UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze e Tecnologie
Laurea Magistrale in Fisica

Models for hierarchical inheritance structures in object-oriented programming languages

Advisor: Prof. Mario Raciti
Co-advisor: Prof. Marco Gherardi

Mariacristina Romano
Matr. n° 823029
A.Y. 2013/2014

PACS: 89.75.-k

Models for hierarchical inheritance structures in object-oriented programming languages

Mariacristina Romano
Dipartimento di Fisica, Università degli Studi di Milano,
Via Celoria 16, 20133 Milano, Italia

April 15, 2015

Abstract

Object-oriented programming languages allow developers to write software in which *objects* have different kinds of relations and interactions: a complex system!

The object paradigm states a list of rules which facilitate the code re-use. Probably the most important mechanism that leads to a minimal effort is the inheritance relation.

Inheritance allows the construction of multilevel hierarchies, which can be modeled by directed acyclic graphs.

How do programmers use inheritance? Which kind of graphs arise from hierarchical relations? What quantity governs the optimization?

Advisor: *P*rof. Mario Raciti
Co-advisor: *P*rof. Marco Gherardi

Contents

Chapter 1

Object-oriented paradigm

Object-oriented programming is the most used programming paradigm in today software design. It allows an intelligent code reuse through inheritance and polymorphism, and it is probably the only one programming paradigm that allows people to design huge projects with a cooperative effort.

Programs and libraries written in object-oriented languages are composed through the concept of *objects*, which can be seen as modular code, composed by data and methods. Objects can interact with each other and there may be different kinds of relations among them.

Through the concept of objects, the whole set of programs can be seen as a huge complex system, with many small elements which interact and cooperate to achieve the same goal, which is the purpose for which each program has been made.

One of the most important relation that can bind objects is the inheritance: a rule introduced in object-oriented programming languages in order to facilitate and automate the reuse of code.

The relation of inheritance arranges objects in hierarchies: when an object inherits from another one, we can imagine a small graph with two nodes connected by a directed link, for example from the original object to the one which has inherited the code. Then, we can add lots of other objects to such graph, and make it enormous and complex, due to the possibility to inherit code from different objects and due to the fact that there is no limitation in the number of objects that can inherit code from the same object.

Such hierarchies are the starting point of this work.

1.1 A little terminology

Class and objects

Object is the word used to refer to an instance of a *class*: there may be lots of objects of the same type in a program. In this thesis there is no necessity to distinguish these two concepts and words will be used without distinctions.

Abstract classes

An abstract class is a class that can not be instantiated and so that can never become an object. Such class contains methods that has no implementation, but only a declaration.

Superclasses and subclasses

When a class inherits from another class, the first is called the *superclass* of the second one, while the second is a *subclass* of the first one.

The subclass usually inherits from the superclass its *public code*.

Multiple Inheritance

Multiple inheritance occurs when a subclass has more than one superclass. Many programming languages allow this possibility, like C++ and Python, while in others it is limited (as in Java) or even not allowed.

Multilevel Inheritance

Multilevel inheritance occurs when a subclass has inherited from one (or more) superclass which is itself the subclass of another class.

In all programming languages studied in this thesis, multilevel inheritance is enabled, but there is a way to prevent the inheritance from a class in Java and C++, through a specific keyword inserted in the class that is considered *final* for the hierarchy.

1.2 Object-oriented programming languages

There are many object-oriented programming languages, with lots of differences among them but united by the object paradigm. To give a complete overview of inheritance hierarchies, three different programming languages have been analized: C++ [Bja], Java [Sun] and Python [Gui].

This thesis can not contain a whole course about three programming languages, but it is useful, however, to highlight some important differences, in order to give you the tools to judge the data analysis.

C++ language

Among the three languages, C++ is the oldest and the one which spread the object-oriented paradigm.

It has been designed to improve *C language*, adding features as virtual functions, operator overloading, references, keywords to control the store-free of memory, and, obviously, the object-oriented paradigm.

It is a typed language and it supports multiple inheritance.

Java language

Java has been designed to be a secure and portable programming language.

Its syntax is identical to the one of C++, but its inheritance rules has some peculiarity. Multiple inheritance is not available in the common sense: a class can inherit code from one superclass only, but it can also inherit declarations of functions and variables from any number of a special kind of classes, called *interfaces*.

In general, rules have been created in order to reduce complexity in programs and to keep the code as simple as possible without limiting its capabilities. Some examples are the prohibition to overload operators, the deletion of the instruction *go to* and of the multiple inheritance.

Python language

Python has been designed with the goals of be useful in any purpose and to allow a really fast programming.

The syntax is quite different respect to the other two languages, since its design philosophy is to allow to express concepts in the minimum possible lines of code and since it is strongly dynamically typed.

It allows multiple inheritance as well as C++, without any limitation.

1.3 Complex noisy data

A complex system analysis always begins from a set of chaotic, complex and noisy data.

Internet is the open-source code home and allows people to obtain a huge amount of libraries and programs in every programming language. The noisy data, which is the subject of the following analysis, has been downloaded from GitHub [PJ], the largest code hoster in the world [GVSZ14].

Hierarchical inheritance structures can be easily modeled as directed acyclic graphs, and then the dataset is a huge ensemble of different kinds of graphs.

In graphs terminology, the oudegree of a node is the quantity which describes the number of nodes pointed by a link from such node. Since it describes how many classes inherit from such class, such relation means speciation.

The indegree instead represents the multiple inheritance, and so it describes the number of classes from which the selected class has inherited code.

Another important point is that each program is made to perform a task, and then each hierarchy is done to solve a problem or a subtask of the whole program.

In principle all classes in a hierarchy can be directly used in programs, excluding abstract classes, but only some of these classes becomes real objects, while the other nodes have structural purposes only.

Chapter 2

Minimal Effort model

The dataset is large and noisy. A model is needful to guide the data analysis and to give an interpretation to all results.

The guiding idea of the Minimal Effort model is that inheritance is designed to better promote code reuse and, in this sense, to minimize the effort in writing programs.

The model is a mean field approximation for graphs and it allows you to obtain some interesting features, useful in all next chapters.

2.1 Reuse and Effort

Programs are created to perform tasks and solve problems and, in object-oriented programming, each task is performed with modular code, the objects, often organized in hierarchies.

In a first approach, let's suppose that only terminal nodes of inheritance graphs compose the effective solution of the problem, while all nodes at higher levels are made for code reuse purposes only.

Terminal nodes are the leaves of the tree which represents the hierarchy. Leaves are given by the problem, but rules are needed in order to build the whole tree from its leaves. How can we group leaves in different levels? Which principle may guide us?

2.1.1 Why use inheritance?

Assuming that inheritance most important effect in graph structure is code reuse, we can start from the $\mathfrak{n}$ classes which represent the solution of the problem.

Suppose that the implementation of two of this classes is very similar. Then you can move mutual code lines in a third node, in the upper level of this tree.

In this way, one can avoid writing twice the mutual lines, creating a node which shares its lines with the two leaves. In this sense, we are guided by a minimum effort principle. Obviously, one can group more than two nodes and try to absorball mutual lines of leaves code in higher level.

The function effort $\mathtt{E}$ of a tree is defined simple as

$$\mathtt{E} = \sum_{\sigma}^{\mathfrak{n}} \mathrm{cost}(\sigma)$$

a sum over all nodes, were each node cost is normalized as

$$\mathrm{cost}(\sigma) = 1 - \varepsilon$$

were ε represent the fraction of code which inheritance permits you to save.

2.1.2 Competition

The trivial minimum is reached when all leaves can share all its code in one upper node, and so when all leaves are identical. To avoid triviality we need a contrasting mechanism: competition arise from the fact that leaves are different in its implementation and more are the classes that one is trying to group together and less is the number of code lines that can be shared, and so $\varepsilon = \varepsilon(\mathfrak{m})$.

To find an explicit form for $\varepsilon(\mathfrak{m})$, consider each class made of an unordered sequence of k symbols. The number of available symbols is $\mathcal{S}$ and symbols are equally likely. Symbols that compose classes represent functions, or private variables, or something that can be considered a unit for the code at this abstracted level.

How much code can be shared by $\mathfrak{m}$ nodes? The probability to draw a selected symbol in a sequence of trials with replacement is given by the Binomial distribution of parameters $p = \frac{1}{\mathcal{S}}$ and k as the number of trials. We can calculate the probability to find the selected symbol once or more as 1 less the probability to not find the selected symbol, and so

$$p = 1 - \left(1 - \frac{1}{\mathcal{S}}\right)^{\mathsf{k}}$$

Now suppose $\mathcal{S} \to \infty$ and $\mathsf{k} \to \infty$ and define β as

$$\beta = \frac{\mathsf{k}}{\mathcal{S}}$$

In this limit, we can expand

$$e^{-\beta} = \lim_{\mathcal{S}\to+\infty} \left(1 - \frac{1}{\mathcal{S}}\right)^{\beta\mathcal{S}}$$

and then we can rewrite the probability to find a selected symbol in a sequence of extractions as

$$p = 1 - e^{-\beta}$$

Consider $\mathfrak{m}$ indipendent sets of k extractions. The probability to find the selected symbol in all $\mathfrak{m}$ sets is simply

$$\rho = \left(1 - e^{-\beta}\right)^{\mathfrak{m}}$$

We are interesting in the fraction of code $\varepsilon(\mathfrak{m})$ that $\mathfrak{m}$ nodes can share. Since all symbols are equal for our groupment, we can multiply the last result by $\mathcal{S}$, and divided it by the length of the sequence k to normalize the result.

$$\varepsilon(\mathfrak{m}) = \frac{\mathcal{S}}{\mathsf{k}} \left(1 - e^{-\beta}\right)^{\mathfrak{m}} = \frac{1}{\beta} \left(1 - e^{-\beta}\right)^{\mathfrak{m}} \equiv e^{-\alpha\mathfrak{m}}$$

It is now possible to evaluate the cost of each single node and then of each graph.

$$\mathrm{cost}(\sigma) = 1 - e^{-\alpha\mathfrak{m}}$$

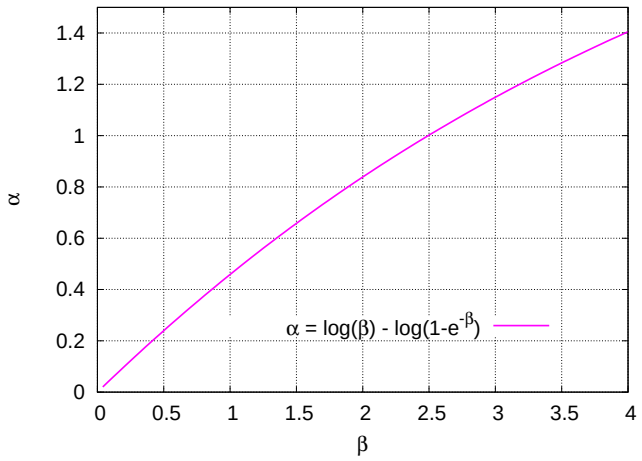

Figure 2.1: ***α* and *β*** - α is small for small β, and viceversa.

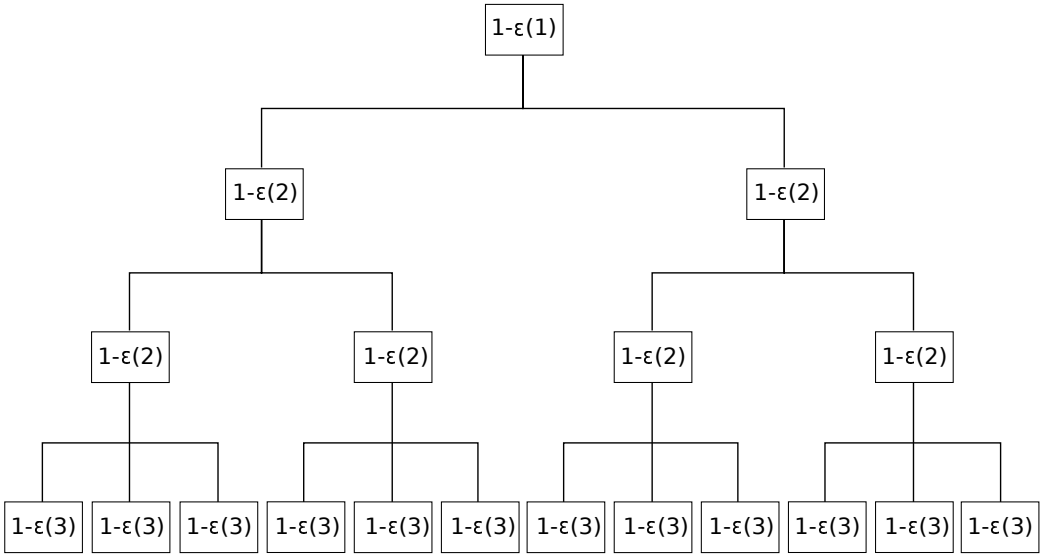

Figure 2.2: **Mean field tree** - The Minimal Effort model estimates the effort necessary to build a graph as the sum of the cost of each node.

2.1.3 Minimizing the effort

Despite the simplicity of the functional $\mathtt{E}$, it's not easy to make calculation keeping the whole complex structure of all possible trees. We need to simplify the topology.

Levels are labeled l, where the level $l = 0$ correspond to the set of leaves and the level $l = \mathtt{L}$ is the set containing the root only. At each level l, the tree is essentially a set of nodes $\mathfrak{n}(l)$.

We can estimate the mean number of brothers that each node may have. It is given by the number of nodes at a level $\mathfrak{n}(l)$ divided by number of available parents, and so the number of nodes at successive level $\mathfrak{n}(l+1)$

$$\mathfrak{m}(l) = \frac{\mathfrak{n}(l)}{\mathfrak{n}(l+1)}$$

Forgetting all details, let's consider a tree configuration all expressed by the number of nodes at each level, ignoring which nodes are linked together and which are not.

$$\{\mathfrak{n}(l)\}_{l=0}^{\mathtt{L}} = \{\mathfrak{n}(0), \mathfrak{n}(1), \ldots, \mathfrak{n}(\mathtt{L}) \equiv 1\}$$

With this simplification we can rewrite the effort as a sum over levels, and then

$$\mathtt{E}[\mathtt{L}, \{\mathfrak{n}(l)\}] = \sum_{l=0}^{\mathtt{L}-1} \left[1 - \varepsilon\left(\frac{\mathfrak{n}(l)}{\mathfrak{n}(l+1)}\right)\right] \mathfrak{n}(l)$$

And expliciting the function ε we obtain

$$\mathtt{E}[\mathtt{L}, \{\mathfrak{n}(l)\}] = \sum_{l=0}^{\mathtt{L}-1} \left(1 - e^{-\alpha \frac{\mathfrak{n}(l)}{\mathfrak{n}(l+1)}}\right) \mathfrak{n}(l) \tag{2.1}$$

It is now possible to obtain the minimum of the Effort $\mathtt{E}$ respect to the total number of levels $\mathtt{L}$ and the number of nodes at each level $\mathfrak{n}(l)$.

$$\mathtt{E}^* = \min_{\mathtt{L}\,\{\mathfrak{n}(l)\}} \mathtt{E}[\mathtt{L}, \{\mathfrak{n}(l)\}]$$

2.2 Numerical solutions

Despite the lack of details about tree shapes, the model allows you the study of some important quantities, like nodes mean degree or population at each level. In this section results of numerical simulations about this quantities are shown for different parameters.

The minimization of the Effort functional (2.1) has been made with the interior point method implemented in the program Wolfram Mathematica [Wol]. The number of nodes, which is a discrete quantity by definition, is allowed to be real numbers in minimization process.

2.2.1 Finding optimal depth

The Effort $\mathtt{E}$ is minimized respect to the number of nodes at each level $\{\mathfrak{n}(l)\}$ and the depth $\mathtt{L}$, which is the maximum level accessible by the tree, that is populated by only one node, the root.

Free parameters of the model are $\mathfrak{n}(0)$, which is defined by the number of leaves, or the number of nodes necessary to solve the problem for which the tree has been created, and α which represents the abstractability.

Fixing reasonably values for these parameters, as $\mathfrak{n}(0) = 1000$ and $\alpha = 0.1$, we can look for the optimal depth $\mathtt{L}^*$ which minimize $\mathtt{E}$.

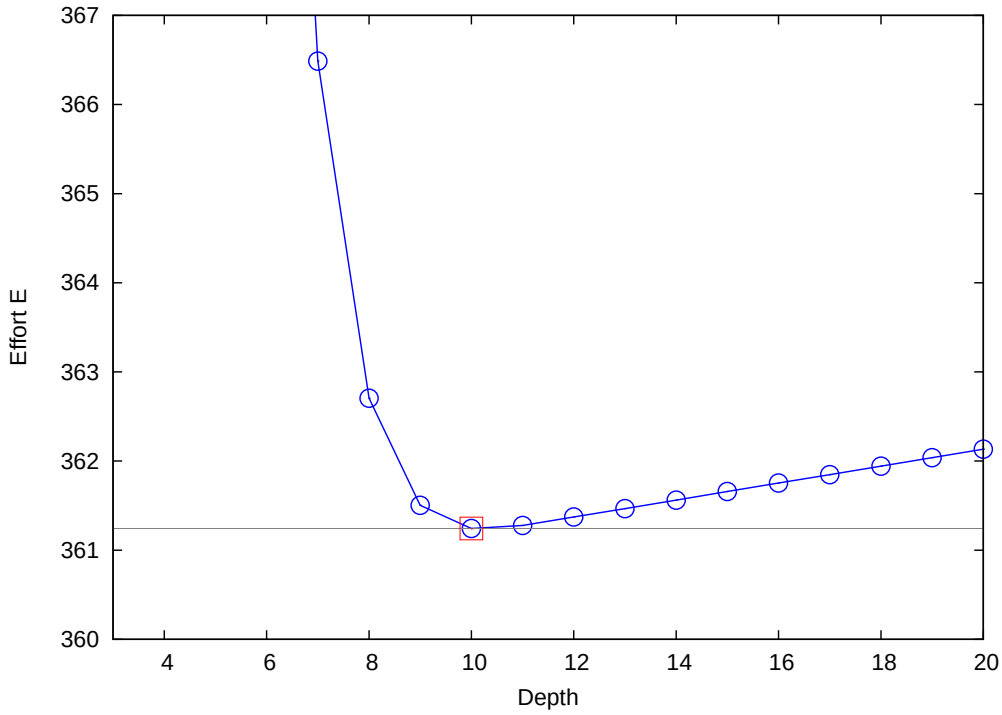

Figure 2.3: **The Effort as a function of depth** - When $\alpha = 0.1$ and $\mathfrak{n}(0) = 1000$, simulations show that the Effort is minimal for $\mathtt{L} = 10$.

The numerical solution leads to $\mathtt{L}^* = 10$, as showed in figure 2.3.

Population at each level

When the depth is optimal, the number of nodes is exponential in its dependence from the level, as shown in figure 2.4.

Starting from a depth value lower than the optimal one and increasing it, the function representing the population at each level increases, approaching gradually the exponential function.

When optimal tree is reached, a further depth increase is ineffective to modify significantly the Effort value. The tree structure changes with the depth increase, but only near the optimal depth. When this point is exceeded of two units, the tree structure remains substantially unchanged and the only effect of depth increase is the addition of a node at the top of the graph, and so the addition of a new root linked to the old one only.

Outdegree mean

Simulations of the function $\mathfrak{m}(l)$ reveal the behavioral change in population distribution respect to fixed depth.

The ratio between the number of nodes of two consecutive levels is always increasing with levels until the optimal depth is reached. Then this quantity tends to be decreasing with levels, as shown in figure 2.4.

This behavior is telling us that the effort is minimized when the outdegree mean is substantially constant with levels.

While number of nodes are allowed to be real number, levels are discrete quantities. We can then suppose that the minimum of functional $\mathtt{E}$ is somewhere between $\mathtt{L} = 10$ and $\mathtt{L} = 11$.

2.2.2 Varying number of leaves

In order to complete this analysis, we have to study how results change varying the fixed initial parameters. In this section, the abstractability is fixed $\alpha = 0.1$.

The parameter $\mathfrak{n}(0)$ is the one which governs the total number of nodes of the tree. The dependence between nodes and leaves is linear, while the number of nodes is an exponential function of nodes for optimal trees, as showed in 2.5. There is a small deviation caused by the discrete nature of levels, that is visible when the level approaches the root.

The optimal depth, the one which minimizes the Effort, moves with the number of leaves. The minimum grows while the number of leaves $\mathfrak{n}(0)$ grows too. The dependence is almost linear, discretized by levels, as showed in 2.6 at the left hand side.

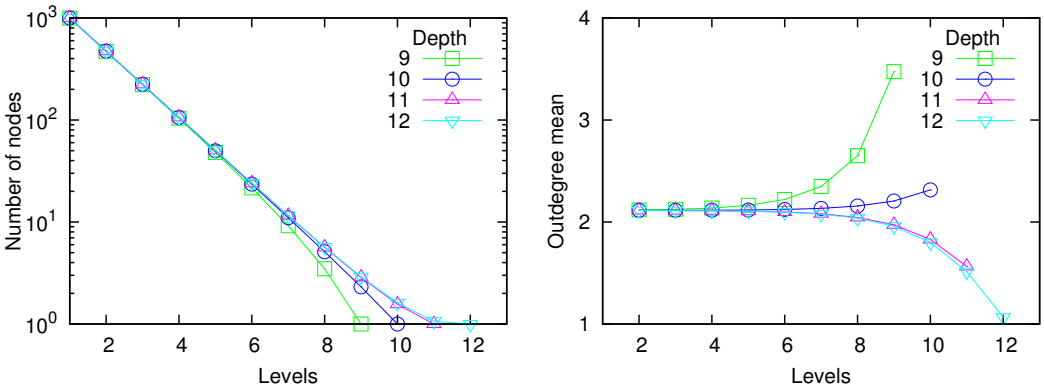

Figure 2.4: **Distribution of nodes and outdegree mean for different values of depth** - When the depth is optimal, $\mathtt{L} = 10$, nodes distribution is exponential, as showed in the left hand side figure. The right hand side shows that the optimal depth is the one for which outdegree vary as less as possible with levels. The costant function is often non accessible due to the fact that levels are discretized.

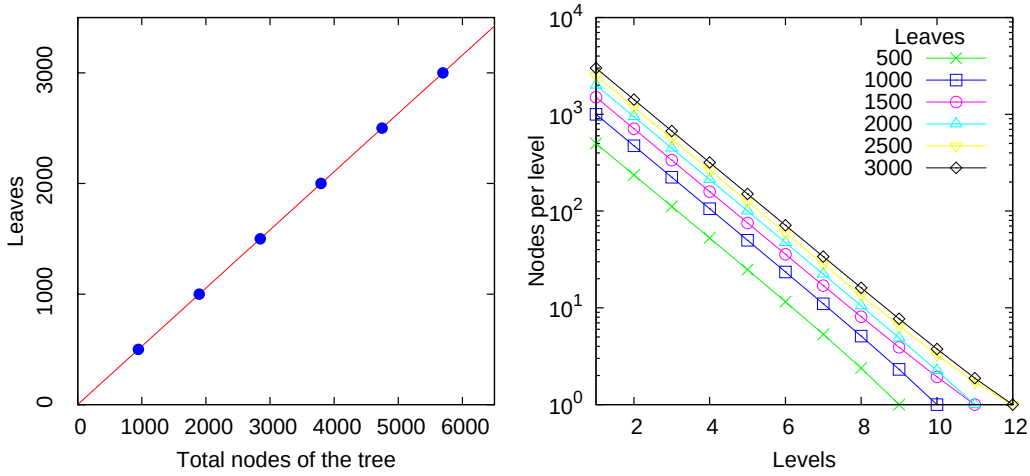

Figure 2.5: **Leaves and nodes of optimal trees** - The figure on the left hand side shows that the dependence between total nodes and fixed initial leaves is linear. The linear fit leads to $y(x) \sim x/2+2$. At the right hand side, populations of nodes at each level of optimal depth for different values of leaves.

Keeping $\alpha = 0.1$ fixed, and moving the number of leaves, we can observe the particular behavior of the outdegree mean as a function of levels showed at the right hand side of figure 2.6. The optimal tree is the one whose outdegree mean is constant respect to levels. When the constant function is not accessible, the nearest function is chosen as optimal. This is the reason for which the derivative of outdegree respect to levels is oscillatory as a function of number of leaves.

2.2.3 Varying abstractability α

The parameter α quantifies the portion of code that can be abstracted in higher levels, and so it represents the number of lines that nodes hold in common. In this section, the number of leaves is fixed $\mathfrak{n}(0) = 1000$.

Numerical simulations show that if α is high, the optimal depth is low and vice versa. The position of the minimal Effort respect to depth moves linearly with α, as showed in the left hand side of figure 2.7.

