Small world characteristics of FLOSS distributions

Jaap Boender[1] and Sara Fernandes[2]

[1] Foundations of Computing Group
Department of Computer Science
School of Science and Technology
Middlesex University, London, UK
J.Boender@mdx.ac.uk
[2] UNU-IIST, Macao SAR China
sara.fernandes@iist.uni.edu

Abstract. Over the years, Free/Libre Open Source Software (FLOSS) distributions have become more and more complex and recent versions contain tens of thousands of packages. This has made it impossible to do quality control by hand. Instead, distribution editors must look to automated methods to ensure the quality of their distributions.
In the present paper, we present some insights into the general structure of FLOSS distributions. We notably show that such distributions have the characteristics of a small world network: there are only a few important packages, and many less important packages. Identifying the important packages can help editors focus their efforts on parts of the distribution where errors will have important consequences.

1 Introduction

It has long been a standard method in computing science to divide complex systems into components [10]. System processes are placed into separate components so that all of the data and functions inside each component are semantically related. Because of this principle, it is often said that components are modular and cohesive.

The modularity of components allows for easy debugging and maintenance, since components are small and generally focus on only one task. Cohesiveness allows components to work together towards a greater goal.

Free/Libre Open Source Software (FLOSS) software distributions are very good examples of component-based systems. They are very large (over 35 000 packages in the latest Debian release), heterogenous (they contain packages written by different teams, in different languages, with different release schedules, etc.).

Since FLOSS distributions are becoming more and more popular and complex, the fact is that to assure quality by hand becomes an impossible task. This forces the editors to search for automated methods in order to ensure the quality of their distributions.

The aim of this paper is to present some insights into the general structure and characteristics of FLOSS distributions. Identifiying these can help distribution editors in concentrating their resources. For example, well-connected packages with errors will have a greater impact than packages that have few connections. As another example, if there is a cluster of strongly connected packages, it might be useful to assign maintenance of these packages to the same person or team.

We have used the Debian and Mandriva distributions for our experiments. Debian was chosen because it is a very large distribution (in number of packages). Also, the semantics of its packaging system are well-defined, which makes it easier to interpret results. Mandriva was chosen because it is one of the distributions using the RPM system. The semantics of RPM are different from those of the Debian packaging system, so it is possible to assess whether characteristics are general for all FLOSS distributions or artefacts of a specific packaging system.

The rest of the paper is structured as follows. Section 2 introduces some background and related work. Section 3 presents the methodology. Section 4 presents the results and its analysis. Finally, Section 5 presents some conclusions pointing to envisaged future work.

2 Background and definitions

FLOSS applications are often distributed in the form of packages, which are bundles of related components necessary to compile or run an application. For many FLOSS packages, the source code is freely available and reuse of the code for derivative works is encouraged.

Because of this, resource reuse is considered to be a natural pillar: a package is often dependent on resources in some other packages to function properly. Package dependencies often span between project development teams, and since there is no central control over which resources from other packages are needed or in what way, the software system self-organizes in to a collection of discrete, interconnected components.

The relationships between packages can be used to compute relevant quality measures, for example in order to identify particularly fragile components [1, 4].

In a distribution, there are two main types of relationships, with totally different significance: dependencies (where one package needs another to function properly) and conflicts (where packages cannot function together). Also, syntactically, dependencies are directional, while conflicts are not. And finally, there are two different types of dependencies, conjunctive (the 'normal' kind, which can only be satisfied in one way) and disjunctive (where a dependency may be satisfied by one out of a list of packages). For formal definitions of these concepts, please refer to [7].

