INTERNSHIP REPORT
Using information theory in network tomography[1]

Rémi Varloot

June 8, 2011 — August 3, 2011

[1]My thanks go to Darryl Veitch, who supervised this project, as well as to François Baccelli for suggesting the topic of the project and putting me in touch with Darryl.

This internship sought to test a new method, based on information theory, for determining the topology of tree-shaped networks by analysing end-to-end packet delays and losses.

Contents

1 Introduction **2**

2 Background **2**
2.1 The network 2
2.2 Packet transmission 2
2.3 Nodes and packet interaction 3

3 A Simple Traffic Model **3**

4 Inferring the Network's Shape **4**
4.1 Computing nearest compression distance 4
4.1.1 Formatting data 5
4.1.2 A generic data compressor 5
4.2 Inferring the network's shape 5
4.2.1 Computing similarity between leaves 6
4.2.2 Regrouping multiple leaves together 6
4.2.3 Allocating time series to new merged nodes 7

5 Analysing Performance **7**
5.1 Means of analysis 7
5.1.1 The probabilist approach 8
5.1.2 Implementation of both methods 8
5.1.3 A new tree distance : the Mean Path Error 8
5.2 Convergence speed 9
5.3 Influence of the network's size 9
5.4 Finding the right α 10
5.5 What about computation time ? 10

6 Testing with Another Model **11**
6.1 The simulator 11
6.1.1 A library providing many random processes 11
6.1.2 A structure representing packet sources 12
6.1.3 A structure representing network nodes 13
6.1.4 A random tree generator 13
6.1.5 A priority heap to store upcoming events 13
6.1.6 A library of methods for outputting data 13
6.1.7 A network configuration file 14
6.1.8 The main execution thread 14
6.2 Observations 14

7 Conclusions **15**

1 Introduction

The Internet today is neither designed nor controlled, to the point that no one really knows what it actually *looks like*. Though that apparently does not prevent it from working well, some knowledge and understanding of what we call the network's topology could nonetheless enhance its performance. For example, flow control algorithms such as TCP could be greatly enhanced by some understanding of the network's layout, and peer-to-peer networks could be improved by identifying bottleneck links.

Making progress in the study of network topologies often involves finding better clustering algorithms for recognising "neighbours". This research is based on the results of [4], which presents a clustering algorithm based on information theory. A generic distance is defined between any two sets of data of the same type. Computing this distance, called normalized compression distance (NCD), between a given number of data sets can then be used to generate a ternary undirected tree in which similar sets of data are grouped close together.

The aim of this research was to use this algorithm on time and/or loss series given by packet end-to-end delays/losses in a tree-shaped network. We furthermore went on to improve the clustering algorithm to better fit our needs, using a simplified model to compare our results to those given by a reference probabilistic method.

2 Background

In order to appropriately study network topology, a primary understanding of networks is necessary. What is a network ? How and why are measurements made ? How do nodes treat packets ?

2.1 The network

Networks are often represented as graphs, in which internal nodes represent routers and end nodes terminals that can operate as both senders and receivers. These terminals exchange packets over the network by means of a path between the two. It is important to note that in most cases, the network is *not* a tree, and more than one path can be found between two terminals.

Now suppose we consider only packets sent by one of the terminals, which we call the *root*, to some of the other terminals, called *leaves*. The paths actually used by these packets are most likely to form a tree-shaped sub-graph. Furthermore, we can make the hypothesis that, throughout our measurements, the network's shape will not evolve, and that packets sent to a given leaf will always take the same path. This justifies our choice of tree-shaped models that, though they will not yield the network's full topology, should be enough to reveal the proximity between certain terminals, especially if this method is reiterated with another choice for the root terminal.

2.2 Packet transmission

When transmitting a packet to a receiving terminal, an emitting terminal produces a single packet that is re-routed to another edge at each node until it reaches its destination.

An interesting question is : what if the emitting terminal wants to send a packet to multiple receivers ? Intuitively, the emitter could send one packet to each of them. Some of these packets will then start off along the same path, splitting off sooner or later in the process.

Multicasting is an alternate method based on the following idea : if we are to send multiple identical packets along a same edge, we might as well send only one. Packets are therefore sent

once along each appropriate edge, and multiple copies are re-routed at each node, so that in the end one packet arrives at each receiver.

With multicasting, it makes sense, when considering a packet heading for two destinations, to introduce the notions of shared and unshared links, and of a branching point. A packet lost along a shared link will indeed be lost for both receivers, whereas a packet lost for one receiver and not the other will certainly have found its way at least up to the branching point.

This point is most important in our study, the proximity of receivers being closely related to the depth of the branching point in the paths to these terminals.

2.3 Nodes and packet interaction

Packets do not instantly travel through the network. They take time to travel through edges, and are delayed at nodes as they are routed along the correct path. The more packets pass by a given edge or node, the more a packet passing there will be delayed, or even lost.

This means that, while we concentrate our study on a reduced part of the network, we must nonetheless take heed of the rest of the network, for it has a strong influence on the packet flows that we consider.

A first approach here could consist in supposing that, the traffic flowing in the sub-network being small compared to that in the real network, and the load of transverse flows being light, packet loads — and therefore losses — between different nodes or edges of the smaller network will be independent.

