

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).

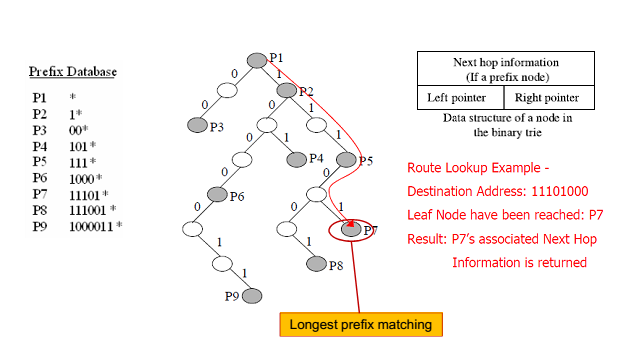
The Openflow protocol is a fundamental element to build SDN solutions.

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:

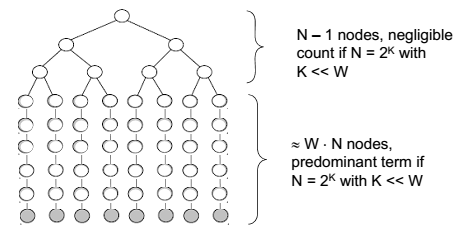
* **Directly programmable**: Network control is directly programmable because it is decoupled from forwarding functions.
* **Agile**: SDN helps organizations rapidly provide new applications, services, and infrastructure to quickly meet changing business goals and objectives.
* **Centrally managed**: 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\*:

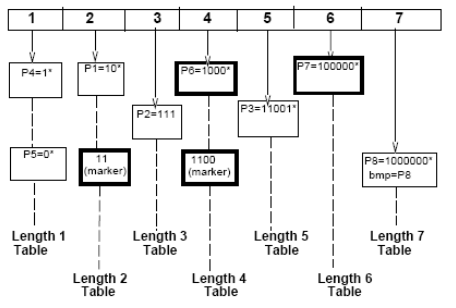
Performance of this algorithm are calculated in terms of:

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

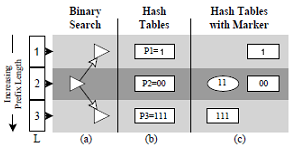
1. **BINARY SEARCH ON PREFIX LENGTHS ALGORITHM**

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.

In order 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.

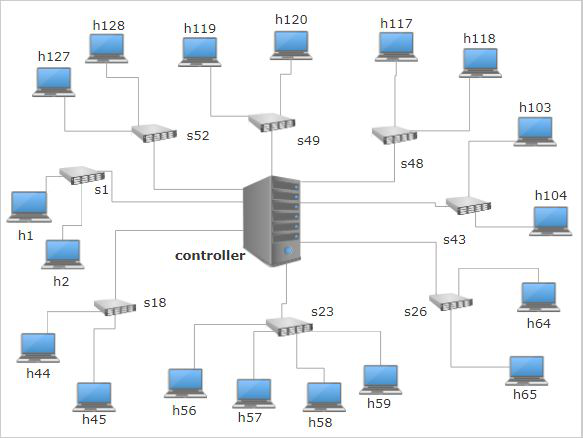
1. **Implementation** 
   1. Network simulation

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.

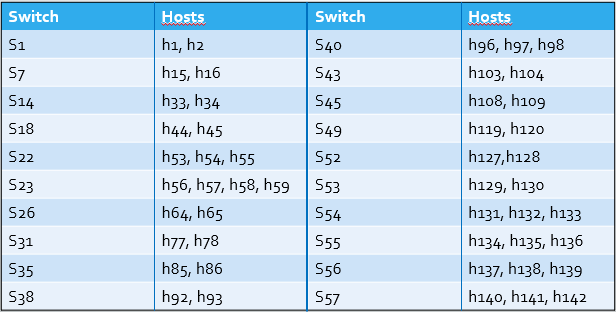
In order to show our simulation, we created three different topologies, one smaller with 8 different swiches and 18 hosts, one medium with 20 switches and 48 hosts and one larger with 35 switches ad 85 hosts. The aim of this procedure is to take over the differences in terms of:

* + - Minimum;
    - Maximum;
    - Average value;
    - RMSD (root mean square deviation);
    - Lookup time.

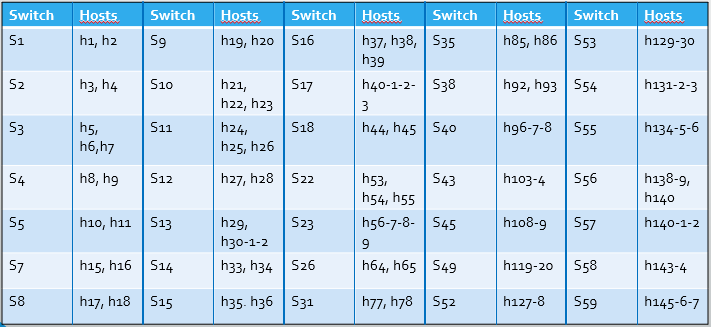
Here is the first topology, the smaller one, with 8 switches and 18 hosts:



Here is the medium one, with 20 switches and 48 hosts:



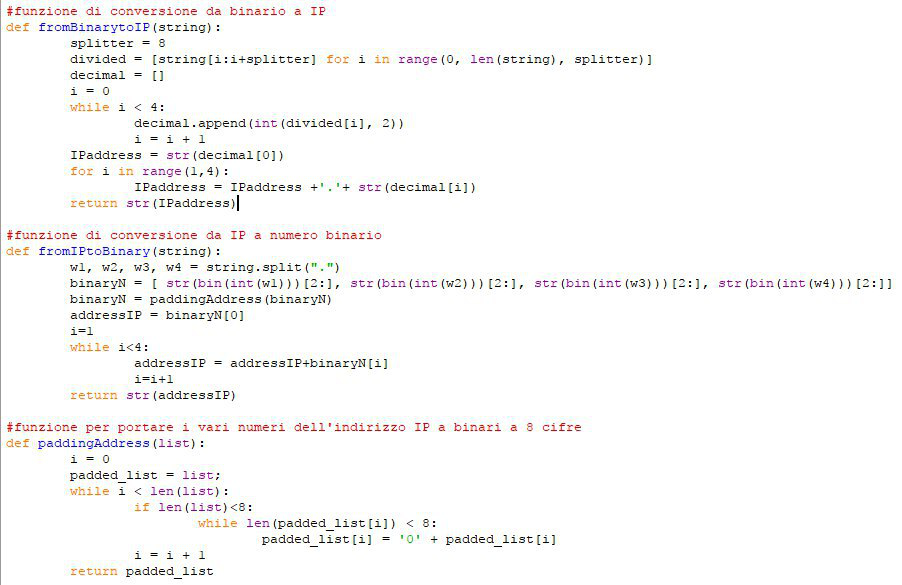
And this is the largest one, with 35 switches and 85 hosts:



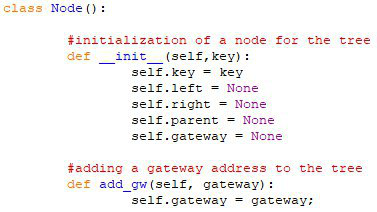
* 1. Algorithm implementation

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

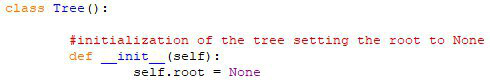
Thanks to padding address every transformed address become 8bit/8bit/8bit/8bit.



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



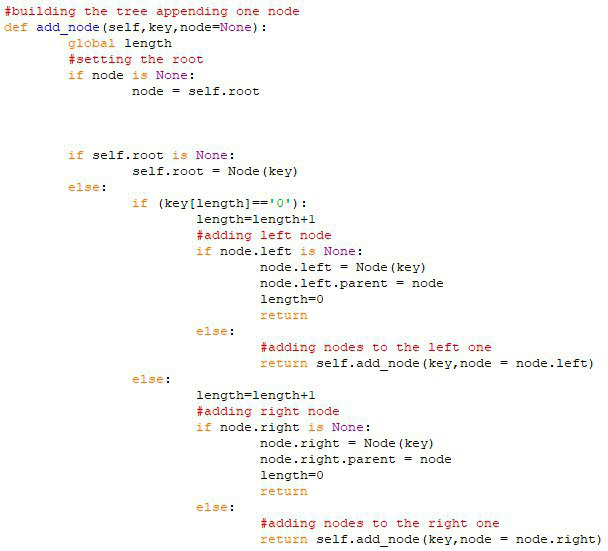
**CLASS TREE**



**FUNCTION ADDING NODES TO THE TREE**

In order to implement the binary trie algorithm, we need to build a binary trie.

The function to implement this starts appending a node to the root, then, after the first node is fixed, every single node is attached iteratively: if the value of the first bit of the prefix is 0 the node is attached to the left branch and the root (or the node analyzed) became the father; if the first bit is 1 the node is attached to the right branch.



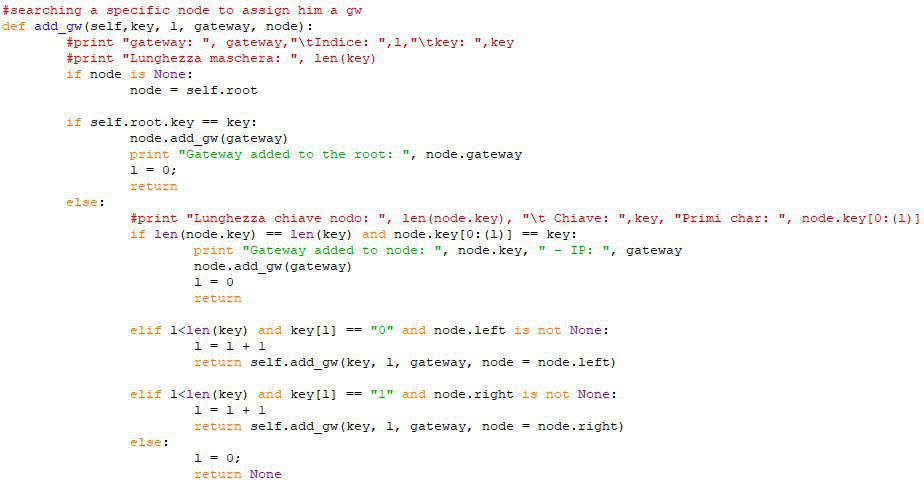
**FUNCTION ASSIGNING A GATEWAY TO A SPECIFIC NODE**

In order to find the longest prefix match for each address (during the algorithm), we must assign to

some nodes an address corresponding to a gateway. This is what the next function does: it takes as

entries the prefix of the address that must become the gateway and the root of the tree. Iteratively the

function goes through the tree until it reaches the right node and assigns the gateway address to it.