An example can be found in figure 1. If we want to install the package `alpha`, we will need to install `bravo` (a conjunctive dependency) and `charlie` or `delta` (a disjunctive dependency). Furthermore, `delta` is in conflict with `foxtrot`, so it is not possible to install them both together. In this case, the disjunctive

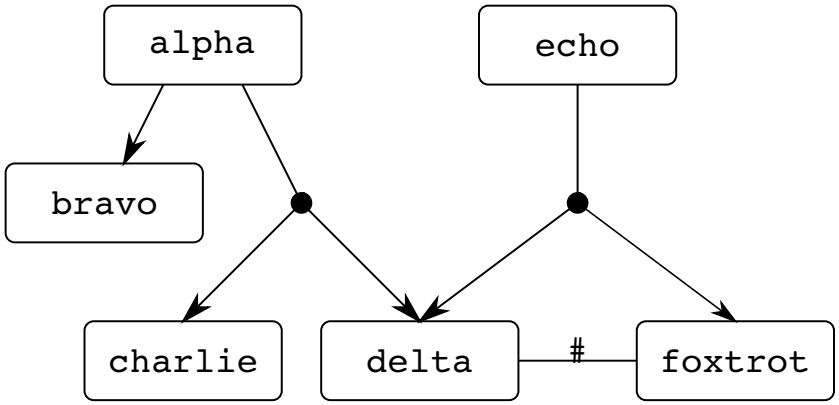

Fig. 1. Simple repository

dependency of `echo` on `delta` or `foxtrot` can be satisfied with either of these packages, but not both. This is because of the conflict: there is no problem in installing `charlie` and `delta` together.

If we look at a FLOSS distribution (which, after all, is nothing more than a set of packages with their relationships) as a whole, we can also identify quality measures. For this purpose, we will look at distributions as *networks*.

A network is an unweighted graph $G = (V, E)$ where V denotes a vertex set and E an edge set. Vertices represent discrete objects in a system, such as social actors, economic agents, computer programs, Internet sites, or biological producers and consumers. Edges represent interactions among these actors, such as Internet sites linking to each other.

Many of these networks exhibit a property known as the *small world* property, first described by Stanley Milgram in his famous paper about the 'six degrees of separation' concept [8]. A small world network is distinguished by the fact that the number of hops needed to reach one vertex from another is relatively small, at least compared to a random network.

This property has been observed in many products of human and biological activity, including the graph of the Internet [5], the graph of the World Wide Web [3] and other complex networks [2].

Formally, a network is deemed small-world if it satisfies two properties:

- The clustering coefficient, defined as the probability that two neighbours of the same vertex are connected, is significantly higher than that of random networks.
- On average, the shortest path length between two vertices is low (on the same order as that of random networks).

A small world graph has a high clustering coefficient (at least with respect to random networks), and also a low average shortest path length. Moreover,

in a small world network, these two properties result in a network that consists of clusters, whose central vertices (or nodes) are highly connected, creating hubs. These have many connections, whereas the other nodes can have very few connections; thus, the degree distribution conforms to the well-known Pareto principle, also known as the 80/20 law—a small number of nodes have a high number of connections.

The application of the small world concept to FLOSS distributions is not new; it has already been proven that FLOSS distributions have small world characteristics [6, 9].

However, in both papers, it is not clear which methodology has been used[3]. As we shall see in the next section, this can have an important effect on results, especially since in FLOSS distributions, not all edges are equivalent (dependencies are radically different from conflicts, for example).

In [6], the numbers suggest that all package relations have been treated equally, without regard for semantics or directionality.

In [9], small world characteristics are shown for both the graph of dependencies and the graph of conflicts. This at least resolves the problem of semantics, because dependencies and conflicts are treated separately.

The paper, though, contains a puzzling claim: it is stated that the small-world method does not hold for packages with very few or very many dependencies (the so-called 'saturation effect'). This claim is puzzling in that the entire basis of the small world phenomenon is the distinction between packages with few and packages with many dependencies. As we shall see, this is especially important in FLOSS distributions, as it gives us insights into the structure of the distribution.

In the next section, we present the methodology used in our measurements.

3 Methodology

In this section, we present the exact methods we have used to generate the distribution graphs and the measurements we have executed on them. We shall also discuss the different significance of these methods.

A distribution can be seen as a graph, where the packages are the vertices and relationships (dependencies and conflicts) are the edges. However, as we have seen before, not all edges are the same—the significance of a dependency is diametrically opposite to that of a conflict, and treating these edges the same can result in confusing results.

It thus becomes clear that, in order to draw any meaningful conclusions from a distribution graph, it is important to know *how* this graph is generated. We propose three methods, each with their own advantages and disadvantages.

