Understanding Software Development Through Networks

Christopher Roach
Apple, Inc.
19333 Vallco Parkway, Building A
Cupertino, CA 95014
croach@apple.com

ABSTRACT

Software engineering, being a relatively new field, has struggled to find ways of gauging the success/failure of development projects. The ability to determine which developers are most crucial to the success of a project, which areas in the project contain the most risk, etc. has remained elusive thus far. Metrics such as SLOC (Source Lines of Code) continues to be used to determine the efficacy of individual developers on a project despite many well-documented deficiencies. In this work, I propose a new way to look at software development using network science. I use networks to to explain and understand the dynamics of the software development process. In this paper I examine several open-source software projects and detail one large open-source software development project—the Python programming language. I begin the analysis with a description of the basic characteristics of the networks used in this project. I follow with the main contribution of this work which is to examine the importance of the developer within their organization based on their centrality measures in networks such as degree, betweenness, and closeness.

1. INTRODUCTION

The discipline of software engineering has been around for over 40 years; the first formal mention of the term can be traced back to a 1968 NATO publication [1] whose main purpose was to provoke thoughtful discussion on the looming "software crisis" of the time. Ever since the establishment of the software engineering discipline, the need for proper evaluation tools has been pursued. In market-driven economies where the performance of employees is directly linked to the success of companies, managers are required to maintain employee satisfaction while keeping up to market demands. In the case of software development, managers should have the ability to identify valuable developers so that they can be rewarded as well as the ones performing poorly so that they can be trained, or in the worst case, replaced.

Over forty years have passed since static, individual-based metrics such as SLOC count were first used and yet the use of these metrics to evaluate individual developers' worth within an organization still pervades the industry. However, software development projects are no longer small with few developers; most are complex, involving sometimes hundreds of developers and millions of lines of code. Despite the fact that it is now common knowledge that traditional metrics are poor indicators of a developer's value in today's world, they can still be found in every corner of the industry [3]. In this study, I argue that individual-based, static statistics such as SLOC count, and other similar size count metrics, are poor indicators of a developer's overall worth within their organization. The major source of problems with these traditional metrics lies within their focus on the individual to the point of excluding the surrounding environment in which the individual does his work. I argue that a network-centric approach to measuring a developer's importance provides a much more realistic means to evaluate individual developers since it takes into account the effect the organization has on the individual developer's productivity and the effect the individual has on the organization in return.

The remainder of this report is organized as follows: In Section 2 I quickly discuss the traditional methods used in software development to evaluate developers; I also review the literature on both the use of network science to study software development and on some of the alternatives that have been prescribed to remedy the known issues with traditional metrics. Section 3 will argue for a network-centric approach because it captures: the importance of a developer to the cohesion of the team, the role of the developer in the maintenance of a stable development group, and the knowledge held by a developer and the effect of his departure to the group as a whole. I follow in this section with our argument for using network-centric approaches. I finish the section with a description of the network measurements used in this paper. Section 4 describes the details of this study and justifies the use of the Python open-source project as well as the results of using network-centric approaches to evaluate developers. Finally, in Section 5, I discuss areas for future work and give some concluding remarks.

2. RELATED WORKS

2.1 Evaluating Software Developers

One of the common problems software development managers face in their daily activities relates to the process of evaluating developers (programmers, testers, etc.). While it would be relatively easy to quantify the amount of work done by a developer, it is quite complicated to qualify the same work. Anyone familiar with software development understands that some solutions to problems may be harder to code than others. Hence the amount of work produced may not reflect the effort a developer has put into the process.

There have been many approaches used to evaluate software developers:

Source Lines of Code: SLOC is likely to have been the first metric to evaluate developers. In a process of software development the main product is a program (a piece of software). Thus it is natural to think that developers could be evaluated by the number of lines of code they contribute to the process. Clearly this is not a good approach as the number of lines does not imply quality, and worse, the complexity of the lines of code produced is disregarded in simple counts.