A second, more precise approach would consist in representing all packet flows in one of two forms : persistent flows consisting in packets that travel within the sub-graph, and transverse flows representing packets passing through only one of its nodes or edges.

This study primarily uses the first approach, though some simulations were also run that used the second.

3 A Simple Traffic Model

To begin, we implemented a simulator based on the first approach above, in which losses throughout the network were time and space independent. Independent loss probabilities were associated to each edge, and packets were sent by multicast.

This model was inspired by [7], in which the same network is studied through probabilistic methods. Furthermore, we know that the inferred tree being a good estimate, as shown in [2] and [6].

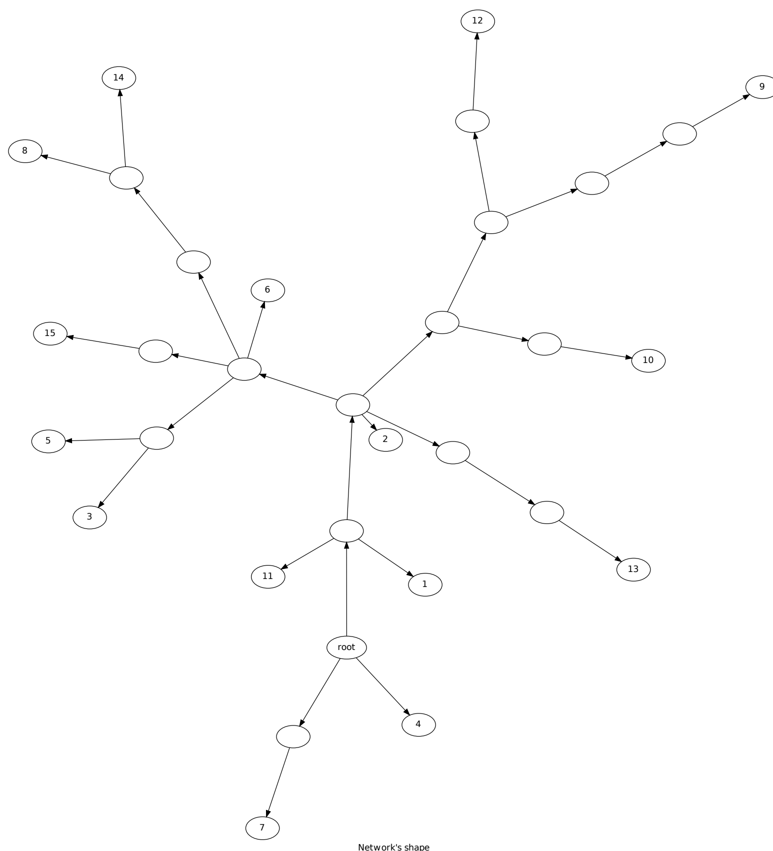
This simulator was very basic. It consisted in a random tree generator and a simulation loop.

The algorithm for generating the tree was as follows : The number of nodes, including leaves, is decided, and the nodes are numbered. Node 0 is designed as the root, and each following node's father is drawn randomly among the preceding nodes. This method gave somewhat irregular trees with a set number of nodes.

The simulation loop in itself works by computing losses at each edge : For each packet, check whether it has reached the different edges of a given depth, starting with the edges starting at the root, then re-iterate moving downwards in network. Record the losses for each leaf, then start again with the next packet. This method yields linear complexity in the tree's size for each packet.

Two things were recorded : a graph of the network (Figure 1) and loss sequences in separate files for each leaf.

Figure 1: Graph produced by the simulator, using [1]



4 Inferring the Network's Shape

Now that loss series could be freely generated, it was necessary to find a means of analysing this data. This is where the whole interest of this project resides : a new distance between data series is introduced, which can then be used to cluster leaves together and infer the network's shape.

This new distance, called the *normalized compression distance* (NCD) is presented in depth in [4]. The authors of this paper have furthermore implemented a tool, CompLearn [3], which can be used to automatically cluster data using this distance.

The hope was to achieve results close to those obtained by the probabilist approach of [7], knowing that it would then be possible to apply the compression-based method to networks with weaker constraints that probabilistic methods cannot easily take into account, such as the possibility of adding dependency between packets over different edges.

4.1 Computing nearest compression distance

The general idea of the NCD is to approximate and normalize the shared information between two data series. In our study, these will be the series $\left(L\left(A\right)_i\right)_{i\leq N}$ of boolean values representing whether the i-th packet reached node A or not.

To compute the NCD between two nodes A and B, we compress data series $L\left(A\right)$, $L\left(B\right)$ and their concatenation $L\left(A\right)L\left(B\right)$ with any efficient compressor, and note C_A, C_B and C_{AB} the size of the compressed series. The NCD is then defined as :

$$NCD(A,B) = \frac{C_{AB} - \min(\{C_A, C_B\})}{\max(\{C_A, C_B\})}.$$

CompLearn computes the NCD between files. It can also automatically generate a matrix of the NCDs between any number of files.

4.1.1 Formatting data

It quickly became clear that there were however great flaws in this method : with multicast sources generating losses, it is important that the fact that two packets are lost at the same time is brought forth, which is clearly not the case when simply concatenating the loss series.

Losses between consecutive packets being independent, the problem is : how can the data be formatted so that the compressor differentiates shared losses and individual losses ?

The most intuitive solution was to combine the two loss series into one two-dimensional loss series, the information about the loss of a given packet at both leaves being thus juxtaposed.

