Six degrees of inclusion

Analyzing the structure of code networks in software projects

Independent research - Pablo Estrada [pabloem@ucla.edu]

Abstract

Computer programs are created using programming languages. With these programming languages, the programmer can define program behavior and functionality. The functionality of a program is usually split among files, which encapsulate functions of a certain type. When a function in file A wants to be able to utilize some functionality defined in file B, file A needs to include file B. In this project, I look at networks formed by file inclusions in software projects developed in the C programming language.

Obtaining data

The data for this research was obtained from the code bases of 3 free software projects. The code of these projects is publicly available on the provided websites or repositories, in the references.

- xv6 – The xv6 operating system is a re-implementation of the original specification for the UNIXv6 operating system. It was developed at MIT for the operating systems class there (6.828: Operating Systems Engineering)[1]. The xv6 project contains about 70 code files.
- git – Git is a distributed revision control and source code management system. It was originally developed by Linus Torvalds (the original developer of the Linux kernel). It contains about 260 files of C code (it also has Perl and Shell code, but these code bases are not analyzed here)[2][3].
- MySQL – MySQL is a relational database management system. It is "the most popular open source database software"[4], it was originally developed in Sweden, and after a few acquisitions, it now belongs to Oracle Corporation[5]. It contains well over 2900 files of C code. In this project, I used the fork of MySQL maintained and utilized at Twitter[6].

The information that was extracted from the code bases was the file inclusions. To extract all the file inclusions in these software projects and format them into a standard network format; a shell script was designed, which by using regular expressions, extracted all the files that included or were included by any other file; as well as all the inclusion data. The shell script is available in the zipball: **include_network.sh**. All specifics of this script are commented in the file.

The chosen network file format was the .NET Pajek format; which although not very well defined[7], is quite flexible and easy to create, given the file inclusion information structure. Again, the information on how the Pajek network files are generated can be reviewed in the **include_network.sh** file. Any special issue that the script could not deal with, was dealt with manually, using Gephi.

To model this project, each file which includes or is included is represented by a single node. An inclusion of one file into another is represented as an edge of weight 1, going from the included file to the including file.

Analyzing the data

Interestingly enough, although different, these networks did converge in several characteristics that will be observed and described in this paper. Table 1 presents some of the metrics calculated and observed in the networks. In the following section, a few characteristics of the networks are analyzed: Power-law distribution and high degree nodes, community structure, relationship between indegree and outdegree in the nodes, and small-world effect.

---

1 Xv6, a simple Unix-like teaching operating system -http://pdos.csail.mit.edu/6.828/2011/xv6.html – Retrieved Nov/2012
2 Git Source Code Mirror - https://github.com/git/git – Retrieved Nov/2012
3 Git(software) – Wikipedia, The Free Encyclopedia - http://en.wikipedia.org/wiki/Git_(software) – Retrieved Nov/2012
4 About MySQL - http://www.mysql.com/about/ - Retrieved Nov/2012
5 MySQL – Wikipedia, The Free Encyclopedia - http://en.wikipedia.org/wiki/MySQL – Retrieved Nov/2012
6 MySQL fork mantained and used at Twitter - https://github.com/twitter/mysql – Retrieved Nov/2012
7 Pajek software website - http://vlado.fmf.uni-lj.si/pub/networks/pajek/ - Retrieved Nov/2012

Project	# of Nodes	# of Edges	Max number of times a file was included (indeg)	Max number of inclusions made by a file (outdeg)	Outdeg distrib. is power-law	Exponent for outdeg distrib	Xmin for outdeg distrib	Node with highest indegree	Node with highest outdegree	Average shortest path**	Number of communities
xv6	70	286	46	11	no			console	types	2.3159	5
git	258	948	157	52	no*	2.80*	11*	git-compat-u	cache	2.5568	10
MySQL	2981	10418	283	90	yes	2.54	15	sql/mysqld	ndb_global	4.7614	35

* The outdegree distribution is not power-law, but the total degree distribution is. Numbers shown are for the total degree distribution.
** The average shortest path was calculated for the equivalent undirected graph.