Number of Commits: It is common for companies to keep track of the number of modifications a developer makes to the project. The idea is that more prolific developers tend to make more modifications (commits). However this is highly questionable, since not all commits have the same importance.

Number of Issues Found: This measure is very similar to the number of commits, but closely related to issues of bugs found in the development. Similarly to the cases above, this is not a good indication because different bugs have different importance. In fact, the location (file) where the bug is found is also important.

Hours worked (extra hours): Many companies evaluate developers by their level of dedication to the project. The number of hours worked on a particular project is often used as an indicator of dedication. Number of hours is a good indicator but in isolation provide little information about the effectiveness of the developer

Innovations Introduced by the Developer: Companies today encourage developers to introduce new ideas and even reward them for introducing ideas and projects. To our knowledge this has not been formally been used as an evaluation mechanism because very few job descriptions formally require it.

Because of problems present in the approaches above, managers generally resort to less precise evaluation mechanisms such as meetings to discuss the worth of each developer from their point of view—a common approach during the evaluation cycle is to have managers discuss each person individually with some or all of the criteria above being considered. While these approaches are useful and bring to the table the important perceptions of managers, the whole process is generally unfocused due to the lack of a precise reliable metric to drive the discussion.

2.2 Development Trackers

The literature also has many examples of software tools aimed at helping track the process of software development as well as the developers working on the project. Some of the common examples of these softwares include SeeSoft, CVSScan and Chronia.

SeeSoft [4] is essentially a versioning tool for visualizing software development at the source-code line level. Modifications in the code (lines) can be associated with any statistic of interest such as: most-recently changed, least-recently changed, number of changes, etc. Although not proposed for evaluation of developers it could be easily adapted to be used to track the number of lines changed by developers and hence automate metrics such as SLOC.

CVSScan [5] is an approach similar to SeeSoft except that CVSScan is based on a single source code visualization. A program evaluator could easily see the changes performed by a programmer. The focus on one source file provide a finer-grain analysis but it hinders the evaluator's ability to see programmer's skill across many projects or source codes—his worth at a macro level.

Finally, Chronia [6] is a program to track ownership of files based on the amount of modifications made by the developers. The idea is that the developer will become the owner of the file when he contributes more to that file than others. Chronia then creates a ownership map that can be analyzed by managers/evaluators. Since this covers many files, it is a reasonable indicator of the importance/performance of a developer but it lacks the ability to see this from a macro-level. In addition, developers who work on a code fixing important parts of it may never get ownership of the file leading the evaluators to have an unfavorable view of them.

2.3 Network analysis of software development

From the descriptions in the previous section one factor should be observed: the current approaches rely on numbers that depend solely on the individual. As new advances in both software development practices and hardware have been made, the size and complexity associated with the development of software has continued to grow. One could argue that any software of interest must be developed by a team of engineers, and it is not uncommon to see team sizes in the hundreds. As software development becomes increasingly more team-oriented so too must the methods we use to measure the individual developers. Traditional metrics are quite simple and typically observe the individual while ignoring the team as a whole. Network measures, on the other hand, take into account the individual's position within the network of developers, and, as a result, can paint a much more accurate picture of each individual's worth within that organization.

There have been studies in the past that have used network measures to examine the software development process. The first, conducted by researchers at Notre Dame University [7], examined the entire Free/Libre/Open Source Software (FLOSS) community as a single network of developers connected by their active participation in common projects. The main goal of this study was to determine if the network of FLOSS developers showed the same characteristics of other communication networks that had been found in previous studies on real networks [8, 9]. They did, in fact, find that the FLOSS community does follow a power-law distribution and that the community seems to grow, not as a random network, but rather as a preferentially attached network.