**BINARY ALGORITHM**

The execution of the algorithm for the lookup is implemented in the function below.

This function acts iterating the tree until the longest prefix match for each address received as input

parameter.

It starts converting the address passed as a string to a binary address, the it verifies if the root is the

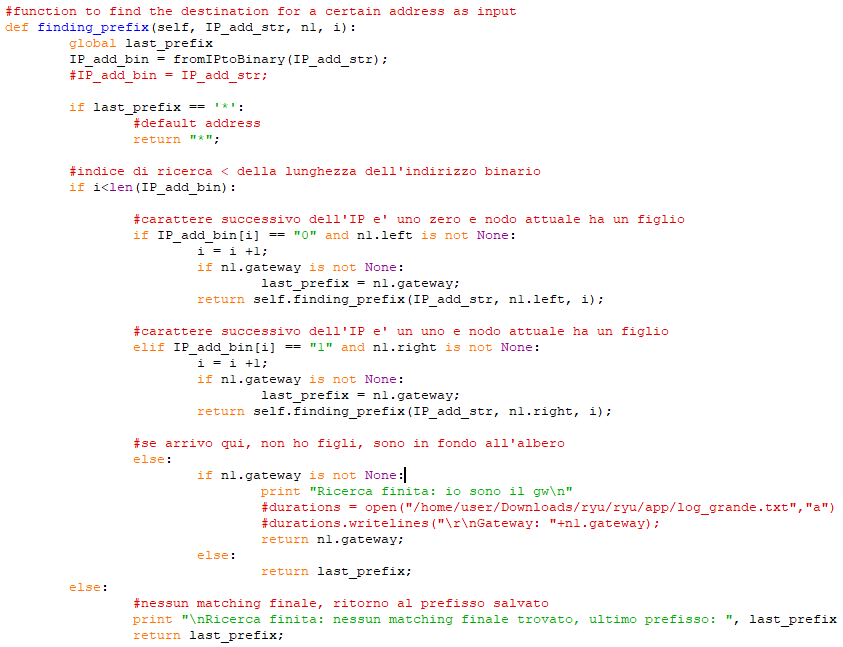
gateway and it goes through the whole tree checking matching conditions between the address of

destination received and the address of a specific path along the tree, depending on matching

conditions.

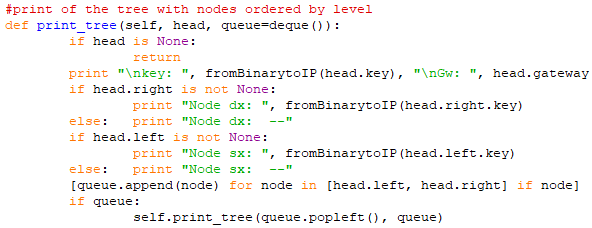
The function assign to a global variable called **last\_prefix** the address of the gateway found at the node

that corresponds to the longest match.



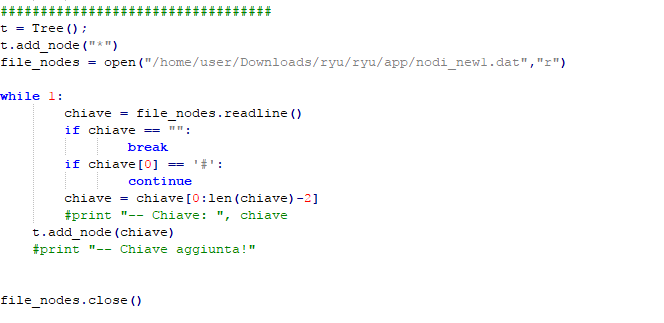
In the code there is also a function created to print the tree but actually not used in the execution but

usefull to verify the right construction of the tree.



In order to keep thin as much as possible the code and to provide an easier lecture of it, we have

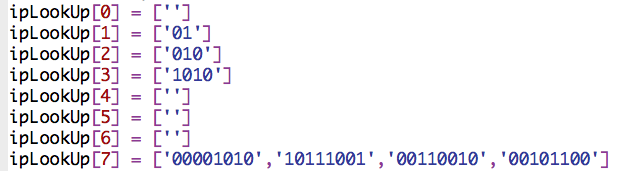
inserted a cycle that import from a file the nodes of the tree in the construction phase.



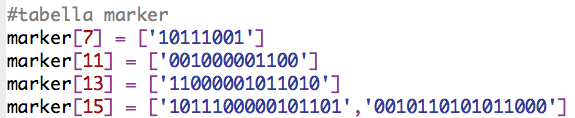
**BINARY SEARCH ON PREFIX LENGTHS ALGORITHM**

The execution of the algorithm for the lookup is implemented in the function below.

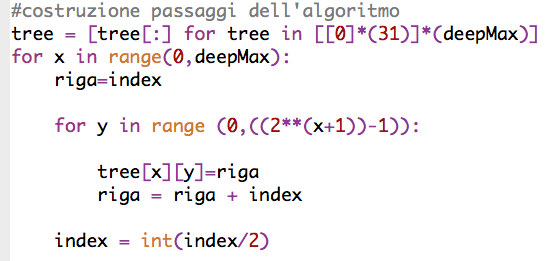
Prefixes are divided by their length and they are stored in the ipLookup[][] matrix.



Also markers are stored in the marker[][] matrix:



Then the *tree* table for the execution of the algorithm is prepared



This table permit in *deepMax* passages to explore all possible cases in the prefix length algorithm.

An example of *tree* with *deepMax* = 3 that corresponds to prefix with length equal to 7:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 |  |  |  |  |  |  |
| 2 | 4 | 6 |  |  |  |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

In this way the algorithm starts to control the *ipLookup* matrix with row index equal to 4 [4-1].

Then if the prefix is in that row (e.g. i) the next index will be the one in the next row (e.g. i+1) of *tree* and the next column (e.g. j+1) respect to the previous index. In this example, if the prefix is found the next prefix will be 6, while if the prefix isn’t found the next prefix will be 2.

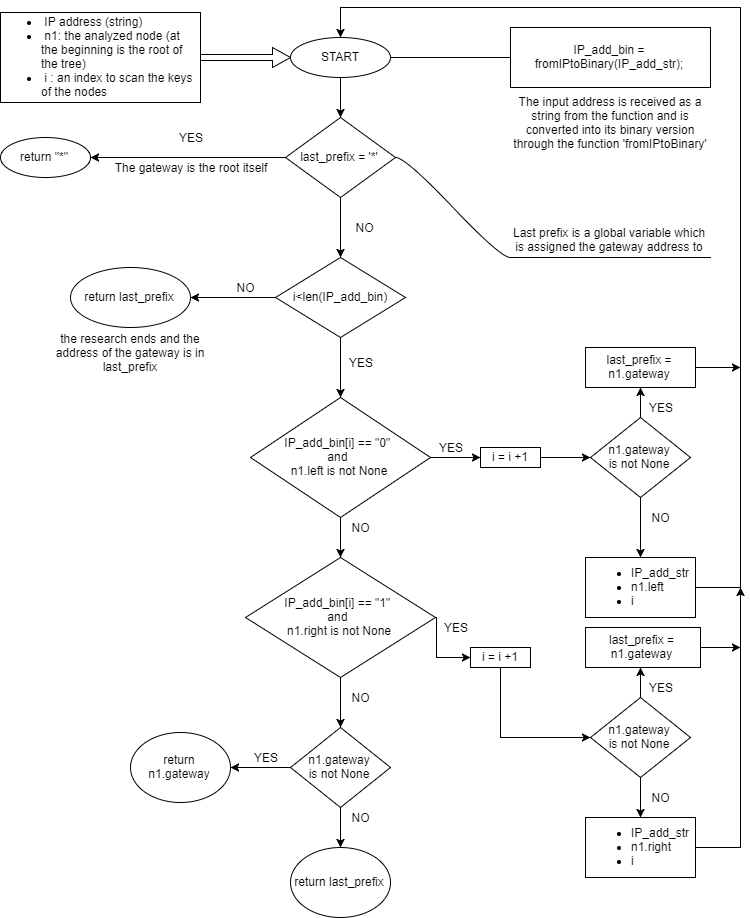
This procedure is in this function: 

The algorithm ends when the last row of the *tree* matrix is explored. In this way the algorithm has a finite number of passages.

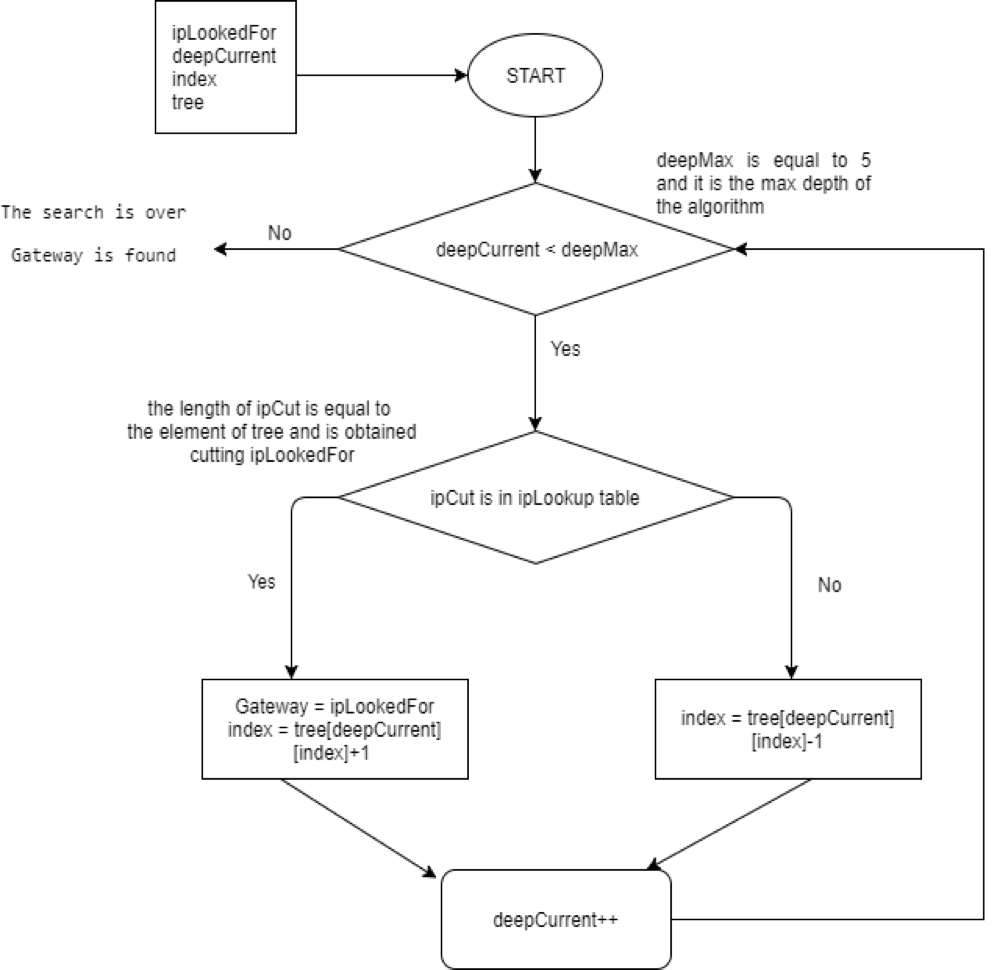
**N.**B. When the prefix is compared in ipLookup, the length of the prefix is cut in order to have the same length of the IP in the ipLookup table.