This however led to the question of how to compress such a loss series. The ideal way to achieve this would be to represent each pair as one lexeme from a four-letter alphabet as this captures the full set of information in a two-dimensional binary vector, and to compress the time series over that alphabet.

Most compressors work on binary inputs, but not all, and it was interesting to realize that four-letter alphabet compressors would surely exist in genetics, seeing the needs for genome compression. Sure enough, there was a lot of information on the subject to be found, but sadly only one implemented compressor was actually available, and it was frustrating to find out that it would not compile.

It therefore ended up being necessary to implement a generic compression algorithm.

4.1.2 A generic data compressor

Implementing a compressor on an arbitrary alphabet actually turned out to be very simple. The algorithm used was the 1978 Lempel-Ziv Algorithm (LZ78) presented in [8], which consists in building a tree-structured dictionary in which previously seen phrases are stored. The compressed stream is a sequence of references to nodes in the dictionary.

This dictionary was implemented as a template class `Dictionary<T>` with a unique field of type `map<T,Dictionary>`, the template argument `T` representing the alphabet. In this study, the alphabet was defined as an enumerable type with four possible values.

One of the nice things here is that, seeing as all that was needed was the size of the actual output, which is easily derived from the size of the final dictionary, it was sufficient to simply build the dictionary without producing an actual output and then retrieve the number of nodes N. The length of the output being equal to the number of nodes times the size of a reference to a node, the compressed size of the input is then $N \log N$.

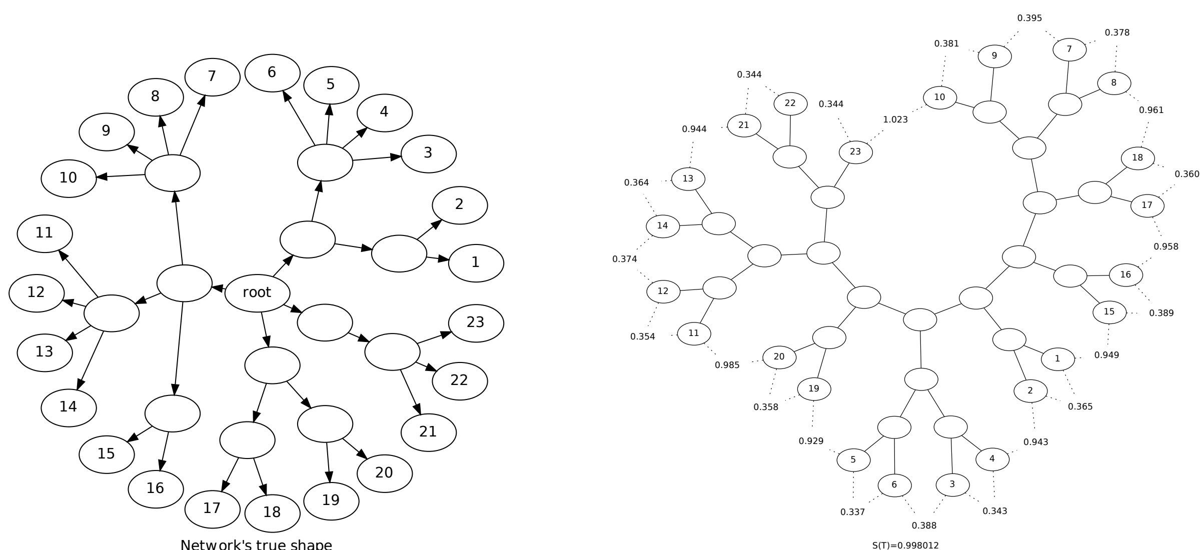
4.2 Inferring the network's shape

Once the distance matrix between nodes has been established, one question still remains : how can one build the tree from these distances ?

The CompLearn tool already offers a clustering method based on quartet topology. The idea here is to find the most consistent undirected ternary tree in which the leaves represent the leaves of our network. The notion of *consistency* is detailed in [4].

This method has a major flaw however : though it will group closer leaves together, it will not give an accurate representation of the network, seeing as it gives no indication as to the location of the network's root, and network's are never structured as binary trees. (Figure 2)

Figure 2: Clustering using CompLearn



Another clustering method had to be found. A usual method in clustering would be to combine the two most similar leaves, replacing them by one unique “leaf”. Three questions still need to be answered :

1. How do you compute the similarity between two leaves ?

2. How do you know when to regroup three or more leaves together ?

3. How do you determine what *merged* time series to allocate to the new merged node ?

The solutions to these problems were primarily inspired from [7].

4.2.1 Computing similarity between leaves

Considering that the distance d used between leaves is normalized, we define the similarity σ between two leaves A and B as :

$$\sigma(A, B) = 1 - d(A, B)$$

4.2.2 Regrouping multiple leaves together

Suppose two leaves A and B are grouped together as (AB), and that we want to combine (AB) with a third leaf C. Two cases occur :

1. If $\sigma(A, B) \approx \sigma((AB), C)$, then it is most likely the three leaves should be combined as (ABC) ;

2. On the contrary, if $\sigma(A, B) \gg \sigma((AB), C)$, then surely the three leaves should be combines as $((AB)C)$.

In order to differentiate these cases, it was necessary to “pick” a permissiveness α to define the relation $\approx$:

$$a \approx b \Leftrightarrow a \geq \alpha b \qquad (a \leq b)\,.$$

In other words, (AB) and C will be combined as (ABC) if $\sigma(A,B) \geq \alpha\sigma((AB),C)$, and as $((AB)\,C)$ if not.