The second study, performed at Syracuse University [10], looked at the internal structure of several projects in the FLOSS community to determine if they followed the widely held belief that FLOSS projects tend to be chaotic and decentralized. The common meme being that FLOSS software is constructed in a "bazaar" style—with little to no real planning and lots of people pitching in where needed—whereas proprietary software is built more like a "cathedral"—ceremonious and carefully thought out with rigid designs and processes. In this study, a network was constructed for each software project by linking individual participants to one another if they had participated in any interaction on a common issue through the project's issue tracking software. The study found that the centrality measures across the 120 different FLOSS projects tended to follow a Gaussian distribution. This suggests that the conventional idea that FLOSS projects are highly decentralized and chaotic—like that of the atmosphere of a "bazaar"—is not entirely correct. In fact, the FLOSS projects showed a broad scope of social structures ranging from highly centralized to thoroughly decoupled. It was also found that a negative correlation exists between the size of the project and the centrality of that project. Therefore, as the projects grow in size, they also tend to grow in entropy as well, becoming more chaotic and decentralized as membership in them increases. This study seemed to prove that neither depiction of FLOSS software—"cathedral" or "bazaar"—was an accurate representation of all FLOSS projects and that, instead, open-source software projects are wildly varied in how each is developed.

3. GENERAL CONCEPTS

3.1 Shortcomings in Traditional Metrics

I have described in earlier sections many metrics that are used in industry to evaluate software engineers. Despite the many attempts, the problem of correctly identifying talent (and consequently identifying under-performers) still challenges managers. Such identification process allow companies to reward developers according to their contribution to the company. Unfortunately, the state-of-the-art of metrics in industry rely on measures that are isolated from the context in which they are applied. For instance, the common SLOC measurement does not take into consideration factors like: difficulty of the code being worked on, quality of the code generated (i.e. n lines of code are not always equal to n lines of code), importance of that source code to the entire project. Due to this lack of metric, managers generally resort to reviews, meetings, interviews to understand the importance of a developer in the entire context of the company.

It should also be highlighted that individual metrics are very susceptible to distortion since each individual can easily inflate his numbers: if one is counting lines of code, the developer could easily produce more lines of code with the same amount of effort. The area of social networks has seen a similar problem. After the advent of sites such as Facebook, MySpace and Twitter, users quickly tried to add more "friends" to their list in an attempt to become more important in those social networks. Adding friends is something over which each user has direct control. However, we now know that the number of connections in a social network is not quite as important as the type of connection (who are your friends). Works inspired by sociology [11, 12] have described that the importance of an individual relates to his centrality in the network, so being a hub (large number of connections) does not always imply being important. On the context of websites, Google has become a power-house search engine when they proposed an algorithm that looks beyond the number of connections a website has with their PageRank algorithm [13].

In this paper I argue that Network Science [14] can be used to model interactions in a software development company; one can create a social network of developers and use this network to measure the importance of each to the company. I demonstrate in this paper that the importance of an individual can, and should, be given by his rank in the network based on approaches such as centrality as well as approaches inspired from his importance to the maintenance of the network stability (as in Random-Targeted attacks in Complex Networks [15, 16]).

3.2 Network Measures for Node Rank

Networks are characterized by many measurements which can generally be classified as micro and macro. Macro measurements are characteristics that pertain to the entire network, such as degree distributions, number of communities, and average path length. Contrasting with these we have micro measures which refer to the characteristics of individual nodes and edges in the network and are used to rank these nodes and edges inside the network.

The most basic example of a micro property is the degree of a node, given by $\deg(v)$ and used as a property of nodes since the beginnings of graph theory with Euler. The $\deg(v)$ in a network represents the number of connections that node has to other nodes where these connections can be directed or undirected. In directed nodes, $\deg(v) = \text{indeg}(v) + \text{outdeg}(v)$, meaning that a node, v, may have edges incoming to it and outgoing from it. Nodes with a high degree are generally thought of as susceptible to being affected by information transmitted on the network. In order to normalize the degree values one refers to the degree centrality of a node, which is given by