6.3 Flowcharts

**BINARY TRIE FLOWCHART**



**BINARY SEARCH ON PREFIX LENGTHS FLOWCHART**

****

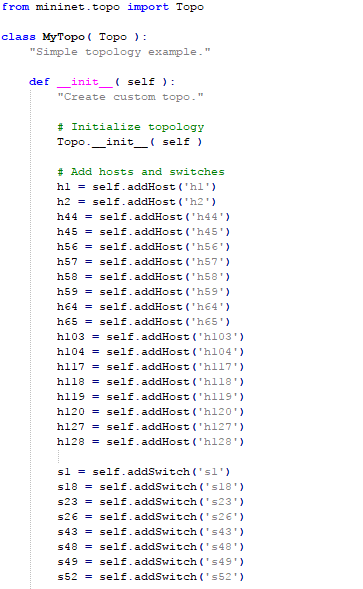
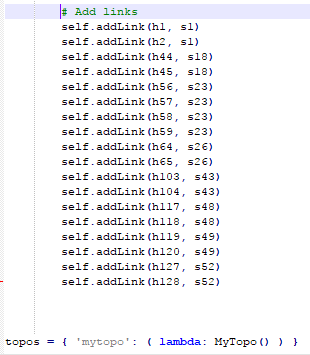
6.4 Support files

In addition to the file containing the algorithms and the communication mechanism in the network, it’s

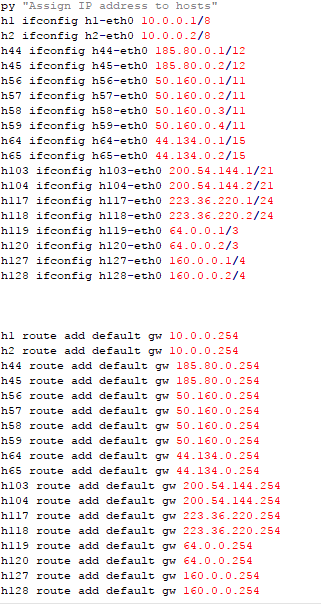
needed to modify some files concerning the different topologies and the configurations of the links.

For example for the smallest topology those two files look like this:

* Topology file



* Configuration file



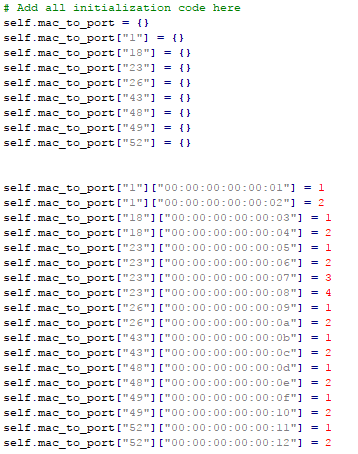
6.5 Configuration part (lookup file)

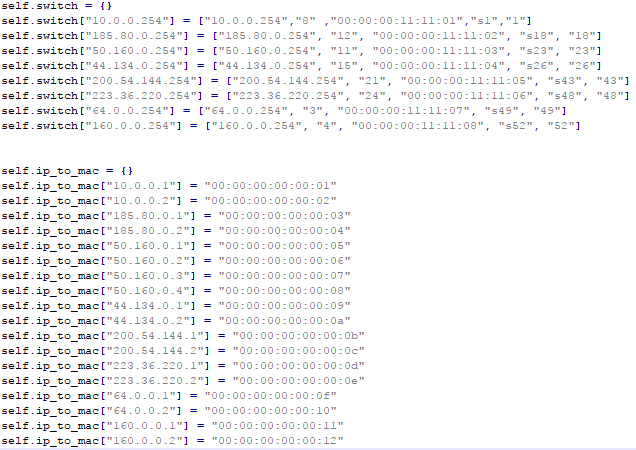
In the file containing all the mechanisms of the network there is an initial part where there are

some importante tables:

* mac\_to\_port: which assigns to every mac address of the hosts the port on the switch correspondent;
* switch: which defines for each switch the IP address, the length of the netmask, the MAC address, and the name.
* ip\_to\_mac: that assigns to every IP address of an host the relative MAC address.

The examples listed below are referred to the smallest topology.





1. **How to run the code**

7.1 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.

Before starting to execute the commands all the files must be in this folder of the virtual machine: **Downloads/ryu/ryu/app**.

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

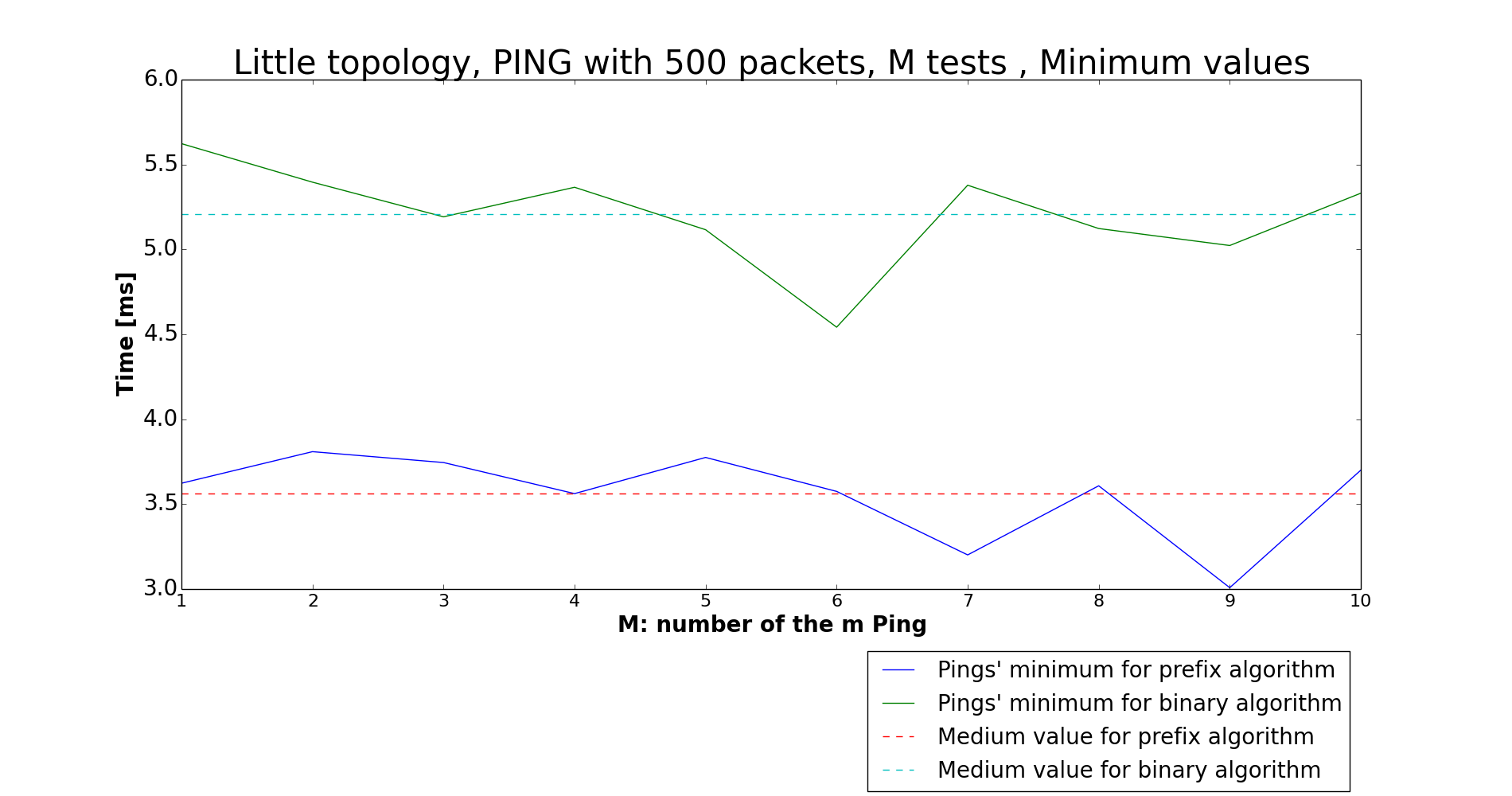
* Open a terminal to run mininet with these commands:
  + cd Downloads/ryu/ryu/app
  + sudo mn --custom *topology\_xxx.py* --topo mytopo --mac --controller remote -- pre config\_xxx.py
* Open another terminal to run the controller with these commands:
  + cd Downloads/ryu
  + sudo PYTHONPATH=. ./bin/ryu-manager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