Table 1. Data obtained by analyzing the code bases.

Power-law distribution and high degree nodes

As it can be seen from Chart 3, the outdegree distribution does seem to be more power-law than the indegree distribution. Nonetheless, after running the power-law fit algorithm on these networks, it turned out that only MySQL presented a power-law outdegree distribution, whereas git had to aggregate both in- and outdegrees to manifest a power-law.

In any case, in all three networks, there are few nodes that manifest much higher outdegree than the average of the network; as well as nodes with quite high indegree. These nodes represent files that are either included by many other files, or that include many other files.

Files that are included by many other files usually define basic functionality that is commonly used across the whole project. As an example, in the xv6 network, the node with the highest outdegree represents the file **“types”** from the project. This file defines the basic types that are utilized across the xv6 operating system. It is natural that most files in the project need to include it.

On the other hand, nodes with a high indegree represent files that include many other files. These are files that implement high level functionality that requires many layers of the project to interact. Again, using xv6 as an example, the node with the highest indegree represents the **“console”** file. This file implements the command interpreter that receives user input, and that should execute any action that the user asks for.

Looking at the data on Table 1, it is clear that although the number of nodes rose sharply per each project, the average shortest path remained quite short. Chart 1 shows the the average shortest path plotted against the file count in a lin-log scale.

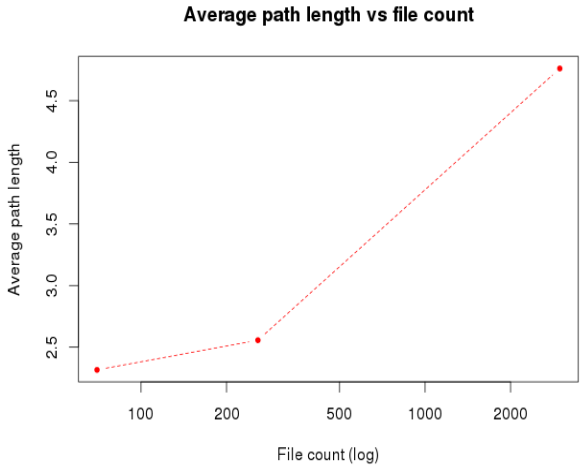

Chart 1. Average path length vs file count

From the plot in Chart 1 can be seen that the speed of growth of the average shortest path for all three projects was not exactly adjusted to a logarithmic curve (as it doesn't show a straight line in a lin-log scale), but the growth rate was quite slow anyway; and we can say that these code networks show small world effect. Having more data points would allow to do a better regression.

The following chart (Chart 2) shows the shortest path distributions for the three networks. They all show a rather normal distribution, somewhat skewed to the left.

Looking at these metrics, it can be understood that in general, networks formed by files in software projects tend to manifest the small-world effect. This means that although these projects may be conceptually divided in several layers, every file is not many hops away from another, and thus the project is closely tied together.

It could be that part of the reason that this occurs is that generally, software projects coded in languages like C tend to be more monolithic[8], and to have complex ties of dependencies; whereas projects coded in other paradigms, such as object-oriented languages tend to promote encapsulation and isolation of functionality. Future work could be focused on the differences in structure between software projects in different languages; and how the characteristics of a language can affect the architecture of a project.