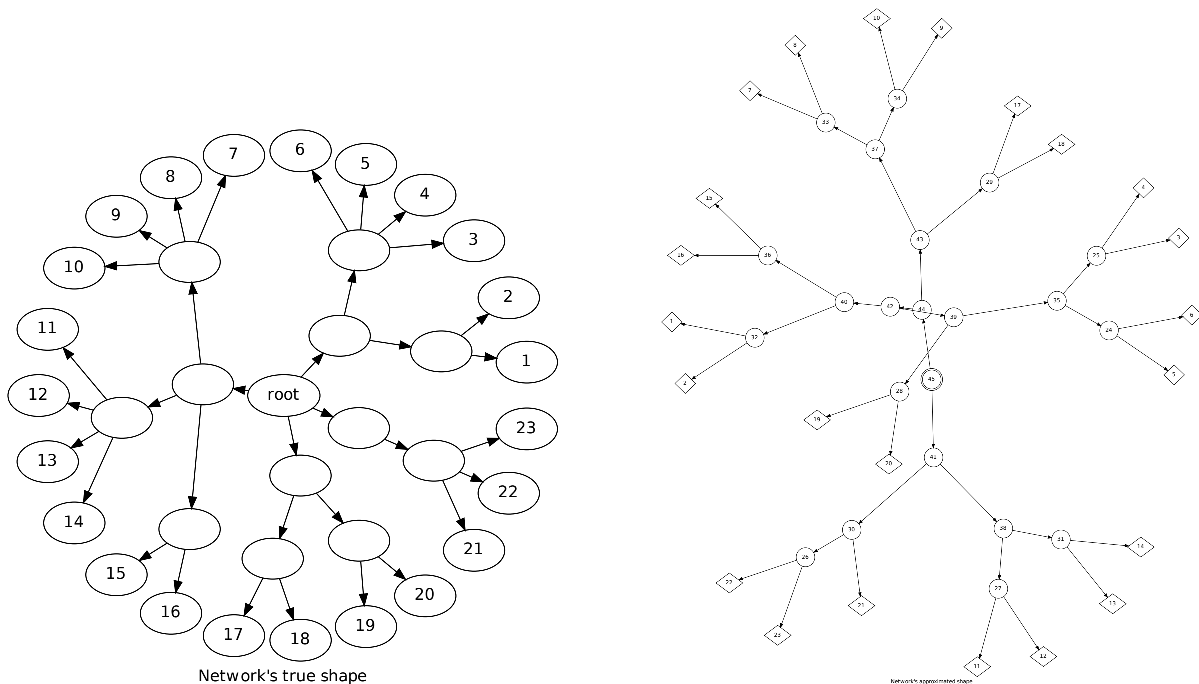
4.2.3 Allocating time series to new merged nodes

All that remains to be done is to define the distance between new “leaves” and the previous ones.

Computing this distance is very simple for loss series : combine the loss series of the two combined leaves by keeping only shared losses ($L(AB) = L(A) \wedge L(B)$), and recompute distances between this new loss series and that of the other leaves. This is also the method used in [7].

Combining the delay series proved to be far more difficult : should one keep the smallest delay ? the longest delay ? the mean of these two delays ? Experience shows that keeping the longest delay worked best. (Figure 3)

Figure 3: Clustering using the new method



5 Analysing Performance

5.1 Means of analysis

In order to evaluate the performance of our method, it was necessary to chose another method to compare it to, and to define a distance between trees so as to compare results automatically.

5.1.1 The probabilist approach

We compared the compression method to the probabilist method described in [7]. This approach consisted in formally computing the probability of shared losses for each pair of leaves, which can be seen as the similarity between these leaves.

Computing the probability P^t_{ab} of having true shared losses between two leaves a and b — as opposed to all shared losses, including those due to losses occurring after the branching point in the paths — yields

$$P^t_{ab} = \frac{P_{ab}P_{b\bar{a}} + P_{b\bar{a}}P_{a\bar{b}} + P_{a\bar{b}}P_{ab} + P^2_{ab} - P_{ab}}{P_{ab} + P_{b\bar{a}} + P_{a\bar{b}} - 1}$$

where P_{ab}, $P_{a\bar{b}}$, and $P_{b\bar{a}}$ are the probabilities of a packet being lost for both a and b, for a and not b, and for b and not a. In practice we estimate these probabilities by the apparition rates of such losses.

