Dependency Networks in Open-Source Software Packages – Report

Jeff Chern: jchern

December 10, 2012

1 Introduction

The real-world practice of software engineering frequently revolves around the central pinnacles of modularity and decomposition, guidelines that aim to produce software that satisfies its motivating needs, is robust to error and changes imposed by future requirements, and is easy to navigate and maintain by its developers. The current state of the art, however, depends on the individual and collected experience of developers (and sometimes their corporate managers) to drive almost all design decisions. There are few, if any, metrics that readily capture the fundamental properties of software codebases; currently employed metrics tend to be very empirical, historical, such as counting bugs per lines of code as a proxy for code quality or stability. But, these empirical and historical measures do not capture the state of the code as it evolves in real time, nor do they yield a "bigger picture" view of how all the software pieces fit together. Since the beginning of the so-called Software Age several decades ago, researchers have been exploring various ways to identify and quantify the most important qualities of software, which were listed above. However, the work to-date has been piecemeal, and hence, there is a goal to move towards a more unified perspective on software: if we could invent or identify a set of metrics that capture and encourage the desired qualities of software, while reducing the undesired, it could lead to a better understanding of software, within the industry and without, regarding exactly what works well, and what doesn't. Having this set of metrics would allow software engineers to compare alternative designs more analytically and scientifically, as compared to current, experientially and qualitatively-based software management practices.

For this project, I studied the Java package dependency network hosted by the Maven Repository (mvnrepository.com). I computed summary statistics to describe this network at coarse granularity, and then attempt several methods to automatically generate some "big picture" comprehensions regarding the contents of the network.

2 Prior Work

Watts and Strogatz (1998) in their classic paper suggest that many disparate types of systems, whether natural or man-made, follow so-called ”small world” behaviors, i.e. the network exhibits high clustering and a short characteristic path length relative to completely random networks of the same size. Researchers have quickly begun to apply this small-world conceptual framework to begin unravelling the heretofore unknown properties of software systems, most saliently, the interactions and dependencies between the ”moving pieces” of software, whether they be classes or methods, or packages and libraries.

LaBelle and Wallingford (2004) analyzed open-source packages from the Debian and FreeBSD software repositories for compile-time dependencies. They found that the dependency networks in both repositories were small world networks. For example, the Debian network was found to have average clustering $C = 0.52$ and characteristic length $L = 3.34$, where characteristic length is the average path length between all pairs of reachable nodes. The largest component in this network contained 88% of all the nodes in the graph, where nodes represent packages, with a diameter of 31 hops. Similarly, the BSD network had $C = 0.56$ and $L = 2.86$. Not surprisingly, these researchers found that some of the most highly depended-upon packages in the Debian repo provided widely-reusable functionality such as programming language libraries (C/C++), reusable GUI widgets, or XML parsing. They also found that both networks' in- and out-degrees followed power-law distributions, i.e. $P(k) \approx k^{-\alpha}$, for positive α, which classify also as so-called ”scale-free” networks, in addition to already being small-worlds. The scale-free degree distribution implies that there exist many nodes with low degree alongside fewer nodes of orders-of-magnitude(s) higher degree, with the number of orders related to the α rate of each particular network. This team explored the dependency network within a diverse sets of software packages, in the C programming language, and placed it on the map of known small-world and scale-free networks. The summary statistics section of my project mimics, in large part, this team's approach, but for packages written in and distributed for use with the Java programming language. My summary statistics will seek to determine whether the new network is indeed a small-world with scale-free distributions.

Valverde and Sole (2007) analyzed a large assortment of object-oriented applications and software projects to find whether software at this level of granularity also exhibit small-world properties. Specifically, they graphed the dependencies between classes in open-source C++ as well as Java programs, and constructed class graphs based on static/compile-time relationships in their code. (Runtime dependencies are the counterpart to compile-time dependencies, and are much more important in newfangled, dynamically typed and interpreted languages such as Python and Ruby, but exploring software in these languages unfortunately lies beyond the scope of my project). This pair also found their networks to be scale-free networks, and small worlds: i.e. path length scales with the log of node count, with much higher clustering than random graphs of similar size. They also noted software's tendency to become disorderly over time, and posited that a ”breakdown of modularity” likely coincides with the emergence, or phase-changes into the small-world regime. In other words, when classes start depending on or interacting with far-flung and unrelated classes, then the path lengths throughout the entire software system begin to shorten. Entering this regime allows the effects of code changes to propagate throughout the software system in complicated ways, making it unstable and disorderly. For my research, though, small-world properties can reveal how heavily used the most popular packages are, and how security or programming flaws can easily propagate to a large segment of the graph, in just a few steps.