$$C_d(v) = \frac{\deg(v)}{|V| - 1}. \tag{1}$$

In network analysis it is common to try to find a node that is shallow on the network, meaning that its distance to all other nodes is small. The measure that captures this geodesic distance to all other nodes is defined as the closeness centrality of a node, given by $C_c(v)$. The most accepted definition of closeness centrality is that

$$C_c(v) = \frac{1}{|V| - 1} \sum_{s \in V \setminus v} d(v, s), \tag{2}$$

where $d(v, s)$ is the minimum distance between v and s. Note however that it is also common to use an alternative definition for $C_c(v)$ defined as the maximum of all minimum-distances between v and the other nodes.

Betweenness is another node property related to centrality. Informally, it represents how much a node seems to be important in the connection of other nodes. Hence, a node with high betweenness appears in many of the shortest paths between pairs of nodes. The definition of betweenness is given by

$$C_b(v) = \frac{1}{\sigma_{st}} \sum_{s,t \in V \setminus v} \sigma_{st}(v), \tag{3}$$

where $\sigma_{st}(v)$ is a count of all the shortest paths that go through node v.

I argue in this paper that centrality measures are better suited to evaluate developers once the network is created than current approaches based solely on the individual. These measures provide managers with a sense of how the network of developers depends on certain individuals. Even more interesting I show that we can use a network of files (see next Section) to evaluate the importance of a developer. In order to confirm my hypothesis, I perform many experiments simulating developers being removed from the project (deleted from the network) according to their centrality measures done here as well as the individual approaches described earlier. My results demonstrate that the removal of developers based on individual metrics such as SLOC count always perform either equivalently to network measures at their best, and equivalent to removing developers at random at their worst.

4. CASE STUDY: THE PYTHON LANGUAGE

The Python programming language was chosen as the main focus of this study for several reasons. The first of which was the author's familiarity with the language and its surrounding community. This familiarity brought with it two main advantages: a slight understanding of the code base, and a knowledge of who the "linchpin" developers are in the community. The latter advantage mentioned acts, to some level, as a qualitative measure of the success of the methods used to determine developer importance.

Next, I chose Python for its size and structure. According to the study done by Crowston and Howison [10], Python, relative to the other 119 FLOSS communities, has a large and decentralized developer community. I chose Python specifically because I wanted a community that would rival, in size, many of the development teams that one would see at some of the larger successful software companies such as Google, Microsoft, and Apple. In addition to the size, the decentralized nature of the project seemed to fit the architecture that one would see in many companies where the software is developed by large numbers of small groups. It is due to these features that I felt that the Python project would stand in as a rough analog for a typical proprietary project in a large company.

I constructed three separate networks in order to study the dynamics of the Python software development community. In the next few sections I will look at each network in turn and discuss the benefits that each provides.

4.1 Source-Code Dependency Network

The Source-Code Dependency Network is essentially a graph-based view of the software's architecture. In other words, I have assembled a directed graph based on the file dependencies (i.e., `#includes`) inside of each source code file. There were a couple of interesting properties to take note of in this network. First, the scaling factor of the power law given by $\alpha \approx 3.5 \pm 0.1$, when the network is analyzed as undirected. This suggests that the degree distribution for this network does indeed follow a power-law distribution. This is a fact that can be easily seen in Figure 1. The major hubs in the graph are highlighted in red (and are larger in size) and labeled with the file's name. As you can see there are 5 main hubs (a few smaller ones, but only those with a degree of 10 or higher are labeled here).

Next, the clustering coefficient is zero—there is no clustering at all.This is due to the fact that software dependencies should form an acyclic graph. As such, a clustering coefficient above 0 would suggest that there are circular references in the software, a property that is usually ardently avoided in most software development practices. So, a zero clustering coefficient can be one sign of a well-architected system.