The outdegree mean increses with α, while its derivative remains oscillatory due to discrete levels. See figure 2.7.

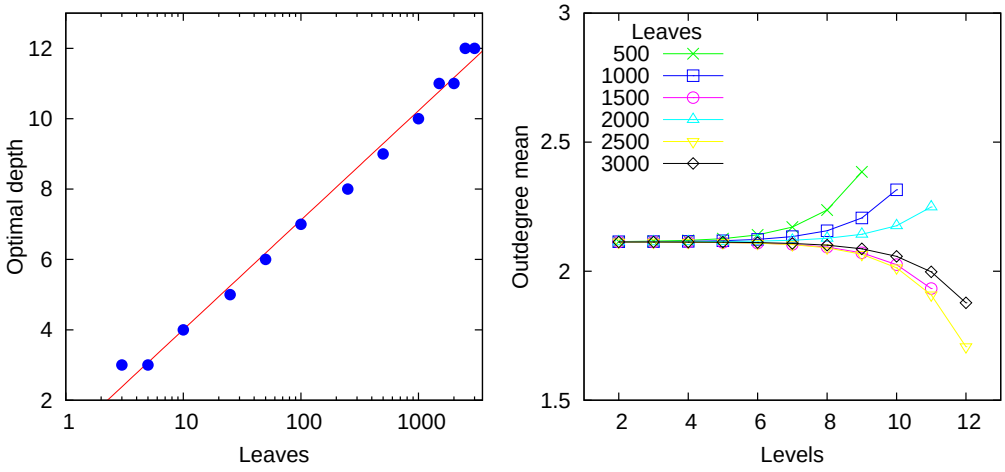

Figure 2.6: **Optimal depth and outdegree mean for different values of leaves** - The optimal depth increases with the number of leaves as $y(x) \sim x/1000 + 9$. At the right hand side, the outdegree mean as a function of levels for optimal depth.

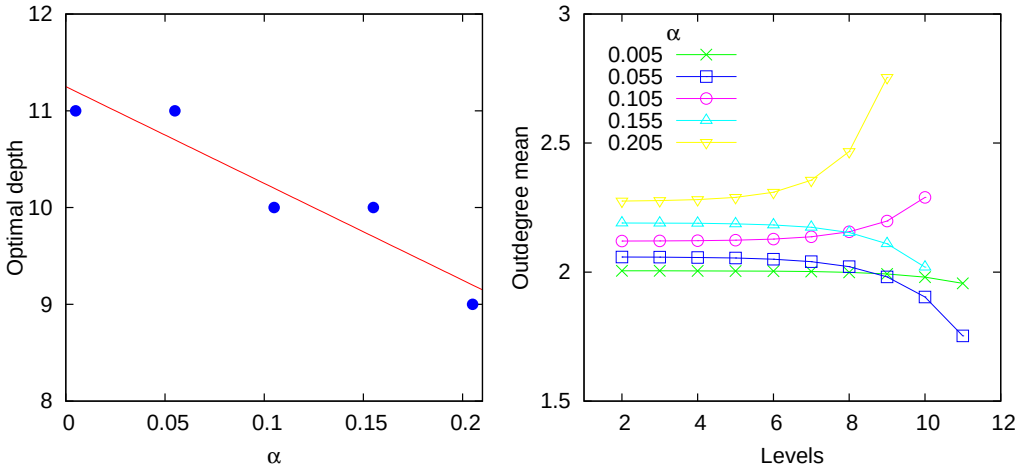

Figure 2.7: **Optimal depth and outdegree mean for different values of α** - The optimal depth decreases with the number of leaves as $y(x) \sim 11 - 10x$. At the right hand side, the outdegree mean as a function of levels for optimal depth.

Chapter 3

Sharing Tree model

The Minimal Effort model is a *mean field model*, needful to capture interesting behaviors in object-oriented programming. In addition, it is useful to introduce a *microscopic model*, the Sharing tree model, which gives you a method to create inheritance structures and so which allows the construction of Monte Carlo trees, for data comparisons.

3.1 Random objects

Our starting point is a set of $\mathcal{N}$ classes which represents the computer science solution of a given problem.

Consider each class composed by a set of different symbols randomly extracted from an alphabet. While the alphabet represents the whole writable code, each symbol represents a unit of code, as in the mean field model.

Each class is composed by a set of k symbols, and there are $\mathcal{S}$ symbols equally likely in the alphabet. This is a first approximation that allows you to find analytic results, but in principle not all units of code are equally important and used and not all classes have the same size.

The metric to measure the size of a code is an open question. The most popular method is *counting source line of code*, subgrouped in physical lines (the whole code) and logical lines (the number of statements), but it is not necessarily an honest method. The same task can be performed by two different programs of different sizes and with different statements and you cannot know in advance whether the best program is the largest or the smallest.

A reasonable starting point is to consider classes of the same size, represented by k, and a simplified alphabet where the $\mathcal{S}$ symbols are equally likely.

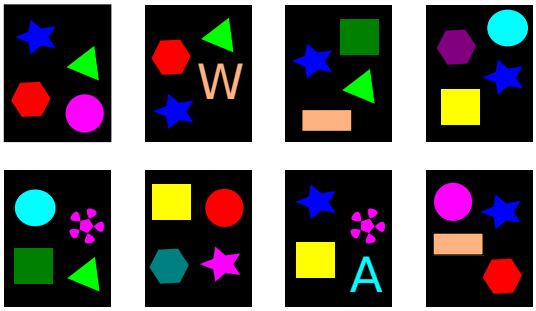

Figure 3.1: **Example of initial sets** - In this $\mathcal{N} = 8$ classes, only $\mathfrak{n} = 6$ contain the most common symbol, the blue star.

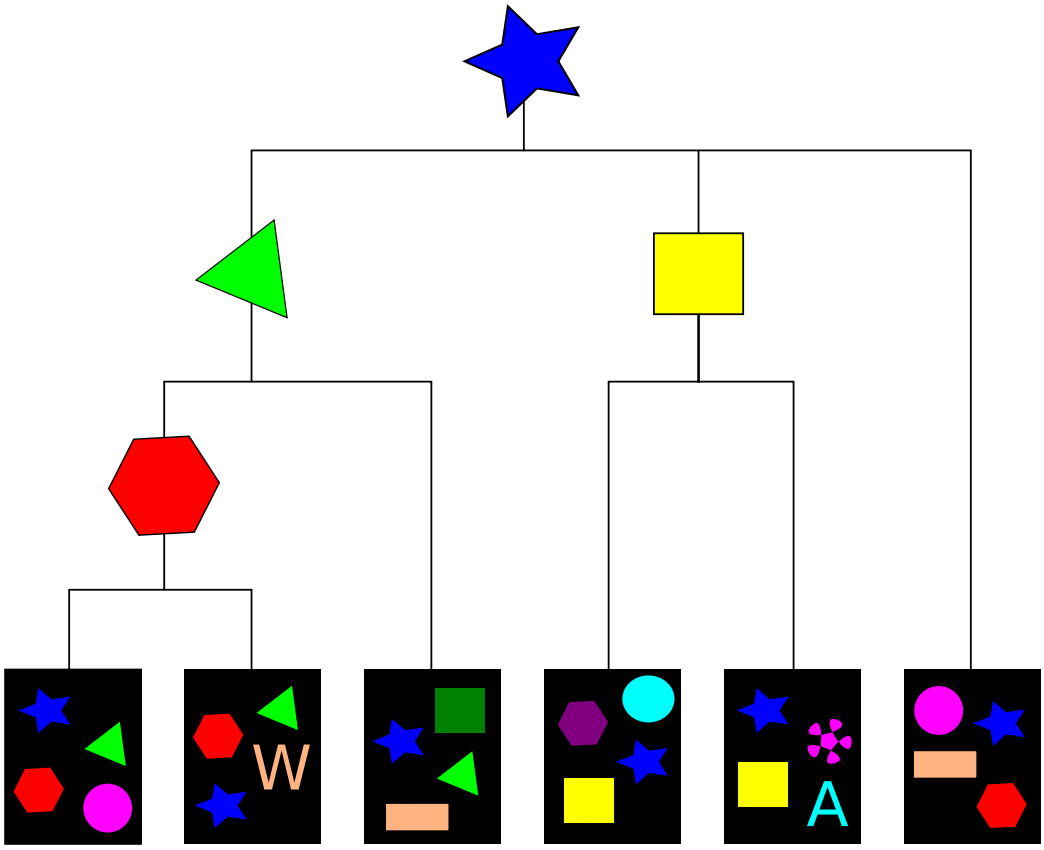

Figure 3.2: **Sharing Tree** - The most common symbol rule allows the creation of a tree. The $\mathfrak{n}$ classes represent the leaves, while each symbol chosen as the most common represents an internal node.

3.2 Building the structure

Once selected the $\mathcal{N}$ classes, rules are necessary to decide the structure of the tree. And so, how can we group classes in order to build a tree structure?

First, select the symbol that appears more often in the $\mathcal{N}$ classes. Keep classes with this symbol and delete the others. This symbol will be our root, and so the part of the code that all nodes of our tree hold in common. With the excluded classes one can in principle constructs other trees, but for now let just delete them.

The next step is the selection of the symbol that appears more often in the remaining $\mathfrak{n}$ classes. This symbol represents the second node of the tree, connected to the root only, for now.

If j classes contain this second symbol, we need to find the most common symbol in the $\mathfrak{n}-j$ remaining classes. The symbol is the third node, connected to the root. This process is iterated for all remaining classes until clustering is finished. The most common symbol groups all the classes in sets composed by different elements and with different number of element in each group. Furthermore, sets of one class are allowed: they are reminders of groupment, appearing once or even more times if the reminder is composed of more classes that hold nothing in common and for which the most common symbol rule does not work.

Each symbol used to group classes represents a node connected to the root, but it represents also a set of classes that have been grouped with such symbol.

When a group contains a single class, it means that its process is finished, but for each group with two or more classes the process can be iterated. The number of iterations necessary to finally arrive to all sets of one class only is $\mathtt{L}$, and it is exactly the depth of the tree. In fact, each step increases tree depth by one while it decreases the available symbols and the elements of each class by one (for this reason, $\mathtt{k}$ cannot be too small!).

Therefore, the process is finished when no group contains more than one class or when classes have no more symbols to be shared.

3.3 Most common symbol

The key concept underlying the mechanism that grows trees from classes is the *most common symbol rule*. Selecting the symbol that appears more often, we are applying an optimization principle: we are not interested in a random groupage, but in the best groupage in a certain defined sense.

It is not trivial to describe how the most common symbol is distributed,

but with some calculations we can obtain lots of interesting results that substantially solve the model.

Then, to be quantitatively and to find analytical solutions we must find the answer to a probabilistic question: what is the occurrence of the most common symbol in $\mathcal{N}$ sets of k draws from an alphabet of $\mathcal{S}$ symbols without replacement?

The answer to this question is central for the model. Each groupage done to build the tree, follows the very same most common symbol rule. The analytic solution that we are looking for can be applied iteratively (with opportune parameters) in order to describe the distribution of the number of classes in each set and at each step. In a certain number of steps, sets become singletons and the process is over.

3.3.1 Find a symbol in a sequence

Unlike the choice made in the Minimal Effort model, Sharing Tree model symbols cannot be repeated in a sequence. Such refinement simplifies the following calculation but it also approaches the model to real classes in which code repetition does not have a well defined sense.

Thus, the probability to find a selected symbol in a sequence is given by the hypergeometric distribution

$$p = \frac{\binom{\mathcal{S}-1}{\mathsf{k}-1}}{\binom{\mathcal{S}}{\mathsf{k}}} = \frac{\mathsf{k}!(\mathcal{S}-1)!}{\mathcal{S}!(\mathsf{k}-1)!}$$

3.3.2 How many classes for each symbol?

In how many classes does such symbol appear? The answer is given by the Binomial distribution of $\mathcal{N}$ trials in which the symbol can appear or not with probability p.

The Binomial distribution then describes the occurrence w of each symbol in the $\mathcal{N}$ sequences. We can compute the probability that w classes contain a selected symbol with

$$Pr(w) = \binom{\mathcal{N}}{w} p^w \, (1-p)^{\mathcal{N}-w}$$

3.3.3 What about the most common symbol?

In the Sharing tree model, optimization is carried by the choice of the most common symbol, and not any one of them.

Let's say that each symbol $s \in \mathcal{S}$ appears in w_s classes. The set $\{w_s\}_{s=1}^{\mathcal{S}}$ contains $\mathcal{S}$ random variables independent and identically distributed, and the most common symbol is the one that appears ω times

$$\omega = \max\{w_1, \ldots, w_{\mathcal{S}}\}$$

In order to find the distribution of the occurrence of the most common symbol, it is necessary to do some calculations. Therefore, consider the cumulative distribution function of ω

$$F_\omega(y) = Pr(\omega \leq y)$$

Since ω is the maximal occurrence and the w_s are independent, it holds that

$$\begin{aligned} Pr(\omega \leq y) &= Pr(w_1 \leq y, w_2 \leq y, \ldots, w_{\mathcal{S}} \leq y) \\ &= Pr(w_1 \leq y) Pr(w_2 \leq y) \ldots Pr(w_{\mathcal{S}} \leq y) \end{aligned}$$

and since all w_s have the same cumulative mass function, we can write

$$F_\omega(y) = F_w^{\mathcal{S}}(y)$$

The probability distribution of ω can now be obtained as the probability that $\omega \leq y$ minus the probability that $\omega \leq y - 1$, and so

$$\begin{aligned} Pr(y = \omega) &= Pr(\omega \leq y) - Pr(\omega \leq y - 1) \\ &= F_w^{\mathcal{S}}(y) - F_w^{\mathcal{S}}(y - 1) \end{aligned}$$

3.3.4 Probability distribution of most common symbol

The occurrence of the most common symbol now is straightforward. Remembering that w_s are binomial random variables, the occurrence ω of the most common symbol is therefore distributed as

$$\Psi(\omega) = \left(\sum_{i=0}^{\omega} \text{Bin}(\mathcal{N}, i) \right)^{\mathcal{S}} - \left(\sum_{i=0}^{\omega-1} \text{Bin}(\mathcal{N}, i) \right)^{\mathcal{S}}$$

Making explicit the formula of the Binomial distribution, we have finally obtained the distribution of the occurrence of the most common symbol

$$\Psi(\omega) = \left(\sum_{i=0}^{\omega} \binom{\mathcal{N}}{i} p^i (1-p)^{\mathcal{N}-i} \right)^{\mathcal{S}} - \left(\sum_{i=0}^{\omega-1} \binom{\mathcal{N}}{i} p^i (1-p)^{\mathcal{N}-i} \right)^{\mathcal{S}}$$

Examples of this distribution for different parameters are shown in figures 3.3 and 3.4.

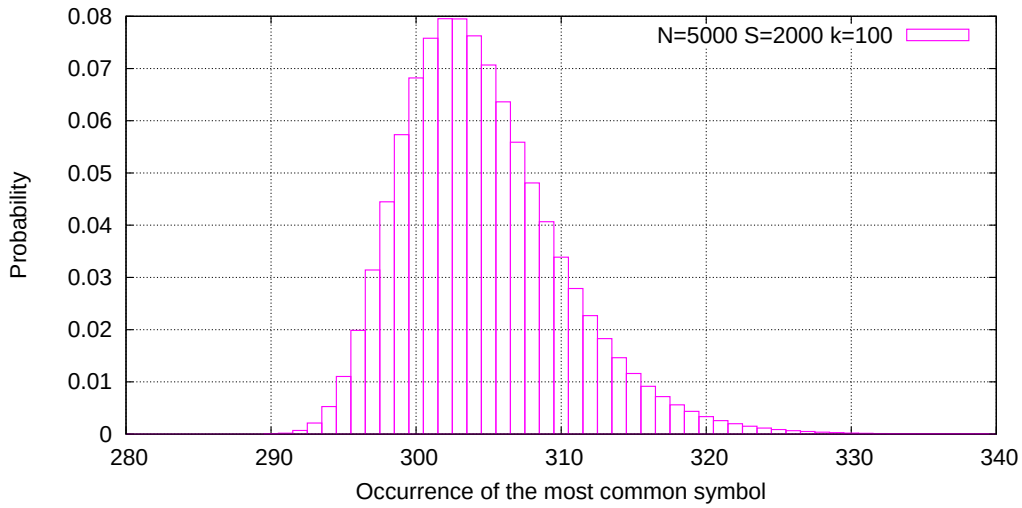

Figure 3.3: **The function** $\Psi(\omega)$ - This figure represent the probability distribution of the occurrence of the most common symbol in 5000 sequences of 100 extracted symbols (without repetitions) from an alphabet of 2000 symbols.

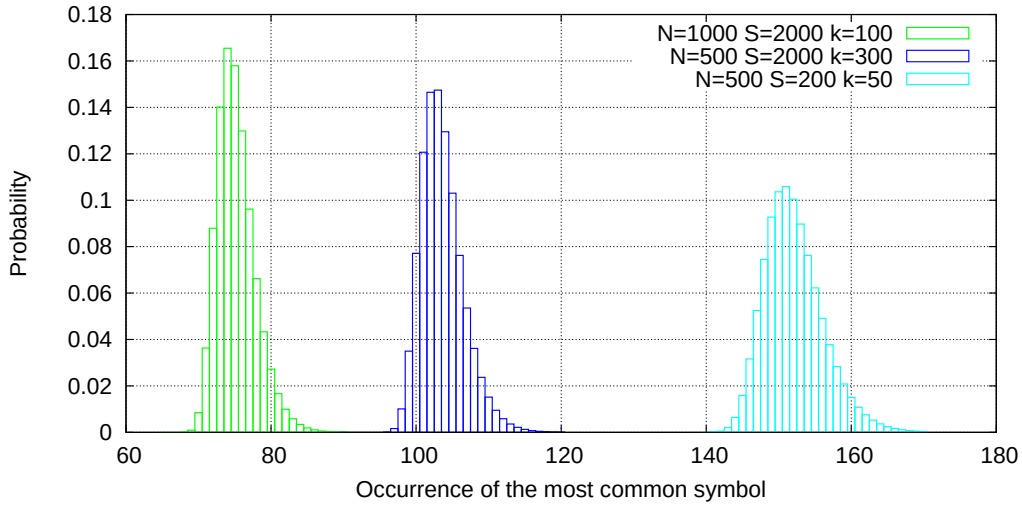

Figure 3.4: $\Psi(\omega)$ **for different parameters** - Some examples of the probability distribution of the occurrence of the most common symbol for different parameters.

3.3.5 Mean occurrence of the most common symbol

To make the calculations less demanding, often it is useful to deal with the mean instead of the whole distribution. We can obtain $\langle\omega\rangle$ expanding

$$\langle\omega\rangle = \sum_{\omega=1}^{\mathcal{N}} \omega\Psi(\omega)$$

To obtain the final simplified formula, some calculations are needed and in order to ease the notation, we can define a_ω as

$$a_\omega = \left(\sum_{i=0}^{\omega} \binom{\mathcal{N}}{i} p^i (1-p)^{\mathcal{N}-i}\right)^{\mathcal{S}}$$

and then rewrite $\Psi(\omega)$ as

$$\Psi(\omega) = a_\omega - a_{\omega-1}$$

Inserting the formula in the mean occurrence of the most common symbol, we have

$$\langle\omega\rangle = \sum_{\omega=1}^{\mathcal{N}} \omega\left(a_\omega - a_{\omega-1}\right)$$

which is a telescopic series, that allows an important simplification

$$\begin{aligned}\langle\omega\rangle &= 1\left(a_1 - a_0\right) + 2\left(a_2 - a_1\right) + \cdots + \mathcal{N}\left(a_{\mathcal{N}} - a_{\mathcal{N}-1}\right) \\ &= \mathcal{N}a_{\mathcal{N}} - a_0 - a_1 - \cdots - a_{\mathcal{N}-1}\end{aligned}$$

Furthermore, considering that $a_{\mathcal{N}}$ is the sum over all possible cases of a probability, it is equal to 1. Then we finally obtain

$$\langle\omega\rangle = \mathcal{N} - \sum_{j=0}^{\mathcal{N}-1} a_j$$

Thus, the mean of the distribution of the occurrence of the most common symbol is explicitly

$$\langle\omega\rangle = \mathcal{N} - \sum_{j=0}^{\mathcal{N}-1} \left(\sum_{k=0}^{j} \binom{N}{k} p^k (1-p)^{\mathcal{N}-k}\right)^{\mathcal{S}}$$

With this formula, we can study directly how parameters change the mean of the number of leaves (but not of total nodes) of simulated trees, as shown in figure 3.5. Considering that, at each step, the groupage of classes in the process is made with the same rule, the formula for $\langle\omega\rangle$ can be applied iteratively to find the mean of the number of classes in each set and for each step of the recursive groupage.

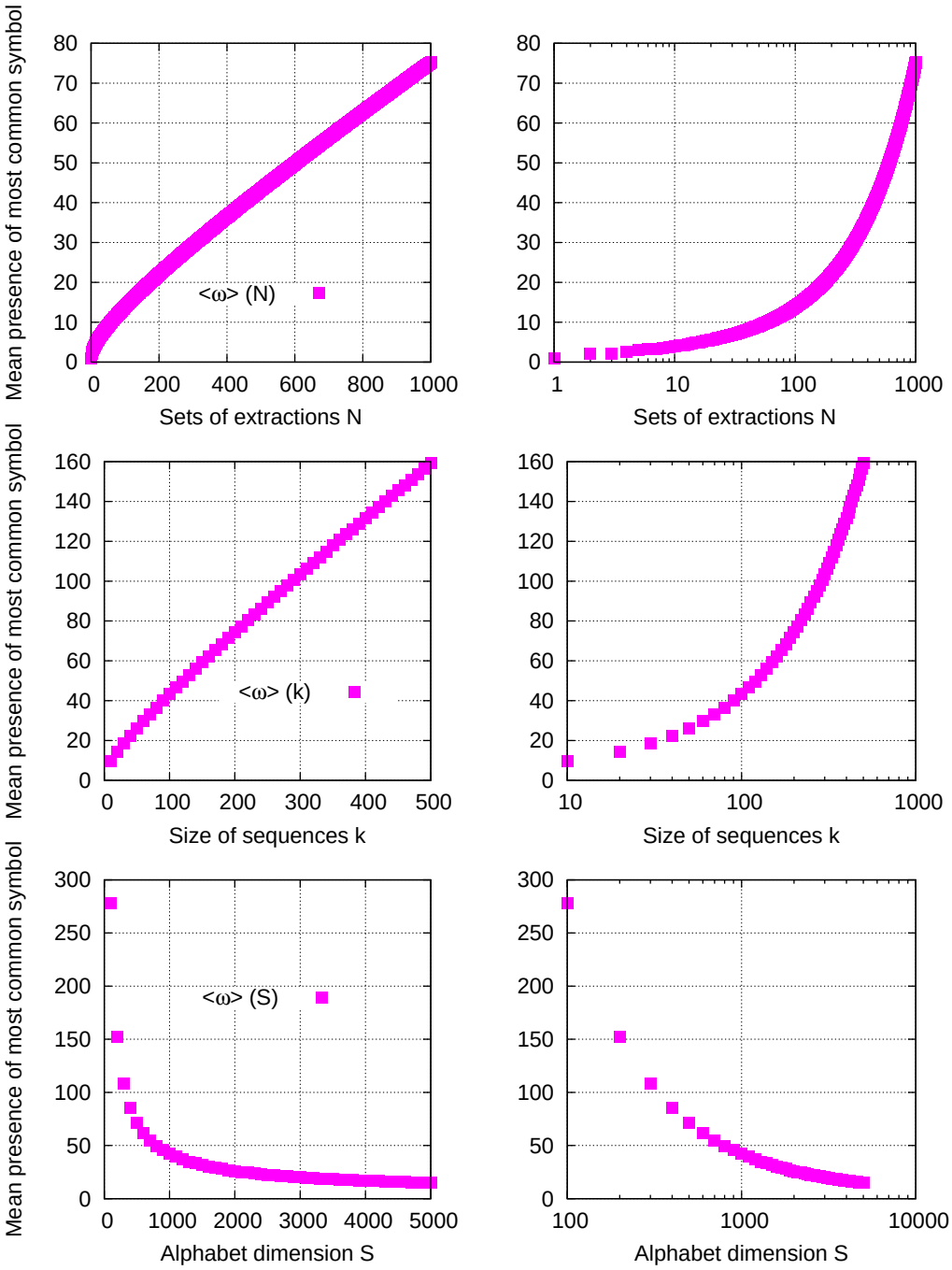

Figure 3.5: **The mean occurrence** $\langle\omega\rangle$ - In the first figure, $\mathcal{S}$ and k are fixed at $\mathcal{S} = 2000$ and $\mathsf{k} = 100$ while and $\langle\omega\rangle$ is plotted as a function of $\mathcal{N}$. In the second figure $\mathcal{N} = 500$ and $\mathcal{S} = 2000$. In the third one, $\mathcal{N} = 500$ and $\mathsf{k} = 50$.