5.1.2 Implementation of both methods

Seeing as the algorithms only differed by the distance method, the code, written in C++, was the same, with the exception of the `Distance` method, which had different implementations. Two competing programs were generated by simply modifying the linkage to this method during compilation. (Figure 4, output examples Figures 5–7)

Figure 4: Implementation structure

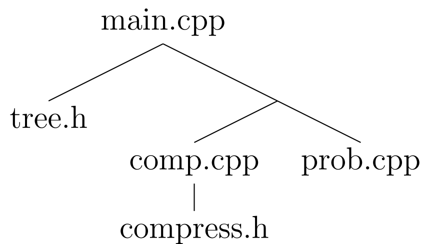


Figure 5: Network's real shape

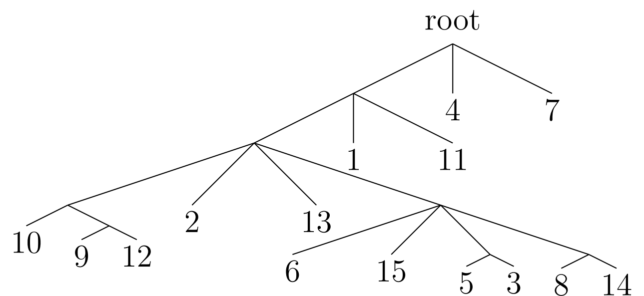


Figure 6: Result of probabilist method

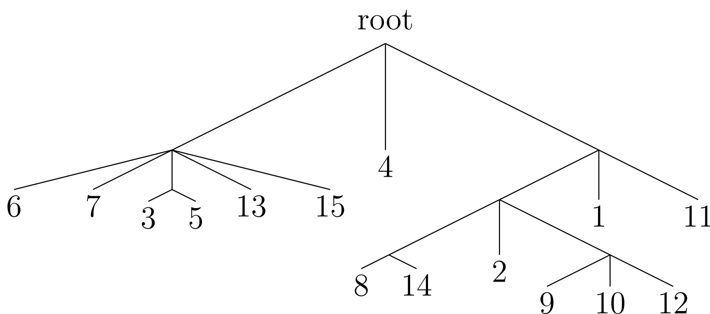
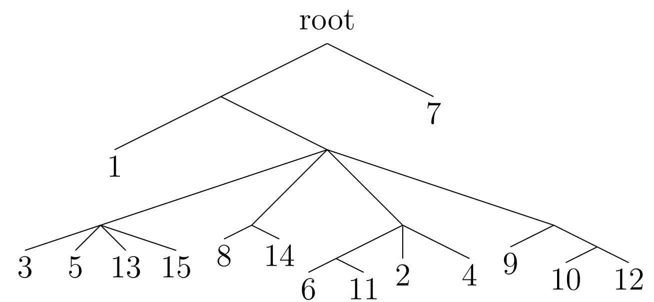


Figure 7: Result of compression method



5.1.3 A new tree distance : the Mean Path Error

Despite there being many theoretical tree-distances in the literature[1], most of these were not suited for implementation, and no already existing program could be found.

[1] See [5] for an example

A distance was therefore introduced, apparently best suited for the needs of this analysis : the *Mean Path Error*.

Due to the frequent extra or missing edges in the inferred trees, it was necessary to find a distance for which such an error would not have an important impact, even if that error appeared near the root.

To define the Mean Path Error, we first define the *edge distance* between two leaves i and j of a tree T as the number $e_{i,j}^T$ of edges between i and j.

Now consider two trees S and T with the same n leaves. The Mean Path Error between S and T is the distance $d(S,T)$ defined by :

$$d(S,T) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \left| e_{i,j}^S - e_{i,j}^T \right|$$

It can be shown that this is indeed a proper distance, and most importantly that $d(S,T) = 0$ if and only if S and T are identical.

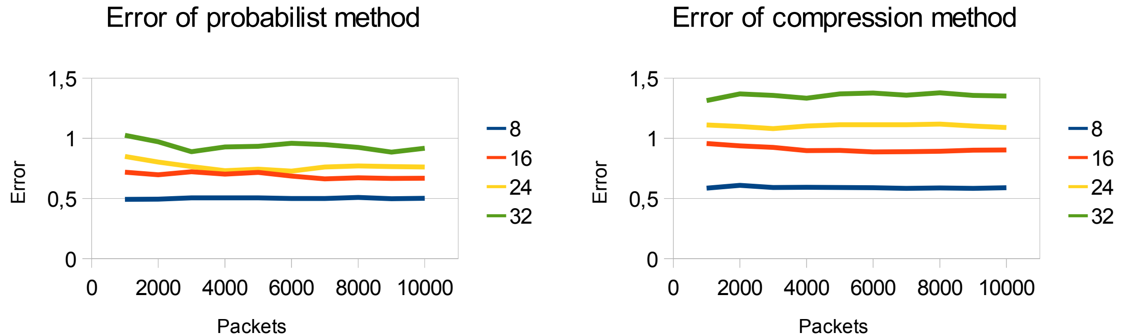
With this distance defined, it was then possible to automatically compare the results of both inference methods on any simulation, and thus to begin to study average performance on large numbers of random networks of varying size.

5.2 Convergence speed

The first batch of simulations aimed at studying the influence of the size of loss series on the average error in inferences, and showed that after 2000 packets, the error would stay about the same. (Figure 8)

This first result was quite important for the rest of my analysis, seeing as studying too few packets would result in inconsistent results, whereas taking into account too many packets would force me to reduce the number of simulations due to time constraints, thus reducing the precision of the results.