Method 1 involves treating every edge equally, irrespective of their significance. We conflate conjunctive dependencies, disjunctive dependencies, conflicts, and any other relations between packages that are present. This gives us a distribution graph where two packages are connected if and only if they are possibly involved in some way in determining each other's installability.

[3] Queries to the authors of these papers have gone unanswered.

In general, this is an overapproximation, since not every package that is mentioned as a dependency is actually used. The main advantage of this method is that it is easy to compute.

In **method 2**, we connect two packages p and q if there is a path in the dependency graph from p to q. Another way of expressing the same concept is that q must appear in the transitive closure of the dependencies of p. In this way, a package p is connected to a package q if there is a possibility that q is installed to satisfy some dependency of p.

This method still is an overapproximation, though less so than the first method. It is not much more difficult to compute, though it no longer takes conflicts into account. The main advantage here is that now transitive dependencies are considered.

For **method 3** we make use of *strong dependencies* [1], a concept that subsumes both dependencies and conflicts. Informally, a package p strongly depends on another package q if and only if it is impossible to install p without also installing q.

Note that strong dependencies are a property of the entire distribution, not just of the packages involved: whether p strongly depends on q depends on the entire distribution, for *every* installation of p has to include q.

The advantage of using the strong dependency graph is that now we have a unified graph, where every edge has the same meaning, but which still takes both conflicts and dependencies into account. It is a slight underapproximation, since conjunctive dependencies where none of the alternatives is obligatory, but one will have to be installed nonetheless, do not end up as strong dependencies.

The main disadvantage is that the strong dependency graph is more difficult to compute, since it involves doing installability checks, e.g. with a SAT solver. However, it can still be done within reasonabletime (a few minutes for the latest distributions).

We have used all of these three methods to generate distribution graphs and measure their characteristics. In the next section we will present the results and discuss their significance.

4 Results and Discussion

In this section we present the results of measurements on the graphs obtained by the three different methods described above, for both the Debian and Mandriva distributions. We also discuss the significance of these measurements and the conclusions that can be drawn from them.

4.1 Debian

Let us start with the raw data for the latest Debian distribution (version 7) on the standard AMD64 architecture. Using the three different methods, we have generated distribution graphs and determined several key indices.

Method	V	E	CC	APL	Comp	CpAvg	LComp
1	35 982	85 190	0.38	3.43	2 251	15.98	33 558
2	35 982	2 386 389	0.26	0.91	2 229	16.15	33 582
3	35 982	1 588 322	0.28	0.91	2 280	15.78	33 537

distribution used: debian/amd64 7 stable

In this table, first we have the number of vertices (V) and edges (E) in the graph. At first glance, it might seem surprising that the method 1 graph has so few edges compared to the other two, especially since it uses every possible package relation, but this can be explained by the fact that the method 2 and 3 graphs are transitive.

Then there are the main small world indices, the clustering coefficient (CC) and average shortest path length (APL). Both these characteristics show a small world effect in all three graphs, though we must note that the average shortest path length index is not indicative for graphs 2 and 3: these graphs being transitive, there is either no path between two vertices or a path of length 1. This results in the average shortest path length being less than 1.

Note that even though the number of edges is vastly higher in graphs 2 and 3, the clustering coefficient is actually lower. This might seem strange (more edges should result in more connection, hence a higher probability of vertices being neighbours) but it is caused by the fact that these graphs are transitive: the fact that vertices have a higher probability of being connected is balanced out by the fact that vertices have more neighbours to begin with.

We also show the component structure of the graph; in this case we use weakly connected components while ignoring direction. We show the number of components (Comp), average component size (CpAvg) and the size of the largest component (LComp). We can see from these measures that distributions consist of one huge connected component, encompassing over 90 percent of the distribuition, with the rest of the distribution consisting of isolated or near-isolated packages.

In the rest of this section, we shall limit ourselves to a discussion of the strong dependency graph (method 3), as it is the most interesting one from a semantic perspective (every edge has an equal, well-defined meaning). All three graphs, however, exhibit small world characteristics.