1. **Results**

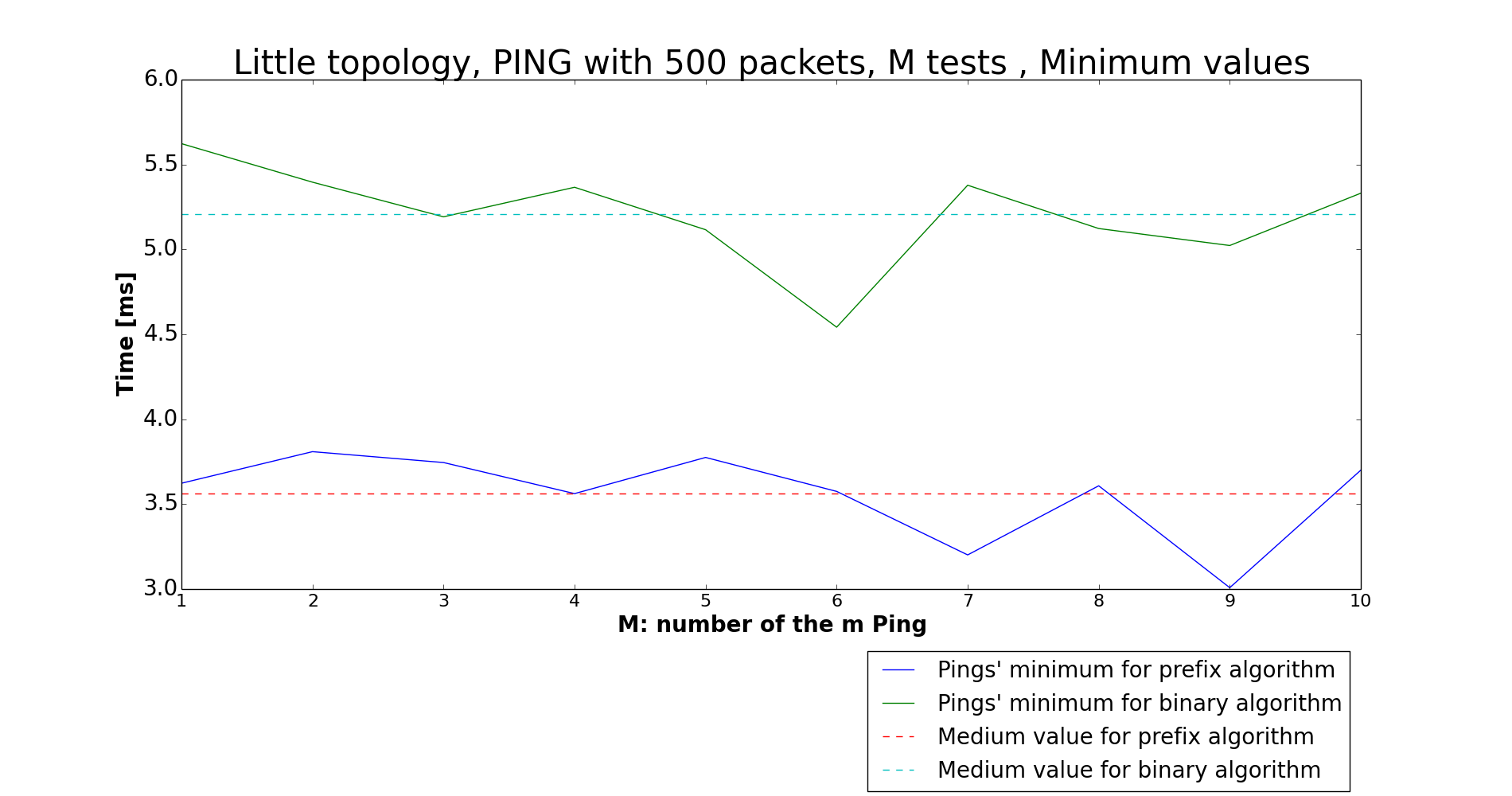
Results are shown divided by topology.

**LITTLE TOPOLOGY**

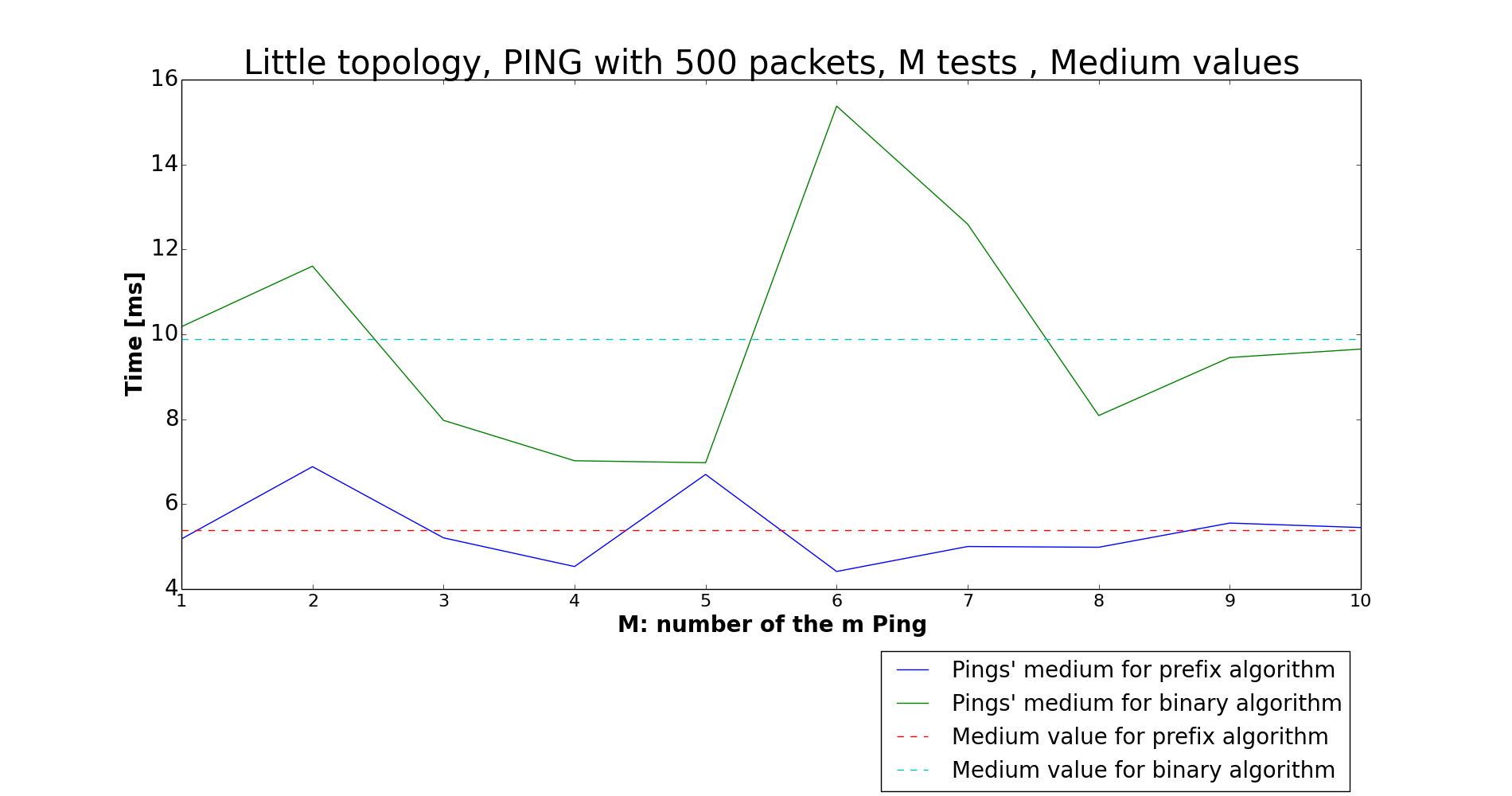
* MINIMUM VALUES:



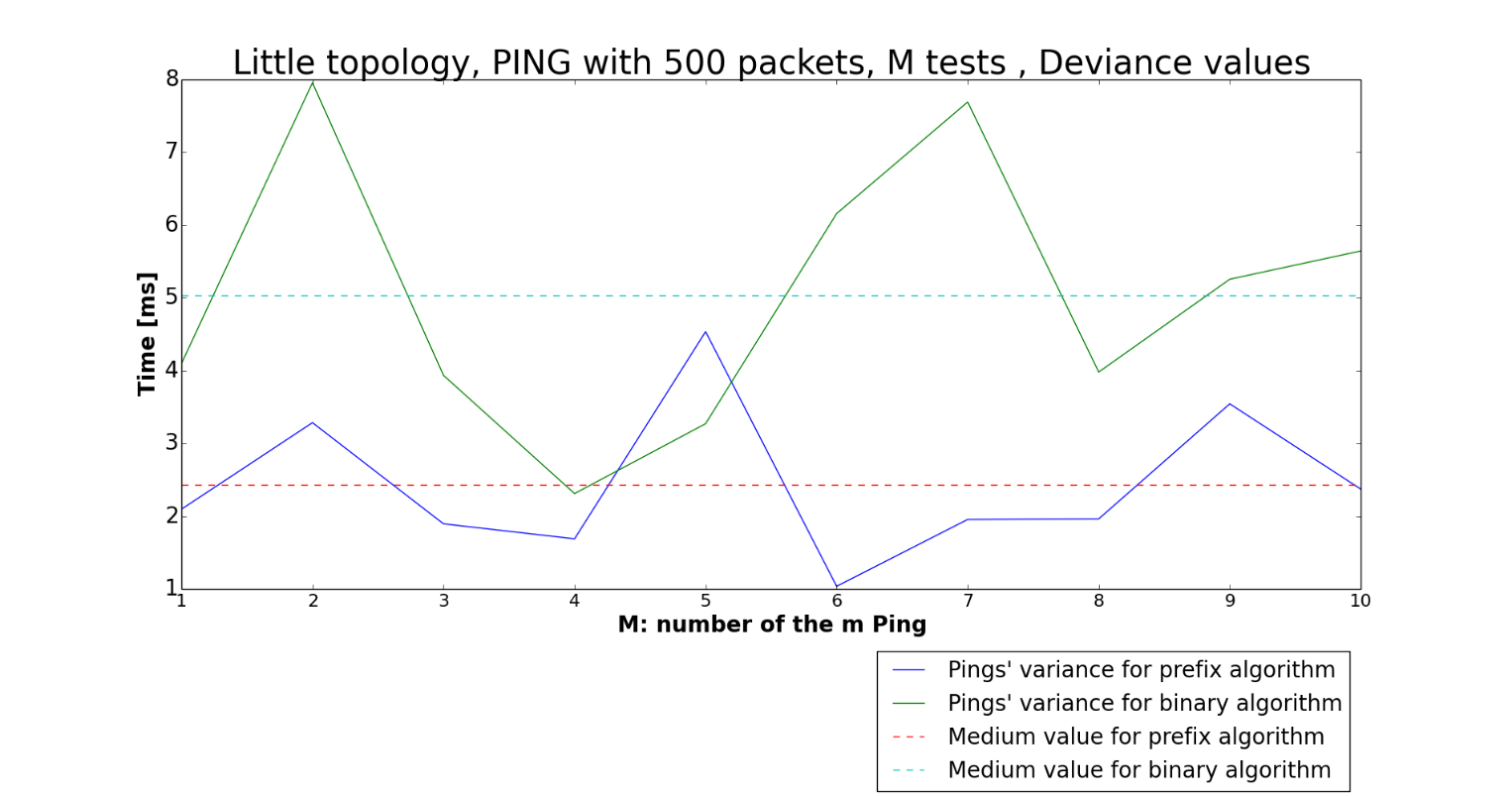
* MAXIMUM VALUES:



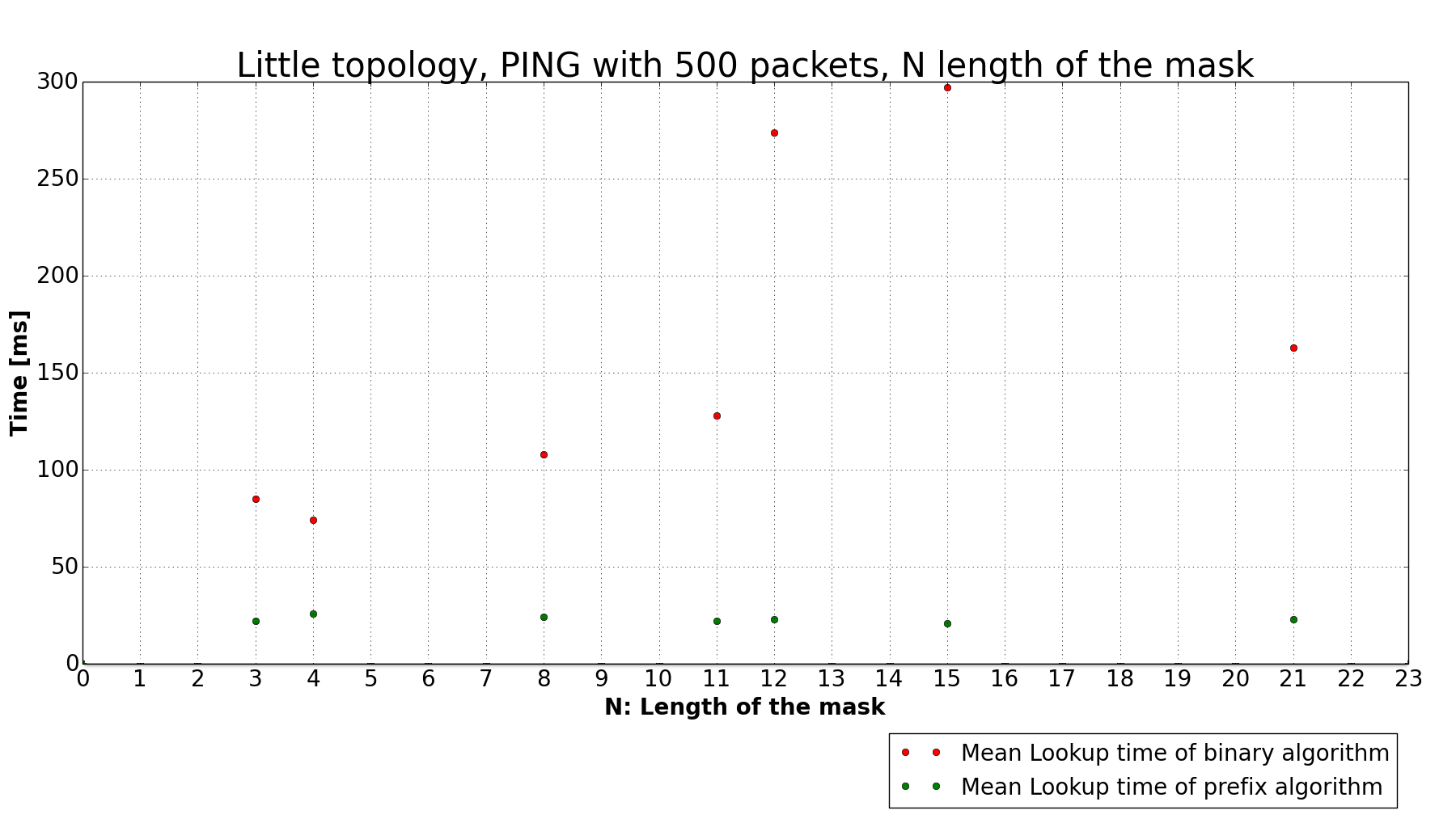
* AVERAGE VALUES:



* RMSD:

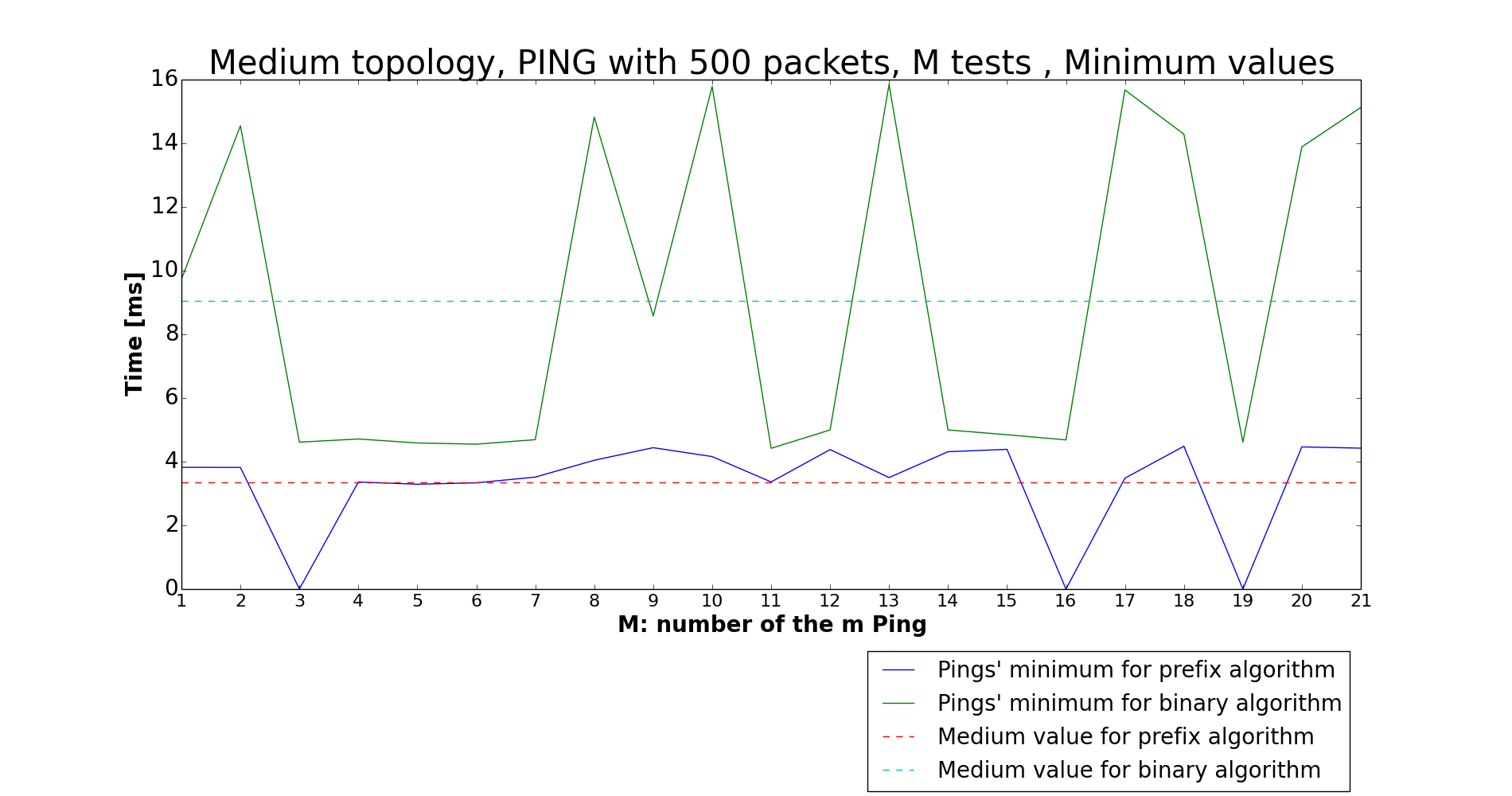


* **LOOKUP TIME:**

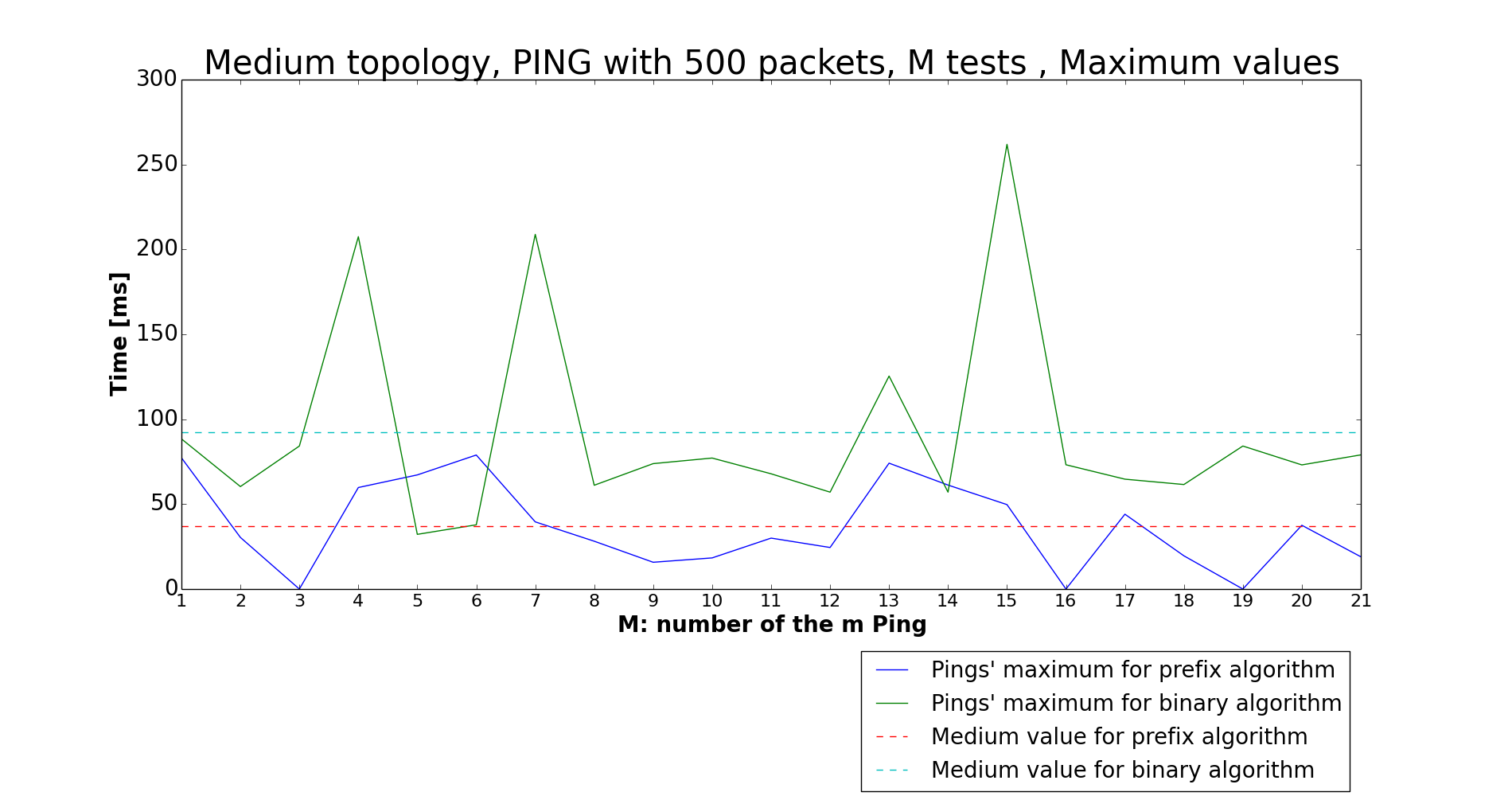


**MEDIUM TOPOLOGY**

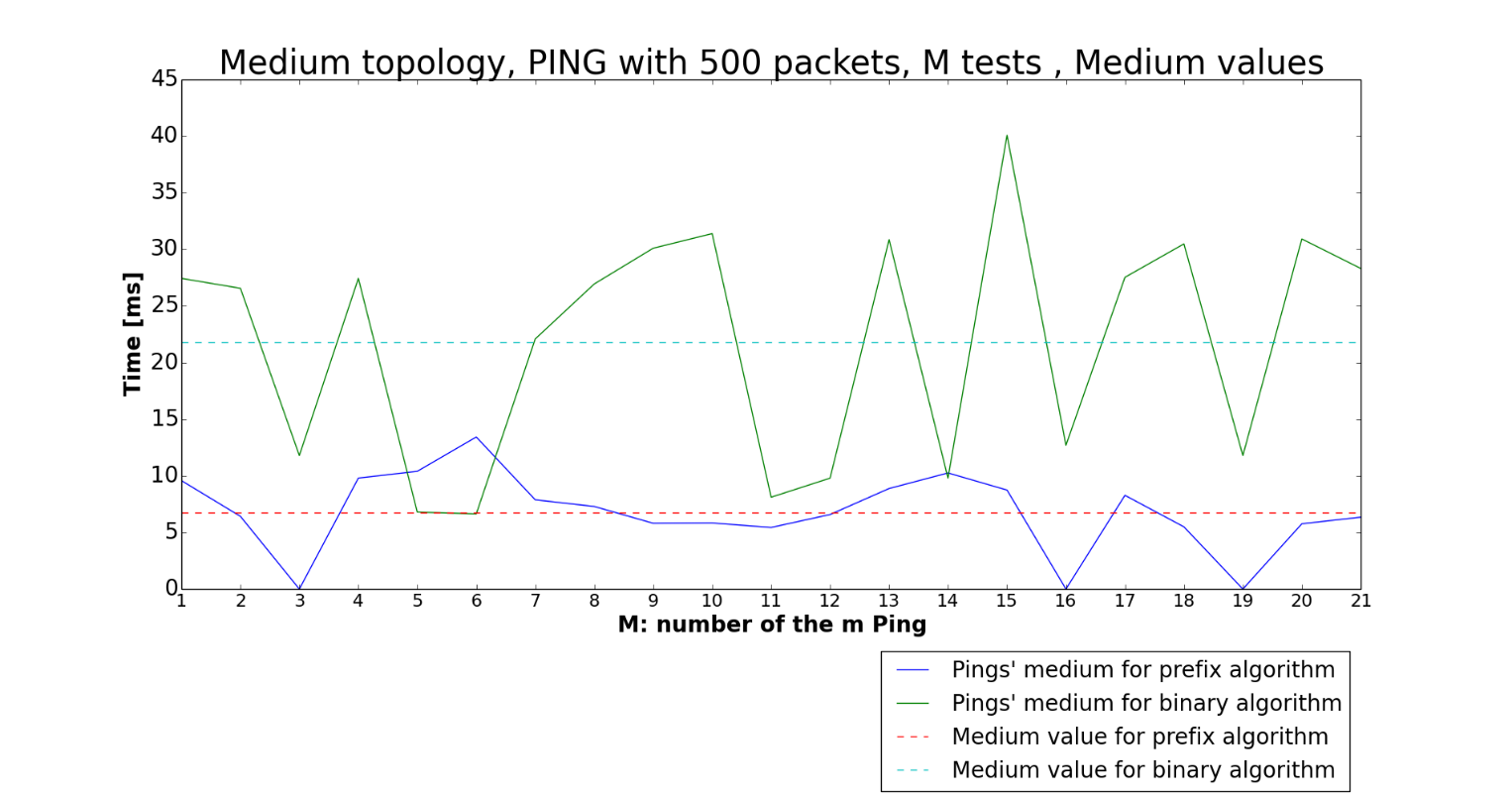
* MINUMUM VALUES:



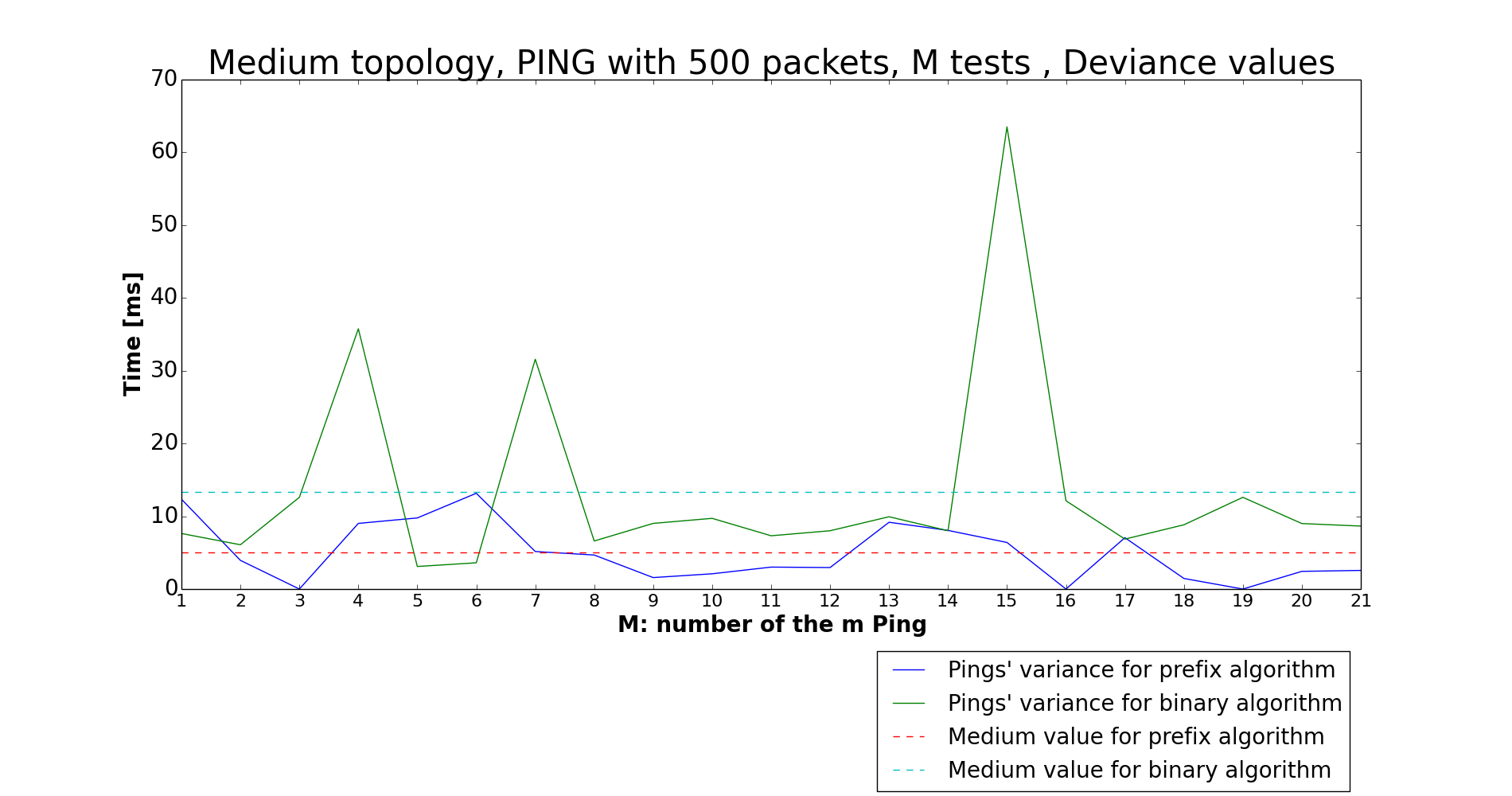
* MAXIMUM VALUES:



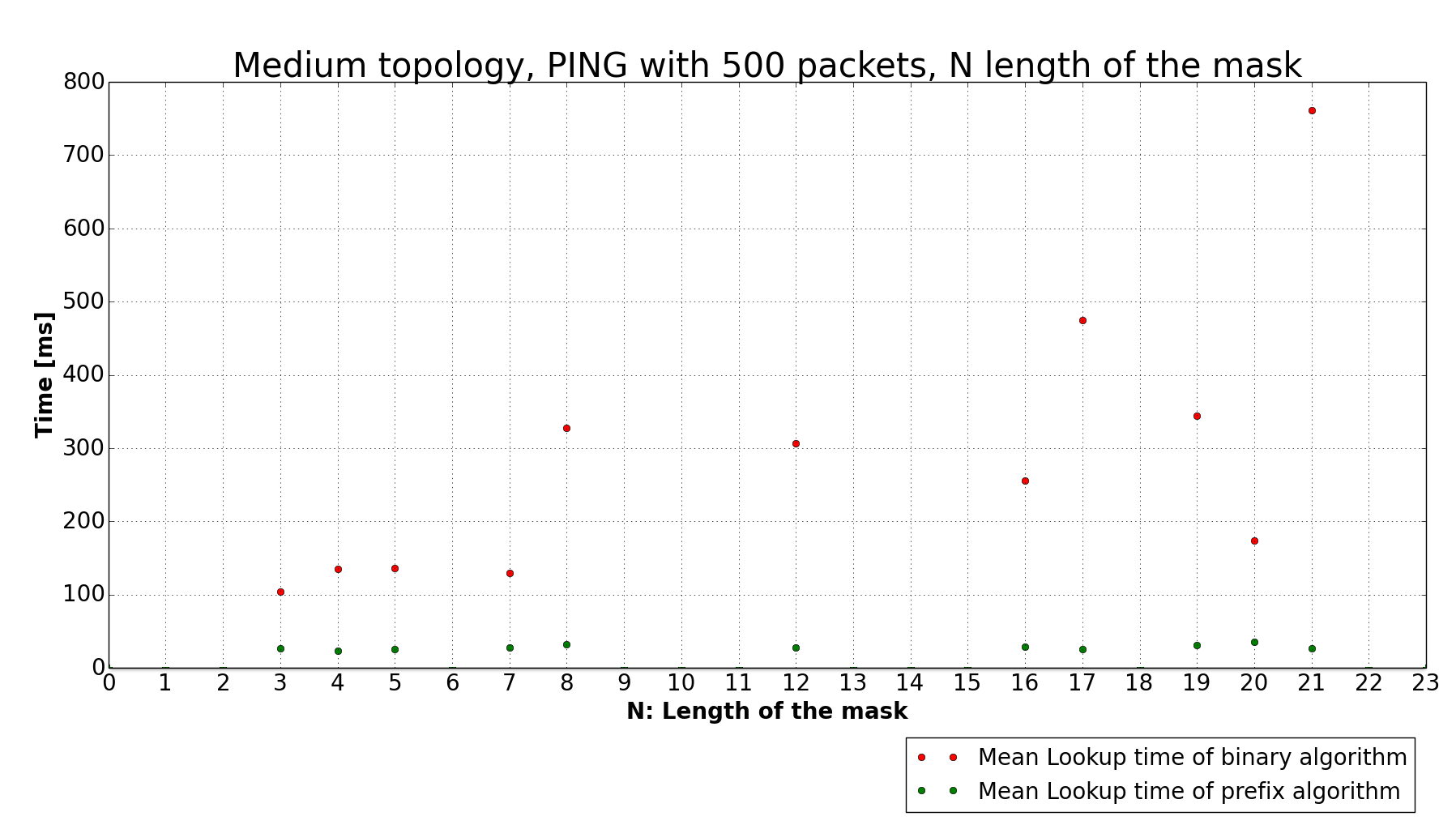
* AVERAGE VALUES: masks number 3, 16 and 19 show the ‘bug’ of the binary search on prefix length algorithm.