3.4 Sharing process

The recipe to build the tree can be seen also as the set of rules defining a process.

At each step, the recipe suggest to look for the symbol that appear more often and to group classes inspired by *most common symbol rule*.

Following the iteration, the number of nodes are $x_0 = \mathcal{N}, x_1, \ldots, x_{\mathtt{L}} = 1$. Every time we divide classes in *with* or *without* the most common symbol, the process generate η subprocesses, with η the minimum number of subgroups necessary to cover all available classes.

Consider only the main process, which starts from $\mathcal{N}$ classes and which divides them into 2 subgroups, iteratively until the $x_{\mathtt{L}}$ is reached. All other subprocesses behave in the same manner with the opportune initial condition.

The mean number of elements of a group at each step t is given by

$$f(x_t, t) = x_t - \sum_{j=0}^{x_t - 1} \left(\sum_{i=0}^{j} \binom{x_t}{i} \Pi_t^i \left(1 - \Pi_t\right)^{x_t - i} \right)^{\mathcal{S}-t}$$

where Π_t is obtained with the hypergeometric distribution and considering that if a symbol has been used as *the most common* then it cannot be reused, and so at each step $\mathcal{S} \to \mathcal{S} - 1$ and $\mathsf{k} \to \mathsf{k} - 1$.

$$\Pi_t = \frac{\binom{\mathcal{S}-1-t}{k-1-t}}{\binom{\mathcal{S}-t}{\mathsf{k}-t}} = \frac{\Gamma(\mathcal{S}-t)\Gamma(\mathsf{k}-t+1)}{\Gamma(\mathsf{k}-t)\Gamma(\mathcal{S}-t+1)}$$

The process for the mean number of elements can so be defined as

$$x_{t+1} = f(x_t, t)$$

where $x_0 = x(0)$ and is equal to $\mathcal{N}$ for the main process.

3.5 Monte Carlo simulations

What kinds of trees arise from Sharing Tree Monte Carlo simulations? To give an answer to this question, I have written a C++ code and I have integrated it with the data analysis library. The program is well optimized and fast and each tree is obtained in few seconds or less for all parameters that have been chosen.

In this section the simulations and the study of the behavior of trees are presented for different parameters.

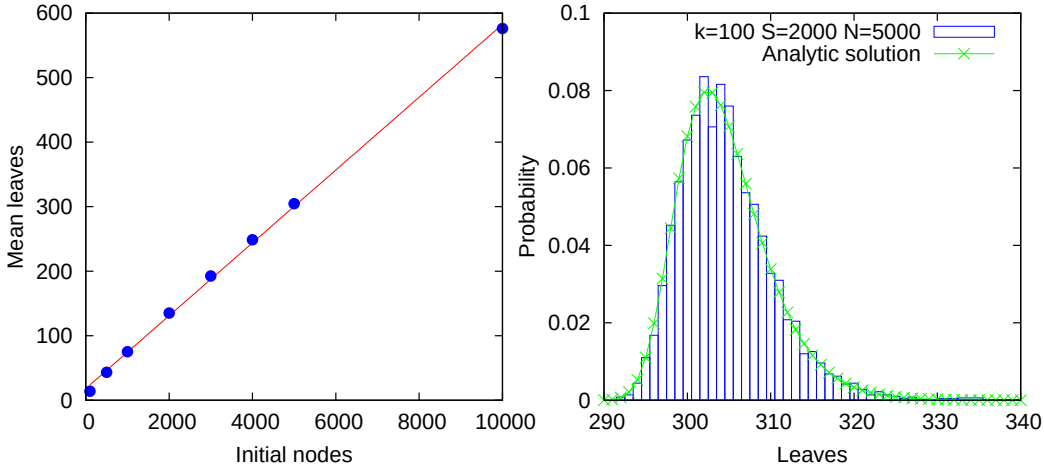

Figure 3.6: **Initial nodes and number of leaves** - The mean number of leaves is a linear function of the initial number of nodes. The fit leads to $y = 0.06x + 18$ when $k = 100$ and $S = 2000$. The function $\Psi(\omega)$ explains perfectly the leaves distribution.

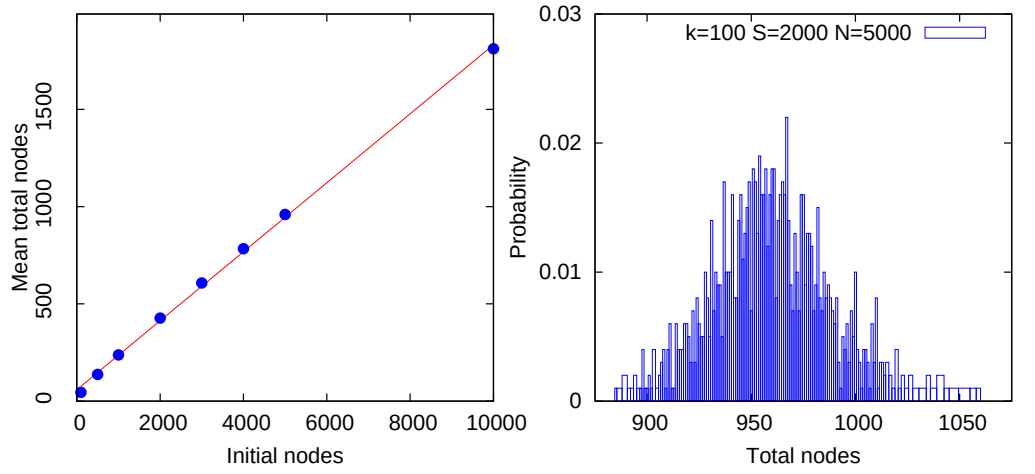

Figure 3.7: **Total nodes for different initial nodes** - While the distribution of total nodes is spread, its mean is linear in the number of initial nodes, with $y = 0.2x + 58.7$ when $k = 100$ and $S = 2000$.

3.5.1 Initial nodes and leaves

This model does not allow you to fix the number of leaves of a tree, but only the number of initial nodes, among which leaves will be selected. Once fixed the $\mathcal{N}$ initial nodes, the tree will be built with a number of leaves whose dependence from this parameter has been obtained in the previous section and showed in figure 3.5.

In the range in which simulations were made, the mean number of leaves $\mathtt{n}$ is substantially linear respect to the number of initial nodes $\mathcal{N}$, as shown on the left hand side of figure 3.6. This means that we can easily decide the mean number of leaves of simulated trees multiplying the number of initial nodes by the factor obtained from the linear fit.

The simulated distribution perfectly matches the analytic solution for the distribution of the most common symbol, as shown on the right hand side of figure 3.6, where the function $\Psi(\omega)$ exactly overlap the distribution of the number of leaves once fixed $\mathcal{N}$, $\mathcal{S}$ and $\mathtt{k}$.

3.5.2 Total number of nodes

The number of nodes in a tree consists in the number of leaves increased by the number of symbols used by the process. Due to the random nature of the number of leaves, the total number of nodes in a tree is usually a spread distribution.

Anyway, the mean is a linear function of the initial nodes and this allows to easily control the mean number of nodes of trees.

3.5.3 Trees depth

The depth $\mathtt{L}$ represents the number of steps in which all sets of classes, or sequences, have become singletons. In simulations, it has a sharp distribution, as showed on the right hand side of figure 3.8.

It is not surprising, since the number of classes in sets is strongly suppressed by each step of the process, with a small variance that decreases as a function of the number of classes, as deducible from figures 3.5 and 3.4.

The mean depth is a logarithmic function of initial nodes, as shown on the left hand side of figure 3.8. Since initial nodes are a linear function of total nodes, the logarithmic behavior of the depth is exactly the same result obtained from the Minimal Effort model. This is an optimal behavior for comparisons with data.

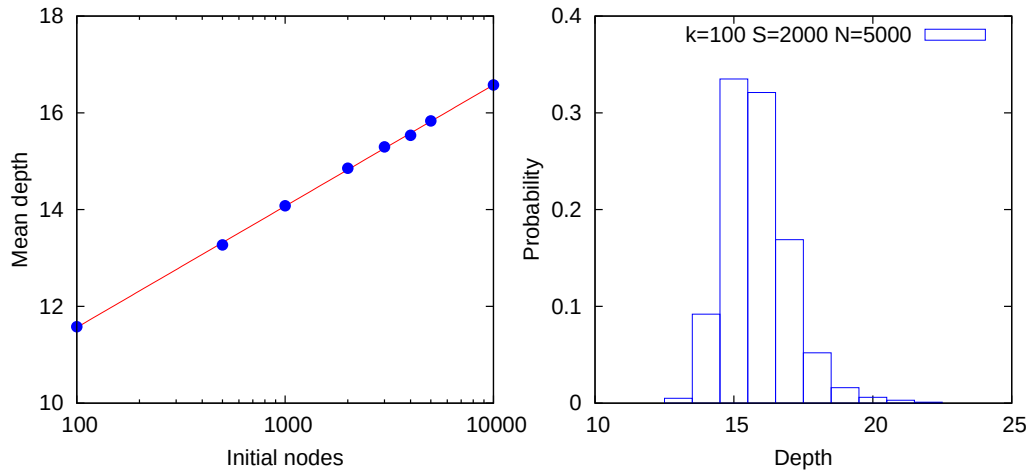

Figure 3.8: **Depth for different initial nodes** - Depth of trees has a sharp distribution, and the mean is a logarithmic function of the initial nodes, as $y = 1.1\log(x) + 6.6$.

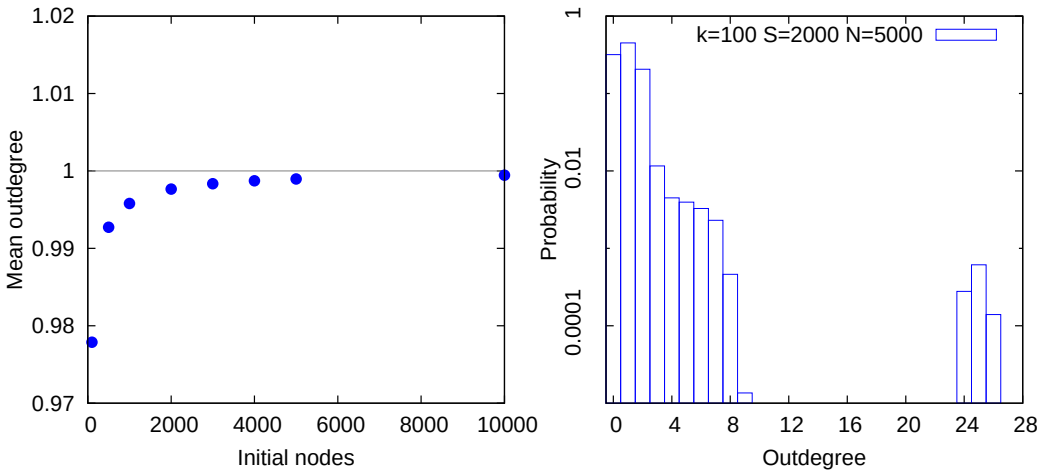

Figure 3.9: **Outdegrees for different initial nodes** - The mean of outdegrees approaches 1 for increasing number of initial nodes, while outdegrees distribution show a bizarre sharp peak for high outdegree.

3.5.4 Outdegrees

The outdegrees distribution shows an interesting behavior. While its mean approaches the value of 1 for increasing number of initial nodes, as shown on the left hand side of figure 3.9, the distribution of outdegrees is concentrated in low outdegree, but with a sharp peak of occurrence for high outdegree, as shown on the right hand side of 3.9.

This means that some nodes share their code with lots of nodes, or in model terminology, that few symbols are used to group lots of nodes that have little more in common.

The gap in the distribution of outdegrees between the *normal nodes* and *special nodes* gradually becomes more and more pronounced while the number of initial node becomes higher, as detailed shown in figure 3.10.

This behavior can be investigated more closely studying the outdegree as a function of some variable, and this will be done in next section.

3.5.5 Outdegree mean per level

A useful description of tree topology is the outdegree mean as a function of levels. In the Minimal Effort model we have seen how this function couples with the depth and that it shows a growth close to the root, strong or weak depending on the depth of the tree.

The microscopic rules of the Sharing Tree model lead to a very similar behavior. As shown on the top of figure 3.11, simulations build trees with an outdegree mean which grows with levels and which has a strong growth near the root.

Parameters govern the position and the height of peaks. In the bottom of figure 3.11, k and $\mathcal{S}$ are fixed, while $\mathcal{N}$ varies. The increase of the number of initial nodes moves the peak, and the depth, to higher levels and to higher outdegrees mean. More nodes need to been grouped, but they does not have many symbols in common.

On the top of figure 3.12, the varying parameter is k. When k is small, the depth is small since few symbols are available for groupage. When k grows, the height of the peaks does not change since it is coupled only with the number of leaves, and so with the number of initial nodes.

On the bottom of figure 3.12, $\mathcal{S}$ is varying. The depth change with $\mathcal{S}$ and peaks move slowly while the number of possible symbols in $\mathcal{S}$ increases.

The behavior of outdegree mean as a function of level can be changed through the probability of each symbol in the alphabet. In this chapter, all symbols in alphabet are equally likely, but if some symbols would have more probability to be chosen in the extractions respect the others, we may expect

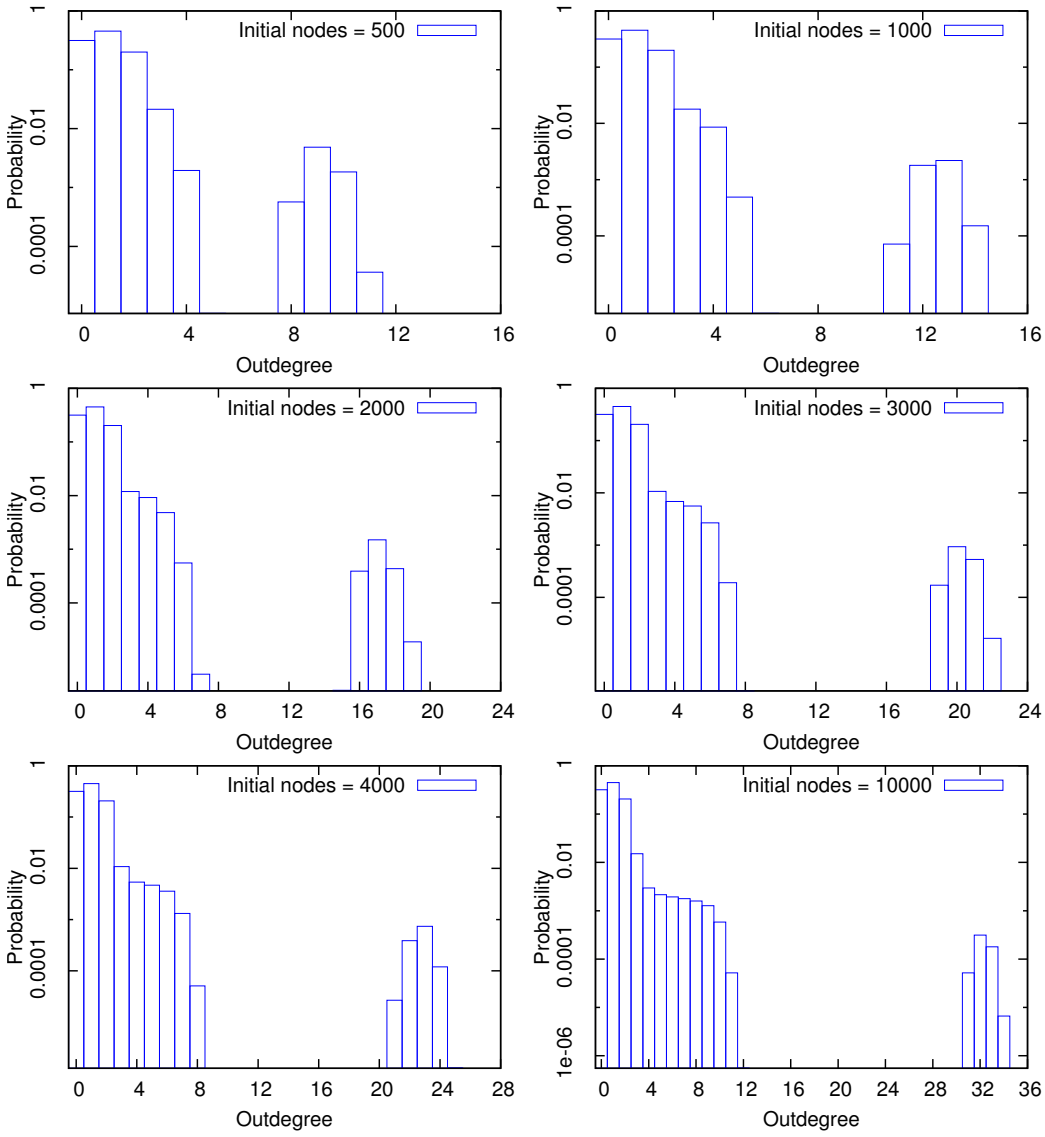

Figure 3.10: **Outdegrees distribution for different initial nodes** - The gap in the distribution of outdegrees becomes more and more pronounced while the number of initial node becomes higher.

a decline of peaks for higher depths. If there are few symbols with which we can initially group classes, then the outdegree mean of high levels will be lower, at least for one level. This argument will be useful in data analysis.

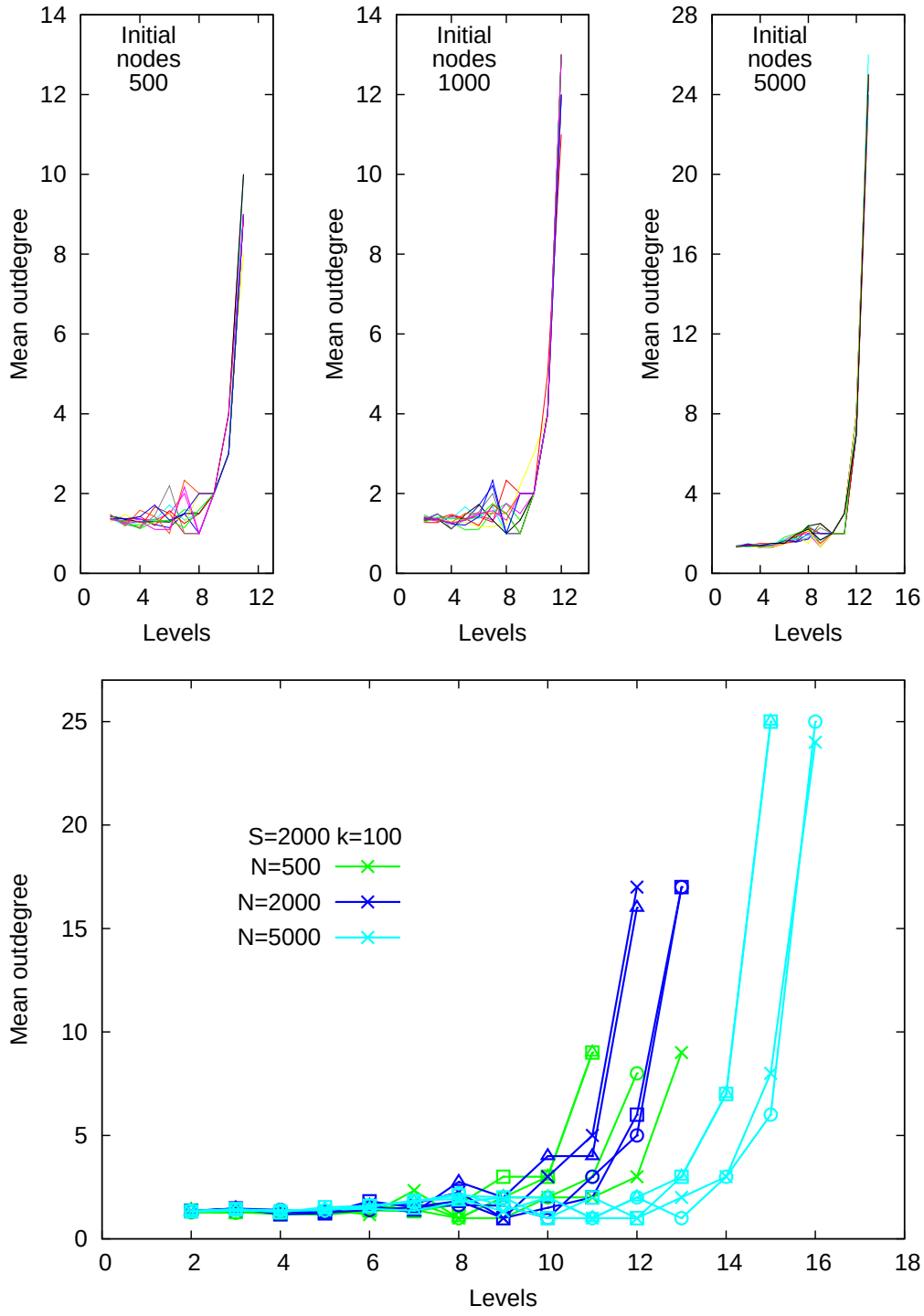

Figure 3.11: **Outdegree mean per level** - On the top, the figure shows some examples of outdegree mean as a function of levels for different initial nodes and with fixed depth. The figure on the bottom shows how the parameter $\mathcal{N}$ can change the simulated trees.

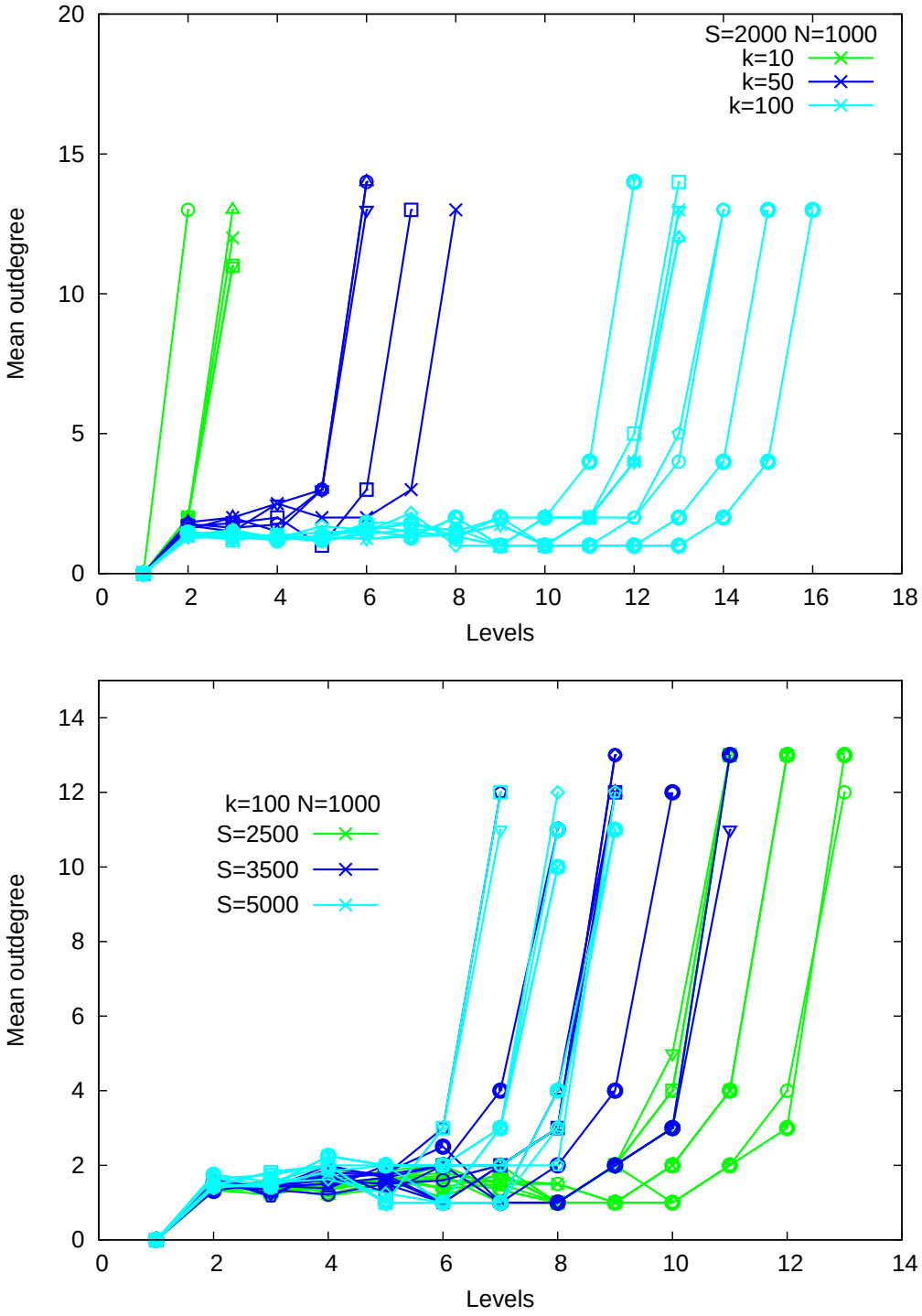

Figure 3.12: **Outdegree mean per level** - Figures shows how parameters move the main quantities in simulated trees, as the depth and outdegrees distribution.