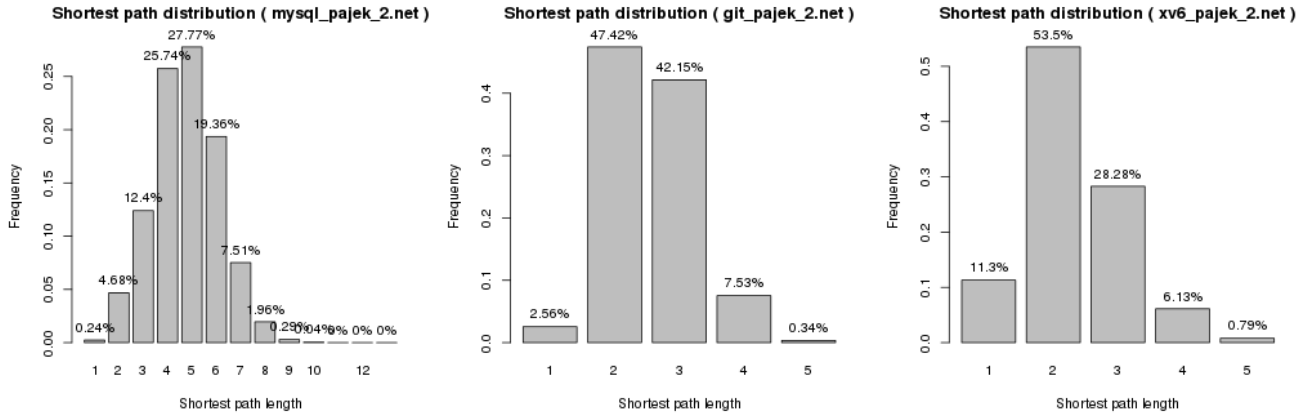

Chart 2. Shortest path distribution for the three networks

Service files and execution files

One of the hypotheses of my analysis is that in software projects there are two kinds of files: Service files, and execution files. The service files define functionality and services that can be utilized by other files; whereas the execution files usually guide the execution path from a high level, and use some functions and services provided by the service files.

Following this theory, I expected to find that files that were included a lot (service files) did not include many files; and the reversal: Files that included a lot of files (execution files) are not included by many other files.

The following plot contains both the outdegree and indegree vectors of each node, ordered first by ascending order on outdegree, and then by descending order on indegree. This makes files with high outdegree to be put on the right side of the plot; and files with low outdegree (which are expected to have a high indegree) will be plotted on the left.

8 Three-tier Applitation Model - http://msdn.microsoft.com/en-us/library/aa480455.aspx – Retrieved Nov/2012

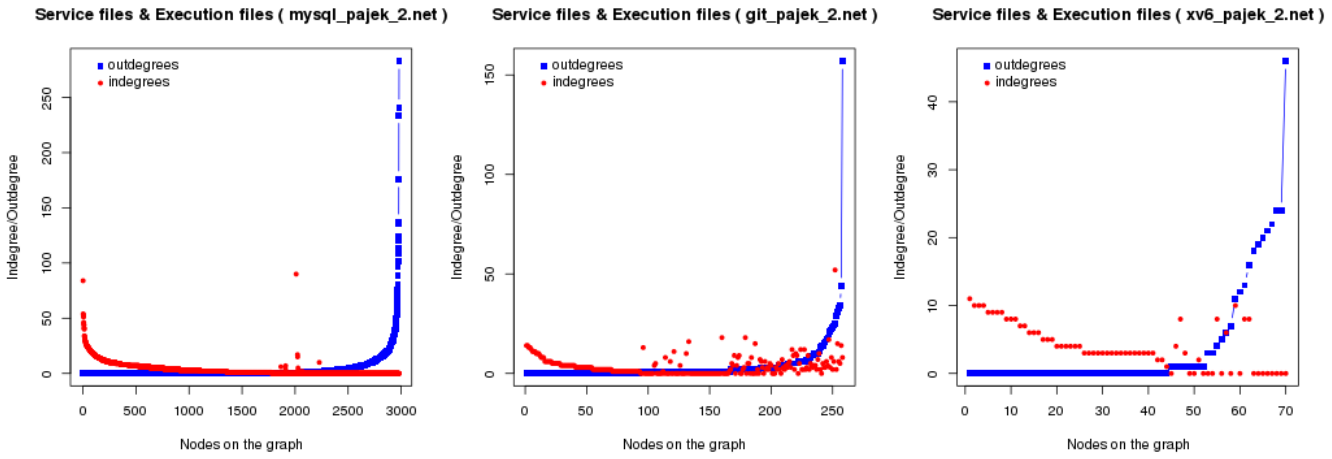

Chart 3. Plots of the indegrees and outdegrees of all nodes, ordered ascending by outdegree and descending by indegree.