* RMSD:

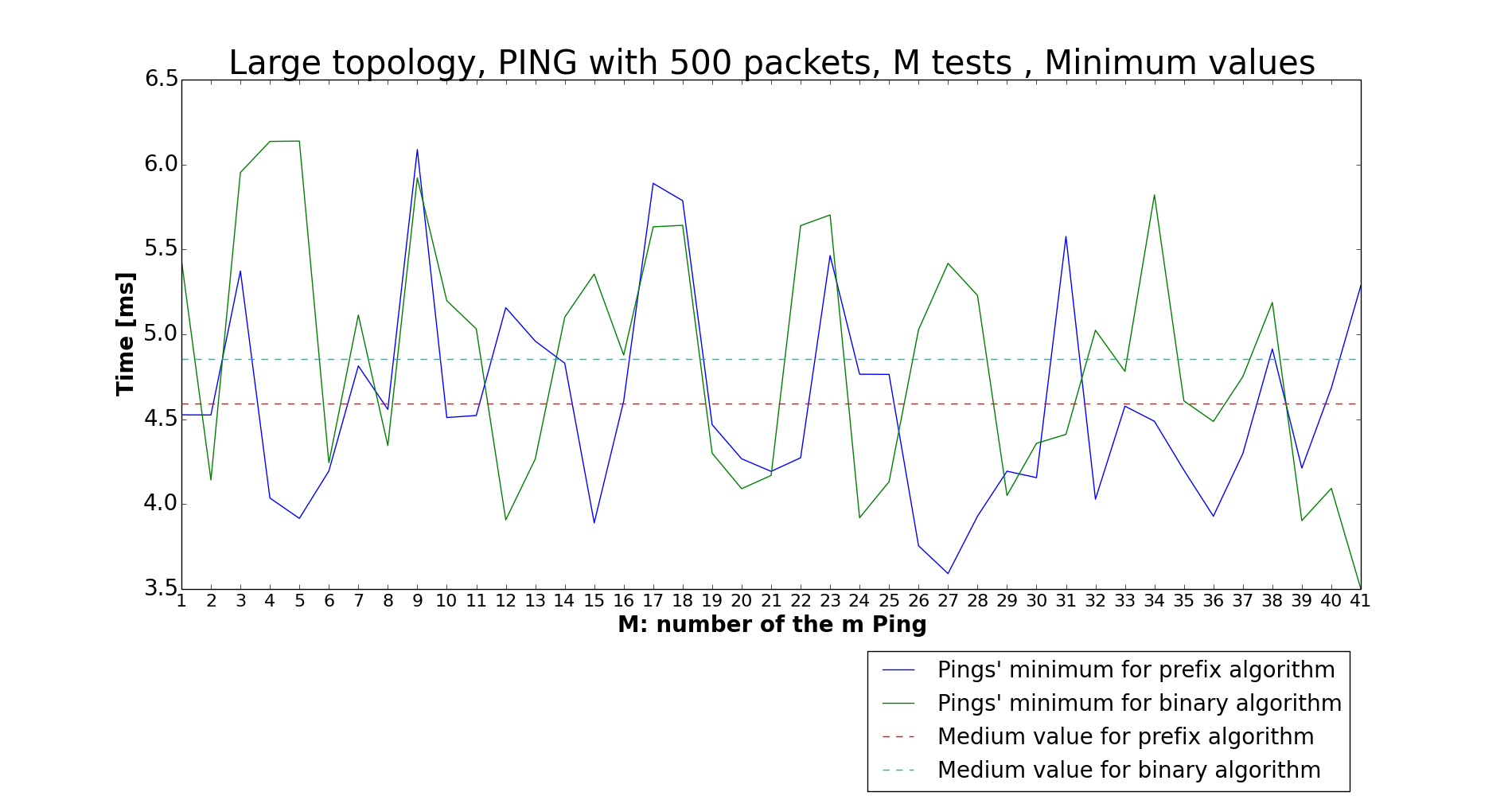


* **LOOKUP TIME:**

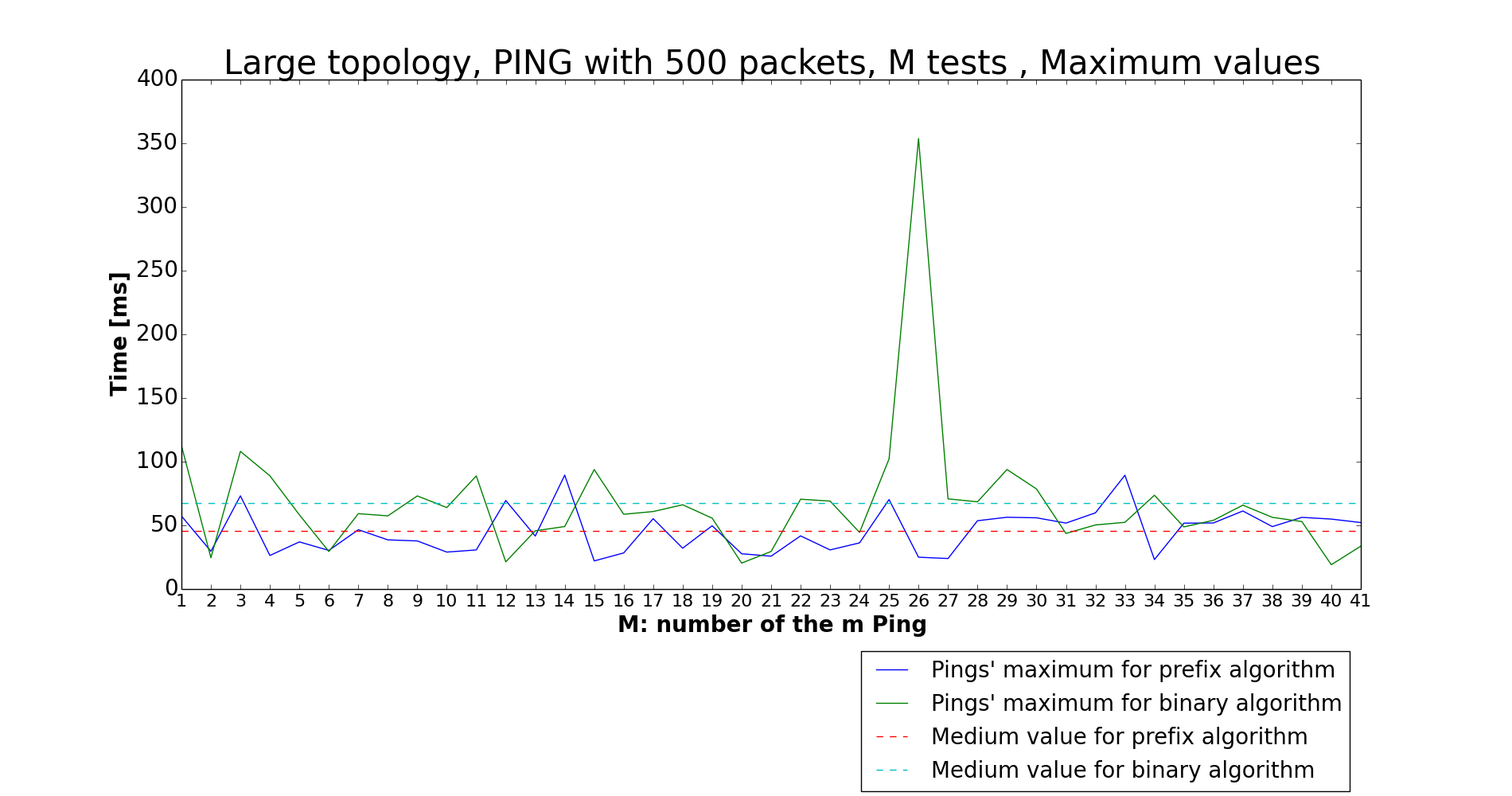


**LARGE TOPOLOGY**

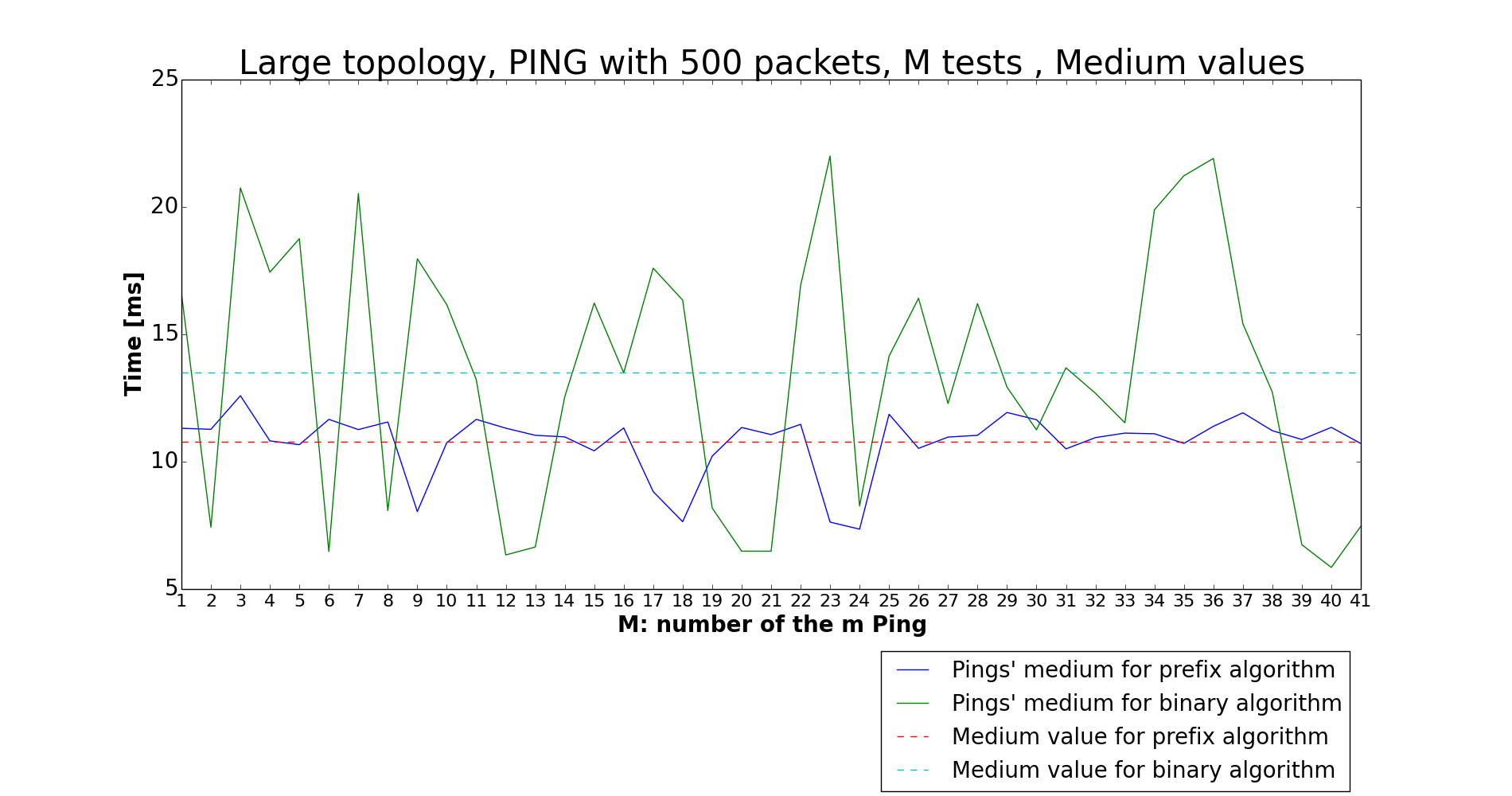
* MINIMUM VALUES:



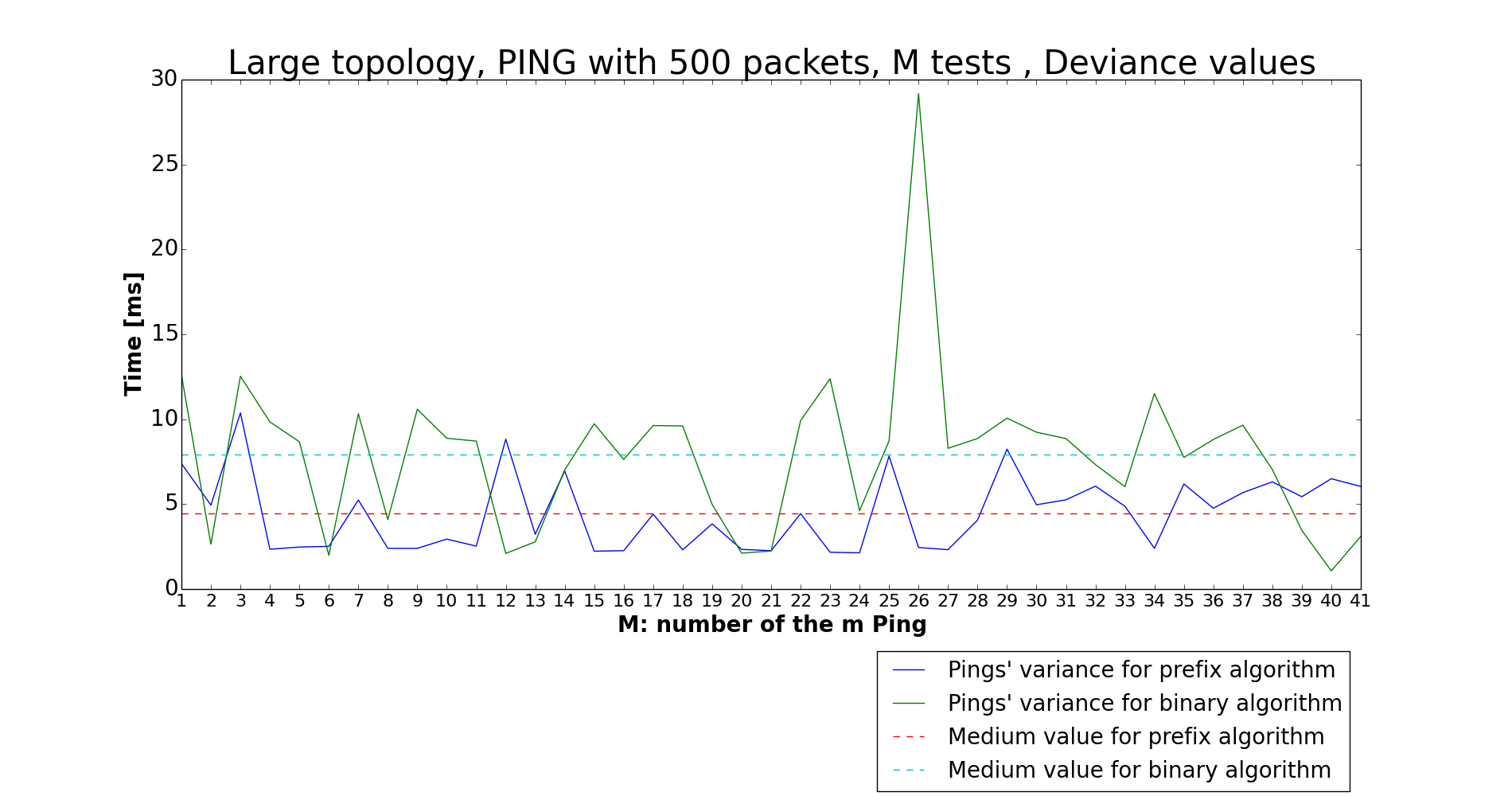
* MAXIMUM VALUES:



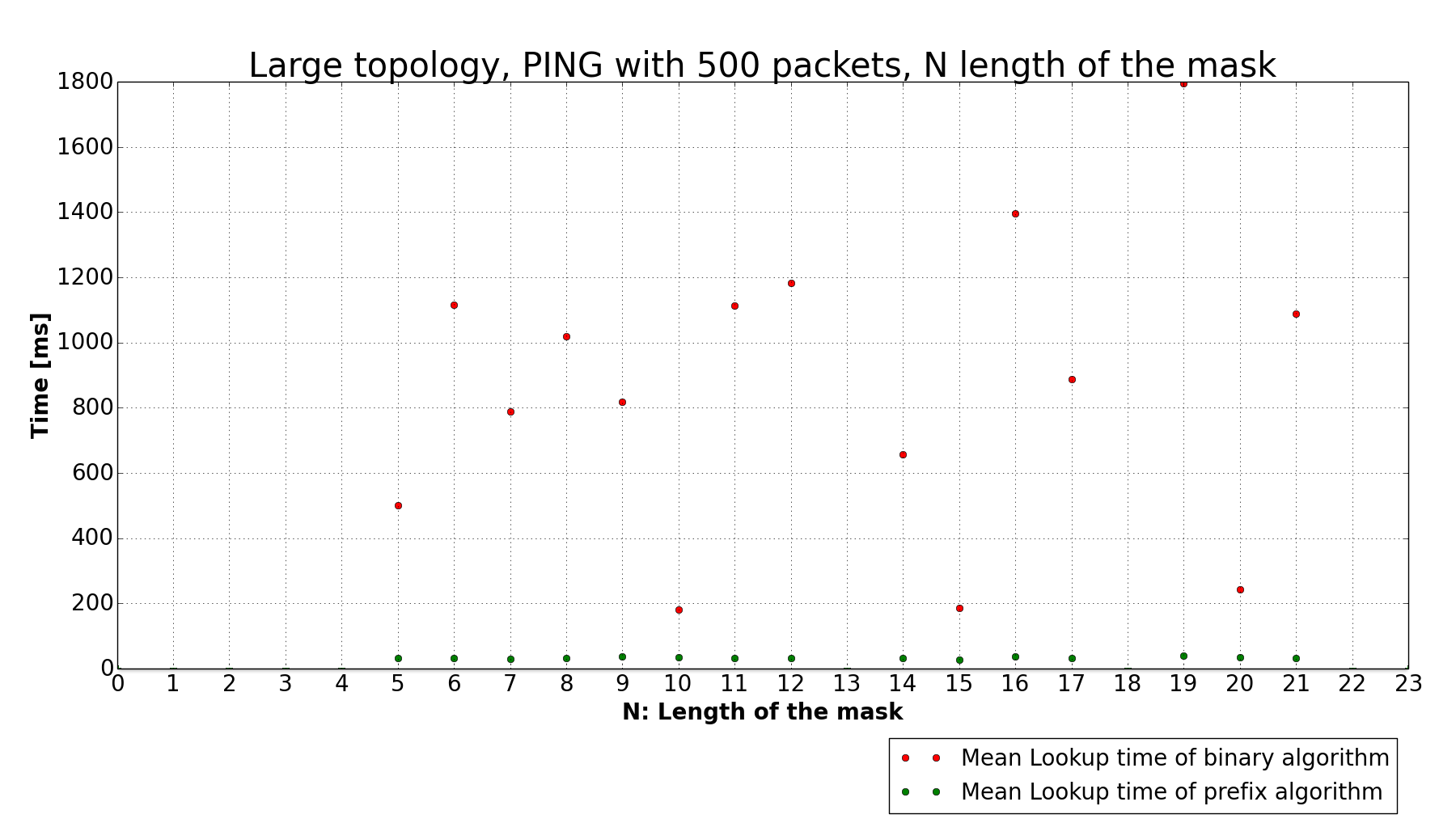
* AVERAGE VALUES:



* 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 simulation 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