Chapter 4

Data analysis

How do programmer use inheritance? What kinds of graphs appear from hierarchical relations? In this chapter we analyze real hierarchical structures made by the Internet programmer community, among which appear users like Google, Facebook, Twitter and Microsoft [MWe].

4.1 Dataset

The dataset is composed by a huge amount of projects downloaded from GitHub, the actual largest code host on the web [GVSZ14]. GitHub offers to its over 9.1 millions users [Git], source code management (directly from the program Git [Lin]) which allows a cooperative programming among all users.

Inheritance hierarchies have been obtained directly from packages through different steps and with a particular attention to possible data bias and programs bugs.

4.1.1 10 millions of hierarchies

Packages have been downloaded in November 2014, and they have been chosen with a research by name (the whole alphabet) and by programming language (C++, Java, Python).

In detail, the dataset studied in this thesis contains:

- 17333 C++ projects (3233447 hierarchies)
- 25318 Java projects (3504681 hierarchies)
- 20010 Python projects (2491603 hierarchies)

Almost 10 millions of inheritance hierarchies!

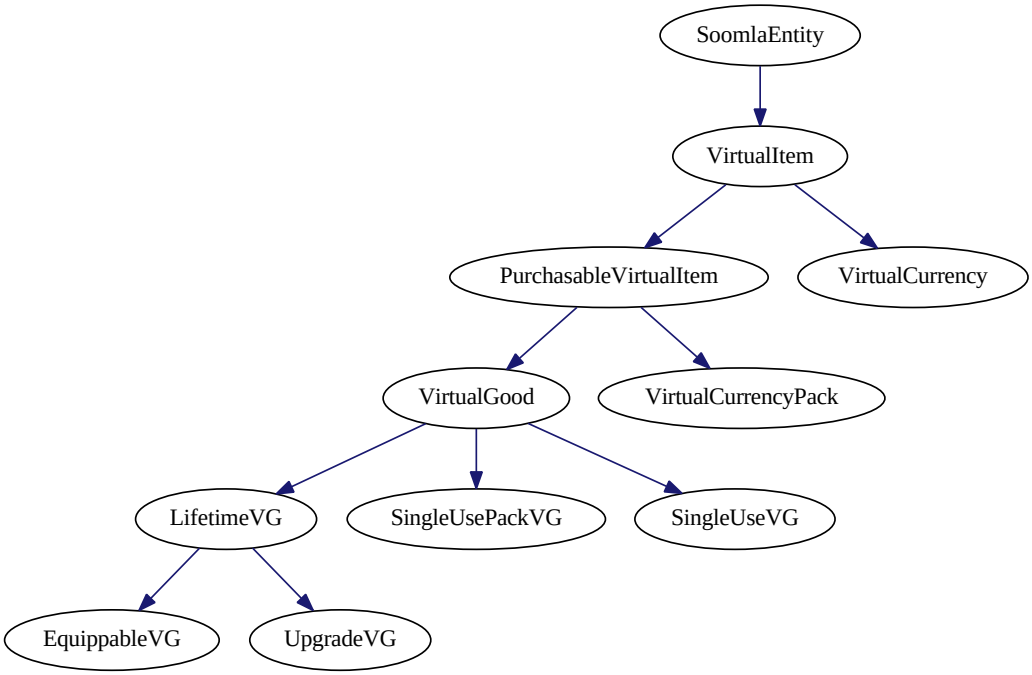

Figure 4.1: **Example of hierarchical inheritance structure** - This is one of the graphs contained in the project Soomla Cocos2dx.

4.1.2 Data conversion

Hierarchies reconstruction have been done by a series of scripts (see A.3 for examples) and with the aid of Doxygen [vH], a program for code documentation.

Doxygen is not written with the goal of reconstructing inheritance hierarchies, but it gives you the chance of obtaining, for each class, a graph which reconstructs the hierarchy of the class *upward* (any sister-classes are not included in graph). Cutting and pasting all the pieces of graphs, one can rebuild the whole hierarchy. This step has been done with the aid of gvpr, an *awk* [Bri] inspired programming language useful for graphs manipulation.

Once obtained the hierarchies in the standard graphs format (.dot), finally the data have been converted in adjacency lists.

To analyze adjacency lists I have written a C++ library (see A.1) that allows the characterization of graphs (to calculate depth, to find roots and leaves, to assign levels to each node, to find eventually nodes not connected, ...) and allows to obtain all principal statistical quantities, distributions and scatterplots.

To complete the analysis, I have written some C and bash scripts and lots of Gnuplot scripts to graphical the representation of the results.

4.1.3 Templates and bias

Analyzing converted data I came across some problems about their structure and composition.

For example, Doxygen consider as inheritance even the relation among a template class and all instances of its template variable. Since I don't consider such relation as inheritance, I wrote a script to exclude fake links among classes.

The second important problem was about data redundancy. Due to the sharing nature of open source code, it often happens that some big libraries are contained in slightly different versions in many packages. I can't exclude all redundancy from data, but I made some general controls to exclude packages that caused the most obvious bias. Instead, I considered good packages those whose libraries are not comparable to the naked eye, since they contain information useful for my analysis.

4.2 Graphs sizes

The first important approach to noisy data is the sizes distribution. Graphs sizes and complexity is in some way coupled with the complexity

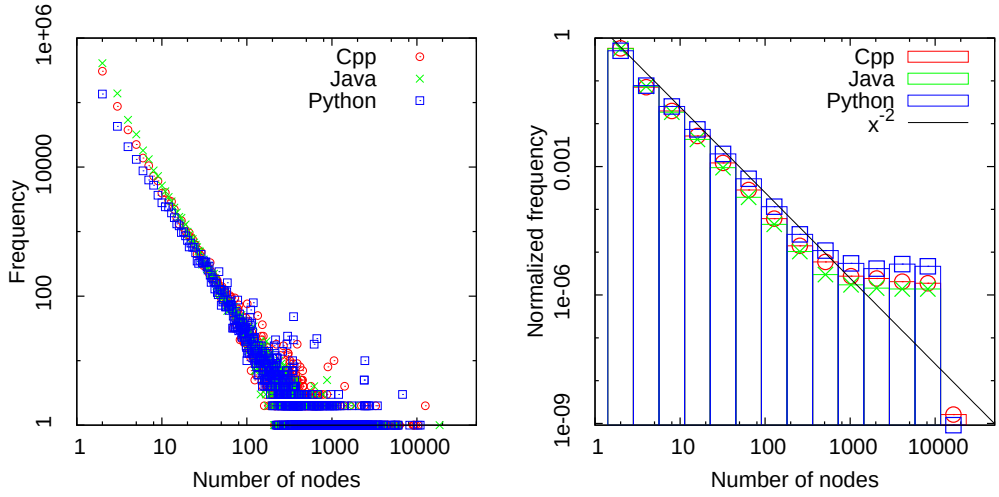

Figure 4.2: **Distribution of the number of nodes in hierarchies** - The number of nodes composing the graphs has a power law distribution. The exponent $\sim x^{-2}$ is universal despite differences between the three languages.

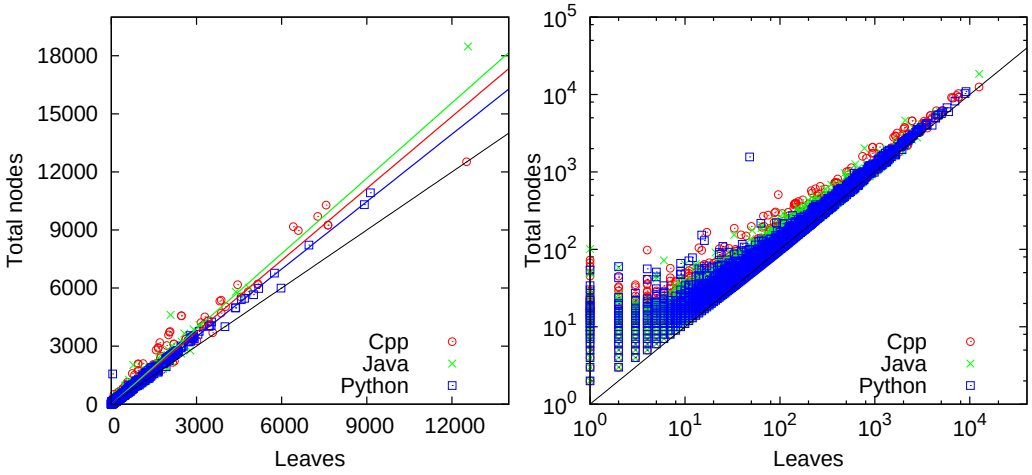

Figure 4.3: **Leaves and total nodes** - If a point (representing a graph) is near the black line, it means that almost all nodes of the graph are leaves, and so that only few nodes contain the shared code, while lots of classes inherit it.

of problems for which the whole set of programs is made for. In this thesis, I am not investigating the *problems space*, that is true head of such distribution, but the whole following analysis will feel the consequences of sizes distribution.

The size of a directed acyclic graph can be defined in different ways. For our purposes, it is important to consider two main quantities: the number of nodes (and so of classes) which is the effective representation of hierarchy purpose, and the depth of graphs, which is the amount representing the hierarchies optimization.

4.2.1 Solutions sizes

Consider each hierarchy as the solution for a computer science problem, or task. How many classes are necessary to solve problems?

First of all, we analyze the total number of nodes of each graph (i.e. the number of classes of each hierarchy), the most important quantity to define the size of structures.

Plotting this quantity for each graph for each package, as showed in figure 4.2, we find a power law distribution with a small exponent ($\sim x^{-2}$).

Despite the differences among C++, Java and Python, this behavior is universal: hierarchies sizes are distributed in the same way, regardless the language in which they are built. This is the first interesting result that arise from the analysis: in principle, programming languages can be used to perform different tasks and they are designed in different ways and for different purposes. This seems not to influence the structure of inheritance hierarchies, at least in sizes distribution.

4.2.2 Sharing classes

Not all classes of a hierarchy are directly used in programs: for example, abstract classes cannot be instantiated, and in general some classes can be built for sharing code purposes, abstracting concepts in *super classes*.

While, in first approximation, we cannot know if an internal node is used or not, we can be sure that the leaves of each hierarchy are instantiable and that such classes are made to solve the task.

How many are the leaves compared to the total number of classes? How many classes share their code?

The answer is shown in figure 4.3: the vast majority of classes in each hierarchy are leaves, i.e. classes that can be used in programs.

This result give us an important clue to argue that inheritance graphs take place only for high abstractability. If almost all nodes are leaves, as

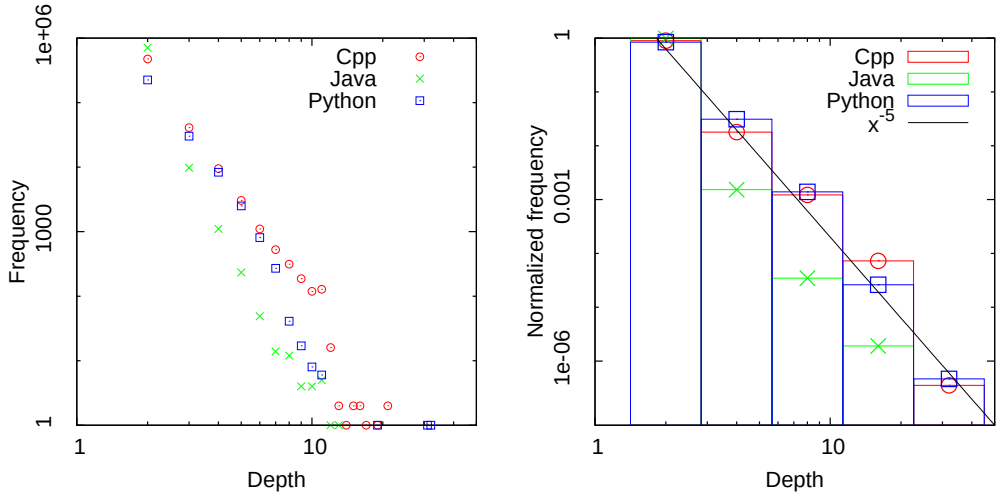

Figure 4.4: **Distribution of depth of each hierarchy** - While C++ and Python have the same power law distribution ($\sim x^{-5}$), Java structures are systematically less deeper than other languages.

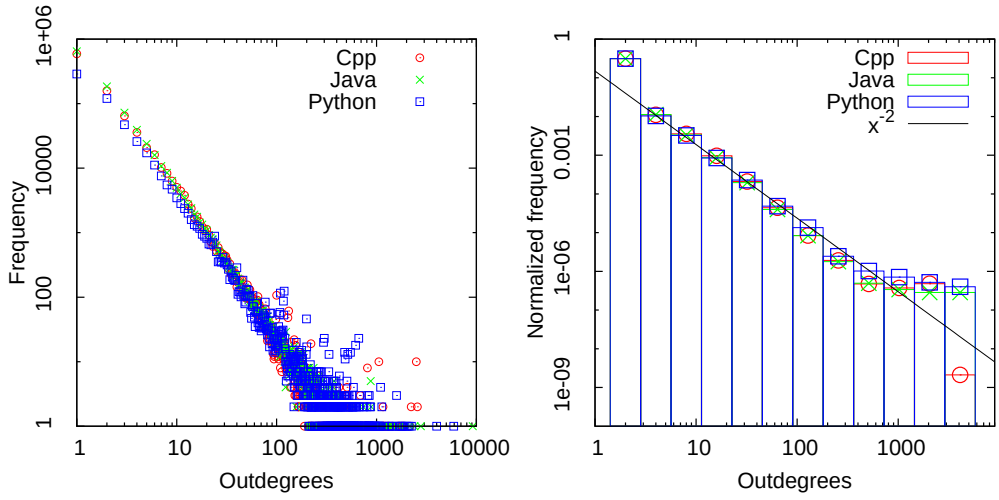

Figure 4.5: **Distribution of outdegrees of all nodes of all graphs** - The outdegree quantifies the number of classes that inherits code from the same class. Its power law behavior can be caused by the power law distribution of the number of nodes of each graph, showed in figure 4.2.

happens in the data, only few nodes are responsible of the collection of all the code that can be shared, while many nodes inherit it. The shared code must be useful for lots of classes, and lots of classes must hold something in common.

This behavior makes us also to think about a general small depth of graphs, and in next section we will check it.

4.2.3 Depth of hierarchies

Hierarchies depth is a surprisingly small quantity: also the largest structures have at most eleven or twelve levels, while the vast majority of graphs have a depth equal to two or three. This result is the second clue that supports the idea for which abstractability is high in all real hierarchies.

Another important result given by figure 4.4 is that Java language does not behave as C++ and Python: depths distribution of Java hierarchies is consistently below the other two distributions. The reasons of this discrepancy can be found directly in Java inheritance implementation rules. In Java, multiple inheritance is allowed only by a special kind of relation given by the keywords *interface* and *implements*. *Interface* is the kind of class that can be inherited without limitations about the multiple inheritance, but *implements* is the keyword that the class which inherits the interface must have: an interface cannot contains *implementation*, but only declarations of methods and functions. The reasons for which a class inherits from interfaces is code reuse as well, but not in the sense expressed by models in sections 2 and 3.

This and other features have been inserted in Java language in order to obtain a simpler and more safe programming. We can expect other differences in data between Java and the other two languages. We can say that Java inheritance rules give rise to less deeper hierarchies respect to other languages (especially C++ from which Java is developed), and in this sense they have a simplified effect as hoped by Java designers.

4.3 Code shareability

The outdegree quantifies the number of classes that inherit code from the same class. The power law distribution of graphs sizes affects outdegrees distribution (over all nodes of all graphs), which appears as a power law with the same exponent ($\sim x^{-2}$). This is not a surprising result, since regardless the outdegrees distribution of each graph, its sum over all graphs (which sizes are power law distributed) will have a power law shape as well. See figure 4.5.

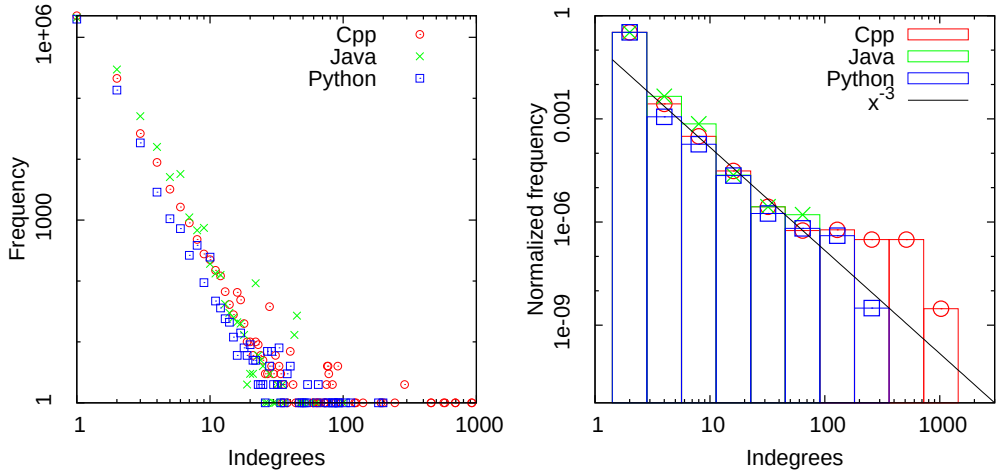

Figure 4.6: **Distribution of indegrees of all nodes of all graphs** - Indegrees are systematically less than outdegrees and distributions have a strong cutoff, especially for Java hierarchies.

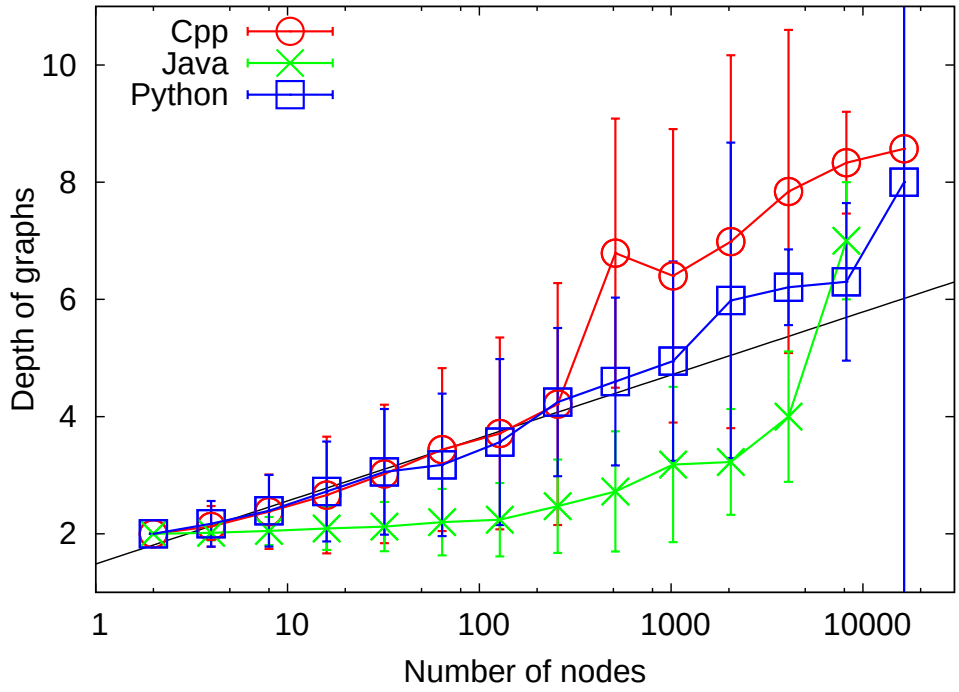

Figure 4.7: **Depth VS Nnodes** - Depth of graphs seems a logarithmic function of the number of object for C++ and Python, while the depth is almost constant in Java hierarchies. The fit for C++ and Python give $y \sim 0.5 \log(x) + 1.5$

For this reason, it's important to look at outdegrees distribution of each graph. Is the outdegrees distribution a real power law or it is an effect caused by the sizes?

As showed in figure 4.8, the power law appears as distribution of outdegrees also in single graphs, at least for biggest hierarchies. Outdegrees distribution is therefore the results of the sum of many power laws.

4.4 Tree approximation

Models presented in sections 3 and 2 schematize hierarchies through trees, while inheritance structures can be in principle directed acyclic graphs. The main difference between this two kind of graphs is in the number of indegrees of each node: if a graph is a tree, all nodes can have at most one indegree, while indegrees have no restrictions for a directed acyclic graph.

In programming languages, multiple inheritance is often not recommended, except in cases where actually required. In data we can observe multiple inheritance, but it is systematically less than single inheritance, as expected. When indegrees distribution is compared to outdegrees distribution in a hierarchy, as in figure 4.8, one can observe how outdegree is the first main quantity to keep into account in an object-oriented hierarchy model.

The outdegrees distribution shows power law behavior with a small exponent ($\sim x^{-2}$ or even less), while indegrees distribution shows a power law behavior with a large exponent ($\sim x^{-3}$ and often more), large enough to allow a first approach considering the indegree constant and equal to its mean, that is about 1. In this sense, tree approximation is an excellent first approach to this system.

Distribution of indegrees of all nodes of all graphs shows a power law behavior, with exponent $\sim x^{-3}$ (see figure 4.6). The three programming languages have quite different behaviors. In Python only the 0.1% of nodes has outdegree greater than one and no one class has more than 300 indegrees. In Java, indegrees are often more respect to other languages, as expected from an *interfaces programming*, but all class has less than 100 indegrees. In C++ indegrees can reach 1000 classes.

4.5 Lateral growth

Graphs are complex structures and their topology has lots of features. A way to give a summary look into its shape is asking in which way the depth is coupled to the number of objects in hierarchies, in order to have an idea

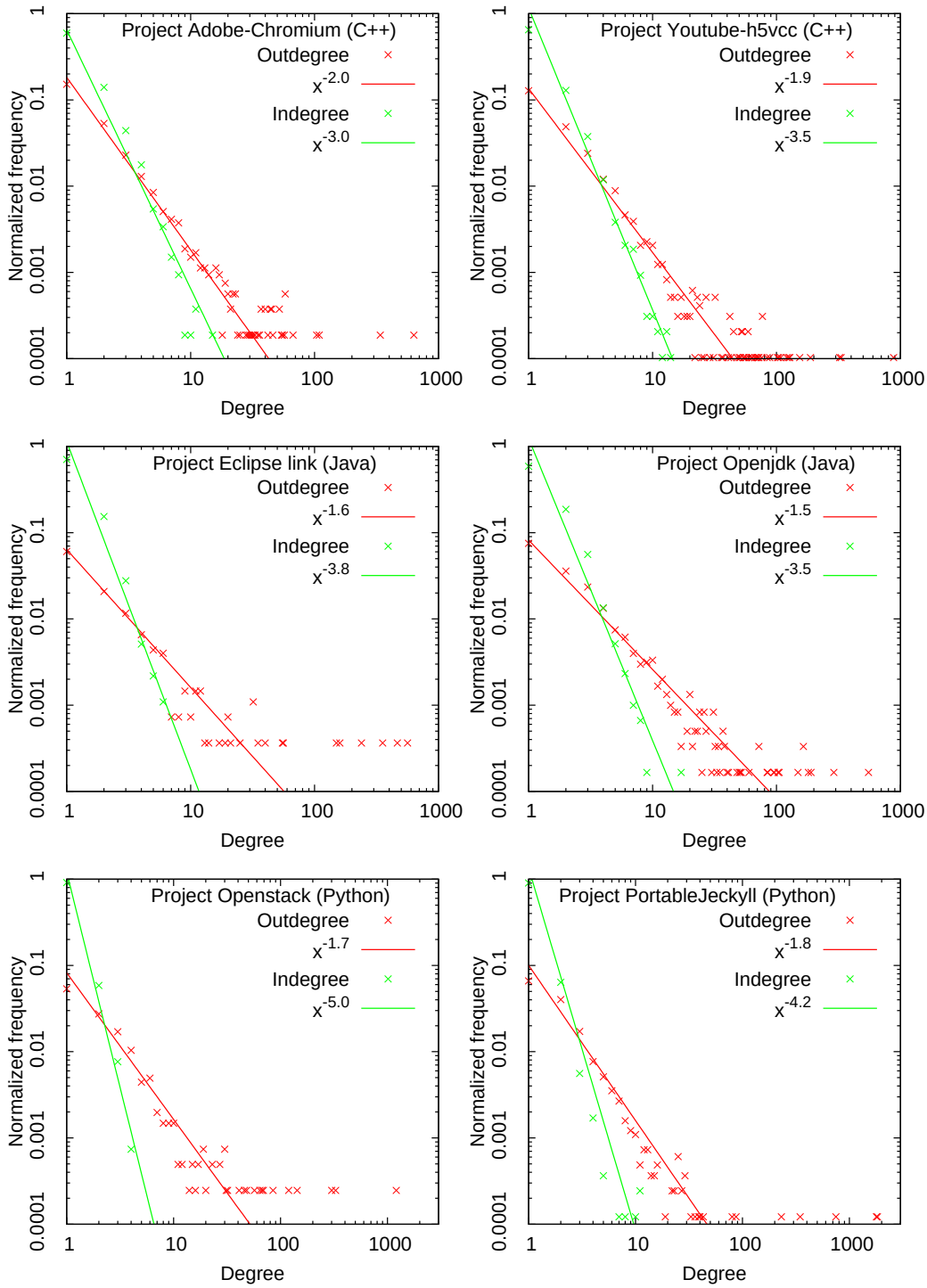

Figure 4.8: **Comparison between indegrees and outdegrees** - The different exponent for indegree and outdegree in each graph shows how tree approximation is an excellent first approach to this system.

about the ratio of the *vertical* dimension (i.e. the depth) and the *horizontal* dimension (i.e. the number of objects in levels).