Another characteristic of small world networks, demonstrated in Figure 2, is that the distribution of degrees of their vertices follows a power law—as mentioned before, the Pareto principle. There should be few vertices with many edges and many vertices with few edges.

We can see this in the figure: the degree distribution forms a straight line in a double logarithmic plot.

In Figure 3, we show the same plot, but now with in degrees and out degrees separated. We can see from this that the distribution consists of three main types of packages:

- Many packages with a small in degree and a small out degree;
- A few packages with a small in degree, but a large out degree;
- A few packages with a large in degree, but a small out degree.

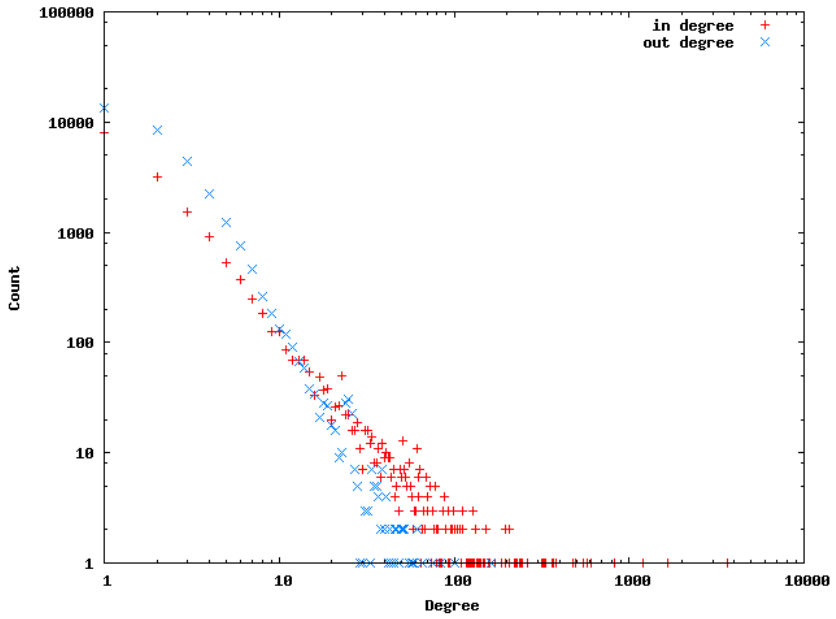

Fig. 2. Distribution of degrees in Debian `stable` (strong dependencies)

Notably, there are no packages that have both a large in degree and a large out degree.

Examining these packages can shed some light on why this is the case. Here is a table with on the left the 10 packages in Debian with the highest in degree, and on the right the 10 packages with the highest out degree.

Highest in degrees			Highest out degrees		
Name	In degree	Out degree	Name	In degree	Out degree
`gcc-4.7-base`	31 708	0	`gnome-desktop-environment`	0	945
`libc-bin`	31 707	2	`gnome`	1	944
`multiarch-support`	31 706	4	`task-gnome-desktop`	0	746
`libgcc1`	31 706	4	`gnome-core-devel`	0	710
`libc6`	31 706	4	`gnome-core`	3	677
`zlib1g`	25 514	5	`kde-full`	0	643
`libselinux1`	21 695	5	`task-kde-desktop`	0	560
`liblzma5`	21 201	5	`ontv`	0	493
`libbz2-1.0`	21 108	6	`kde-standard`	1	473
`tar`	20 681	5	`kde-telepathy`	0	382

We see that the packages on the left are mostly libraries and base packages (`libc`, for example, or `tar`), and that on the right there are mostly high-level packages (*metapackages*) such as KDE or GNOME.

Figure 4 shows this in a schematic way.