Chart 3 shows that although there are some nodes that do not adjust to the hypothesis of execution and service files (particularly in git), both MySQL and xv6 show a strong separation between files that include many others and files that are included by many others. Then again, the availability of more data (graph information of other projects) will be helpful to determine if this is a general trend in software, or just a trend in the selected sample.

Community structure and functionality clusters

I applied the fastgreedy community finding algorithm on these networks. The expectation was that communities would emerge where nodes represented files that implemented the same, or related functionality.

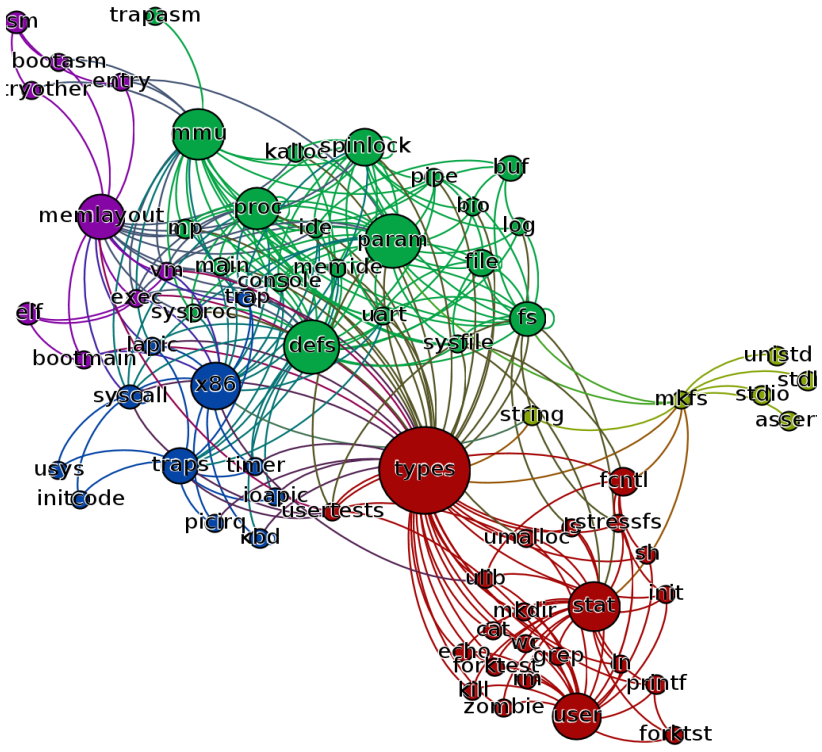

Figure 1. The graph representation of the xv6 project (generated in Gephi)

Figure 1 shows the graph for the xv6 project, with nodes colored according to their community, and sized according to their outdegree (how many times they were included). In this case, related functionality was indeed grouped in communities, with the following patterns:

- Red community – This community groups mostly user-side programs. These programs can be executed by the user, interacting with the console.
- Yellow community – This community includes, mostly, standard Unix files (stdio, assert, stdlib, etc.)
- Purple community – Here are files that take part in the boot up of the machine, as well as basic memory layout and process structure files.
- Green community – Green files are mostly files implementing inter-process communication, file interaction and memory allocation. These are process-level APIs.
- Blue community – These are hardware management files. Interrupt traps, x86 architecture code, timers and other hardware-close functionality.

Although the community division was not perfect, it was quite precise; and although it is difficult to look at the files in MySQL or git, and tell exactly what functionality they implement, Figure 2 shows that communities of files seem to be emerging correctly according to functionality.