3 Data Collection

The package-repository site I chose for this project, mvnrepository.com, does not provide an API that exposes metadata about the artifacts it contains, so I created a web scraper to crawl through the site, looking for artifacts and their dependencies and users. The site, does not provide a full list of artifacts it hosts, either, but it does provide a list of about 250 popular tags and artifacts filed under them. I obtained a starting set of artifacts for my web crawl by extracting metadata for the latest 'release' version all artifacts mentioned in this tag list, or any latest version if an artifact did not have a marked 'release' version. This list totaled about 11,572 artifacts with different names. I did not take measures to detect and unify duplicates with different org or package names; for example, org.eclipse.jetty jetty-server and org.mortbay.jetty jetty, share the same evolutionary history, provide the same functionalities, and are both called Jetty Server Core. Taking artifacts' names at face value is a limitation of my approach, but still a systematic one at that.

Mvnrepository.com advertises that it current hosts about 378,000 artifacts in a small plot on its front page. Granted, many of these artifacts may be isolated by themselves or with other artifacts in small islands, or represent the myriad versions of every artifact, but the majority of artifacts, particularly the most well-known and well-used, will be reachable in some sort of large connected component of dependencies.

My web crawler works on the following basic principle: given an artifact, the crawler identifies its dependencies and users (dependants), and enqueues previously unseen packages for future crawling, BFS style. It then adds edges to the graph, one for each package –> dependency pair it finds. It does not add an edge for dependant/user –> package yet, deferring it till the dependant/user itself is crawled. (Gratuitous metaphor: it's like painting with a paintbrush in only one direction, to keep the strokes consistent.) Given a single initial package, or an initial set of packages, I should be able to find the entire connected component, by following dependency and dependent edges until there are no new artifacts to visit. In practice, loading each package's web page is slow, so I wrote my web crawler to support multiple fetcher threads, for faster crawling.

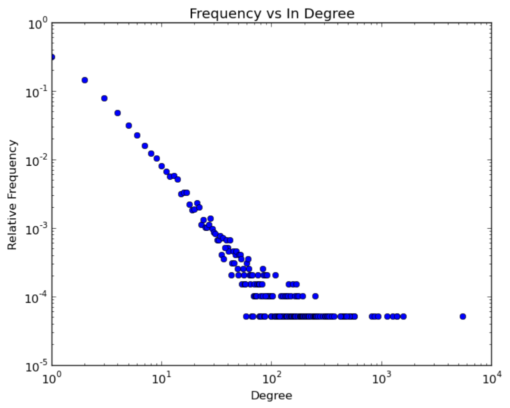
For simplicity, I started the crawl with just a single Java package, a popular open-source graphing package called jfreechart. My crawler's output contains 107,143 edges among 19,591 nodes. Some of the artifacts' pages on the site were unreachable with a 404 error, and this resulted in 7 separate connected components, whereas there should have been only one, by definition, since everything was reached from one starting node. I simply discarded the 6 smaller CCs, containing 33 nodes in total. Worse, still, a dependency network, by its very nature, should not contain any cyclical dependencies, but for some unknown reasons, several cycles and self-loops appeared in my edgelist; instead of preprocessing this data, like for the disconnected CCs, I simply marked traversed nodes in any BFS or DFS, so that they wouldn't be followed more than once.

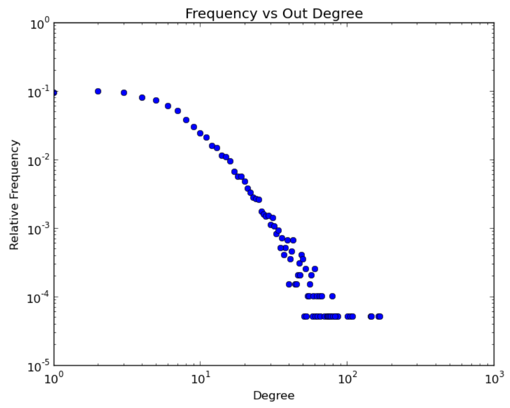
4 Small World and Scale-Free Distribution

For small world analysis, I flattened my network to be a undirectd graph. Average clustering $\bar{C} = 0.260$, characteristic distance (mean shortest path, over all node pairs) was 3.71, and diameter was 13. For a corresponding $G_{n,m}$ random network, average clustering was $\bar{C} = 0.00635$, characteristic distance was 4.40, and diameter was the same.