This data can be corroborated in a different way as well: in Debian, most packages carry *tags* that identify things like their role, whether they are part of

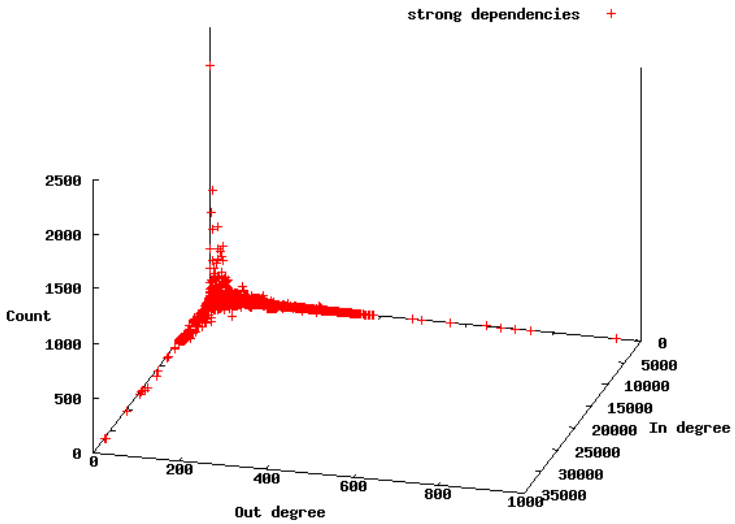

Fig. 3. In and out degrees in Debian `stable` (strong dependencies)

a larger software suite, or the programming language they are implemented in. If we look at the tags that occur more than once in the packages in the top 10 shown above, we get the following table:

High in degrees		High out degrees	
Tag	Count	Tag	Count
implemented-in::c	7	role::metapackage	6
role::shared-lib	4	interface:x11	5
devel::packaging	2	uitoolkit::gtk	5
interface::commandline	2	suite::gnome	4
role::program	2	suite::kde	2
scope::utility	2		
suite::gnu	2		
use::storing	2		
works-with::archive	2		
works-with::file	2		

It seems that packages with a high in degrees are often shared libraries and implemented in C. Both of these characteristics point to system libraries.

Similarly, the packages with a high out degree are mostly metapackages, using the X window system and part of the GNOME or KDE suites. This also confirms the structure as shown in Figure 4.

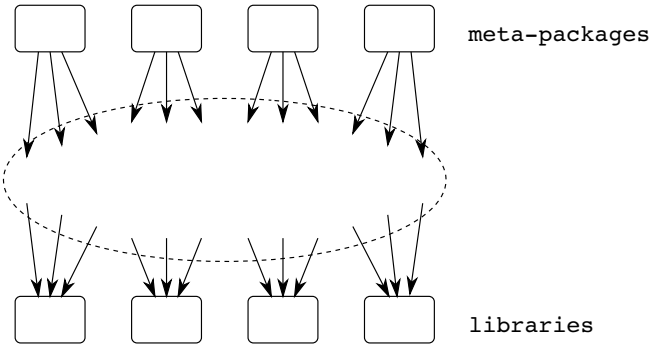

Fig. 4. Schematic repository structure

4.2 Mandriva

Debian, of course, is not the only distribution available. We have also analysed another distribution, Mandriva, which is based on RPM, a different but comparable packaging system. Let us see if the conclusions drawn for Debian also hold for Mandriva. First the raw data:

Method	V	E	CC	APL	Comp	CpAvg	LComp
1	7 566	84 855	0.47	7.49	289	26.18	7 273
2	7 566	1 170 721	0.25	0.94	333	22.72	7 230
3	7 566	721 162	0.25	0.94	339	22.32	7 223

distribution used: mandriva/x86_64 2010.1 main

Allowing for the smaller size of the distribution, the values are roughly similar. However, if we look at the first graph, we see that it has almost the same number of edges as its Debian equivalent, for roughly a fifth of the packages. This can be explained by a difference in semantics between the Debian package format and RPM: RPM packages and dependencies are more fine-grained, which makes for more edges in the graph[4]. We can also see this from the higher average shortest path length: there are on average more intermediate dependencies between two packages than in Debian.

We also see that the clustering coefficient of Mandriva is higher than that of Debian in the first graph, but slightly lower in the second and third. The higher clustering coefficent in the first graph can be explained by the difference in semantics mentioned above—there are simply much more dependencies, and the balancing effect of transitive graphs is not present here. For the difference in the second and third graphs, we will have to look at the actual degree distribution.

This degree distribution is shown in Figure 5. We see that there is still a power law distribution, but it is not as clear as for Debian.

[4] This might seem at odds with the fact that there are many less packages in Mandriva than in Debian. The Debian distribution is, hoewever, very extensive and contains many packages not present in Mandriva.