Figure 8: Average error of both methods depending on the number of packet samples for networks with 8, 16, 24 and 32 nodes

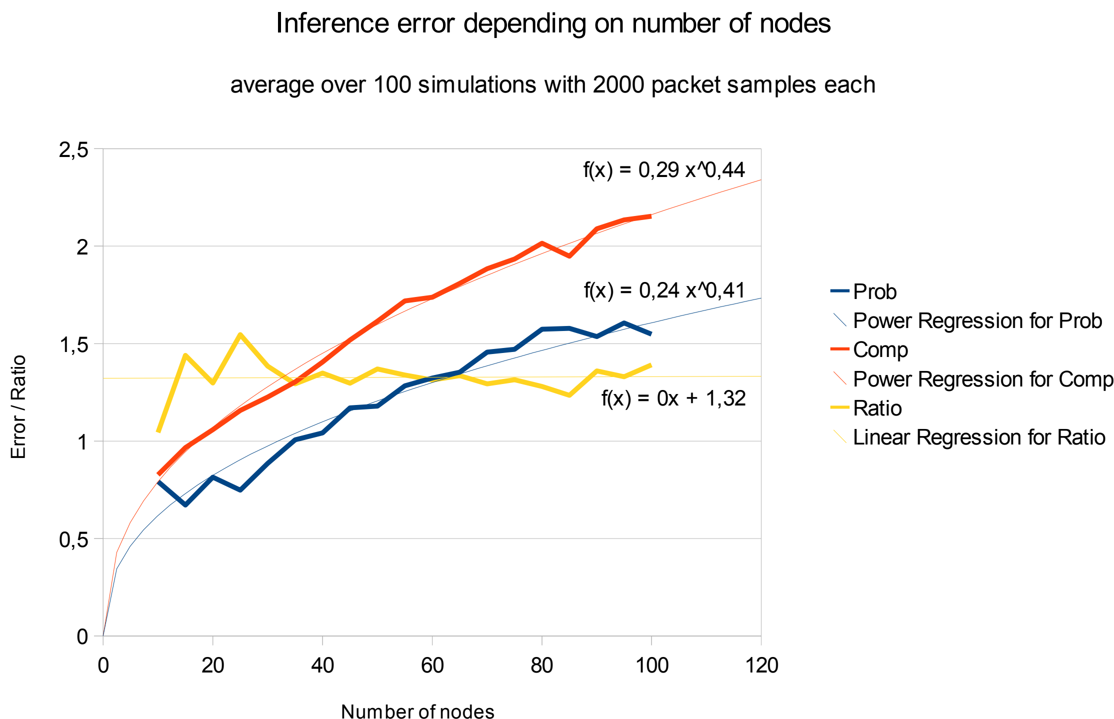


5.3 Influence of the network's size

The next series of simulations were aimed at studying the evolution of errors as the size of the network increased. These experiments tend to show that, though the compression method

is less efficient than the probabilist one (which is easily conceivable, the latter being far more specific), the error ratio between the two is pretty much constant as the number of nodes in the network increases (the error being proportional to more or less the same power of the number of nodes), suggesting that the method will do just as well as the probabilist approach on both small and big networks (though computation time may prove to be an issue for very big networks, as discussed latter). (Figure 9)

Figure 9: Average error of both methods depending on the network's size



5.4 Finding the right α

All the preceding experiments were made for an arbitrary permissiveness α which seemed to suit well (0.95), but it felt necessary to study the influence of this parameter on performance.

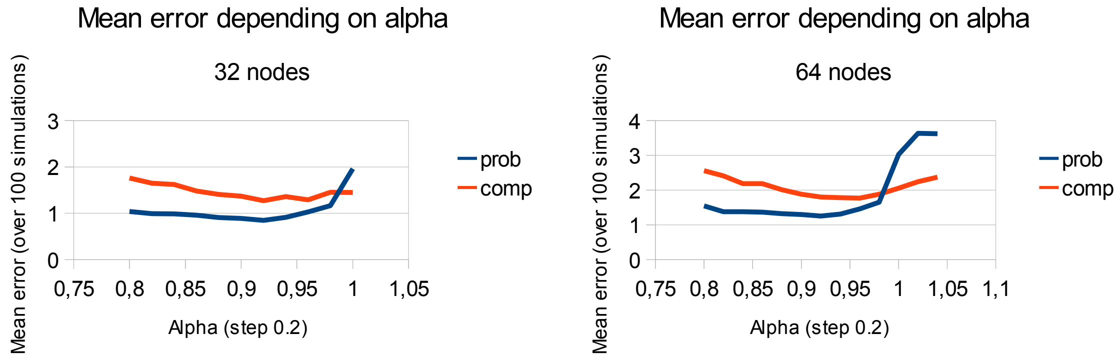
It turns out the original estimate was close to optimal, which appears to be 0.94 for the probabilist approach, and 0.96 for the compression approach (Figure 10). It is interesting to note that the second method continues to work for values of α greater than 1. This is due to the fact that, when combining leaves, the combined loss series can sometimes compress better than previous ones, reducing distances to other leaves and thus increasing similarity.

5.5 What about computation time ?

There is however one major drawback to this new method : experiments show that the probabilist approach is far quicker.

Designating n as the number of leaves and m as the number of packets, the algorithmic complexity of the compression method is only $O\left(n^2 m \log m\right)$, that is to say hardly more than

Figure 10: Average error of both methods depending on the permissiveness α



that of the probabilist method : $O(n^2m)$. This suggests that the difference resides in the constant.

6 Testing with Another Model

One of the strengths of this method is that, unlike the probabilistic model, it makes no hypothesis on the network's characteristics.

In real networks, packet losses and delays are neither time nor space independent. The nature of traffic (UDP or TCP) and of nodes (queuing systems in routers) indeed creates strong dependencies here.

