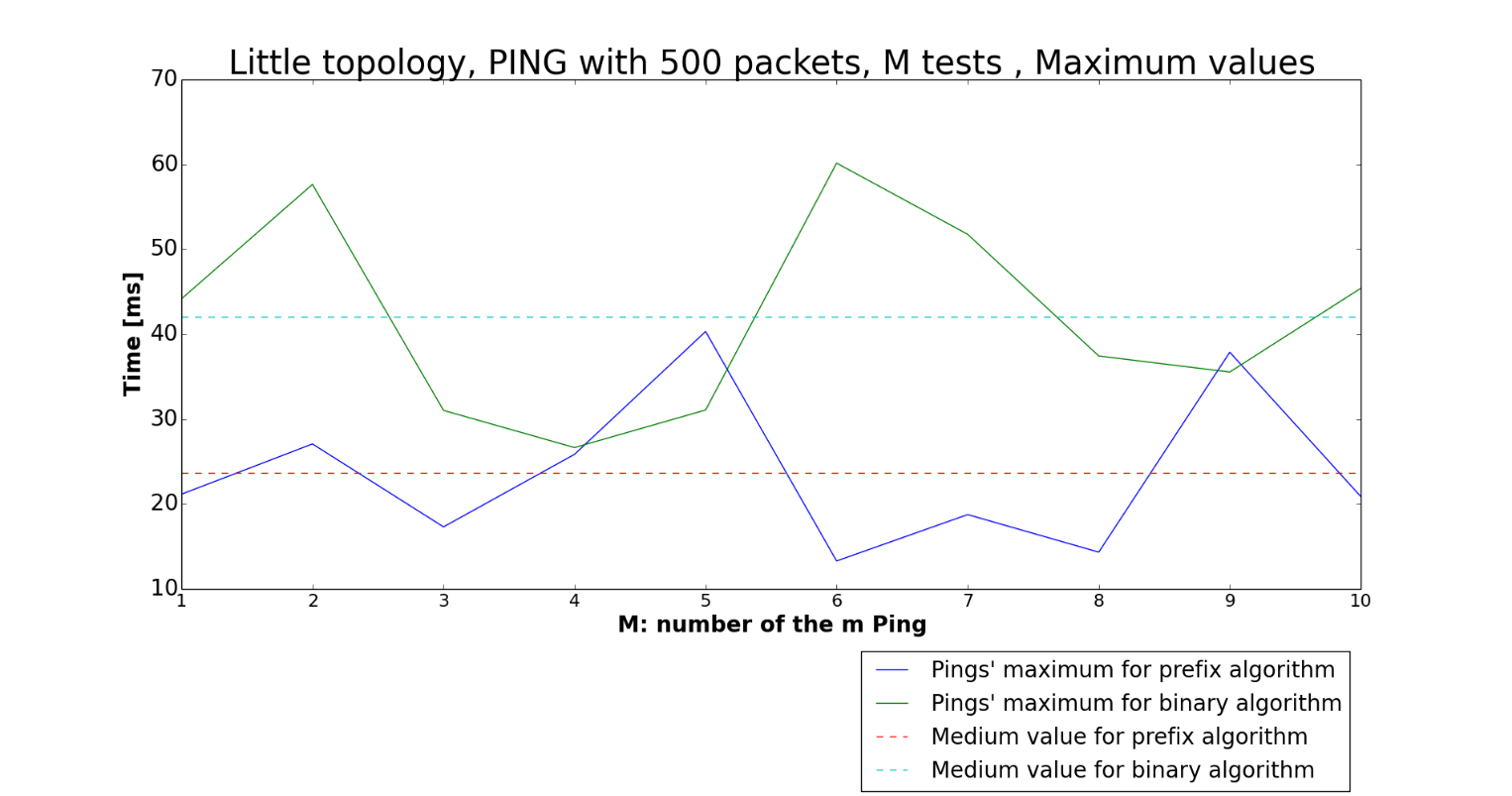
We show our results dividing them by topology.

**TOPOLOGY LITTLE**

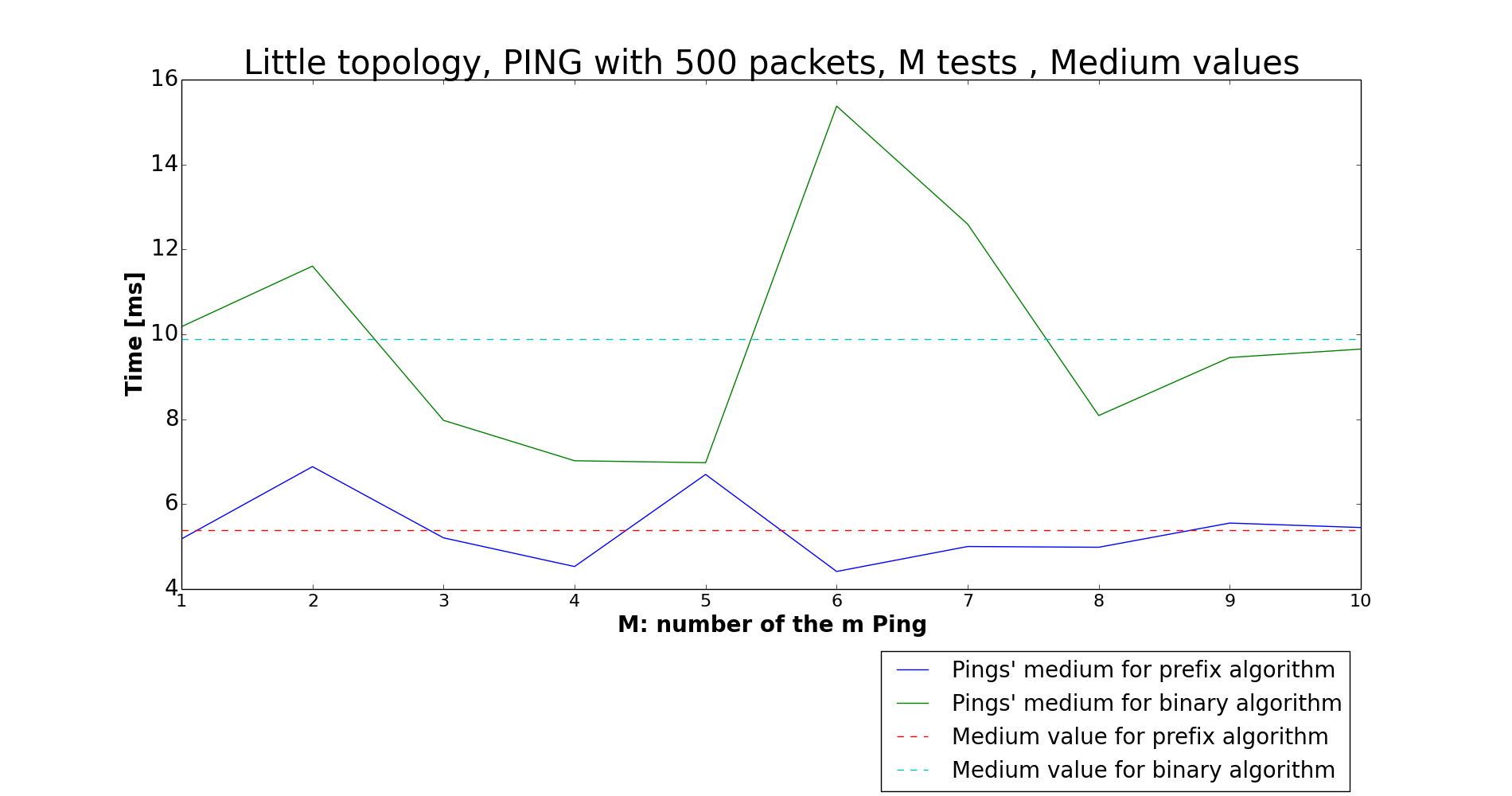
MINIMUM VALUES:



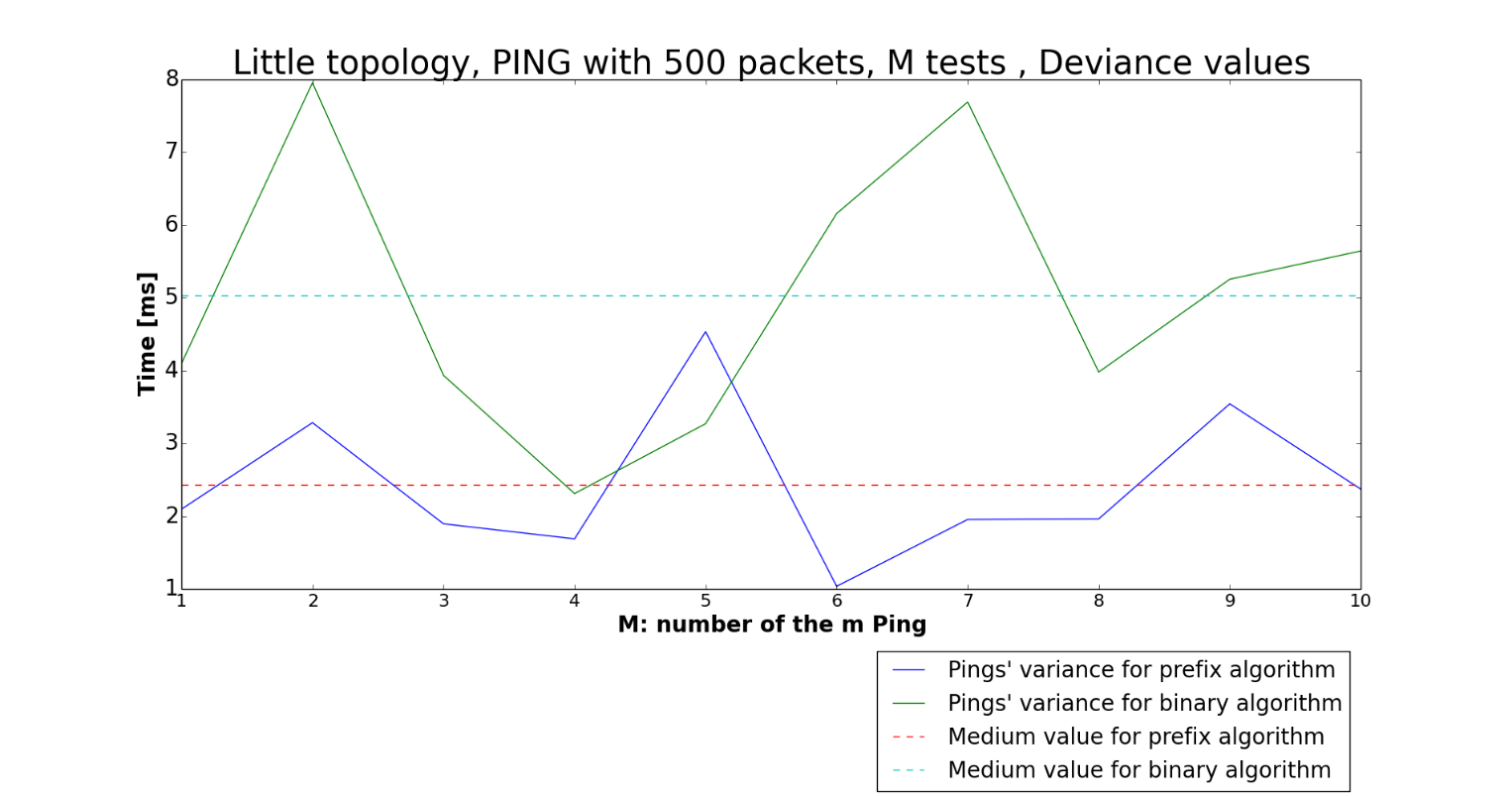
MAXIMUM VALUES:



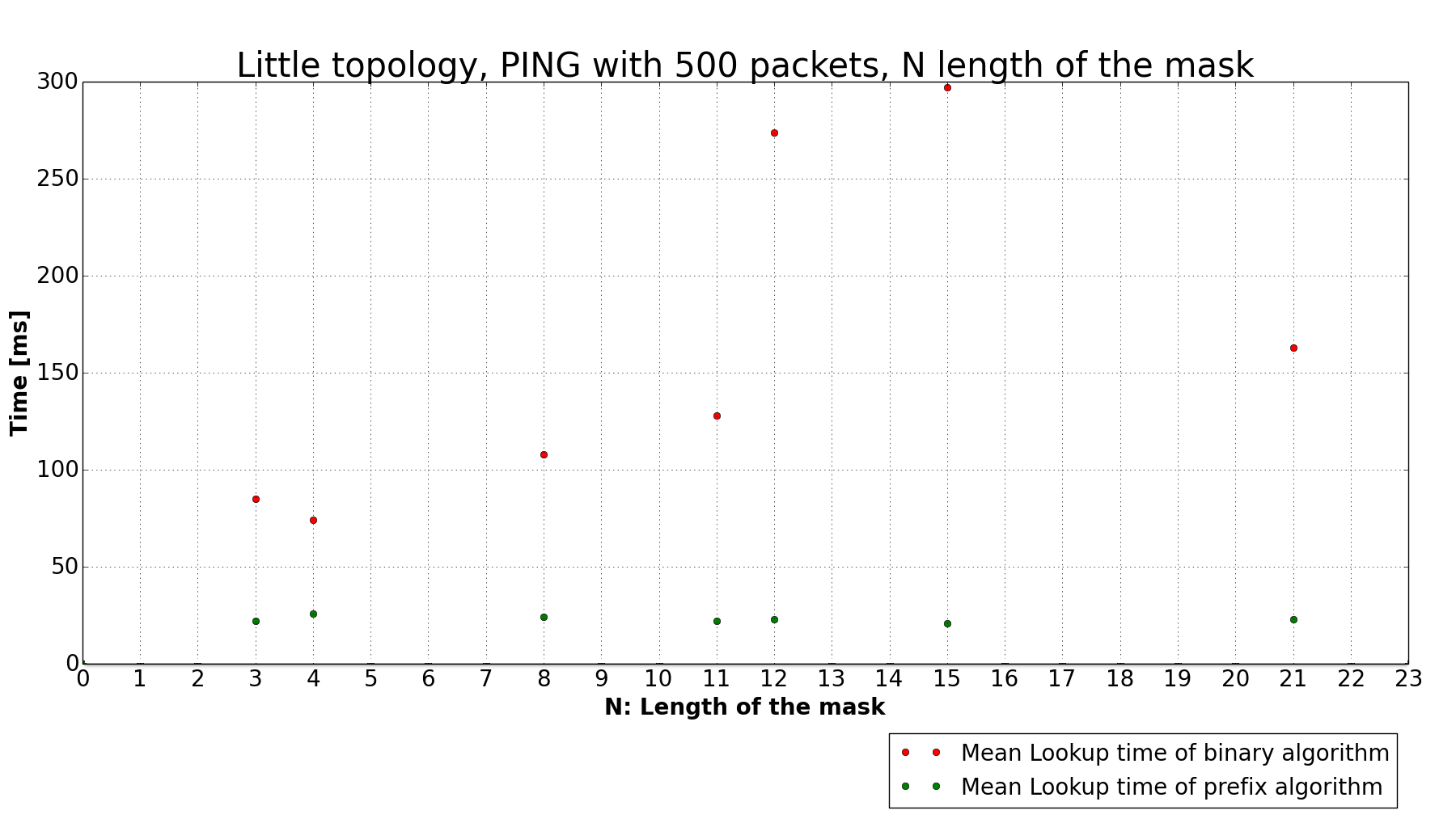
AVERAGE VALUES:



RMSD:

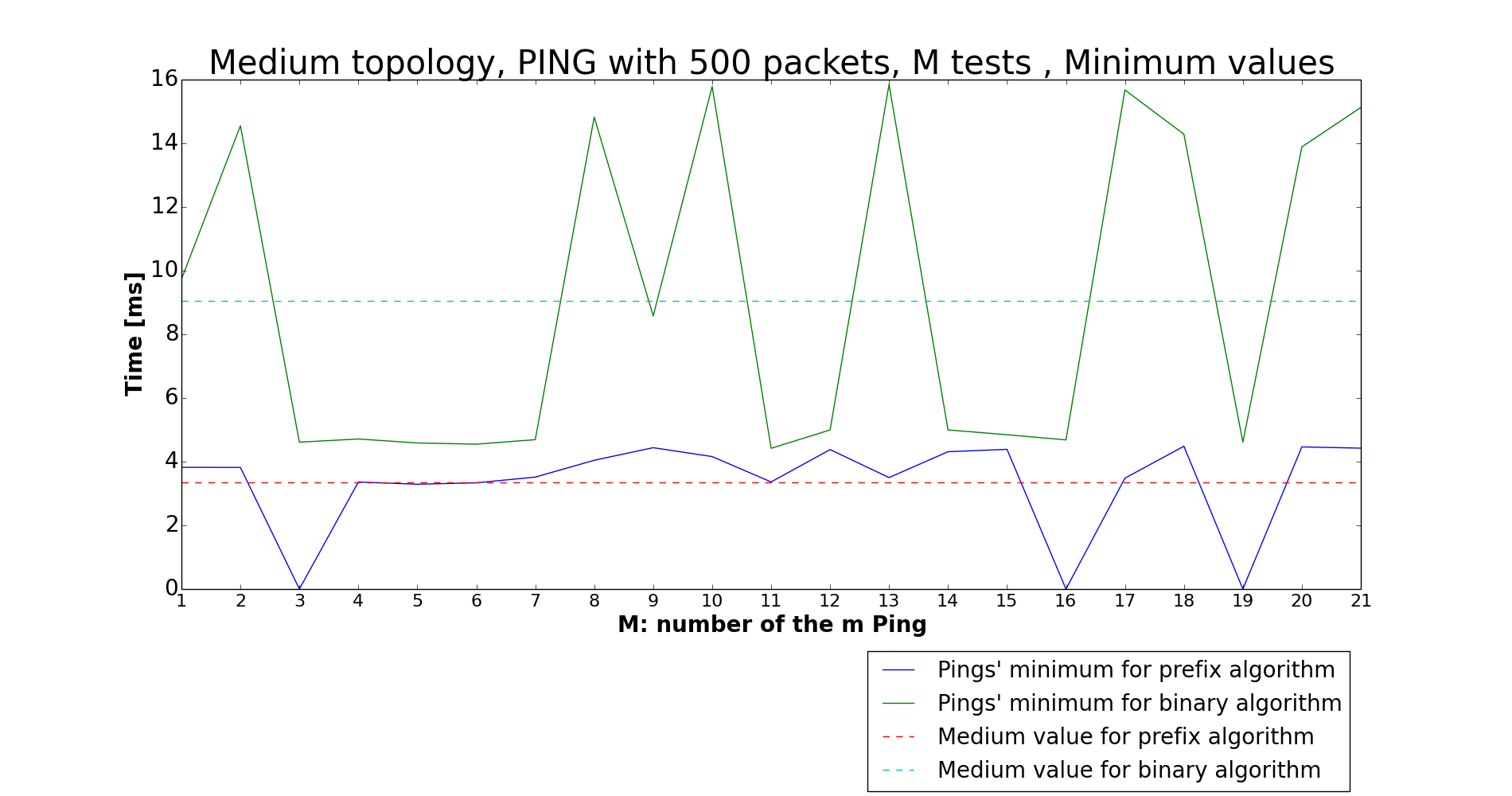


LOOKUP TIME:

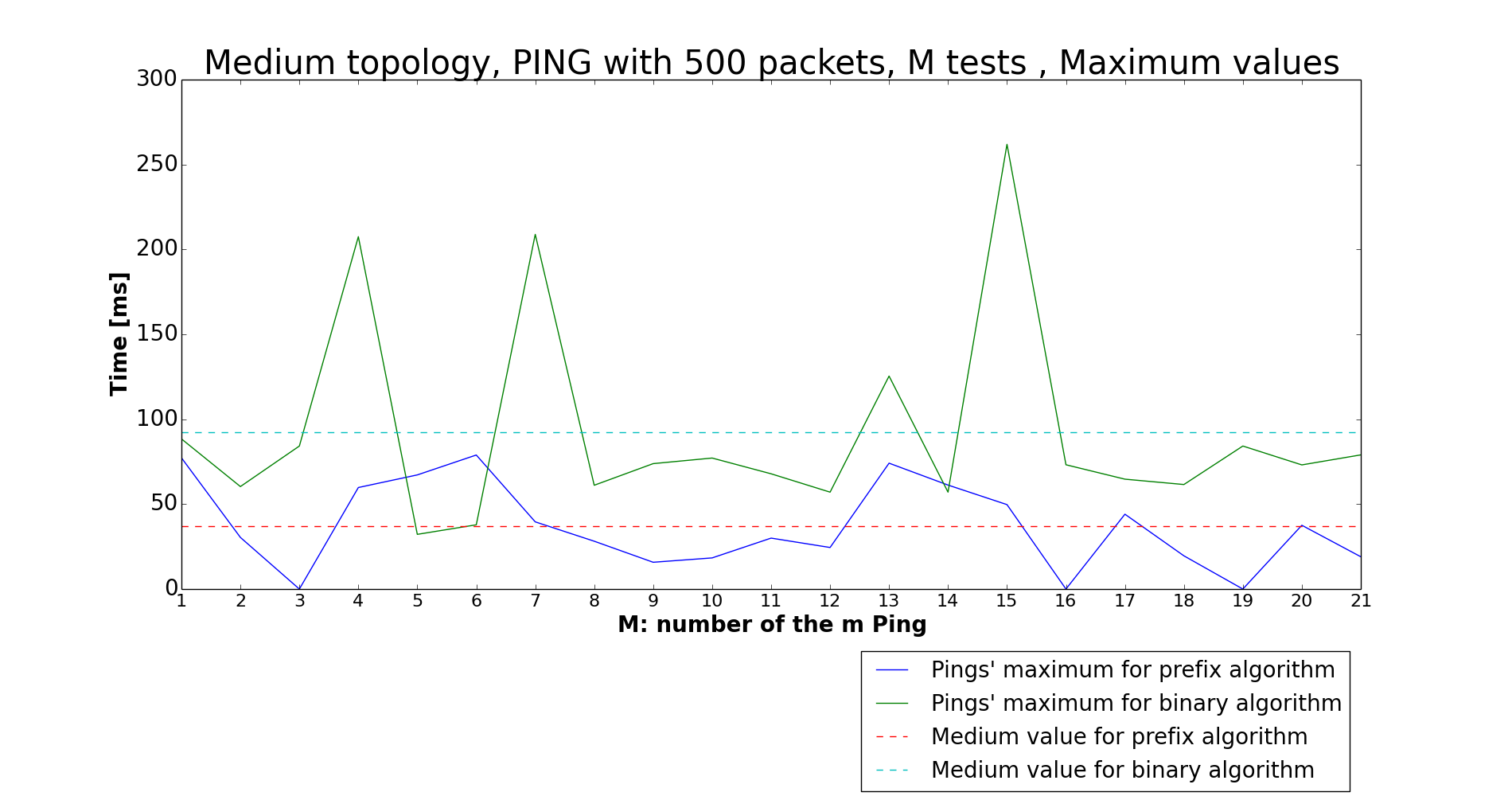


**TOPOLOGY MEDIUM**

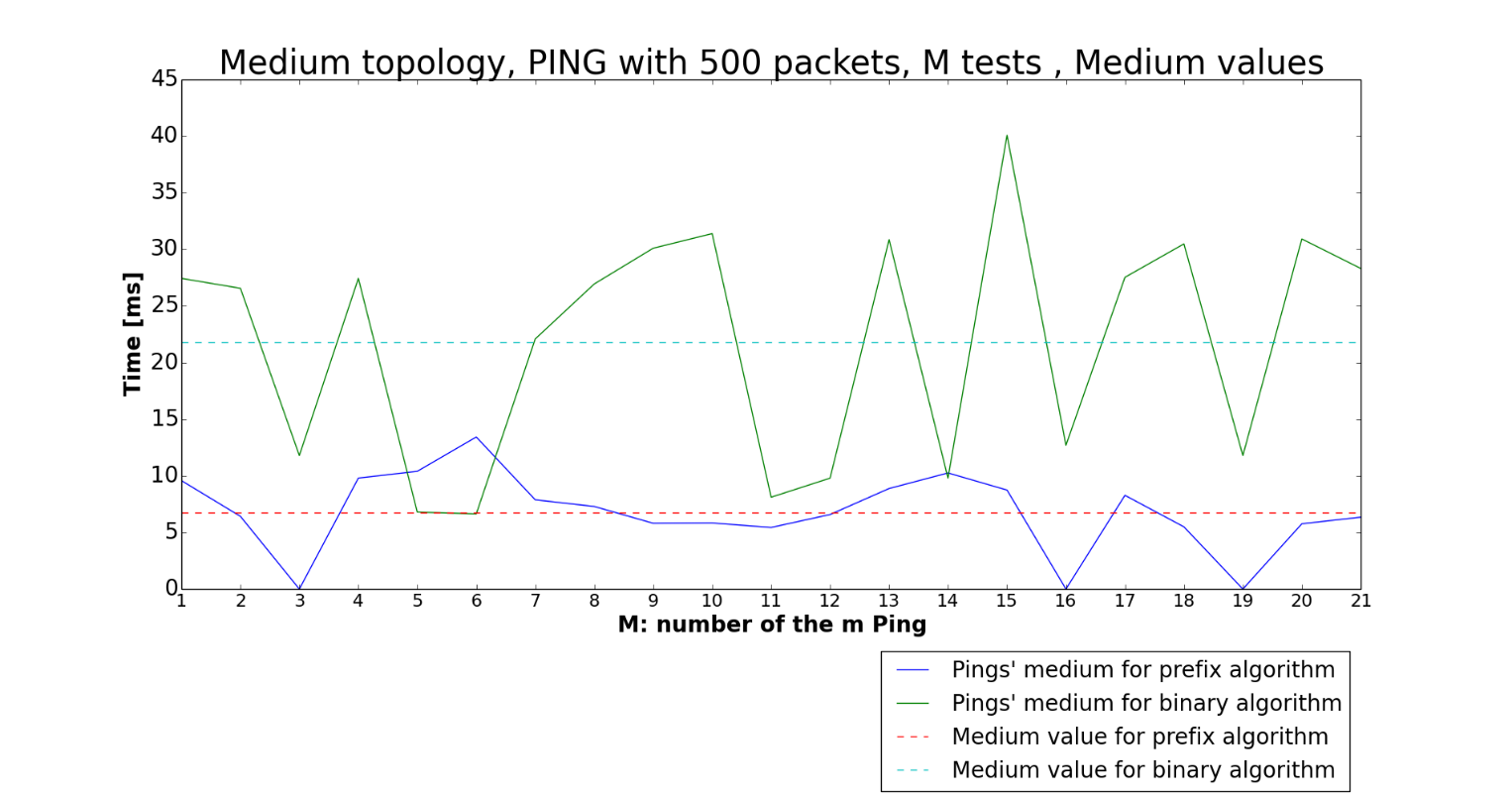
MINIMUM VALUES:



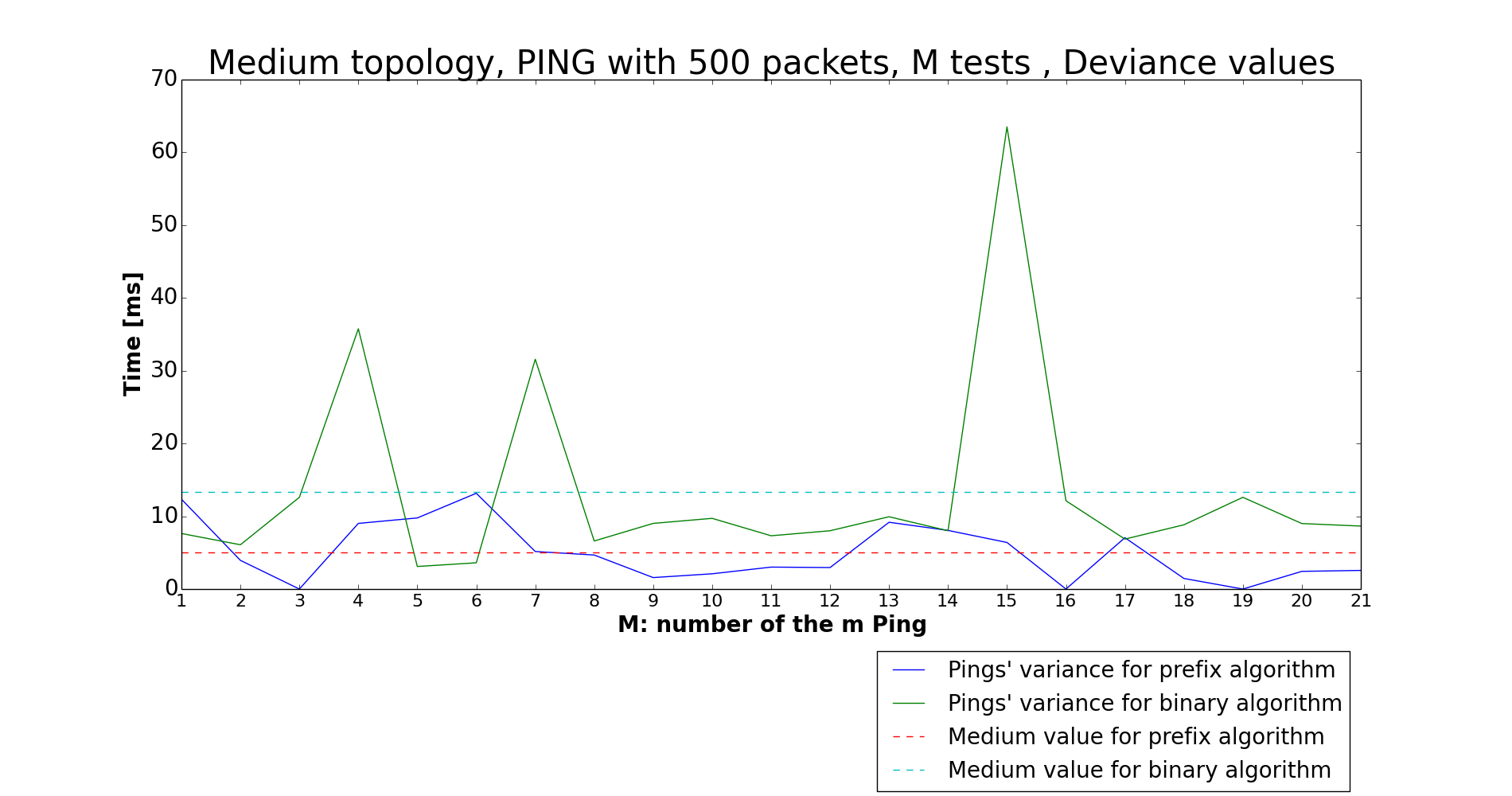
MAXIMUM VALUES:



AVERAGE VALUES:

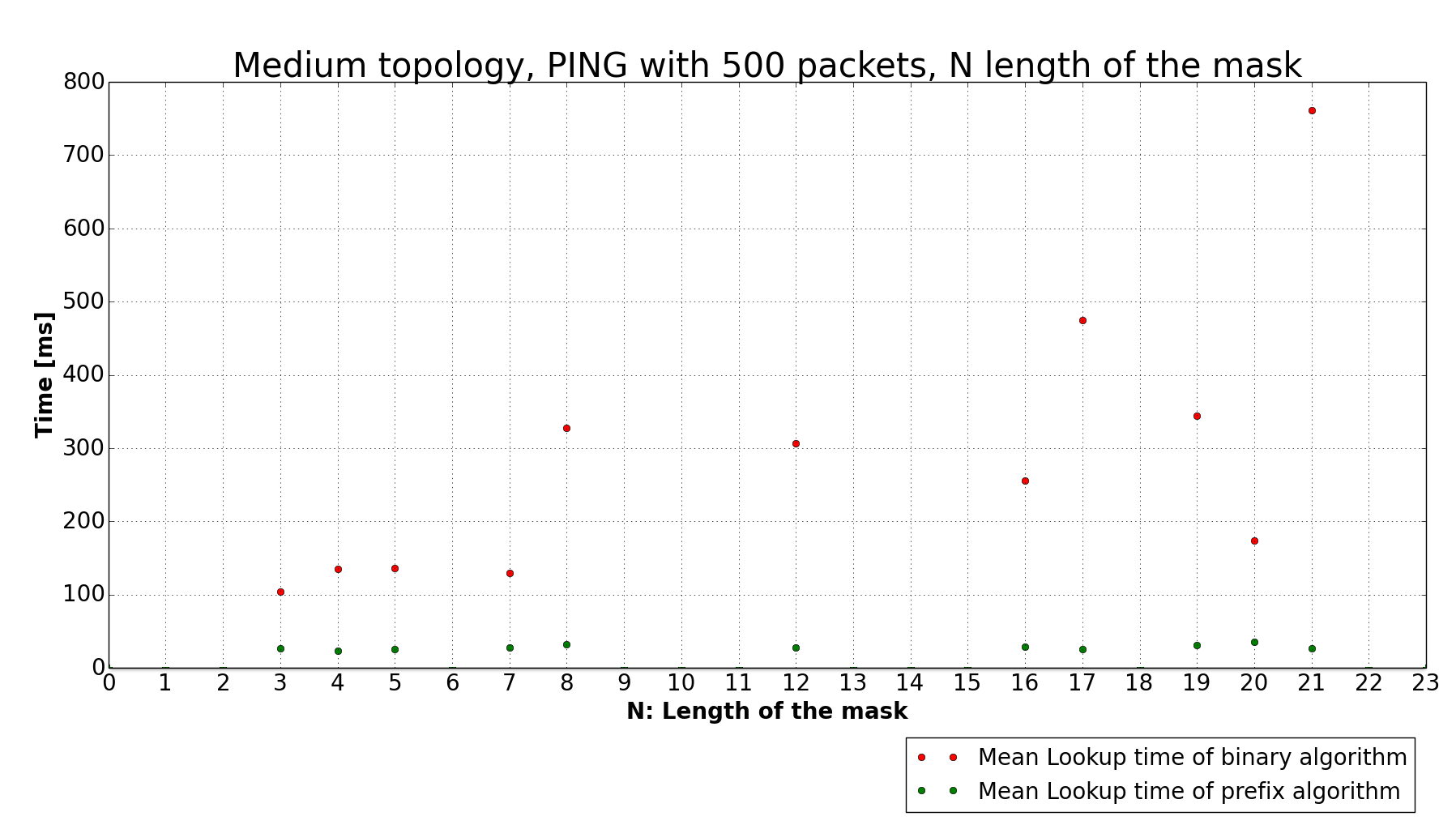


RMSD:



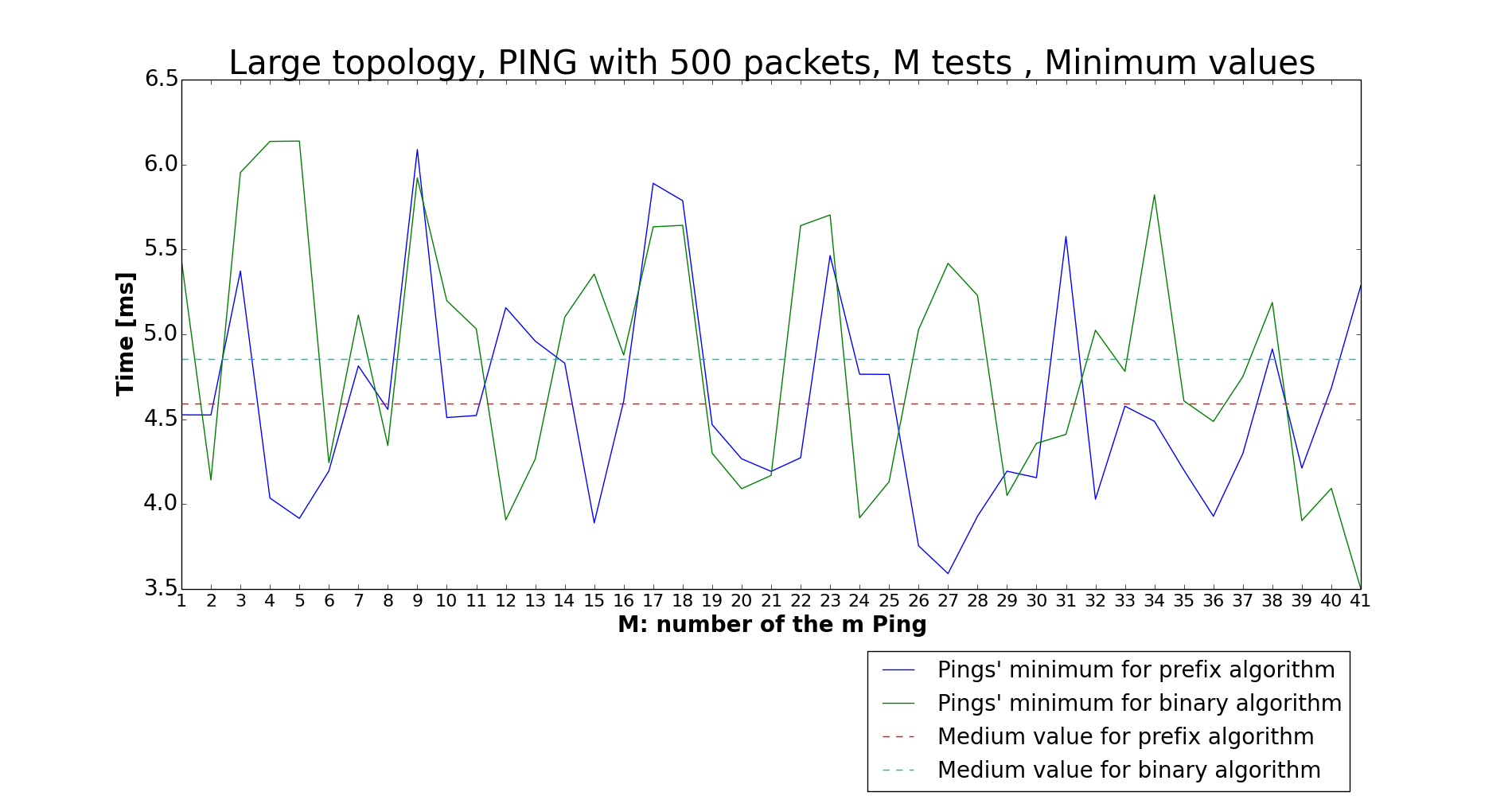
In the Prefix algorithm, the 3°, the 16° and the 19° algorithm show the “bug” of the algorithm.

LOOKUP TIME:

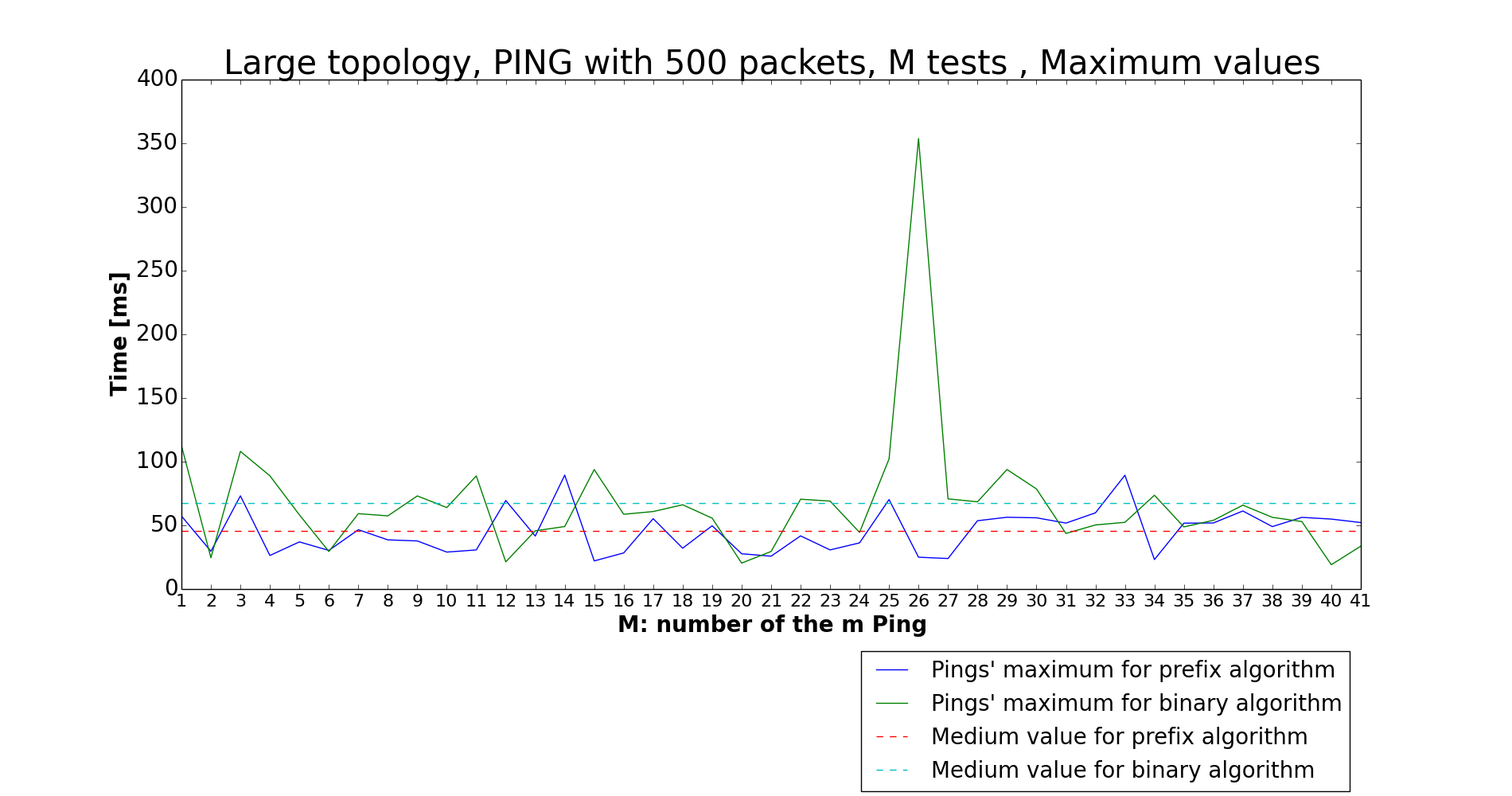


**TOPOLOGY LARGE**

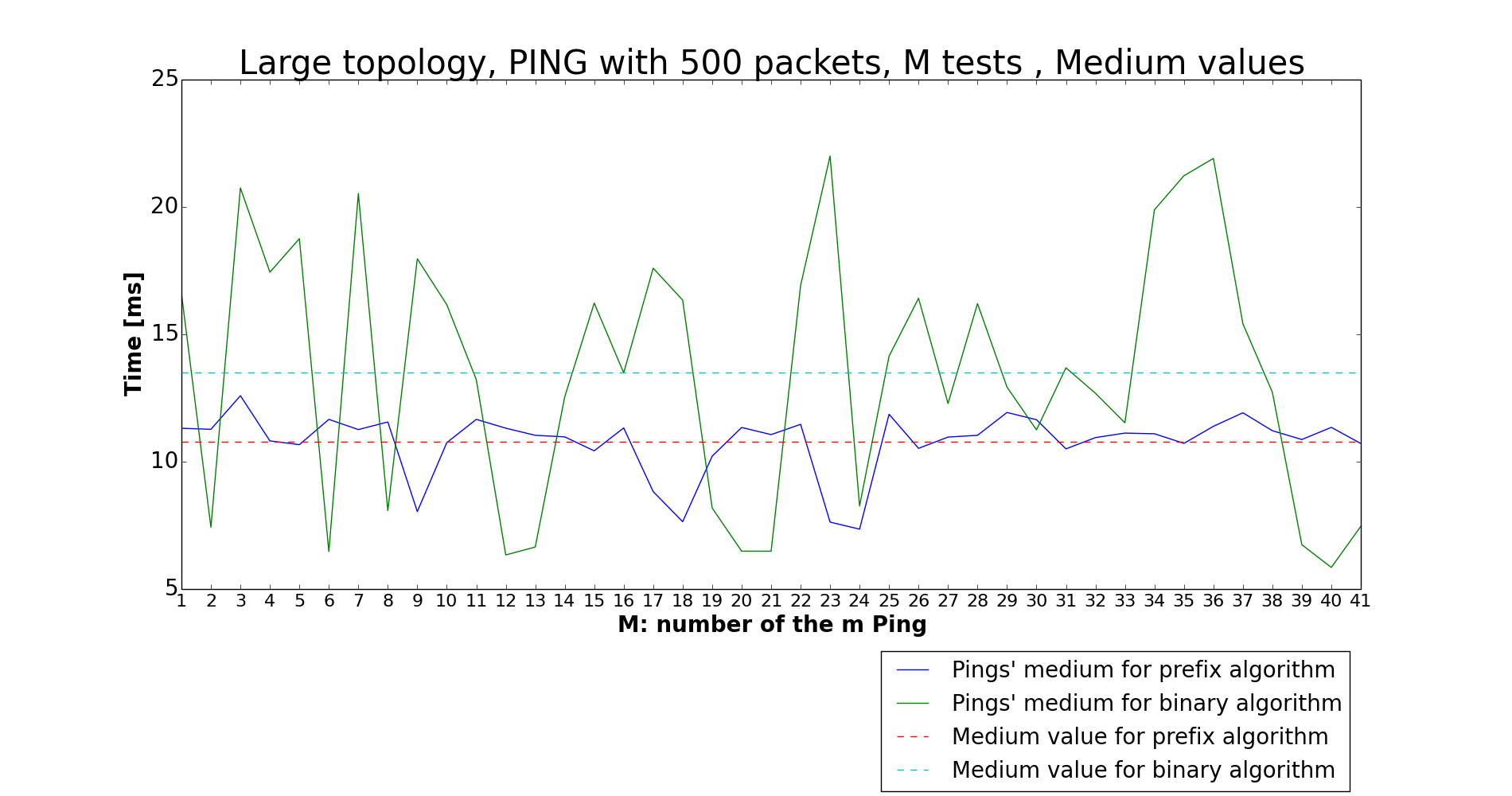
MINIMUM VALUES:



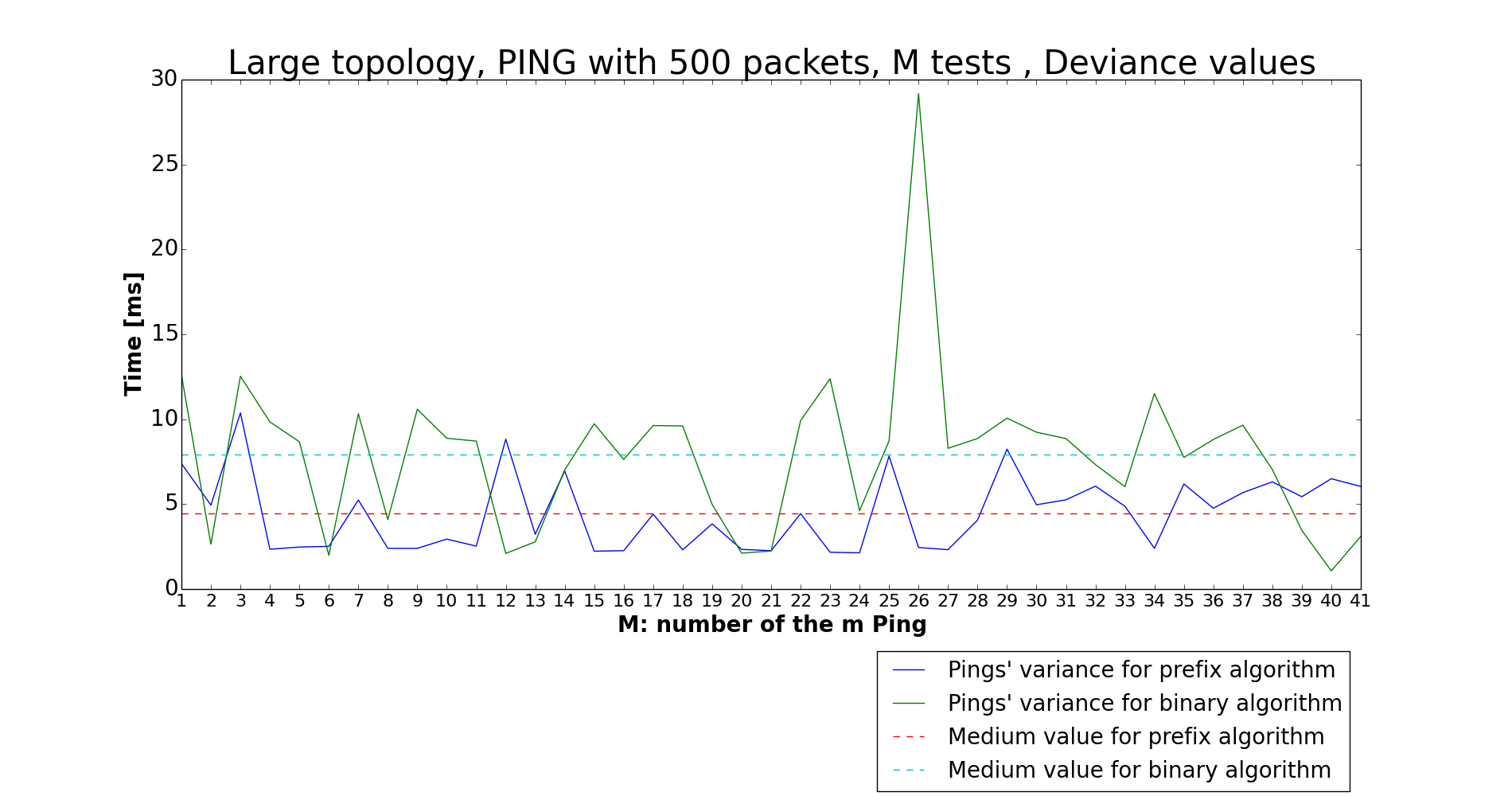
MAXIMUM VALUES:



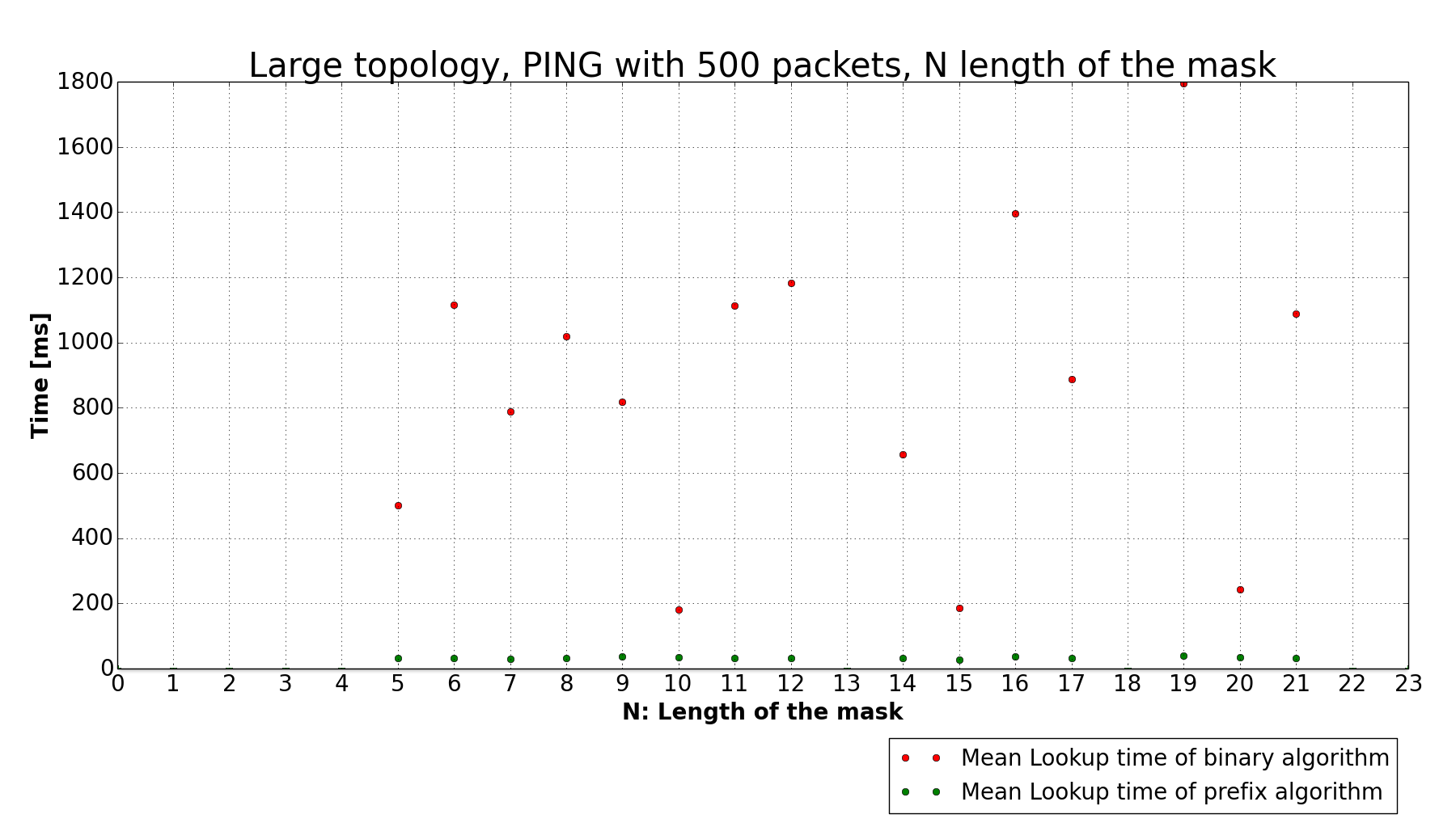
AVERAGE VALUES:



RMSD:



LOOKUP TIME:



1. Conclusion

This project had the aim of proving better performance of binary search on prefix algorithm than the binary trie algorithm. In every simulations we can see in fact all the better performance obtained by the prefix algorithm than that obtained by the binary algorithm. This is clear in theory and it is proved by our different simulations: obviously in theory the performance should be a little better than ours simulated, but our simulations are however a good expression of what we wanted to find.

1. Bibliography

* https://www.opennetworking.org/sdn-resources/sdn-definition
* Slides of “Switching and Routing” course of Professor Maier
* Slides on Python provided by Sebastian Troia
* *”High performance switches and routers”*, H. Jonathan Chao, Bin Liu