4.5.1 Horizontal and vertical dimensions

Plotting the depth versus the number of objects for each graph allow to reconstruct a distribution of occurrence for each programming language, which means and standard deviations are showed in figure 4.7.

Python and C++ behave in the same way: the depth is in good approximation a logarithmic function of the number of objects (at least for most of sizes). For Java hierarchies, instead, the depth is almost a constant function of the number of classes that grows only for big graphs sizes. Both results are descriptive of a *later growth* of graphs instead of a vertical one.

Consider two hierarchies with a very different number of objects. Despite the gap between the two effective sizes of the structures, we may expect two graphs with similar depths, at most one increased or decreased by one respect the other, especially for Java structures.

The logarithmic behavior of the depth as a function of the number of objects is well captured by the Sharing Tree model and the Effort model. Both mean field hierarchies and simulated trees well describe this feature.

4.5.2 Shallow is better

A really interesting result is the change in behavior of the depth as a function of the number of objects, as highlight in figure 4.9 through a heat map for each programming language.

In object-oriented programming languages there are general advices that suggest to keep hierarchies as shallow as possible, in order to avoid the anti-pattern known as *Yo-yo problem* [TGP89]. If inheritance graph is too much long and complicated, then a programmer that has to read and understand the hierarchy, has to keep flipping between different parts of the code.

For these reasons, the huge amount of shallow hierarchies was foreseeable. But what happens when hierarchy size get over 1000 classes?

The artificial attempt to keep hierarchies as shallow as possible seems to fail, and all hierarchies show a depth greater than that expected.

Can the guess for which *shallow is better* be wrong? If shallow hierarchies are produced by a false assumption, can the optimization be achieved through deep hierarchies?

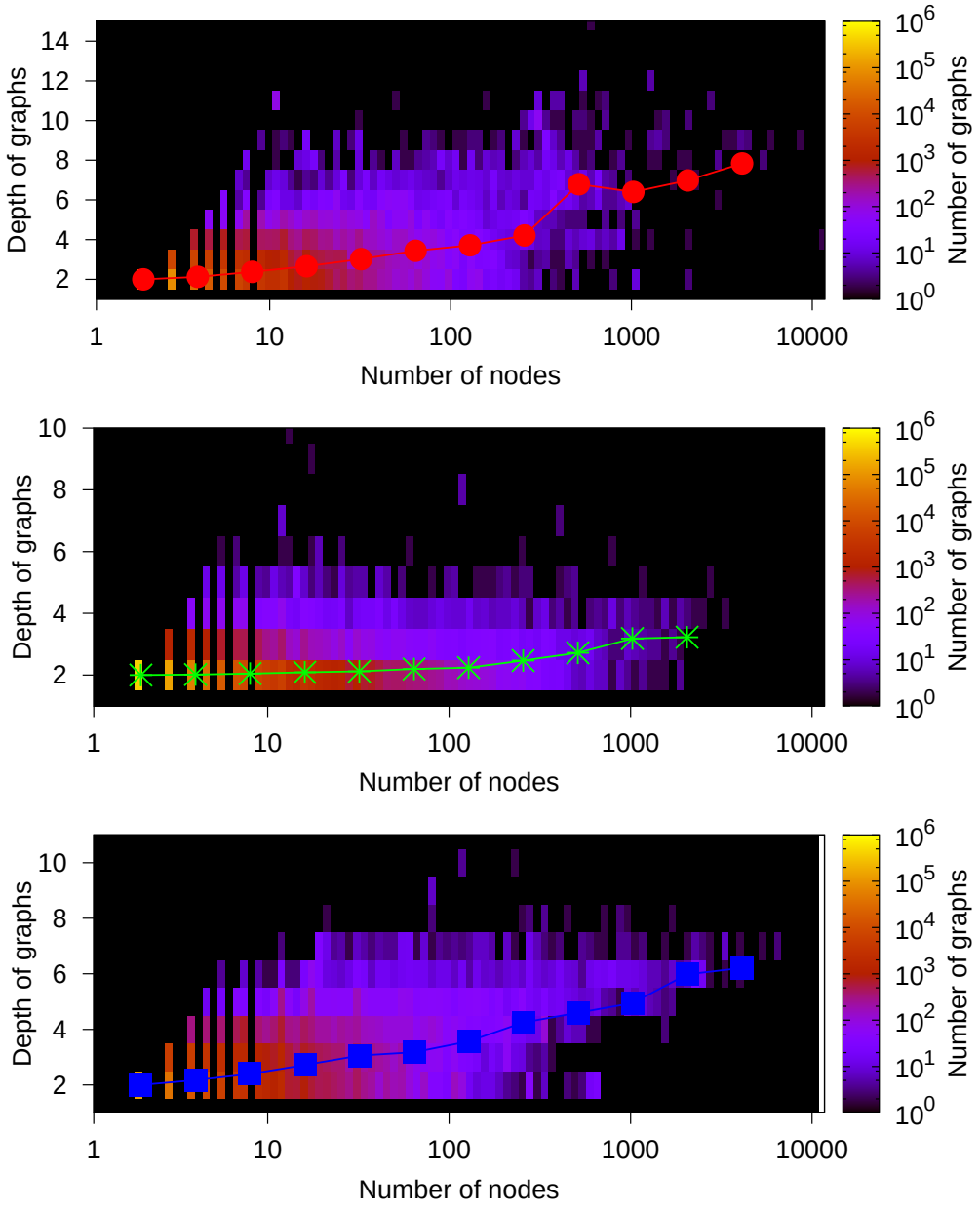

Figure 4.9: **Depth VS nodes in C++, Java and Python hierarchy** - Colors (used in log scale) are representative of how many graphs have the x number of nodes and depth equal to y. Colors represent the whole distribution, while lines trace the mean. Comparison of this three trends is given in figure 4.7.

4.6 Vertical balancing

Inspired by models, we can investigate more accurately the topology of hierarchies studying the behavior of the outdegree mean of nodes as a function of the levels. Is it constant across graphs? If not, where are situated objects with the most shared code?

4.6.1 Code shareability per level

Once established that graphs are wider than high, we can probe the *vertical balancing*. An interesting quantity that we can study as a function of a vertical variable (i.e. of levels) is the outdegree mean of nodes.

When the outdegree is high, the code contained in classes is shared with lots of other classes: high outdegree means high shareability.

Some examples of this function are given in figures 4.10 and 4.11. We can observe that the outdegree mean is a quantity that tends to grow with levels, with pronounced peaks close to the root.

Why this behavior? Why the outdegree mean grows with level?

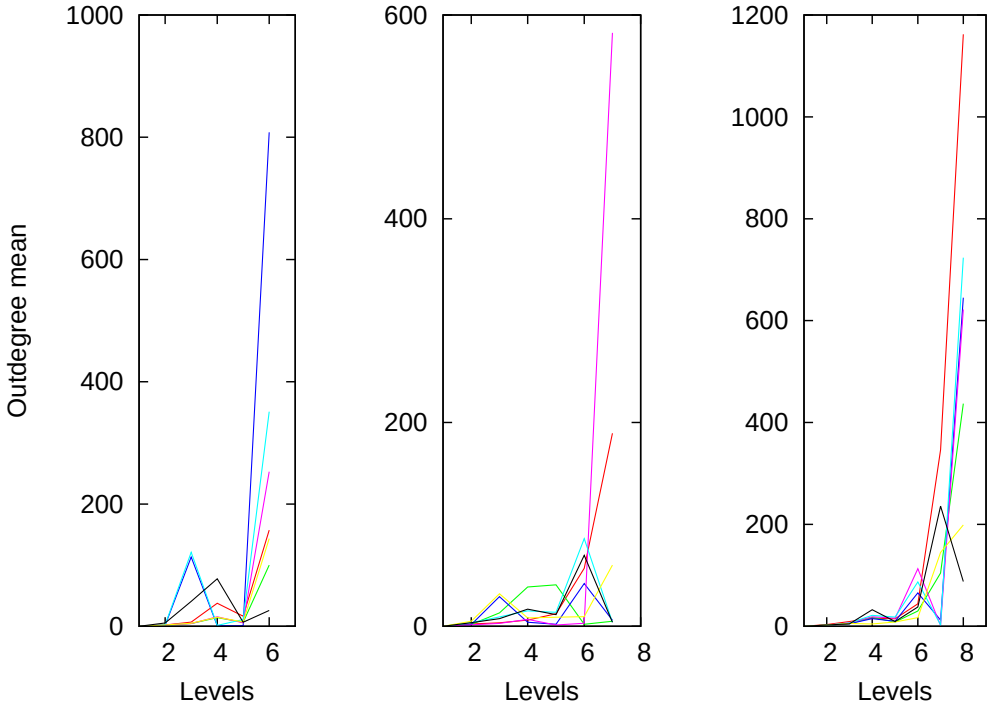

Figure 4.10: **Outdegree mean for Cpp hierarchies** - Examples of structure with more than 1000 nodes and depth equal to 6, 7 and 8 (in order).

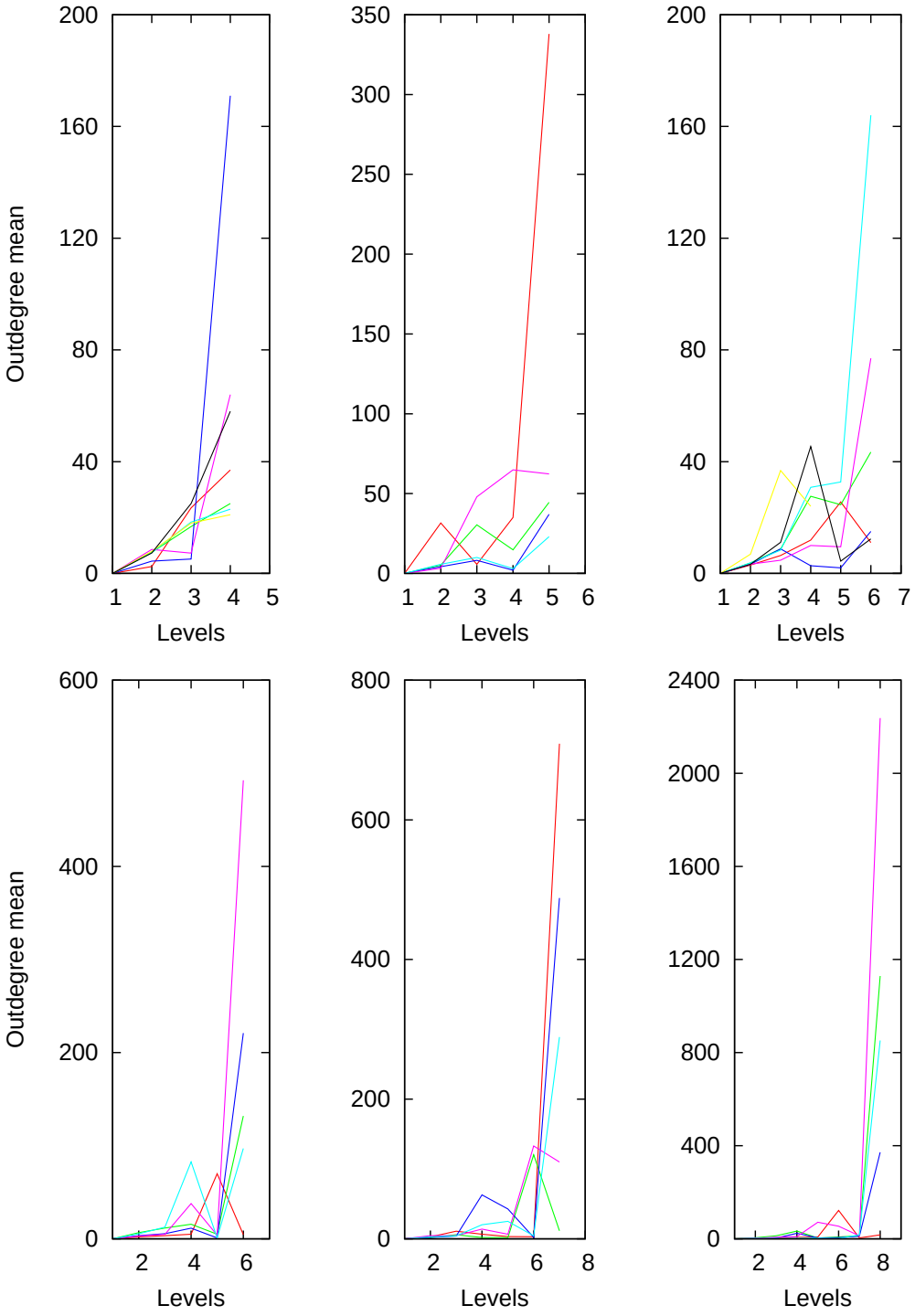

Figure 4.11: **Outdegree mean for Java and Python hierarchies** - In the upper side figures, examples of Java structure with more than 500 nodes and depth equal to 4, 5 and 6 (in order). The other figures represent examples of python structure with more than 500 nodes and depth equal to 6, 7 and 8 (in order).

4.6.2 Shallow regime

Thanks to the mean field model (Effort model), we can give an interpretation to the growth of outdegree with levels.

We have showed in chapter 2 how the outdegree mean behavior, as function of levels, is very sensible to *depth constrain* and that it can have a strong growth close to the root in the case of insufficient depth. This is exactly the situation in which we find inheritance hierarchies in data.

This is yet another clue that leads us to argue that *shallow hierarchies* are not the best solution. Code reuse appears not optimized by a decrease of the depth in Effort model, and data appears as trapped in an artificial *shallow regime*.

A most efficient documentation system could avoid the *yo-yo problem* as well as decreasing the depth of hierarchies, without sacrificing the optimal depth.

4.7 Optimal or random?

The models proposed in this thesis are based on the assumption that inheritance graphs are made with optimizing spirit. This is a perfectly reasonable assumption since there are no reasons to hypothesize that inheritance arise from chaos.

Anyway, it is interesting to compare Sharing Tree graphs with graphs obtained through models completely random and made without optimization mechanisms.

4.7.1 Random Branching Tree model

A Random Branching Tree can be built starting from an initial node, the root, and choosing a distribution with a well-defined mean.

As a first step, we extract a random variable from the distribution: the extraction represents the root outdegree and so the number of nodes that are connected to the root. Once added new nodes, the process is iterated for each new node until a certain condition is reached.

The outdegree mean in function of levels is obviously a constant value, equal to the mean of the chosen distribution for the outdegree. Simulations are shown in figure 4.12.

Which random mechanism can be introduced to obtain a non constant function?

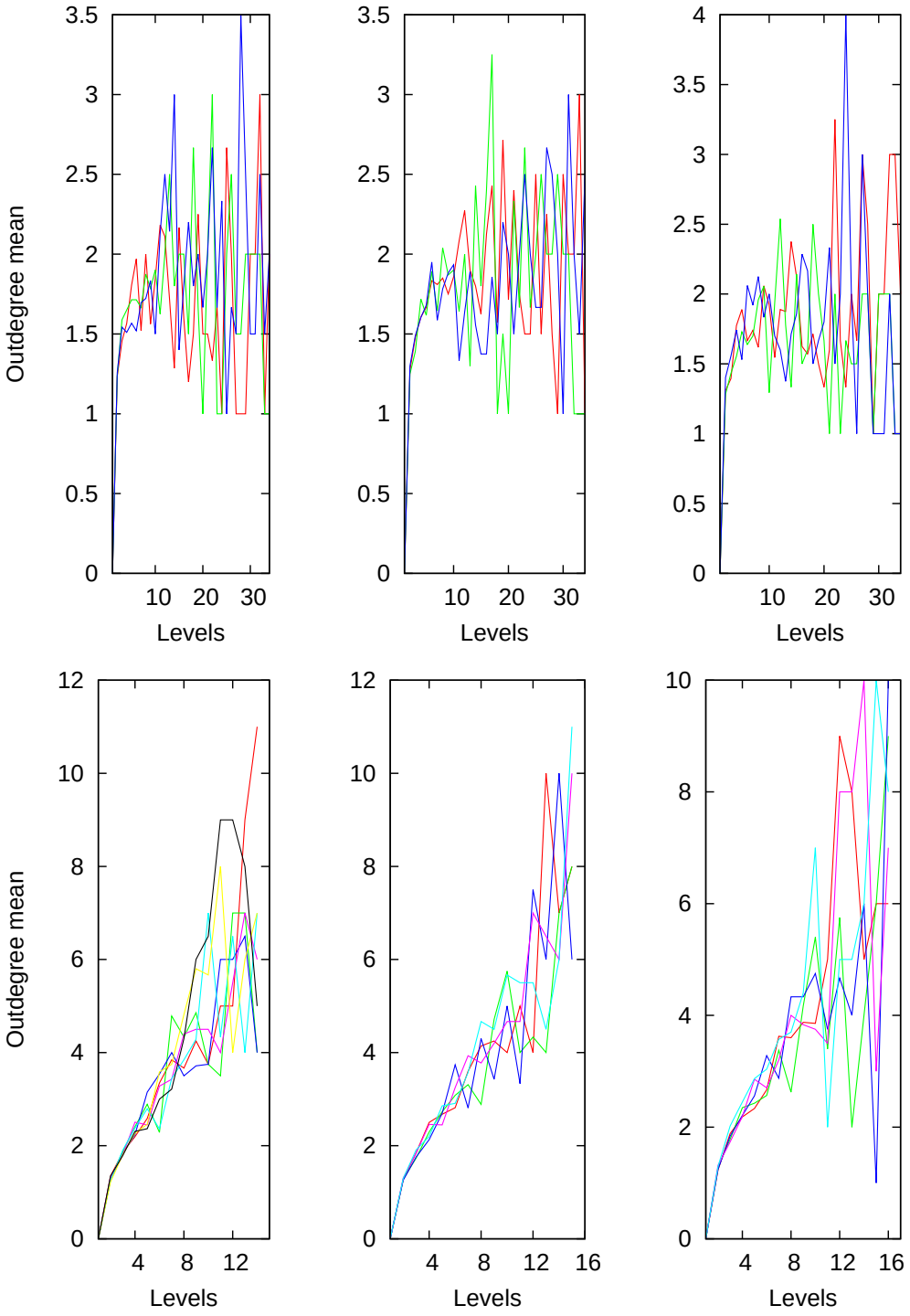

Figure 4.12: **Random branching tree and Pang-Maslov models simulation** - The outdegree mean in function of levels does not reproduce the behavior observed in data.

4.7.2 Pang-Maslov model

The Pang-Maslov model [PM13] can be used to construct a tree. Consider an initial node an a process for which, at each step, a new node arrives and sticks to one of the existing nodes. The process can be iterated until a certain condition is reached.

This model prevents the constancy of outdegree mean, since the older nodes have had more chance of being chosen by another node, and so they must have an outdegree greater than young nodes. Growth, however, is constant and it does not represent well the behavior of outdegree mean in data.

4.7.3 Sharing Tree comparison

Through the Sharing Tree model, we can can produce Monte Carlo trees to explicitly compare optimization ideas with data. Some examples are given in figure 4.13.

The outdegree mean has not a trivial behavior due to its strong growth close to the root, that is also the main feature observable in data.

Therefore, there is a very good agreement between data and optimization models.

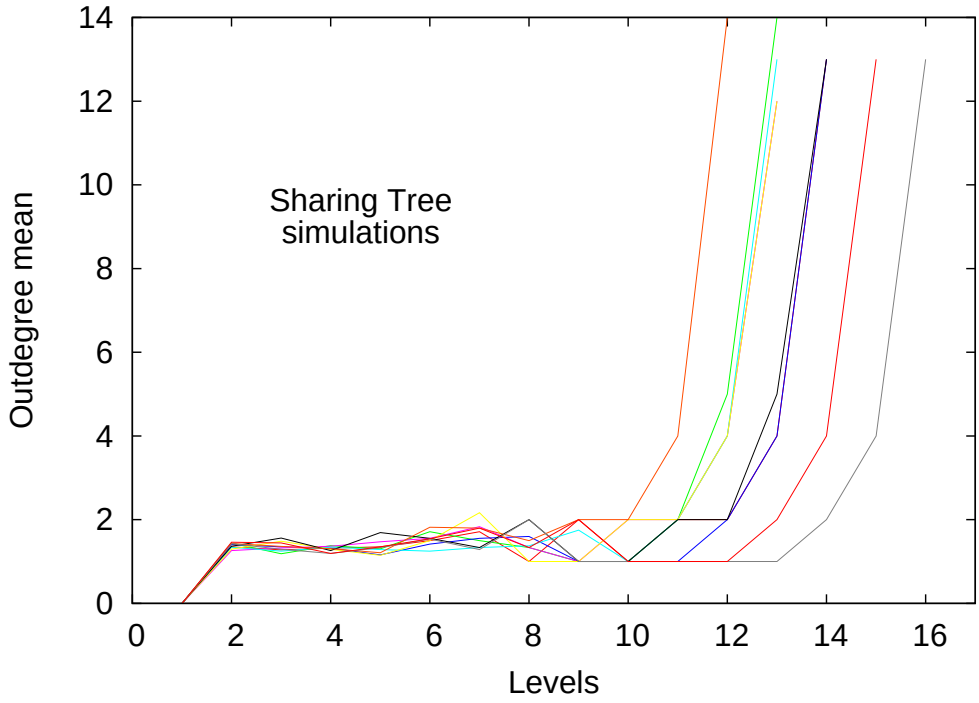

Figure 4.13: **Sharing Tree Monte Carlo simulations** - The Sharing Tree model well capture the behave of the outdegree mean as function of levels. In simulations, parameters are $\mathcal{N} = 1000$, $\mathsf{k} = 100$ and $\mathcal{S} = 2000$.

Chapter 5

Conclusions

The noisy data have proved to be full of interesting information: they have showed universal behaviors among object-oriented programming languages but they also highlight structural differences in inheritance management.

Hierarchical sizes distribution is universal

Hierarchies sizes distribution is power law distributed and there are no differences among the three programming languages.

The speciation of classes has a universal behavior

Despite the differences among C++, Java and Python, the speciation is used in the same way, and so the distribution of the number of classes that inherits code from a selected class is the same.

Multiple inheritance is ever less used than speciation

The multiple inheritance is used with differences between the three languages. It is always used less than speciation, even in Java, where it has a different and more useful meaning.

Hierarchies are shallow

The vast majority of graphs have a depth equal to two or three while the deepest hierarchies have at most a depth of about eleven, or twelve.

Java hierarchies are shallower

Java hierarchies show a depth sistematically lower than the Python and C++, while graph sizes behave in the same way.

The depth is a logarithmic function of the hierarchy size

In C++ and Python, the mean of depths of hierarchies as a function of the number of classes has a logarithmic behavior.

Huge hierarchies behave differently

The depth of huge hierarchies doesn't fit the logarithmic behavior of small and medium hierarchies. It is systematically over such function.

Speciation is a function of levels

Classes that does lots of speciation are always at the higher levels.

The Minimal Effort model gives a link between the hierarchical structures and the effort done to build them. It allows to give an interpretation to data, and it is a guide to key quantities.

The Sharing Tree model extends the mean field model and allows Monte Carlo simulations. The mechanism implemented is close to the natural approach of programmers in objects organization and Sharing Monte Carlo trees well reproduce the data hierarchies.