A second simulator, implemented in part before the internship, was thus used to test this clustering method on a more complex model, though there was no time to analyse results thoroughly.

6.1 The simulator

This simulator took quite some time to get together, and was regularly enriched throughout the internship. A lot of time was also spent debugging it.

The simulator was written in C++, and was split into many independent sections.

6.1.1 A library providing many random processes

This section provided numerous random variable generators.

Two type of generators were provided, based on whether the random variables were drawn from $\mathbf{N}$ or $\mathbf{R}^+$. The generators were represented by classes inheriting from an abstract class with one virtual method : `draw()`.

This had two uses. To begin with, parameters could be assigned at the creation of the generator, and then stored within the structure. This then enabled sources and other structures using random variables to be given a generator at their creation and to draw random variables regardless of the law and of parameters.

Here is an illustration of how these generators work :

```
Exponential myGenerator(2.);
// Instantiation of a real number generator using the exponential law with
   parameter 2
```

```
float a = myGenerator.draw(), b = myGenerator.draw();
// Two different values drawn using the exponential law with parameter 2

Node myNode(myGenerator);
// myNode can use myGenerator to draw random variables despite it not knowing
    the type and parameters of the law
```

Many simple random variables were implemented, as well as two more complex ones corresponding to the distance between consecutive points of two specific random processes :

Cyclic processes Inter-arrivals were deterministic but varied in a cyclic fashion, so as to gather data from both slow and rapid packet rates.

Cluster processes This process represents packet inter-arrivals throughout sessions consisting of multiple clicks, each of which was in turn interpreted as a batch of packets.

6.1.2 A structure representing packet sources

Here is where the data structures for packets and sources were defined.

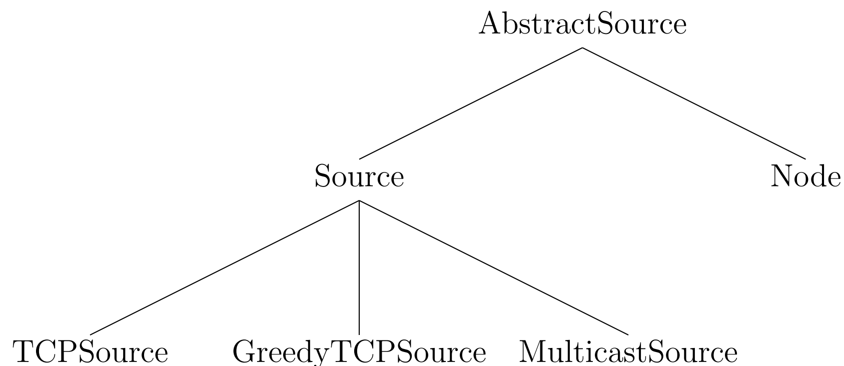
Packets A packet was defined as follows :

```
struct Packet
{
    Source* origin; // Pointer to the source that emitted the packet (for TCP
        purposes)
    bool tagged; // Is this packet being traced ?
    bool multicast; // Is this packet part of a multicast flow ?
    unsigned int destination; // packet's destination (0 if part of a transverse
        or multicast flow)
    float creation; // Date at which the packet was emitted from the source
};
```

Sources The structure of sources was a bit more complicated due to the different types of sources that were implemented : TCP controlled or not, greedy TCP, multicasting or unicasting, and of course nodes.

Due to their similarities, these data structures are all linked through inheritance. (Figure 11)

Figure 11: Inheritance scheme for sources



6.1.3 A structure representing network nodes

Nodes are a very specific kind of sources : they do not simply emit packets every so often, but must also keep track of the packets queuing up, allow other sources to send packets to it, have a table detailing which nodes packets heading for a given direction should be directed to, etc.

The nodes were implemented so as to allow both FIFO and LIFO services, multiple servers, and the possibility to limit storage space, which gives rise to packet losses.

6.1.4 A random tree generator

This is where the tree's shape was determined. This section was divided into two parts : first creating an empty tree structure, then taking that structure and creating the appropriate sources and nodes. This separation meant that a new tree generation algorithm could latter be implemented without having to worry about creating the sources and nodes whilst building the tree.

The algorithm for determining the tree's shape was as follows :

1. Choose a maximum depth for leaves ;
2. Every node that has not reached the maximum depth has its number of children determined by an integer random variable (most often binomial, so as to limit the number of sons of a given node), a node without children being replaced by a terminal.

Once the shape of the tree has been determined, sources and nodes are created as follows : For each node of the tree, a node structure is created, the routing table is computed, and sources are introduced to represent transverse flows. For each leaf, sources are created to generate both the persistent flows and the tagged packet flows.

The number and speed of non-tagged sources, as well as the capacity, the number of servers, and the service speed of nodes are randomly generated, and are taken to be functions only of the depth of nodes in the network.

6.1.5 A priority heap to store upcoming events

This section was very straightforward : use a priority queue to store upcoming events by date, and implement functions to either add an event to the heap or retrieve the top event.