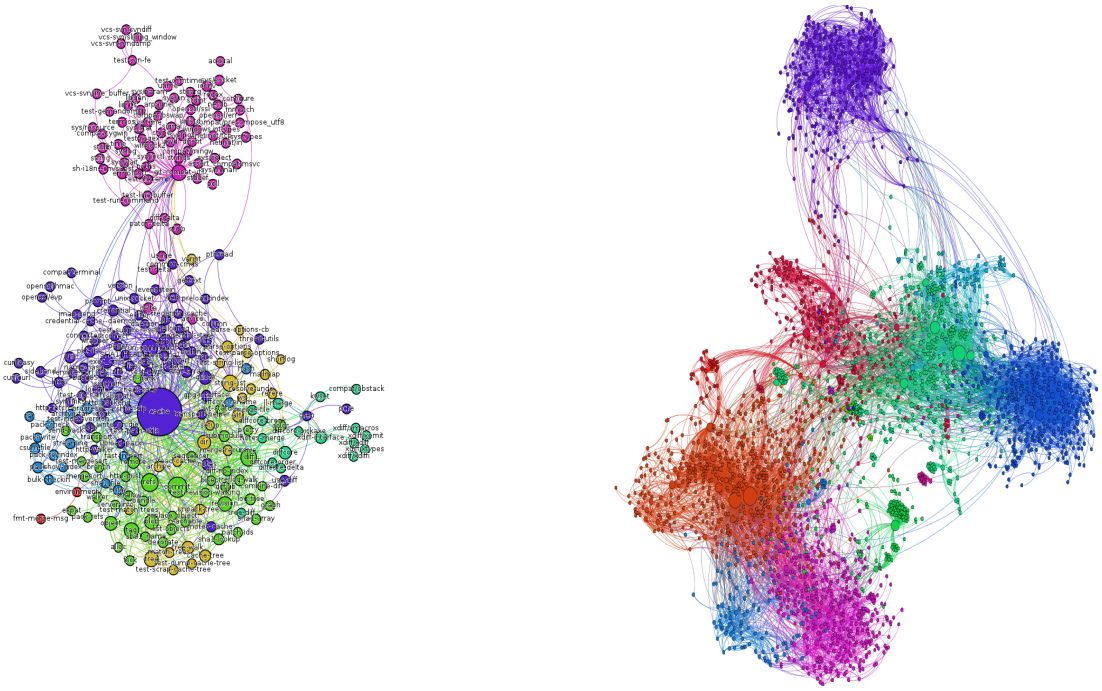

Figure 2. Left: graph representation of the git project; Right: graph representation of the MySQL project. (both generated in Gephi)

A remarkable example is the pink community that seems to be isolated in git (upper side of the visualization). This community implements user-visible functionality that is apart from the core of git, which is organized surrounding the larger **cache** node in the center.

Concluding remarks

The inclusion network of a software project is still quite coarse-grained. Regarding software projects; it is still possible to generate more fine-grained network information; regarding the interaction between functions and global variables, as well as other symbols in the program. Future work can focus on the interaction between smaller structure units in software projects.

Nonetheless, the file structure of a project does manifest interesting and insightful information about it. These networks show preferential attachment; which is naturally explained by the widespread need of basic

functionality in all software projects.

Also, the rather clear separation between service files and execution files suggests better strategies to focus testing paradigms: While unit testing can be helpful if done in service files, execution files might prove harder to test, and need system wide testing. This can help software makers decide how to focus their Quality Assurance engineering capacity, and adapt particular code coverage and testing requirements and strategies to the kind of file/project they are working with.

An interesting analysis yet to be done is the motiff characterization of the networks to find general local structure trends that show themselves in these networks. Also, the analysis of the growth of software networks promises some interesting insight.

One of the potential applications of this network approach into software engineering itself is the development of testing and compilation tools that use the network structure of the project to minimize the number of tests that have to be run or the number of files that need to be recompiled. Google has published some work regarding this[9].

Future work will try to formalize the knowledge derived here by augmenting the number of projects that are studied, and go on to study the local structure and motiff characterization of software projects.

9 Testing at the speed and scale of Google - http://google-engtools.blogspot.mx/2011/06/testing-at-speed-and-scale-of-google.html – Retrieved Nov/2012