Since the network has comparable distance and diameter, but much higher clustering, I conclude that it is a "small-world".

As for degree distributions, "plfit" reported the in-degrees to follow a power law with $\alpha = 2.11$ and $x_m in = 11$, and for out-degrees, $\alpha = 3.50$ and $x_m in = 18$. It is worth noting briefly that the in-degrees span approximately an order of magnitude higher than the out-degrees, which makes intuitive sense, because a popular package can be cited or used by others ad infinitum, while it makes no sense for a package to depend on more and more others. Interestingly, the out-degree α is higher than the typically observed values between 2 and 3, although it is close to the value for references to papers, of approximately 3, from an example in lecture.





I also computed, for each node, the number of other nodes that directly or indirectly depend on it. This led to some nodes with values on the order of 10^5, i.e. being depended on by almost half of the entire network in some way. Some very popular packages included log4j, a widely used logging library; JBoss and Jetty web servers; Spring, an enterprise objects framework; and JUnit, the standard Java testing framework. As a side note, given the popularity of the two web servers just mentioned, and the number of packages using them, a security flaw in one could easily affect many projects. Exactly how many projects, my data does not say – it only counts the number of packages, not the number of real-world projects using each.

5 Detecting Layers

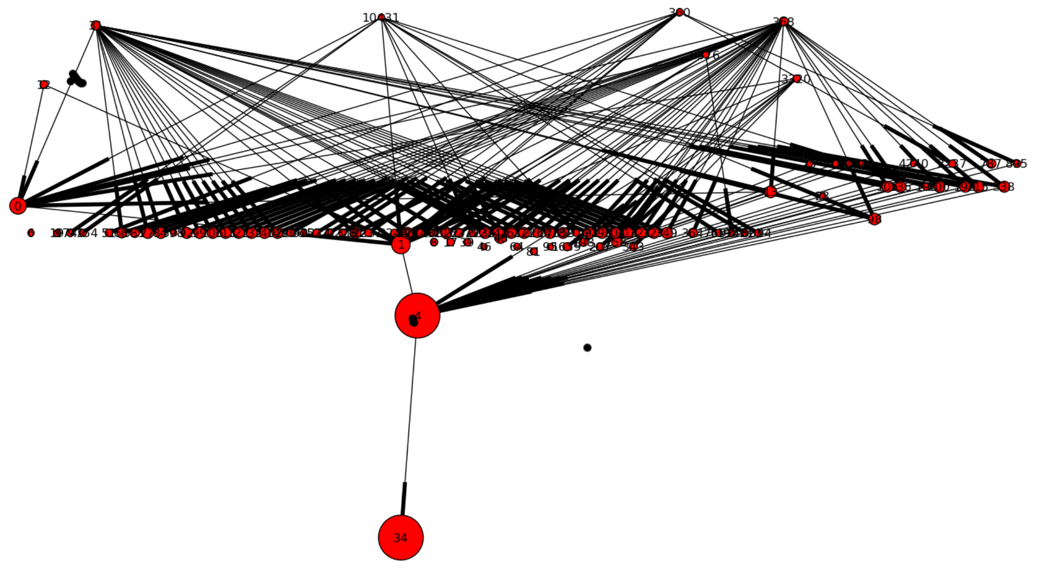
I used the following heuristic to ”detect” layers of packages in the network.