The depth is the central quantity for optimization mechanism, since it governs the hierarchical structures for fixed size and the proportion between the vertical and the horizontal dimension. Programmers seems to follow the advise to keep the depth as shallow as possibile to avoid the yo-yo problem instead of looking for the optimal structure for code reuse.

The optimization resulting from the competition between the reuse and the abstractability of the code seems to play an important role in the construction of hierarchical inheritance structures in object-oriented programming languages.

Appendix A

Source code

A.1 Library for data analysis

In this section, some of the classes implemented in the library are listed. Classes omitted are Scatterplot, Lineplot and Distribuzioni that allow a suitable print of data obtainable by the following classes.

A.1.1 class Node

A class that realize the concept of node.

node.hpp

```
1  #ifndef _NODE_HPP_
2  #define _NODE_HPP_
3
4  #include <string>
5  #include <vector>
6  #include <iostream>
7
8  using namespace std;
9
10 class Node {
11
12   public:
13     //Costruttore/Distruttore
14     Node();
15     ~Node();
16
17     //Assegnamento
18     Node &operator=(Node &obj);
19
20     //Funzioni
21     void Add_edge_in(size_t number);
22     void Add_edge_out(size_t number);
23
24     //Set
```

```
25      void Set_level(size_t level);
26      void Set_Rlevel(size_t Rlevel);
27
28      //Get
29      size_t Get_Npar() const;
30      size_t Get_Nsons() const;
31      size_t Get_level() const;
32      size_t Get_Rlevel() const;
33
34      size_t Get_son(size_t i) const;
35      size_t Get_par(size_t i) const;
36
37
38    private:
39      vector<size_t> _edgein;
40      vector<size_t> _edgeout;
41      size_t _level;
42      size_t _Rlevel;
43
44  };
45
46  #endif
```

node.cpp

```
1   #include "node.hpp"
2
3   /*------------------------------------------------------------------------
4    Costruttore
5   ------------------------------------------------------------------------*/
6   Node::Node(): _level(0), _Rlevel(0) {};
7   /*------------------------------------------------------------------------
8    Assegnamento
9   ------------------------------------------------------------------------*/
10  Node &Node::operator=(Node &obj){
11    _edgein=obj._edgein;
12    _edgeout=obj._edgeout;
13    _level=obj._level;
14    _Rlevel=obj._Rlevel;
15    return *this;
16  };
17  /*------------------------------------------------------------------------
18   Distruttore
19  ------------------------------------------------------------------------*/
20  Node::~Node(){
21    _edgein.clear();
22    _edgeout.clear();
23  };
24  /*------------------------------------------------------------------------
25    Functions
26  ------------------------------------------------------------------------*/
27  void Node::Add_edge_in(size_t number){
28    bool found = false;
29    for(size_t i=0; i<_edgein.size(); i++) {
30      if(number == _edgein[i]) found = true;
31    }
32    if(!found) {
33      _edgein.push_back(number);
34    }
35  };
```

```
36 /*--------------------------------------------------------------------*/
37 void Node::Add_edge_out(size_t number){
38   bool found = false;
39   for(size_t i=0; i<_edgeout.size(); i++) {
40     if(number == _edgeout[i]) found = true;
41   }
42   if(!found) {
43     _edgeout.push_back(number);
44   }
45 };
46 /*--------------------------------------------------------------------
47   Set
48 --------------------------------------------------------------------*/
49 void Node::Set_level(size_t level){
50   if(_level<level) _level=level;
51 };
52 /*--------------------------------------------------------------------*/
53 void Node::Set_Rlevel(size_t Rlevel){
54   if(_Rlevel<Rlevel) _Rlevel=Rlevel;
55 };
56 /*--------------------------------------------------------------------
57   Get
58 --------------------------------------------------------------------*/
59 size_t Node::Get_Npar() const {
60   return _edgein.size();
61 };
62 /*--------------------------------------------------------------------*/
63 size_t Node::Get_Nsons() const {
64   return _edgeout.size();
65 };
66 /*--------------------------------------------------------------------*/
67 size_t Node::Get_level() const {
68   return _level;
69 };
70 /*--------------------------------------------------------------------*/
71 size_t Node::Get_Rlevel() const {
72   return _Rlevel;
73 };
74 /*--------------------------------------------------------------------*/
75 size_t Node::Get_son(size_t i) const {
76   return _edgeout[i];
77 };
78 /*--------------------------------------------------------------------*/
79 size_t Node::Get_par(size_t i) const {
80   return _edgein[i];
81 };
82 /*--------------------------------------------------------------------*/
```

A.1.2 class Graph

The class Graph is central for the library. It allows to obtain all main quantities about a graph, or a tree.

graph.hpp

```
1 #ifndef _GRAPH_HPP_
2 #define _GRAPH_HPP_
```

```
3
4  #include <string>
5  #include <vector>
6  #include <fstream>
7  #include <cmath>
8  #include <map>
9  #include <set>
10 #include "node.hpp"
11 #include "funzioni.hpp"
12
13 using namespace std;
14
15 class Graph {
16
17   public:
18     //Costruttori/Distruttore
19     Graph();
20     Graph(const char *edgefile);
21     ~Graph();
22
23     //Assegnamento
24     Graph &operator=(const Graph &obj);
25
26     //Funzioni
27     void HTD(double **scores, string **namescores, size_t nrows);
28
29     //Set
30     void set(vector<size_t> from, vector<size_t> to, bool connection = 0)
           ;
31     void clear();
32
33     //Get
34     size_t depth();
35     size_t Nnodes();
36     size_t Nnodes(size_t level);
37     size_t RNnodes(size_t level);
38     size_t diamonds();
39
40     size_t Nleaves();
41
42     size_t outdeg(size_t nodo);
43     size_t indeg(size_t nodo);
44
45     size_t brothers(size_t level);
46     size_t Rbrothers(size_t level);
47
48     vector<size_t> leaves_depth();
49     vector<size_t> nodes_depth();
50
51     vector<size_t> indegrees();
52     vector<size_t> outdegrees();
53     vector<size_t> leaves_indegrees();
54     vector<size_t> roots_outdegrees();
55
56     vector<double> mean_Routdegrees();
57     vector<double> mean_outdegrees();
58
59     vector<size_t> Kin();
60     vector<size_t> Kout();
61
62     //Statistics
63     double mean_depth();
```

```
64      double mean_leaves_depth();
65      double mean_Rdepth();
66      double mean_outdeg();
67      double mean_Routdeg(size_t level);
68      double mean_outdeg(size_t level);
69
70      double cost(double lambda);
71      double cost(size_t nodo, double lambda);
72      double recursive_cost(double lambda);
73      double recursive_cost(size_t nodo, double lambda);
74
75  //    OVERLOADING DEL <<
76      friend ostream &operator<<(ostream &os, Graph &g);
77
78    private:
79      Node *_nodes;
80      size_t _Nnodes;
81      size_t *_ordernodes;
82      size_t *_Rordernodes;
83      vector<size_t> _leaves;
84      vector<size_t> _roots;
85      size_t _diamonds;
86      bool _diamonds_check;
87
88      vector<size_t> _Kin;
89      vector<size_t> _Kout;
90
91      void find_levels();
92      void find_reversed_levels();
93      void find_leaves();
94      void find_roots();
95      void visita(size_t node, std::set<size_t> &visitati);
96      void find_connection();
97      void find_Kin();
98      void visita_down(size_t node, std::set<size_t> &visitati);
99      void find_Kout();
100     void visita_up(size_t node, std::set<size_t> &visitati);
101     void find_diamonds();
102     void visita_diamond(size_t analyzed, size_t node, std::set<size_t> &
            visitati, std::set<size_t> &diamanti);
103 };
104 #endif
```

graph.cpp

```
1   #include "graph.hpp"
2
3   /*------------------------------------------------------------------
4    Costruttori
5   ------------------------------------------------------------------*/
6   Graph::Graph(): _nodes(NULL), _ordernodes(NULL), _Rordernodes(NULL),
        _diamonds(0), _diamonds_check(0) {};
7   /*------------------------------------------------------------------*/
8   Graph::Graph(const char *edgefile): _ordernodes(NULL), _Rordernodes(NULL)
        , _diamonds(0), _diamonds_check(0) {
9     fstream f;
10    string help;
11    vector<string> from,to;
12
13    f.open(edgefile,ios::in);
```

```
14    f>>help;
15    while(!f.eof()){
16      from.push_back(help);
17      f>>help;
18      to.push_back(help);
19      f>>help;
20    }
21          f.close();
22
23    if (from.size()!=to.size()) {
24      cerr<<"Graph:: costruttore - controlla il file di input!"<<endl;
25    }
26    if (from.size()==0 || to.size()==0) {
27      cerr<<"Graph:: costruttore - creazione di grafo mononodo!"<<endl;
28      _Nnodes=1;
29      _nodes = new Node;
30      _roots.push_back(0);
31      _leaves.push_back(0);
32      _nodes[0].Set_level(1);
33      _nodes[0].Set_Rlevel(1);
34      _ordernodes=new size_t;
35      _ordernodes[0]=0;
36      _Rordernodes=new size_t;
37      _Rordernodes[0]=0;
38    }
39    else {
40      map<string,size_t> conversione;
41      size_t cont=0;
42      for(size_t i=0; i<from.size(); i++) {
43        if(conversione.find(from[i]) == conversione.end()) {
44          conversione[from[i]]=cont;
45          cont++;
46        }
47        if(conversione.find(to[i]) == conversione.end()) {
48          conversione[to[i]]=cont;
49          cont++;
50        }
51      }
52      _Nnodes = conversione.size();
53      _nodes = new Node[_Nnodes];
54      //Costruzione del grafo
55      for(size_t i=0;i<from.size();i++){
56        if(from[i] == to[i]) {
57          cerr << "Graph:: set - Loop, padre di se stesso!" << endl;
58          continue; // looooooop :(
59        }
60        _nodes[conversione[from[i]]].Add_edge_out(conversione[to[i]]);
61        _nodes[conversione[to[i]]].Add_edge_in(conversione[from[i]]);
62      }
63      find_roots();
64      find_connection();
65      find_levels();
66      find_leaves();
67      find_reversed_levels();
68    }
69  };
70  /*-------------------------------------------------------------------------
71   Assegnamento
72  -------------------------------------------------------------------------*/
73  Graph &Graph::operator=(const Graph &obj){
74    (*this).clear();
75    _Nnodes=obj._Nnodes;
```

```
76    _leaves=obj._leaves;
77    _roots=obj._roots;
78    _diamonds=obj._diamonds;
79    _diamonds_check=obj._diamonds_check;
80    _Kin=obj._Kin;
81    _Kout=obj._Kout;
82    _ordernodes = new size_t[_Nnodes];
83    _Rordernodes = new size_t[_Nnodes];
84    _nodes = new Node[_Nnodes];
85    for(size_t i=0; i<_Nnodes; i++) {
86      _ordernodes[i]=obj._ordernodes[i];
87      _Rordernodes[i]=obj._Rordernodes[i];
88      _nodes[i]=obj._nodes[i];
89    }
90    return *this;
91  };
92  /*---------------------------------------------------------------------
93   Distruttore
94  ---------------------------------------------------------------------*/
95  Graph::~Graph(){
96    if(_nodes!=NULL) {
97      if(_Nnodes==1) delete _nodes;
98      else delete[] _nodes;
99    }
100   if(_ordernodes!=NULL) {
101     if(_Nnodes==1) delete _ordernodes;
102     else delete[] _ordernodes;
103   }
104   if(_Rordernodes!=NULL) {
105     if(_Nnodes==1) delete _Rordernodes;
106     else delete[] _Rordernodes;
107   }
108   _nodes = NULL;
109   _ordernodes = NULL;
110   _Rordernodes = NULL;
111   _leaves.clear();
112   _roots.clear();
113   _Nnodes = 0;
114 };
115 /*---------------------------------------------------------------------
116    Settings
117 ---------------------------------------------------------------------*/
118 void Graph::set(vector<size_t> from, vector<size_t> to, bool connection )
        {
119   clear();
120   if (from.size()!=to.size()) {
121     cerr<<"Graph:: set - controlla i vettori in input!"<<endl;
122   }
123   if (from.size()==0 || to.size()==0) {
124     cerr<<"Graph:: set - creazione di grafo mononodo!"<<endl;
125     _Nnodes=1;
126     _nodes = new Node;
127     _roots.push_back(0);
128     _leaves.push_back(0);
129     _nodes[0].Set_level(1);
130     _nodes[0].Set_Rlevel(1);
131     _ordernodes=new size_t;
132     _ordernodes[0]=0;
133     _Rordernodes=new size_t;
134     _Rordernodes[0]=0;
135   }
136   else {
```

```
137     map<size_t,size_t> conversione;
138     size_t cont=0;
139     for(size_t i=0; i<from.size(); i++) {
140       if(conversione.find(from[i]) == conversione.end()) {
141         conversione[from[i]]=cont;
142         cont++;
143       }
144       if(conversione.find(to[i]) == conversione.end()) {
145         conversione[to[i]]=cont;
146         cont++;
147       }
148     }
149     _Nnodes = conversione.size();
150     _nodes = new Node[_Nnodes];
151     //Costruzione del grafo
152     for(size_t i=0;i<from.size();i++){
153       if(from[i] == to[i]) {
154         cerr << "Graph:: set - Loop, padre di se stesso!" << endl;
155         continue; // looooooop :(
156       }
157       _nodes[conversione[from[i]]].Add_edge_out(conversione[to[i]]);
158       _nodes[conversione[to[i]]].Add_edge_in(conversione[from[i]]);
159     }
160     find_roots();
161     if(connection==0) find_connection();
162     find_levels();
163     find_leaves();
164     find_reversed_levels();
165   }
166 };
167 /*-------------------------------------------------------------------------*/
168 void Graph::clear(){
169   if(_nodes!=NULL) {
170     if(_Nnodes==1) delete _nodes;
171     else delete[] _nodes;
172   }
173   if(_ordernodes!=NULL) {
174     if(_Nnodes==1) delete _ordernodes;
175     else delete[] _ordernodes;
176   }
177   if(_Rordernodes!=NULL) {
178     if(_Nnodes==1) delete _Rordernodes;
179     else delete[] _Rordernodes;
180   }
181   _nodes = NULL;
182   _ordernodes = NULL;
183   _Rordernodes = NULL;
184   _leaves.clear();
185   _roots.clear();
186   _Kin.clear();
187   _Kout.clear();
188   _Nnodes = 0;
189   _diamonds_check = 0;
190   _diamonds = 0;
191 };
192 /*-------------------------------------------------------------------------
193   Finding ...
194 -------------------------------------------------------------------------*/
195 void Graph::find_levels(){
196   size_t iter_level=1, help;
197   vector<size_t> support1, support2;
198   //selezionare le root del grafo
```

```
199   for(size_t i=0;i<_roots.size();i++){
200     support1.push_back(_roots[i]);
201   }
202   //percorrere il grafo top-down per definire i livelli
203   while(support1.size()!=0){
204     iter_level++;
205     for(size_t i=0;i<support1.size();i++){
206       for(size_t j=0;j<_nodes[support1[i]].Get_Nsons();j++){
207         help=_nodes[support1[i]].Get_son(j);
208         _nodes[help].Set_level(iter_level);
209         support2.push_back(help);
210       }
211     }
212     support1.clear();
213     iter_level++;
214     for(size_t i=0;i<support2.size();i++){
215       for(size_t j=0;j<_nodes[support2[i]].Get_Nsons();j++){
216         help=_nodes[support2[i]].Get_son(j);
217         _nodes[help].Set_level(iter_level);
218         support1.push_back(help);
219       }
220     }
221     support2.clear();
222   }
223   //salvo l'indice delle posizioni dei nodi ordinati secondo il loro
        livello
224   _ordernodes=new size_t[_Nnodes];
225   for(size_t i=0;i<_Nnodes;i++) _ordernodes[i]=i;
226   size_t vettore[_Nnodes];
227   for(size_t i=0;i<_Nnodes;i++) vettore[i]=_nodes[i].Get_level();
228   quicksort(_Nnodes,vettore,_ordernodes);
229 };
230 /*-------------------------------------------------------------------
      */
231 void Graph::find_reversed_levels(){
232   size_t iter_level=1, help;
233   vector<size_t> support1, support2;
234   //selezionare le leaves del grafo
235   for(size_t i=0;i<_leaves.size();i++){
236     support1.push_back(_leaves[i]);
237   }
238   //percorrere il grafo down-top per definire i livelli
239   while(support1.size()!=0){
240     iter_level++;
241     for(size_t i=0;i<support1.size();i++){
242       for(size_t j=0;j<_nodes[support1[i]].Get_Npar();j++){
243         help=_nodes[support1[i]].Get_par(j);
244         _nodes[help].Set_Rlevel(iter_level);
245         support2.push_back(help);
246       }
247     }
248     support1.clear();
249     iter_level++;
250     for(size_t i=0;i<support2.size();i++){
251       for(size_t j=0;j<_nodes[support2[i]].Get_Npar();j++){
252         help=_nodes[support2[i]].Get_par(j);
253         _nodes[help].Set_Rlevel(iter_level);
254         support1.push_back(help);
255       }
256     }
257     support2.clear();
258   }
```

```
259   //salvo l'indice delle posizioni dei nodi ordinati secondo il loro
          livello
260   _Rordernodes=new size_t[_Nnodes];
261   for(size_t i=0;i<_Nnodes;i++) _Rordernodes[i]=i;
262   size_t vettore[_Nnodes];
263   for(size_t i=0;i<_Nnodes;i++) vettore[i]=_nodes[i].Get_Rlevel();
264   quicksort(_Nnodes,vettore,_Rordernodes);
265 };
266 /*--------------------------------------------------------------------------
      */
267 void Graph::find_leaves(){
268   for(size_t i=0;i<_Nnodes;i++) {
269     if(_nodes[i].Get_Nsons()==0) {
270       _leaves.push_back(i);
271       _nodes[i].Set_Rlevel(1);
272     }
273   }
274 };
275 /*--------------------------------------------------------------------------
      */
276 void Graph::find_roots(){
277   for(size_t i=0;i<_Nnodes;i++) {
278     if(_nodes[i].Get_Npar()==0) {
279       _roots.push_back(i);
280       _nodes[i].Set_level(1);
281     }
282   }
283 };
284 /*-------------------------------------------------------------------------*/
285 void Graph::find_connection() {
286   std::set<size_t> supporto, tosave;
287   //cerco componente piu' grande
288   for (size_t i=0; i<_roots.size(); i++) {
289     if(supporto.find(_roots[i])==supporto.end()) {
290       visita(_roots[i], supporto);
291       if(supporto.size() > tosave.size()) tosave = supporto;
292       supporto.clear();
293     }
294   }
295   if(tosave.size() != _Nnodes) {
296     cerr<<"Graph:: Errore! Il grafo non e' connesso (o non ci sono radici
          )!"<<endl;
297     vector<size_t> new_from, new_to;
298     for (std::set<size_t>::iterator i = tosave.begin(); i != tosave.end()
          ; i++) {
299       for(size_t j=0; j<_nodes[*i].Get_Nsons(); j++){
300         new_from.push_back(*i);
301         new_to.push_back(_nodes[*i].Get_son(j));
302       }
303     }
304     (*this).clear();
305     this->set(new_from,new_to, 1);
306   }
307 }
308 /*-------------------------------------------------------------------------*/
309 void Graph::find_Kin() {
310   std::set<size_t> linkati_up;
311   for(size_t i=0; i<_Nnodes; i++) {
312     visita_up(i,linkati_up);
313     _Kin.push_back(linkati_up.size()-1);
314     linkati_up.clear();
315   }
```

```
316
317 }
318 /*--------------------------------------------------------------------*/
319 void Graph::find_Kout() {
320   std::set<size_t> linkati_down;
321   for(size_t i=0; i<_Nnodes; i++) {
322     visita_down(i,linkati_down);
323     _Kout.push_back(linkati_down.size()-1);
324     linkati_down.clear();
325   }
326 }
327 /*--------------------------------------------------------------------*/
328 void Graph::find_diamonds() {
329   std::set<size_t> linkati;
330   std::set<size_t> diamanti;
331   for(size_t i=0; i<_Nnodes; i++) {
332     visita_diamond(i,i,linkati, diamanti);
333     linkati.clear();
334   }
335   _diamonds = diamanti.size();
336   _diamonds_check = 1;
337 }
338 /*--------------------------------------------------------------------
339   Visiting ...
340 --------------------------------------------------------------------*/
341 //per vedere se il grafo e' connesso
342 void Graph::visita(size_t node, std::set<size_t> &visitati) {
343   std::set<size_t> linkati;
344   visitati.insert(node);
345   for (size_t j=0; j<_nodes[node].Get_Nsons(); j++) {
346     linkati.insert(_nodes[node].Get_son(j));
347   }
348   for (size_t j=0; j<_nodes[node].Get_Npar(); j++) {
349     linkati.insert(_nodes[node].Get_par(j));
350   }
351   for (std::set<size_t>::iterator j = linkati.begin(); j != linkati.end()
       ; j++) {
352     if(visitati.find(*j) == visitati.end()) {
353       visita(*j, visitati);
354     }
355   }
356 };
357 /*--------------------------------------------------------------------*/
358 void Graph::visita_up(size_t node, std::set<size_t> &visitati) {
359   std::set<size_t> linkati;
360   visitati.insert(node);
361   for (size_t j=0; j<_nodes[node].Get_Npar(); j++) {
362     linkati.insert(_nodes[node].Get_par(j));
363   }
364   for (std::set<size_t>::iterator j = linkati.begin(); j != linkati.end()
       ; j++) {
365     if(visitati.find(*j) == visitati.end()) {
366       visita_up(*j, visitati);
367     }
368   }
369 };
370 /*--------------------------------------------------------------------*/
371 void Graph::visita_down(size_t node, std::set<size_t> &visitati) {
372   std::set<size_t> linkati;
373   visitati.insert(node);
374   for (size_t j=0; j<_nodes[node].Get_Nsons(); j++) {
375     linkati.insert(_nodes[node].Get_son(j));
```

```
376   }
377   for (std::set<size_t>::iterator j = linkati.begin(); j != linkati.end()
          ; j++) {
378     if(visitati.find(*j) == visitati.end()) {
379       visita_down(*j, visitati);
380     }
381   }
382 };
383 /*--------------------------------------------------------------------------*/
384 void Graph::visita_diamond(size_t analyzed, size_t node, std::set<size_t>
       &visitati, std::set<size_t> &diamanti) {
385   std::set<size_t> linkati;
386   visitati.insert(node);
387   for (size_t j=0; j<_nodes[node].Get_Npar(); j++) {
388     linkati.insert(_nodes[node].Get_par(j));
389   }
390   if (_diamonds_check == 0 ) {
391     for (std::set<size_t>::iterator j = linkati.begin(); j != linkati.end
          (); j++) {
392       if(visitati.find(*j) == visitati.end()) {
393         visita_diamond(analyzed, *j, visitati, diamanti);
394       }
395       else {
396         diamanti.insert(analyzed);
397         break;
398       }
399     }
400   }
401
402 };
403 /*--------------------------------------------------------------------------
404   Get
405 --------------------------------------------------------------------------*/
406 size_t Graph::brothers(size_t level) {
407   size_t cont=0;
408   for(size_t i=0;i<_Nnodes;i++){
409     if(_nodes[_ordernodes[i]].Get_level()==level) cont++;;
410     if(_nodes[_ordernodes[i]].Get_level()>level) break;
411   }
412   return cont;
413 }
414 /*--------------------------------------------------------------------------*/
415 size_t Graph::Rbrothers(size_t level) {
416   size_t cont=0;
417   for(size_t i=0;i<_Nnodes;i++){
418     if(_nodes[_Rordernodes[i]].Get_Rlevel()==level) cont++;;
419     if(_nodes[_Rordernodes[i]].Get_Rlevel()>level) break;
420   }
421   return cont;
422 }
423 /*--------------------------------------------------------------------------*/
424 size_t Graph::depth() {
425   return _nodes[_ordernodes[_Nnodes-1]].Get_level();
426 }
427 /*--------------------------------------------------------------------------*/
428 size_t Graph::Nnodes() {
429   return _Nnodes;
430 }
431 /*--------------------------------------------------------------------------*/
432 size_t Graph::outdeg(size_t nodo) {
433   return _nodes[_ordernodes[nodo]].Get_Nsons();
434 }
```

```
435 /*------------------------------------------------------------------------*/
436 size_t Graph::indeg(size_t nodo) {
437   return _nodes[_ordernodes[nodo]].Get_Npar();
438 }
439 /*------------------------------------------------------------------------*/
440 size_t Graph::Nnodes(size_t level) {
441   size_t control=0, sum=0;
442   for(size_t i=0;i<_Nnodes;i++){
443     control=_nodes[_ordernodes[i]].Get_level();
444     if(control==level) sum++;
445     if(control>level) break;
446   }
447   return sum;
448 }
449 /*------------------------------------------------------------------------*/
450 size_t Graph::RNnodes(size_t level) {
451   size_t control=0, sum=0;
452   for(size_t i=0;i<_Nnodes;i++){
453     control=_nodes[_Rordernodes[i]].Get_Rlevel();
454     if(control==level) sum++;
455     if(control>level) break;
456   }
457   return sum;
458 }
459 /*------------------------------------------------------------------------*/
460 size_t Graph::diamonds() {
461   if (_diamonds_check == 0) {
462     find_diamonds();
463   }
464   return _diamonds;
465 };
466 /*------------------------------------------------------------------------*/
467 size_t Graph::Nleaves() {
468   return _leaves.size();
469 }
470 /*------------------------------------------------------------------------*/
471 vector<size_t> Graph::leaves_depth(){
472   vector<size_t> depths;
473   for(size_t i=0;i<_leaves.size();i++)
474      depths.push_back(_nodes[_leaves[i]].Get_level());
475   return depths;
476 };
477 /*------------------------------------------------------------------------*/
478 vector<size_t> Graph::nodes_depth(){
479   vector<size_t> depths;
480   for(size_t i=0;i<_Nnodes;i++)
481      depths.push_back(_nodes[i].Get_level());
482   return depths;
483 };
484 /*------------------------------------------------------------------------*/
485 vector<size_t> Graph::leaves_indegrees(){
486   vector<size_t> indegrees;
487   for(size_t i=0;i<_leaves.size();i++)
488      indegrees.push_back(_nodes[_leaves[i]].Get_Npar());
489   return indegrees;
490 };
491 /*------------------------------------------------------------------------*/
492 vector<size_t> Graph::roots_outdegrees(){
493   vector<size_t> outdegrees;
494   for(size_t i=0;i<_roots.size();i++)
495      outdegrees.push_back(_nodes[_roots[i]].Get_Nsons());
496   return outdegrees;
```

```
497 };
498 /*--------------------------------------------------------------------------*/
499 vector<size_t> Graph::indegrees(){
500   vector<size_t> indegrees;
501   for(size_t i=0;i<_Nnodes;i++)
502      indegrees.push_back(_nodes[i].Get_Npar());
503   return indegrees;
504 };
505 /*--------------------------------------------------------------------------*/
506 vector<size_t> Graph::outdegrees(){
507   vector<size_t> outdegrees;
508   for(size_t i=0;i<_Nnodes;i++)
509      outdegrees.push_back(_nodes[i].Get_Nsons());
510   return outdegrees;
511 };
512 /*--------------------------------------------------------------------------*/
513 vector<double> Graph::mean_Routdegrees(){
514   vector<double> outdegrees;
515   for(size_t i=0;i<depth();i++)
516      outdegrees.push_back(mean_Routdeg(i+1));
517   return outdegrees;
518 };
519 /*--------------------------------------------------------------------------*/
520 vector<double> Graph::mean_outdegrees(){
521   vector<double> outdegrees;
522   for(size_t i=0;i<depth();i++)
523      outdegrees.push_back(mean_outdeg(i+1));
524   return outdegrees;
525 };
526 /*--------------------------------------------------------------------------*/
527 vector<size_t> Graph::Kin(){
528   find_Kin();
529   return _Kin;
530 }
531 /*--------------------------------------------------------------------------*/
532 vector<size_t> Graph::Kout(){
533   find_Kout();
534   return _Kout;
535 }
536 /*--------------------------------------------------------------------------
537  Statistics
538 --------------------------------------------------------------------------*/
539 double Graph::mean_depth(){
540   size_t sum=0;
541   for(size_t i=0;i<_Nnodes;i++)
542     sum+=_nodes[i].Get_level();
543   return double(sum)/_Nnodes;
544 };
545 /*--------------------------------------------------------------------------*/
546 double Graph::mean_leaves_depth(){
547   size_t sum=0;
548   for(size_t i=0;i<_leaves.size();i++)
549     sum+=_nodes[_leaves[i]].Get_level();
550   return double(sum)/_leaves.size();
551 };
552 /*--------------------------------------------------------------------------*/
553 double Graph::mean_Rdepth(){
554   size_t sum=0;
555   for(size_t i=0;i<_Nnodes;i++)
556     sum+=_nodes[i].Get_Rlevel();
557   return double(sum)/_Nnodes;
558 };
```

```
559 /*------------------------------------------------------------------------*/
560 double Graph::mean_outdeg(){
561   size_t sum=0,cont=0;
562   for(size_t i=0;i<_Nnodes;i++){
563     sum+=_nodes[i].Get_Nsons();
564     cont++;
565   }
566   return double(sum)/cont;
567 };
568 /*------------------------------------------------------------------------*/
569 double Graph::mean_outdeg(size_t level){
570   size_t sum=0,cont=0;
571   for(size_t i=0;i<_Nnodes;i++){
572     if(_nodes[i].Get_level()==level){
573       sum+=_nodes[i].Get_Nsons();
574       cont++;
575     }
576   }
577   return double(sum)/cont;
578 };
579 /*------------------------------------------------------------------------*/
580 double Graph::mean_Routdeg(size_t level){
581   size_t sum=0,cont=0;
582   for(size_t i=0;i<_Nnodes;i++){
583     if(_nodes[i].Get_Rlevel()==level){
584       sum+=_nodes[i].Get_Nsons();
585       cont++;
586     }
587   }
588   return double(sum)/cont;
589 };
590 /*------------------------------------------------------------------------*/
591 double Graph::cost(double lambda) {
592   double sum = 0.;
593   for(size_t i=0;i<_Nnodes;i++)
594     sum += cost(i, lambda);
595   return sum/_Nnodes;
596 };
597 /*------------------------------------------------------------------------*/
598 double Graph::cost(size_t nodo, double lambda) {
599   size_t Npar = _nodes[nodo].Get_Npar();
600   size_t kout = 0;
601   for(size_t i=0;i<Npar;i++){
602     size_t par = _nodes[nodo].Get_par(i);
603     kout += _nodes[par].Get_Nsons();
604   }
605   if(Npar == 0) return 1.0;
606   else return 1.0 - exp(-lambda*kout);
607 };
608 /*------------------------------------------------------------------------*/
609 double Graph::recursive_cost(double lambda) {
610   double sum = 0.;
611   for(size_t i=0;i<_Nnodes;i++)
612     sum += recursive_cost(i, lambda);
613   return sum/_Nnodes;
614 };
615 /*------------------------------------------------------------------------*/
616 double Graph::recursive_cost(size_t nodo, double lambda) {
617   size_t Npar = _nodes[nodo].Get_Npar();
618   size_t kout = 0;
619   double prev = 0.0;
620   for(size_t i=0;i<Npar;i++){
```

```
621     size_t par = _nodes[nodo].Get_par(i);
622     kout += _nodes[par].Get_Nsons();
623     prev += recursive_cost(par, lambda);
624   }
625   prev /= Npar;
626   if(Npar == 0) return 1.0;
627   else return prev - exp(-lambda*kout);
628 };
629 /*-------------------------------------------------------------------------
630   Miscellaneous
631 -------------------------------------------------------------------------*/
632 void Graph::HTD(double **scores, string **namescores, size_t nrows){
633   size_t position, Npar, dad;
634   //applico l'algoritmo top-down al grafo
635   for(size_t i=0;i<_Nnodes;i++){
636     position=_ordernodes[i];
637     Npar=_nodes[position].Get_Npar();
638     for(size_t k=0;k<nrows;k++){
639       for(size_t j=0;j<Npar;j++){
640         dad=_nodes[position].Get_par(j);
641         if(scores[k][position] > scores[k][dad]){
642           scores[k][position]=scores[k][dad];
643           namescores[k][position]=namescores[k][dad];
644         }
645       }
646     }
647   }
648 };
649 /*-------------------------------------------------------------------------*/
650 std::ostream &operator<<( std::ostream &os,  Graph &g){
651   os<<0<<endl;
652   for(size_t i=0; i<g._Nnodes; i++){
653     for(size_t j=0; j<g._nodes[i].Get_Nsons(); j++){
654       os<< i+1 << "\t" << g._nodes[i].Get_son(j)+1<< "\n";
655     }
656   }
657   return os;
658 };
659 /*-------------------------------------------------------------------------*/
```