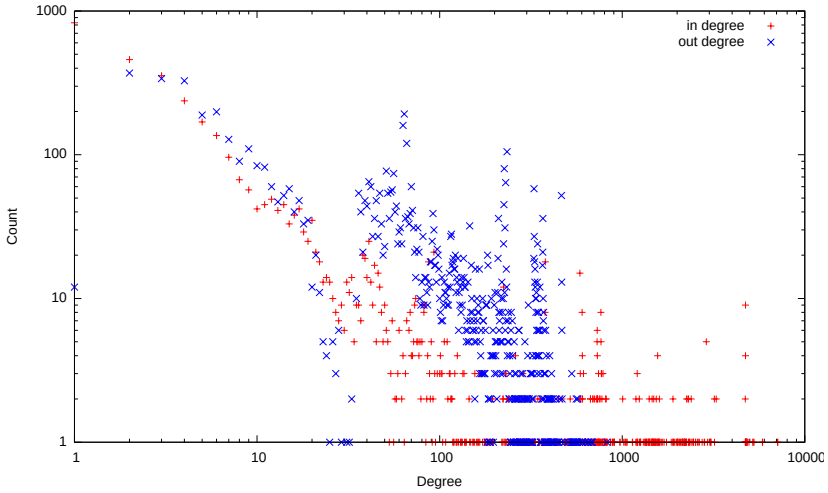

Fig. 5. Distribution of degrees in Mandriva `2010.1`, strong dependencies

In Figure 6, we have the degree distribution with in and out degrees broken down. This figure explains best why the clustering coefficient is lower: the figure looks comparable with its Debian equivalent (Figure 3), but there are several outlying packages with a high out degree.

This is due to a specificity in Mandriva packaging: there are several packages that do not install files themselves, but are only there to fulfill a certain task, such as installing the X window system. These are similar to the meta-packages mentioned above, but they can have dependencies that are not at all related to each other. This explains both the lower clustering coefficient (dependencies of these meta-packages may not depend on each other) and the outlying packages (these will be like meta-packages in that they have a high out degree, but a low in degree).

Looking at the top 10 of high degree packages in Mandriva corroborates this:

Highest in degrees			Highest out degrees		
Name	In degree	Out degree	Name	In degree	Out degree
`dash-static`	7 106	0	`task-kde4-devel`	0	824
`glibc`	7 105	1	`kimono-devel`	0	683
`lib64termcap2`	5 862	2	`ruby-kde4-devel`	0	682
`bash`	5 861	3	`qyoto-devel`	1	681
`perl-base`	5 274	2	`ruby-qt4-devel`	1	680
`libgcc1`	5 206	2	`smoke4-devel`	4	675
`libstdcc++6`	5 201	3	`kdenetwork4-devel`	0	655
`lib64pcre0`	4 946	4	`kipi-plugins-devel`	0	651
`lib64lzma2`	4 836	2	`kdepim4-devel`	0	645
`xz`	4 825	5	`kdeplasma-addons-devel`	0	630

We see the same distribution structure: library packages on the left, with high in degree andlow out degree, and metapackages (and tasks) on the right, with high out degree and low in degree.

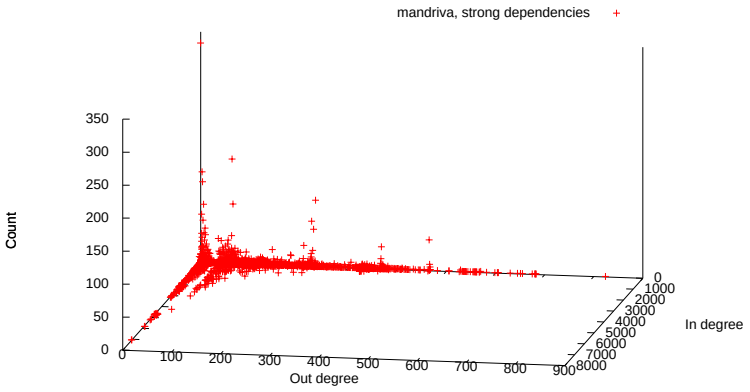

Fig. 6. In and out degrees in Mandriva `2010.1`, strong dependencies

If we look at the list of task packages in Mandriva, they all have a low in degree and a high out degree. `task-kde4-devel` is simply the most glaring example; there are about 30 task packages in the entire Mandriva distribution, but they all have an out degree of over 100, and an in degree of under 10.