1. Choose some starting node, assign it $level = 0$.
2. Assign a vote to all successors (dependencies) and predecessors (users) of this node. Successors get $level - 1$ since they are lower level, and predecessors get $level + 1$.
3. Repeat this process recursively on neighboring nodes:
4. For each node: Take the mean/median of the current votes for this node, as our tentative ”level”, and use it to assign votes to all of its neighbors.
5. Repeat on unvisited nodes (i.e. nodes that haven't voted yet)

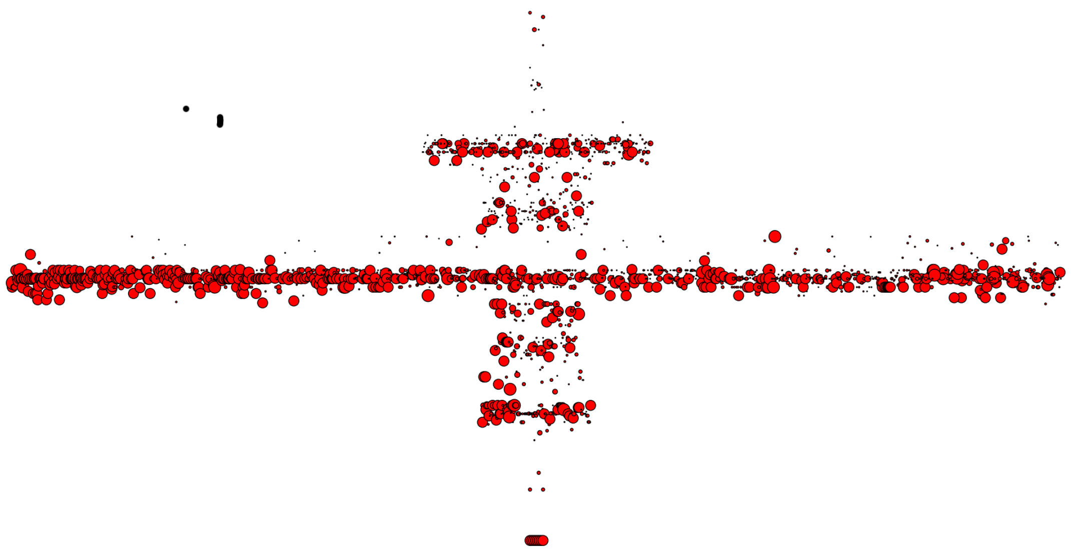
Note that each edge will be voted with twice, once going with the arrow, and once against. This approach is not mathematically elegant, but it was simple to implement, and allows nodes a chance to adjust their neighbors once. If the mean is used, fractional levels will become possible, which is useful for identifying nodes that are ”in between” two other layers of nodes.

5.1 Evaluation

The layering seems to work acceptably for relatively shallow graphs, as seen below:



But it doesn't work as well for very large, complex graphs, such as the entire dependency network, shown here with edges removed for clarity:



One glaring problem with my level heuristic is that it doesn't accommodate software stacks that are shallow or deep together. For example, if one stack featured 8 levels of modularity between lowest and highest level, while another featured 3, the shallow stack wouldn't be able to "expand" to fit across the range of high and low level. This problem arises from the way my heuristic doesn't understand anything about the dependency network's semantics at all.

Another is that it is sensitive to which node is used as the starting node, as levels can tend to diverge more if starting from the fringes of the network, as opposed to the center.

However, it does provide an "at a glance" overview of how the entire repository is laid out.

5.2 Other methods attempted

1. I attempted to use spectral decomposition on the adjacency matrix, to recursively divide the network into disjoint clusters, but this approach did not work at all, as each eigenvector yielded little information on the nodes (almost all nodes were assigned a group near 0, with very few, having significant values). 2. After the level grouping from above, I implemented a market basket algorithm to try to detect which packages were used together frequently. This approach worked for small graphs, but was too slow for any level with lots (more than 10) nodes. I had thought that partitioning the graph first by "level" would allow Market Basket to run more effectively, since packages do not usually choose from the entire world of dependencies, but from the layers or levels that are directly underneath it. In a sense, I theorized that each level would essentially be "shopping" for nodes from the level directly below, but I wasn't able to verify this hypothesis.

6 Conclusion

In conclusion, I have successfully identified the Maven repository as another small-world and scale-free network, but was not very successful in developing a high-level automatic comprehension of the relationships in the network.

References

[1] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *Journal of Theoretical Computer Science*, 2004.

[2] Lovro Subelj and Marko Bajec. Community structure of complex software systems. *Physica*, 2011.

[3] Sergi Valverde and Ricard Solé. Hierarchical small-worlds in software architecture. *Special Issue on Software Engineering and Complex Networks Dynamics of Continuous, Discrete and Impulsive Systems Series B*, 2007.

[4] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440–442, 1998.