A.1.3 class Packagegraph

This class is a tool that allows to analyze automatically more graphs at the same time.

packagegraph.hpp

```
1  #ifndef _PACKAGEGRAPH_HPP_
2  #define _PACKAGEGRAPH_HPP_
3
4  #include <string>
5  #include <vector>
6  #include <fstream>
7  #include "node.hpp"
8  #include "graph.hpp"
9  #include "funzioni.hpp"
10
```

```
11 using namespace std;
12
13 class Packagegraph {
14
15   public:
16     //Costruttori/Distruttore
17     Packagegraph();
18     Packagegraph(const char *edgefile);
19     ~Packagegraph();
20
21     //Assegnamento
22     Packagegraph &operator=(Packagegraph &obj);
23
24     //Setting
25     void add(Graph *g);
26     void clear();
27
28     //Get
29     string name();
30     size_t N();
31     size_t depth(size_t i);
32     size_t Nnodes(size_t i);
33     size_t outdeg(size_t grafo,size_t nodo);
34     size_t indeg(size_t grafo,size_t nodo);
35     size_t Nleaves(size_t grafo);
36
37     //Set
38     void set_name(string name);
39
40     //Quantities
41     size_t max_depth();
42     size_t Nnodes(size_t i, size_t level);
43     size_t RNnodes(size_t i, size_t level);
44
45     //Order Parameters
46     size_t diamanti();
47     size_t diamanti(size_t i);
48     double cost(size_t grafo, double lambda);
49     double recursive_cost(size_t grafo, double lambda);
50
51     //Means
52     double mean_Nnodes();
53     double mean_depth();
54     double mean_Rdepth();
55     double mean_leaves_depth();
56     double mean_leaves_depth(size_t i);
57     double mean_outdeg(size_t grafo);
58
59     double mean_meandepth();
60     double mean_depth(size_t i);
61     double mean_meanRdepth();
62
63     vector<size_t> brothers(size_t graph);
64
65     vector<size_t> Rbrothers(size_t graph);
66
67     //Distributions
68     vector<size_t> indegrees();
69     vector<size_t> outdegrees();
70     vector<size_t> nodes_depth();
71
72     vector<size_t> indegrees(size_t i);
```

```
73      vector<size_t> outdegrees(size_t i);
74      vector<size_t> nodes_depth(size_t i);
75
76      vector<double> mean_Routdegrees(size_t grafo);
77      vector<double> mean_outdegrees(size_t grafo);
78
79      vector<size_t> roots_outdegrees();
80      vector<size_t> roots_outdegrees(size_t i);
81
82      vector<size_t> leaves_indegrees();
83      vector<size_t> leaves_depth();
84
85      vector<size_t> leaves_indegrees(size_t i);
86      vector<size_t> leaves_depth(size_t i);
87
88    private:
89      vector<Graph*> _grafi;
90      size_t _Ngrafi;
91      string _nome;
92  };
93
94  #endif
```

packagegraph.cpp

```
1   #include "packagegraph.hpp"
2
3   /*------------------------------------------------------------------------
4    Costruttori
5   ------------------------------------------------------------------------*/
6   Packagegraph::Packagegraph(): _Ngrafi(0){};
7   /*------------------------------------------------------------------------*/
8   Packagegraph::Packagegraph(const char *edgefile) {
9     fstream f;
10    size_t help;
11    vector<size_t> edges1,edges2;
12    Graph *graphelp;
13    _nome=edgefile; _nome.pop_back(); _nome.pop_back(); _nome.pop_back();
          _nome.pop_back();
14    f.open(edgefile,ios::in);
15    f>>help;
16    f>>help;
17    while(!f.eof()){
18      if (help!=0) {
19        edges1.push_back(help);
20        f>>help;
21        edges2.push_back(help);
22        f>>help;
23      }
24      else {
25        graphelp = new Graph();
26        if(edges1.size()==0 || edges2.size()==0) {
27          f>>help;
28          continue;
29        }
30        graphelp->set(edges1,edges2);
31        _grafi.push_back(graphelp);
32        edges1.clear();
33        edges2.clear();
34        f>>help;
```

```
35      }
36    }
37          f.close();
38    if(edges1.size()!=0 && edges2.size()!=0) {
39      graphelp = new Graph();
40      graphelp->set(edges1,edges2);
41      _grafi.push_back(graphelp);
42    }
43    edges1.clear();
44    edges2.clear();
45    _Ngrafi=_grafi.size();
46  };
47  /*-------------------------------------------------------------------
48   Assegnamento
49  -------------------------------------------------------------------*/
50  Packagegraph &Packagegraph::operator=(Packagegraph &obj){
51    _Ngrafi=obj._Ngrafi;
52    Graph* help;
53    for(size_t i=0; i<_Ngrafi; i++) {
54      help=new Graph();
55      *help=*(obj._grafi[i]);
56      _grafi.push_back(help);
57    }
58    return *this;
59  };
60  /*-------------------------------------------------------------------
61   Distruttore
62  -------------------------------------------------------------------*/
63  Packagegraph::~Packagegraph(){
64    for(size_t i=0;i<_Ngrafi; i++){
65      delete _grafi[i];
66    }
67    _grafi.clear();
68  };
69  /*-------------------------------------------------------------------
70   Settings
71  -------------------------------------------------------------------*/
72  /*-------------------------------------------------------------------*/
73  void Packagegraph::add(Graph *g) {
74    _grafi.push_back(g);
75    _Ngrafi=_grafi.size();
76  };
77  /*-------------------------------------------------------------------*/
78  void Packagegraph::clear() {
79    for(size_t i=0;i<_Ngrafi; i++){
80      delete _grafi[i];
81    }
82    _grafi.clear();
83    _Ngrafi=0;
84    _nome="";
85  };
86  /*-------------------------------------------------------------------
87   Get
88  -------------------------------------------------------------------*/
89  string Packagegraph::name() {
90    return _nome;
91  };
92  /*-------------------------------------------------------------------*/
93  size_t Packagegraph::N() {
94    return _Ngrafi;
95  };
96  /*-------------------------------------------------------------------*/
```

```
97  size_t Packagegraph::Nnodes(size_t i) {
98    return _grafi[i]->Nnodes();
99  };
100 /*------------------------------------------------------------------------*/
101 size_t Packagegraph::depth(size_t i) {
102   return _grafi[i]->depth();
103 };
104 /*------------------------------------------------------------------------*/
105 size_t Packagegraph::outdeg(size_t grafo,size_t nodo){
106   return _grafi[grafo]->outdeg(nodo);
107 };
108 /*------------------------------------------------------------------------*/
109 size_t Packagegraph::indeg(size_t grafo,size_t nodo){
110   return _grafi[grafo]->indeg(nodo);
111 };
112 /*------------------------------------------------------------------------
113  Set
114 ------------------------------------------------------------------------*/
115 void Packagegraph::set_name(string name) {
116   _nome=name;
117 };
118 /*------------------------------------------------------------------------
119  Quantities
120 ------------------------------------------------------------------------*/
121 size_t Packagegraph::max_depth() {
122   size_t max = depth(0);
123   for(size_t i=0; i<_Ngrafi; i++)
124     max = _max(depth(i),max);
125   return max;
126 };
127 /*------------------------------------------------------------------------*/
128 size_t Packagegraph::Nnodes(size_t i, size_t level) {
129   return _grafi[i]->Nnodes(level);
130 };
131 /*------------------------------------------------------------------------*/
132 size_t Packagegraph::RNnodes(size_t i, size_t level) {
133   return _grafi[i]->RNnodes(level);
134 };
135 /*------------------------------------------------------------------------*/
136 size_t Packagegraph::Nleaves(size_t grafo) {
137   return _grafi[grafo]->Nleaves();
138 };
139 /*------------------------------------------------------------------------
140  Order parameters
141 ------------------------------------------------------------------------*/
142 double Packagegraph::cost(size_t grafo, double lambda) {
143   return _grafi[grafo]->cost(lambda);
144 };
145 /*------------------------------------------------------------------------*/
146 double Packagegraph::recursive_cost(size_t grafo, double lambda) {
147   return _grafi[grafo]->recursive_cost(lambda);
148 };
149 /*------------------------------------------------------------------------*/
150 size_t Packagegraph::diamanti() {
151   size_t help=0;
152   for(size_t i=0;i<_Ngrafi; i++){
153     help+=_grafi[i]->diamonds();
154   }
155   return help;
156 };
157 /*------------------------------------------------------------------------*/
158 size_t Packagegraph::diamanti(size_t i) {
```

```
159   return _grafi[i]->diamonds();
160 };
161 /*-------------------------------------------------------------------------
162  Means
163 -------------------------------------------------------------------------*/
164 double Packagegraph::mean_Nnodes(){
165   double sum=0;
166   for(size_t i=0; i<_Ngrafi; i++)
167     sum+=_grafi[i]->Nnodes();
168   return sum/_Ngrafi;
169 };
170 /*-------------------------------------------------------------------------*/
171 double Packagegraph::mean_depth(){
172   double sum=0;
173   for(size_t i=0; i<_Ngrafi; i++)
174     sum+=_grafi[i]->depth();
175   return sum/_Ngrafi;
176 };
177 /*-------------------------------------------------------------------------*/
178 double Packagegraph::mean_outdeg(size_t grafo){
179   return _grafi[grafo]->mean_outdeg();
180 };
181 /*-------------------------------------------------------------------------*/
182 double Packagegraph::mean_meandepth(){
183   double sum=0;
184   for(size_t i=0; i<_Ngrafi; i++)
185     sum+=_grafi[i]->mean_depth();
186   return sum/_Ngrafi;
187 };
188 /*-------------------------------------------------------------------------*/
189 double Packagegraph::mean_depth(size_t i){
190   return _grafi[i]->mean_depth();
191 };
192 /*-------------------------------------------------------------------------*/
193 double Packagegraph::mean_leaves_depth(){
194   double sum=0;
195   for(size_t i=0; i<_Ngrafi; i++)
196     sum+=_grafi[i]->mean_leaves_depth();
197   return sum/_Ngrafi;
198 };
199 /*-------------------------------------------------------------------------*/
200 double Packagegraph::mean_leaves_depth(size_t i){
201   return _grafi[i]->mean_leaves_depth();
202 };
203 /*-------------------------------------------------------------------------
204  Distributions
205 -------------------------------------------------------------------------*/
206 vector<size_t> Packagegraph::leaves_depth(size_t i){
207   return _grafi[i]->leaves_depth();
208 };
209 /*-------------------------------------------------------------------------*/
210 vector<size_t> Packagegraph::brothers(size_t graph){
211   vector<size_t> out;
212   for(size_t i=0;i<_grafi[graph]->depth();i++) {
213     out.push_back(_grafi[graph]->brothers(i+1));
214   }
215   return out;
216 };
217 /*-------------------------------------------------------------------------*/
218 vector<size_t> Packagegraph::Rbrothers(size_t graph){
219   vector<size_t> out;
220   for(size_t i=0;i<_grafi[graph]->depth();i++) {
```

```
221     out.push_back(_grafi[graph]->Rbrothers(i+1));
222   }
223   return out;
224 };
225 /*------------------------------------------------------------------------*/
226 vector<size_t> Packagegraph::leaves_depth(){
227   vector<size_t> out;
228   for(size_t i=0;i<_Ngrafi;i++) {
229     vector<size_t> temp = leaves_depth(i);
230     for(size_t j=0; j<temp.size(); j++)
231       out.push_back(temp[j]);
232   }
233   return out;
234 };
235 /*------------------------------------------------------------------------*/
236 vector<size_t> Packagegraph::nodes_depth(size_t i){
237   return _grafi[i]->nodes_depth();
238 };
239 /*------------------------------------------------------------------------*/
240 vector<size_t> Packagegraph::nodes_depth(){
241   vector<size_t> out;
242   for(size_t i=0;i<_Ngrafi;i++) {
243     vector<size_t> temp = nodes_depth(i);
244     for(size_t j=0; j<temp.size(); j++)
245       out.push_back(temp[j]);
246   }
247   return out;
248 };
249 /*------------------------------------------------------------------------*/
250 vector<size_t> Packagegraph::leaves_indegrees(size_t i){
251   return _grafi[i]->leaves_indegrees();
252 };
253 /*------------------------------------------------------------------------*/
254 vector<size_t> Packagegraph::leaves_indegrees(){
255   vector<size_t> out;
256   for(size_t i=0;i<_Ngrafi;i++) {
257     vector<size_t> temp = leaves_indegrees(i);
258     for(size_t j=0; j<temp.size(); j++)
259       out.push_back(temp[j]);
260   }
261   return out;
262 };
263 /*------------------------------------------------------------------------*/
264 vector<size_t> Packagegraph::roots_outdegrees(size_t i){
265   return _grafi[i]->roots_outdegrees();
266 };
267 /*------------------------------------------------------------------------*/
268 vector<size_t> Packagegraph::roots_outdegrees(){
269   vector<size_t> out;
270   for(size_t i=0;i<_Ngrafi;i++) {
271     vector<size_t> temp = roots_outdegrees(i);
272     for(size_t j=0; j<temp.size(); j++)
273       out.push_back(temp[j]);
274   }
275   return out;
276 };
277 /*------------------------------------------------------------------------*/
278 vector<size_t> Packagegraph::indegrees(size_t i){
279   return _grafi[i]->indegrees();
280 };
281 /*------------------------------------------------------------------------*/
282 vector<size_t> Packagegraph::indegrees(){
```

```
283   vector<size_t> out;
284   for(size_t i=0;i<_Ngrafi;i++) {
285     vector<size_t> temp = indegrees(i);
286     for(size_t j=0; j<temp.size(); j++)
287       out.push_back(temp[j]);
288   }
289   return out;
290 };
291 /*--------------------------------------------------------------------*/
292 vector<size_t> Packagegraph::outdegrees(size_t i){
293   return _grafi[i]->outdegrees();
294 };
295 /*--------------------------------------------------------------------*/
296 vector<double> Packagegraph::mean_Routdegrees(size_t grafo){
297   return _grafi[grafo]->mean_Routdegrees();
298 };
299 /*--------------------------------------------------------------------*/
300 vector<double> Packagegraph::mean_outdegrees(size_t grafo){
301   return _grafi[grafo]->mean_outdegrees();
302 };
303 /*--------------------------------------------------------------------*/
304 vector<size_t> Packagegraph::outdegrees(){
305   vector<size_t> out;
306   for(size_t i=0;i<_Ngrafi;i++) {
307     vector<size_t> temp = outdegrees(i);
308     for(size_t j=0; j<temp.size(); j++)
309       out.push_back(temp[j]);
310   }
311   return out;
312 };
313 /*--------------------------------------------------------------------*/
```

A.1.4 class Treemaker

This class builds trees and graphs through Sharing Tree model, Pang-Maslov model and Branching Tree model.