All in all, Mandriva shows much the same structure as Debian, and if we consider the task packages to be metapackages as well (which, in a sense, they are), the structure of Mandriva conforms to Figure 4 as well.

4.3 General

We can conclude that there are two kinds of vulnerable packages in a distribution: the meta-packages that are vulnerable because they pull in a great amount of other packages, each with its own possible bugs, and libraries that are vulnerable because if they contain bugs, a large number of other packages will be influenced.

Identifying these packages in a distribution can help distribution editors focus their efforts.

5 Conclusion and future work

In the previous sections, we have presented a clear and precise method for creating graphs of FLOSS distributions, using three different methods. The most interesting of these three involves strong dependencies, where we create a single graph that incorporates information from both dependencies and conflicts.

We have shown that these graphs have small world characteristics for both Debian and Mandriva, and that packages can be divided into three distinct

groups: meta-packages (top-level packages with many dependencies), libraries (base packages that many other packages depend on), and other packages.

Distribution editors can use these data to identify packages that are in need of extra surveillance, or that must be treated with extra care during upgrades or repairs.

Meta-packages have many dependencies, and therefore have a high probability of depending on a faulty package. This makes them excellent yardsticks for measuring the health of an entire software suite, since they will easily be influenced by errors in their dependencies.

On the other hand, library packages must be treated with care, since errors in them can have huge effects on the rest of the distribution. Release policies for these packages should therefore be more conservative than for less crucial packages.

The fact that FLOSS distributions have small world characteristics, provided that the methodology is clear, allows us interesting insights into the structure of these distributions that, we hope, will be used to make distribution editors' lives easier.

5.1 Future work

We have so far used Debian and Mandriva for our tests. We do not expect huge differences in the results for other distributions such as Ubuntu and OpenSUSE, but it would be good to test these nonetheless—as we have seen from the discussion of the results for Mandriva, even small differences can be of interest.

It would also be interesting to have these tests run on a daily basis over a distribution to see how the data changes. This could not only be interesting for scientists who want to track changes to the structure of a distribution, but also for distribution editors, who could then identify vulnerable packages daily. They could also identify the effect of changes in dependencies on the distribution structure.

6 Acknowledgments

This work is partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project. Work developed at IRILL. This work is also supported by Macao Science and Technology Development Fund (MSTDF), File No. 019 / 2011 / A1.

References

1. Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S.: Strong dependencies between software components. In: ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. pp. 89–99. IEEE Computer Society, Washington, DC, USA (2009)

2. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. 74(1), 47–97 (Jan 2002)
3. Barabási, A.L., Albert, R.: Emergence of Scaling in Random Networks. Science 286(5439), 509–512 (1999), http://www.sciencemag.org/cgi/content/abstract/286/5439/509
4. Boender, J.: Efficient computation of dominance in component systems (short paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) Software Engineering and Formal Methods, Lecture Notes in Computer Science, vol. 7041, pp. 399–406. Springer Berlin Heidelberg (2011)
5. Caldarelli, G., Marchetti, R., Pietronero, L.: The fractal properties of internet. EPL (Europhysics Letters) 52(4), 386 (2000), http://stacks.iop.org/0295-5075/52/i=4/a=386
6. LaBelle, N., Wallingford, E.: Inter-package dependency networks in open-source software. CoRR cs.SE/0411096 (2004)
7. Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-bas ed software distributions. In: ASE. pp. 199–208 (2006)
8. Milgram, S.: The small world problem. Psychology Today 1(1), 60–67 (1967)
9. Nair, R., Nagarjuna, G., Ray, A.K.: Semantic structure and finite-size saturation in scale-free dependency networks of free software. ArXiv e-prints (Jan 2009)
10. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, second edition. Addison Wesley Professional (2002)