An event was in fact no more than an activation time and a pointer to the node or source to be activated, so that the main loop of simulator was brought down to :

```
while (Time() < EndOfSimulation)
{
    NextEvent()->Activate();
}
```

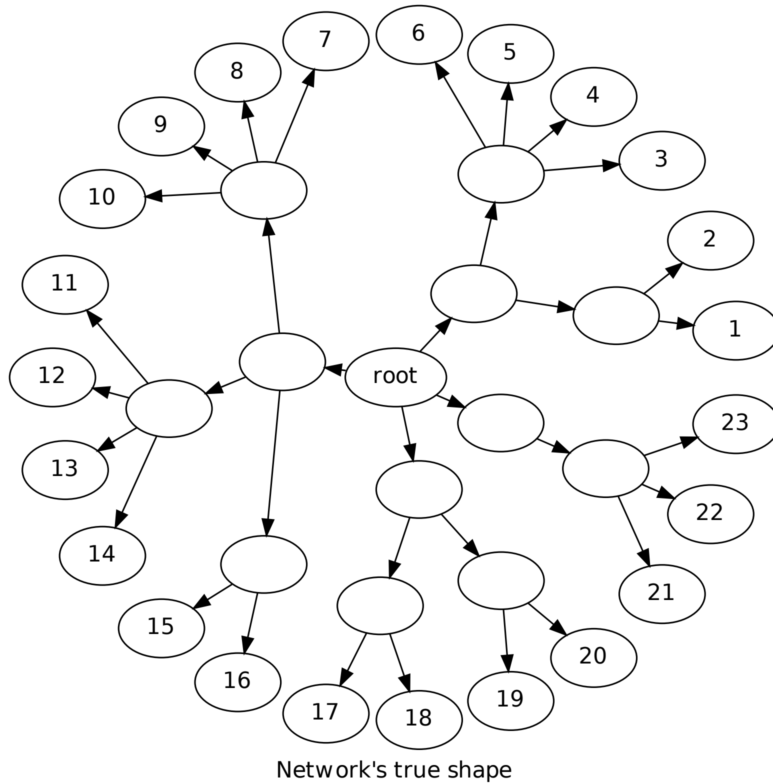
The time used during this simulation was also stored in this section, which seemed most appropriate seeing as it was only modified during calls to NextEvent(). The Time() function is the only means of accessing it elsewhere.

6.1.6 A library of methods for outputting data

There was a lot of data to output throughout the simulation, so in order not to carry output streams around in the rest of my code, these were all grouped in one place.

The information that needed to be stored into files was the packet losses and delays, in different files for each leaf, as well as the random seed, the number of terminals, and a graph of the network to be able to compare the inferred topology to the real one. (Figure 12)

Figure 12: Graph produced by the simulator



6.1.7 A network configuration file

This file contains every parameter of the simulator. These include the different random processes and other parameters for nodes and sources, the random variables used to generate the tree, the number of transverse/persistent flows per node/leaf, and the nature of the tagged flows.

These were used in parameterless functions called upon by the rest of the program throughout the network building phase, and which build every atomic element themselves.

6.1.8 The main execution thread

This part is really simple, since it consists of using all that's been done in order. Basically, the main function does the following things :

1. Initialize the random seed and record it ;
2. Generate the network's shape and output it ;
3. Build the tree (sources and nodes) ;
4. Run the simulation.

6.2 Observations

Though time constraints prevented me from thoroughly testing this second model, I noticed a couple of tendencies after a few simulations.

The first of these is that nodes were sometimes grouped in clusters according to the number of "siblings" that they had, which meant that siblings were indeed often placed closed together, but that their other theoretical neighbours often appeared to come from just about anywhere else in the network.

The other observation is that clustering appeared to be more sensitive to losses than to delays (the more losses, the better the result). More simulations would however be required to confirm this hypothesis.

There was sadly no time to test the probabilist method to the loss series generated by this model, though it is hoped that compression distance would do little worst here, if not better.

7 Conclusions

This study of compression-based network inference is not complete, but it nonetheless suggests that such an approach could eventually yield reliable information for determining the shape of most networks.

Though the experiments on a simplified model are neither as accurate nor as fast to compute as those of a probabilistic method, they are nonetheless highly satisfactory, and have the advantage of not placing any restrictions on the model.

With the diversity of network topologies in today's world, such a universal inference method could prove most useful in understanding what the Internet looks like.

There is still much that needs to be done before truly reliable results can be extracted from a network this way. This is true of every method known to this day, however, so compression-based inference should not be overlooked.

References

[1] Graphviz - graph visualization software. `http://www.graphviz.org`.

[2] R. Caceres, N.G. Duffield, J. Horowitz, F. Lo Presti, and D. Towsley. Loss-based inference of multicast network topology. In *Proceedings 1999 IEEE Conference on Decision and Control*, 1999.

[3] Rudi Cilibrasi, Anna Lissa Cruz, Steven de Rooij, and Maarten Keijzer. Complearn. `http://www.complearn.org`, October 2010.

[4] Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.

[5] Mark Coates. Maximum likelihood network topology identification from edge-based unicast measurements. pages 11–20, 2002.

[6] N.G. Duffield, J. Horowitz, F. Lo Presti, and D. Towsley. Multicast topology inference from measured end-to-end loss, 2001.

[7] Sylvia Ratnasamy and Steven Mccanne. Inference of multicast routing trees and bottleneck bandwidths using end-to-end measurements. In *Proceedings of IEEE INFOCOM 1999*, 1999.

[8] Christina Zeeh. The lempel ziv algorithm. In *Seminar "Famous Algorithms"*.