treemaker.hpp

```
1  #ifndef _TREEMAKER_HPP_
2  #define _TREEMAKER_HPP_
3
4  #include <string>
5  #include <vector>
6  #include <cmath>
7  #include <map>
8  #include <unordered_set>
9  #include "graph.hpp"
10 #include <gsl/gsl_randist.h>
11 #include <sys/time.h>
12
13 using namespace std;
14
15 class TreeMaker {
16
17   public:
18     //Costruttore/Distruttore
```

```
19      TreeMaker();
20      ~TreeMaker();
21
22      //Funzioni a Steps
23      Graph *YuleTree_Time(size_t Nsteps, double outdeg_mean);
24      Graph *RussianTree_Time(size_t NrootsInit, size_t Nsteps, double
            indeg_mean);
25      Graph *ReversedRussianTree_Time(size_t NleavesInit, size_t Nsteps,
            double outdeg_mean);
26
27      void YuleTree_Time(size_t Nsteps, double outdeg_mean, vector<size_t>
            &from, vector<size_t> &to);
28      void RussianTree_Time(size_t NrootsInit, size_t Nsteps, double
            indeg_mean, vector<size_t> &from, vector<size_t> &to);
29      void ReversedRussianTree_Time(size_t NleavesInit, size_t Nsteps,
            double outdeg_mean, vector<size_t> &from, vector<size_t> &to);
30
31      //Funzioni a Nnodes
32      Graph *YuleTree_Nodes(size_t Nnodes, double outdeg_mean);
33      Graph *RussianTree_Nodes(size_t NrootsInit, size_t Nnodes, double
            indeg_mean);
34      Graph *ReversedRussianTree_Nodes(size_t NleavesInit, size_t Nnodes,
            double outdeg_mean);
35
36      void YuleTree_Nodes(size_t Nodes, double outdeg_mean, vector<size_t>
            &from, vector<size_t> &to);
37      void RussianTree_Nodes(size_t NrootsInit, size_t Nnodes, double
            indeg_mean, vector<size_t> &from, vector<size_t> &to);
38      void ReversedRussianTree_Nodes(size_t NleavesInit, size_t Nnodes,
            double outdeg_mean, vector<size_t> &from, vector<size_t> &to);
39
40      //MC
41      Graph *MCTree(size_t NleavesInit, size_t Nsequence, size_t Nsymbols);
42      void MCTree(size_t NleavesInit, size_t Nsequence, size_t Nsymbols,
            vector<size_t> &from, vector<size_t> &to);
43
44
45    private:
46      gsl_rng *r;
47      vector<size_t> _from, _to;
48      bool **_presenza;
49      size_t _NleavesInit;
50      size_t _Nnodes;
51      size_t _Nsequence;
52      size_t _Nsymbols;
53
54      void Simboloricorrente(vector<size_t> &supporto, vector<vector<size_t
            >> &splittati);
55      void Ricorrente(vector<vector<size_t>> split1, size_t &esclude,
            size_t &esclude2, size_t nextnode, size_t &newnode, size_t &
            foglie);
56      size_t RandomSymbol();
57
58
59
60  };
61
62  #endif
```

treemaker.cpp

```
1  #include "treemaker.hpp"
2
3  /*--------------------------------------------------------------------
4   Costruttore
5  --------------------------------------------------------------------*/
6  TreeMaker::TreeMaker() {
7    r = gsl_rng_alloc(gsl_rng_default);
8    struct timeval deltatime;
9    gettimeofday(&deltatime, NULL);
10   gsl_rng_set(r,deltatime.tv_usec);
11 };
12 /*--------------------------------------------------------------------
13  Distruttore
14 --------------------------------------------------------------------*/
15 TreeMaker::~TreeMaker(){
16   gsl_rng_free(r);
17 };
18 /*--------------------------------------------------------------------
19  Functions for Time
20 --------------------------------------------------------------------*/
21 void TreeMaker::RussianTree_Time(size_t NrootsInit, size_t Nsteps, double
       indeg_mean, vector<size_t> &from, vector<size_t> &to){
22   from.clear();
23   to.clear();
24   size_t newnode=NrootsInit+1, rho, dad;
25
26   for(size_t i=0; i<Nsteps;i++){
27     rho = gsl_ran_poisson (r, indeg_mean);
28     for(size_t j=0; j<rho; j++) {
29       dad = floor(gsl_ran_flat (r, 1.0, newnode));
30       from.push_back(dad);
31       to.push_back(newnode);
32     }
33     newnode++;
34   }
35 };
36 /*--------------------------------------------------------------------*/
37 void TreeMaker::ReversedRussianTree_Time(size_t NleavesInit, size_t
      Nsteps, double outdeg_mean, vector<size_t> &from, vector<size_t> &to)
      {
38   from.clear();
39   to.clear();
40   size_t newnode=NleavesInit+1, rho, son;
41
42   for(size_t i=0; i<Nsteps;i++){
43     rho = gsl_ran_poisson (r, outdeg_mean);
44     for(size_t j=0; j<rho; j++) {
45       son = floor(gsl_ran_flat (r, 1.0, newnode));
46       from.push_back(newnode);
47       to.push_back(son);
48     }
49     newnode++;
50   }
51 };
52 /*--------------------------------------------------------------------*/
53 void TreeMaker::YuleTree_Time(size_t Nsteps, double outdeg_mean, vector<
      size_t> &from, vector<size_t> &to){
54   from.clear();
55   to.clear();
```

```
56    size_t selected = 1, generati = 1, rho=0, exrho=0;
57
58    while(rho==0) rho = gsl_ran_poisson (r, outdeg_mean);
59
60    for(size_t j=0; j<Nsteps;j++){
61      if (rho!=0) exrho=rho;
62      for(size_t i=0; i<rho; i++) {
63        generati++;
64        from.push_back(selected);
65        to.push_back(generati);
66      }
67      rho = gsl_ran_poisson (r, outdeg_mean);
68      if(selected == generati) {
69        selected -= exrho;
70        j--;
71      }
72      selected++;
73    }
74  };
75  /*-----------------------------------------------------------------------*/
76  Graph *TreeMaker::YuleTree_Time(size_t Nsteps, double outdeg_mean) {
77    vector<size_t> from, to;
78    YuleTree_Time(Nsteps, outdeg_mean, from, to);
79    Graph *g=new Graph;
80    g->set(from, to);
81    from.clear();
82    to.clear();
83    return g;
84  }
85  /*-----------------------------------------------------------------------*/
86  Graph *TreeMaker::RussianTree_Time(size_t NrootsInit, size_t Nsteps,
        double indeg_mean) {
87    vector<size_t> from, to;
88    RussianTree_Time(NrootsInit, Nsteps, indeg_mean, from, to);
89    Graph *g=new Graph;
90    g->set(from, to);
91    from.clear();
92    to.clear();
93    return g;
94  }
95  /*-----------------------------------------------------------------------*/
96  Graph *TreeMaker::ReversedRussianTree_Time(size_t NleavesInit, size_t
        Nsteps, double outdeg_mean) {
97    vector<size_t> from, to;
98    ReversedRussianTree_Time(NleavesInit, Nsteps, outdeg_mean, from, to);
99    Graph *g=new Graph;
100   g->set(from, to);
101   from.clear();
102   to.clear();
103   return g;
104 }
105 /*-----------------------------------------------------------------------
106  Functions
107 -----------------------------------------------------------------------*/
108 void TreeMaker::RussianTree_Nodes(size_t NrootsInit, size_t Nnodes,
        double indeg_mean, vector<size_t> &from, vector<size_t> &to){
109   from.clear();
110   to.clear();
111   std::set<size_t> linkati;
112   size_t newnode=NrootsInit+1, rho, dad;
113
114   while(linkati.size() <= Nnodes) {
```

```
115     rho = gsl_ran_poisson (r, indeg_mean);
116     for(size_t j=0; j<rho; j++) {
117       dad = floor(gsl_ran_flat (r, 1.0, newnode));
118       from.push_back(dad);
119       linkati.insert(dad);
120       to.push_back(newnode);
121       linkati.insert(newnode);
122       if (linkati.size() >= Nnodes) return;
123     }
124     newnode++;
125   }
126 };
127 /*-----------------------------------------------------------------*/
128 void TreeMaker::ReversedRussianTree_Nodes(size_t NleavesInit, size_t
        Nnodes, double outdeg_mean, vector<size_t> &from, vector<size_t> &to)
        {
129   from.clear();
130   to.clear();
131   std::set<size_t> linkati;
132   size_t newnode=NleavesInit+1, rho, son;
133 
134   while(linkati.size() <= Nnodes) {
135     rho = gsl_ran_poisson (r, outdeg_mean);
136     for(size_t j=0; j<rho; j++) {
137       son = floor(gsl_ran_flat (r, 1.0, newnode));
138       from.push_back(newnode);
139       linkati.insert(newnode);
140       to.push_back(son);
141       linkati.insert(son);
142       if (linkati.size() >= Nnodes) return;
143     }
144     newnode++;
145   }
146 };
147 /*-----------------------------------------------------------------*/
148 void TreeMaker::YuleTree_Nodes(size_t Nnodes, double outdeg_mean, vector<
        size_t> &from, vector<size_t> &to){
149   from.clear();
150   to.clear();
151   size_t selected = 1, generati = 1, rho=0, exrho=0;
152 
153   while(rho==0) rho = gsl_ran_poisson (r, outdeg_mean);
154 
155   while(generati != Nnodes) {
156     if (rho!=0) exrho=rho;
157     for(size_t i=0; i<rho; i++) {
158       generati++;
159       from.push_back(selected);
160       to.push_back(generati);
161       if (generati==Nnodes) break;
162     }
163     rho = gsl_ran_poisson (r, outdeg_mean);
164     if(selected == generati) {
165       selected -= exrho;
166     }
167     selected++;
168   }
169 };
170 /*-----------------------------------------------------------------*/
171 Graph *TreeMaker::YuleTree_Nodes(size_t Nnodes, double outdeg_mean) {
172   vector<size_t> from, to;
173   YuleTree_Nodes(Nnodes, outdeg_mean, from, to);
```

```
174   Graph *g=new Graph;
175   g->set(from, to);
176   from.clear();
177   to.clear();
178   return g;
179 }
180 /*-------------------------------------------------------------------*/
181 Graph *TreeMaker::RussianTree_Nodes(size_t NrootsInit, size_t Nnodes,
        double indeg_mean) {
182   vector<size_t> from, to;
183   RussianTree_Nodes(NrootsInit, Nnodes, indeg_mean, from, to);
184   Graph *g=new Graph;
185   g->set(from, to);
186   from.clear();
187   to.clear();
188   return g;
189 }
190 /*-------------------------------------------------------------------*/
191 Graph *TreeMaker::ReversedRussianTree_Nodes(size_t NleavesInit, size_t
        Nnodes, double outdeg_mean) {
192   vector<size_t> from, to;
193   ReversedRussianTree_Nodes(NleavesInit, Nnodes, outdeg_mean, from, to);
194   Graph *g=new Graph;
195   g->set(from, to);
196   from.clear();
197   to.clear();
198   return g;
199 }
200 /*-------------------------------------------------------------------*/
201 void TreeMaker::MCTree(size_t NleavesInit, size_t Nsequence, size_t
        Nsymbols, vector<size_t> &from, vector<size_t> &to) {
202   from.clear();
203   to.clear();
204   _from.clear();
205   _to.clear();
206
207   _NleavesInit=NleavesInit;
208   _Nsequence=Nsequence;
209   _Nsymbols=Nsymbols;
210
211   unordered_set<size_t> nodi[_NleavesInit];
212   vector<size_t> supporto;
213   vector<vector<size_t>> split1;
214
215   for(size_t i=0;i<_NleavesInit;i++) supporto.push_back(i);
216
217   for(size_t i=0; i<_NleavesInit; i++){
218 //    for(size_t j=0; j<(gsl_ran_flat(r, 1.0, Nsequence+1)); j++){
219     for(size_t j=0; j<_Nsequence; j++){
220 riprovaci:
221       nodi[i].insert(RandomSymbol());
222       if(nodi[i].size()!=j+1) goto riprovaci;
223     }
224   }
225   // matrice dei simboli //
226   _presenza=new bool*[_NleavesInit];
227   for(size_t i=0; i<_NleavesInit; i++) _presenza[i]=new bool[_Nsymbols];
228
229   for(size_t i=0; i<_NleavesInit; i++){
230     for(size_t j=0; j<Nsymbols; j++){
231       if(nodi[i].find(j+1)!=nodi[i].end()){
232         _presenza[i][j]=1;
```

```
233       }
234       else _presenza[i][j]=0;
235     }
236   }
237
238   size_t mas=0;
239   size_t cont=0;
240   size_t sim=0;
241   for(size_t j=0; j<_Nsymbols; j++){
242     for(size_t i=0; i<supporto.size(); i++){
243       if(_presenza[supporto[i]][j]) cont++;
244     }
245     if(mas<cont) {
246       mas=cont;
247       sim=j;
248     }
249     cont=0;
250   }
251
252   vector<size_t> add;
253   split1.push_back(add);
254   for(size_t k=0; k<supporto.size(); k++){
255     if(_presenza[supporto[k]][sim]){
256       split1[0].push_back(supporto[k]);
257     }
258   }
259   _Nnodes=split1[0].size();
260
261   size_t esclude=0, esclude2=0, nextnode=1, newnode=1;
262   size_t foglie=0;
263
264   Ricorrente(split1, esclude, esclude2, nextnode, newnode, foglie);
265
266   from=_from;
267   to=_to;
268   _from.clear();
269   _to.clear();
270
271   for(size_t j=0; j<_NleavesInit; j++) delete _presenza[j];
272   delete[] _presenza;
273 };
274 /*-------------------------------------------------------------------*/
275 Graph *TreeMaker::MCTree(size_t NleavesInit, size_t Nsequence, size_t
      Nsymbols){
276   vector<size_t> from, to;
277   MCTree(NleavesInit, Nsequence, Nsymbols, from, to);
278   Graph *g=new Graph;
279   g->set(from, to);
280   from.clear();
281   to.clear();
282   return g;
283 }
284 /*-------------------------------------------------------------------*/
285 size_t TreeMaker::RandomSymbol(){
286   return floor(gsl_ran_flat (r, 1.0, _Nsymbols+1));
287 //  return floor(gsl_ran_gaussian (r, 100))+(_Nsymbols/2.);
288 };
289 /*-------------------------------------------------------------------*/
290 void TreeMaker::Simboloricorrente(vector<size_t> &supporto, vector<vector
      <size_t>> &splittati){
291   vector<size_t> ecco, esclusi;
292   bool chk;
```

```
293   size_t ciclo=0, max_sum, massimo, contatore;
294   do {
295     massimo=0;
296     contatore=0;
297     for(size_t j=0; j<_Nsymbols; j++){
298       chk=0;
299       for(size_t h=0; h<ecco.size(); h++){
300         if(j==ecco[h]) {
301           chk=1;
302           break;
303         }
304       }
305       if(chk) continue;
306       for(size_t i=0; i<supporto.size(); i++){
307         if(_presenza[supporto[i]][j]) contatore++;
308       }
309       if(massimo<contatore) {
310         massimo=contatore;
311         max_sum=j;
312       }
313       contatore=0;
314     }
315     if(massimo==0) {
316       vector<size_t> add;
317       if(ciclo>0){
318         splittati.push_back(add);
319         for(size_t k=0; k<supporto.size(); k++){
320           splittati[ciclo].push_back(supporto[k]);
321         }
322       }
323       if(ciclo==0){
324         for(size_t k=0; k<supporto.size(); k++){
325           splittati.push_back(add);
326           splittati[ciclo].push_back(supporto[k]);
327           ciclo++;
328         }
329       }
330       break;
331     }
332     vector<size_t> add;
333     splittati.push_back(add);
334     for(size_t k=0; k<supporto.size(); k++){
335       if(_presenza[supporto[k]][max_sum]){
336         splittati[ciclo].push_back(supporto[k]);
337         _presenza[supporto[k]][max_sum]=0;
338       }
339       else{
340         esclusi.push_back(supporto[k]);
341       }
342     }
343     supporto=esclusi;
344     esclusi.clear();
345     ciclo++;
346     if(supporto.size()==0) break;
347   } while(1);
348 };
349 /*--------------------------------------------------------------------*/
350 void TreeMaker::Ricorrente(vector<vector<size_t>> split1, size_t &esclude
      , size_t &esclude2, size_t nextnode, size_t &newnode, size_t &foglie)
      {
351   vector<vector<size_t>> split2;
352   vector<vector<size_t>> split3;
```

```
353   for(size_t i=0;i<split1.size();i++){
354     if(split1[i].size()==1) {
355       nextnode++;
356       foglie++;
357       if(foglie==_Nnodes) return;
358     }
359     if(split1[i].size()>1){
360       split2.clear();
361       Simboloricorrente(split1[i], split2);
362       for(size_t r=0;r<split2.size();r++){
363         newnode++;
364         _from.push_back(nextnode);
365         _to.push_back(newnode);
366       }
367       nextnode++;
368     }
369     for(size_t j=0;j<split2.size();j++) {
370       if(split2[j].size()>1)  split3.push_back(split2[j]);
371       else{
372         esclude2++;
373         foglie++;
374         if(foglie==_Nnodes) return;
375       }
376     }
377   }
378   nextnode+=esclude;
379   esclude=esclude2;
380   esclude2=0;
381   if(split2.size()!=0) Ricorrente(split3, esclude, esclude2, nextnode,
          newnode, foglie);
382   else return;
383 };
384 /*---------------------------------------------------------------------*/
```

A.2 Programs for simulations and analysis

A.2.1 mrmodel.cpp

This program use the library to simulate and analyze Sharing Trees.

```
1  /*---------------------------------------------------------------------
2  MRmodel nome nnodi kappa esse
3  ---------------------------------------------------------------------*/
4
5  #include <iostream>
6  #include <fstream>
7  #include <cstdlib>
8
9  #include "libreria.hpp"
10
11 using namespace std;
12
13 int main(int argc, char **argv) {
14   size_t N_STEPS = 1;
15   size_t NALBERI = 10;
16
17   string language=argv[1];
18   size_t nnodi=atoi(argv[2]);
```

```
19    size_t kappa=atoi(argv[3]);
20    size_t esse=atoi(argv[4]);
21
22
23    TreeMaker alberelli;
24    Packagegraph *scatola;
25    scatola=new Packagegraph;
26
27    string path="../STmodel/";
28
29    Distribuzioni distr(path,language);
30    Scatterplot scat(path,language);
31
32    cerr<<"ST simulation..."<<endl;
33    for(size_t n=1; n<=N_STEPS; n++) {
34      cerr<<n<<"\t";
35      for(size_t i=0; i<NALBERI; i++) {
36        scatola->add(alberelli.MCTree(nnodi, kappa, esse));
37      }
38      distr.outdegrees(scatola);
39      distr.Nnodes(scatola);
40      distr.depth(scatola);
41      distr.leaves(scatola);
42      distr.leaves_position(scatola);
43
44      scat.levels_outdegrees(scatola);
45      scat.depth_Nnodes(scatola);
46      scat.levels_outdegmean(scatola,4);
47      scat.Rlevels_outdegmean(scatola,1,1);
48      scat.levels_Nnodes(scatola,2);
49
50      scatola->clear();
51    }
52    cerr<<endl;
53    distr.clear();
54    scat.clear();
55
56    delete scatola;
57
58    return 0;
59  }
```

A.2.2 tarantola.cpp

A program to obtain the whole data analysis through the library.

```
1   /*-------------------------------------------------------------------------
2   tarantola language `ls`
3   -------------------------------------------------------------------------*/
4
5   #include <iostream>
6   #include <fstream>
7   #include <cstdlib>
8   #include <string>
9
10  #include "libreria.hpp"
11
12  using namespace std;
```

```
13
14 int main(int argc, char **argv) {
15
16   string path="../outputC/";
17   string language=argv[1];
18
19   Packagegraph *pac;
20   Distribuzioni distr(path,language);
21   Scatterplot scat(path,language);
22   Lineplot line(path,language);
23
24   for(int i=2; i<argc; i++) {
25     cout<<argv[i]<<"\n";
26     pac = new Packagegraph(argv[i]);
27
28     distr.indegrees(pac);
29     distr.roots_outdeg(pac);
30     distr.leaves_indeg(pac);
31     distr.Nnodes(pac);
32     distr.depth(pac);
33     distr.leaves_position(pac);
34     distr.outdegrees(pac, 5, 1000);
35
36     scat.levels_indegrees(pac);
37     scat.levels_outdegrees(pac);
38     scat.levels_Nnodes(pac,3);
39     scat.levels_outdegmean(pac,4);
40     scat.Rlevels_Nnodes(pac,3);
41     scat.Rlevels_outdegmean(pac,3, 500);
42     scat.LeavesIndegrees_LeavesLevels(pac);
43     scat.depth_Nnodes(pac);
44     scat.Vnodes_Nnodes(pac);
45     scat.leaves_Nnodes(pac,1);
46     scat.depth_Nnodes(pac);
47     scat.depth_Nnodes_outdeg(pac, 5, 2);
48     scat.meanoutdeg_Nnodes_depth(pac);
49
50     line.outdegree(pac,5,1000);
51     delete pac;
52   }
53
54   return 0;
55 }
```

A.3 Programs for data conversion

Programs in this section are written in gvpr [GN00].

A.3.1 Papera.gv

A program to paste together the subgraphs produced by doxygen.

```
1 #!/bin/bash
2 BEGIN {
3   graph_t out = graph("", "D");
```

```
}
N {
  edge_t e = fstin($);
  node(out, $.label);
  while(e) {
    node_t h = e.head;
    node_t t = e.tail;

    node_t oh = node(out, h.label);
    node_t ot = node(out, t.label);

    if(isEdge_sg(out,ot,oh,0) != NULL) {
      edge_t found = isEdge_sg(out,ot,oh,0);
      if(found.style != e.style)
        printf("ERRORE:: due stili diversi per lo stesso edge!\n");
    }

    edge_t oe = edge_sg(out,ot,oh,0);
    oe.style = e.style;
    oe.color = e.color;
    e = nxtin(e);

  }
}
END {
        write(out);
}
```

A.3.2 Bilimetta.gv

A program to erase templates from graphs.

```
BEG_G {
  $tvtype = TV_en;
  graph_t out = clone(NULL,$G);
}
E {
  if($.color=="orange") {
    edge_t linkOUT = fstout($.head);
    while(linkOUT) {
      if(isNode(out, linkOUT.tail.name) != NULL) {
        if(isNode(out, linkOUT.head.name) != NULL) {
          if(isEdge_sg(out, isNode(out, linkOUT.tail.name), isNode(out,
              linkOUT.head.name), 0) != NULL) {
            delete(out,isEdge_sg(out, isNode(out, linkOUT.tail.name),
                isNode(out, linkOUT.head.name), 0));
            if(isNode(out, linkOUT.head.name) != NULL) {
              if(isNode(out, $.tail.name) != NULL) {
                edge_sg(out, isNode(out, $.tail.name), isNode(out,
                    linkOUT.head.name), 0);
              }
            }
          }
        }
      }
      linkOUT = nxtout(linkOUT);
    }
    edge_t linkIN = fstin($.head);
```

```
24      while(linkIN) {
25        if(isNode(out,linkIN.tail.name) != NULL && isNode(out,linkIN.head.
              name) != NULL) {
26          if(isEdge_sg(out, isNode(out,linkIN.tail.name), isNode(out,linkIN
                .head.name), 0) != NULL){
27            delete(out,isEdge_sg(out, isNode(out,linkIN.tail.name), isNode(
                  out,linkIN.head.name), 0));
28            if($.tail!=linkIN.tail) {
29              if(isNode(out, linkIN.tail.name) != NULL) {
30                if(isNode(out, $.tail.name) != NULL) {
31                  edge_sg(out,isNode(out, linkIN.tail.name),isNode(out, $.
                        tail.name),0);
32                }
33              }
34            }
35          }
36        }
37        linkIN = nxtin(linkIN);
38      }
39      if(isNode(out,$.head.name) != NULL){
40        delete(out, isNode(out,$.head.name));
41      }
42    }
43  }
44  END {
45    write(out);
46  }
```

A.3.3 Numerello.gv

A program to convert .dot files in adjacency lists.

```
1   BEG_G {
2     printf("0\n");
3     $tvtype = TV_ne;
4     int i = 1;
5   }
6
7   N {
8     node_t n = $;
9     n.mionome = i;
10    i++;
11  }
12
13  E {
14    edge_t e = $;
15    node_t h = e.head;
16    node_t t = e.tail;
17    printf("%s %s\n", t.mionome, h.mionome);
18  }
```

Bibliography

[Bja] Bjarne Stroustrup. C++ programming language. http://www.stroustrup.com/. Accessed: 2015-04-06.

[Bri] Alfred Aho Peter Weinberger Brian Kernighan. Awk programming language. http://cm.bell-labs.com/cm/cs/awkbook/index.html. Accessed: 2015-04-06.

[Git] GitHub. People and repositories on github. https://github.com/about/press. Accessed: 2015-04-06.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and experience*, 30(11):1203–1233, 2000.

[Gui] Guido van Rossum. Python programming language. https://www.python.org/. Accessed: 2015-04-06.

[GVSZ14] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387, New York, NY, USA, 2014. ACM.

[Lin] Linus Torvalds. git. http://git-scm.com/. Accessed: 2015-04-06.

[MWe] Cade Metz (Wired editor). Github conquered google, microsoft and everyone else. http://www.wired.com/2015/03/github-conquered-google-microsoft-everyone-else/. Accessed: 2015-04-06.

[PJ] Tom Preston-Werner Chris Wanstrath PJ Hyett. Github. https://github.com/. Accessed: 2015-04-06.

[PM13] Tin Yau Pang and Sergei Maslov. Universal distribution of component frequencies in biological and technological systems. *Proceedings of the National Academy of Sciences*, 110(15):6235–6239, 2013.

[Sun] James Gosling Sun Microsystems. Java programming language. `http://www.oracle.com/technetwork/java/index.html`. Accessed: 2015-04-06.

[TGP89] David H. Taenzer, Murthy Ganti, and Sunil Podar. Problems in object-oriented software reuse. In *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989.*, pages 25–38, 1989.

[vH] Dimitri van Heesch. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`. Accessed: 2015-04-06.

[Wol] Wolfam Research. Mathematica. `http://www.wolfram.com/mathematica/`. Accessed: 2015-04-06.