I extracted the bugs that were reported as fixed in the SCM (Source Configuration Management) logs and overlaid that information on our network. I did the same for the developer data. This information can be seen in Figure 2. The two networks show the bug data, Figure 2(a), and the developer data, Figure 2(b). The data shown is the number of bugs and developers per file as a heat-map where blue (cold) nodes represent files with relatively few bugs and developers and red (hot) nodes representing files with many bugs and developers; to make this even clear, the size of the nodes are also proportional to the number bugs and developers associated with each file. As can be seen from the networks, the number of bugs per file tends to correspond to the number of developers per file. In other words, the more developers that worked on a file, the better chance that more bugs would be reported on that file as well. The data supports what we can intuitively tell from the network, as the Pearson correlation coefficient for the data was $\approx 0.70 \pm 0.05$ representing a high correlation between the two. It is also worth mentioning that the distribution of the number of bugs found per file and the number of developers per file follows a power law. My findings suggest that both distributions (bugs/file and developers/file) have scaling given by 1.95 ± 0.02 and 2.0 ± 0.05 respectively.

Recall however that the main claim in this paper relates to using the network to evaluate developers. The results in the previous section, given in Figure 1 by hub nodes. To study the importance of a developer I have created another network from the SCM logs. I generated a bipartite network with files (source codes) and developers as nodes, where the edges represent the fact that a developer has worked on a file. From this bipartite graph, I created two single-mode projections, both of which are discussed in the following sections.

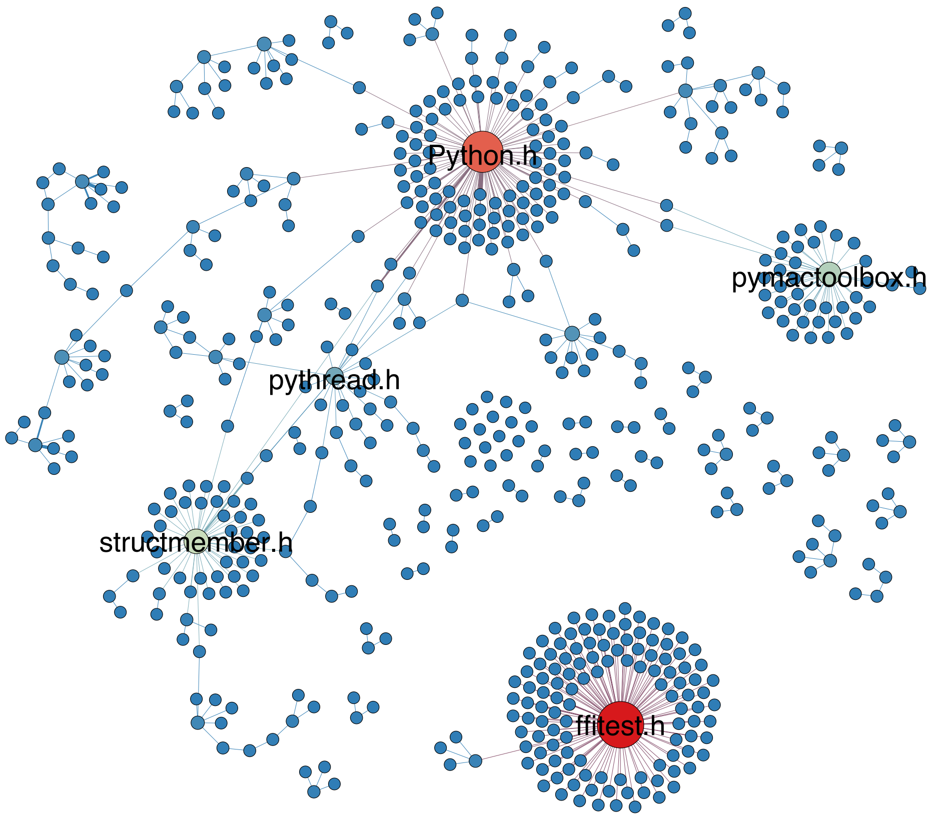
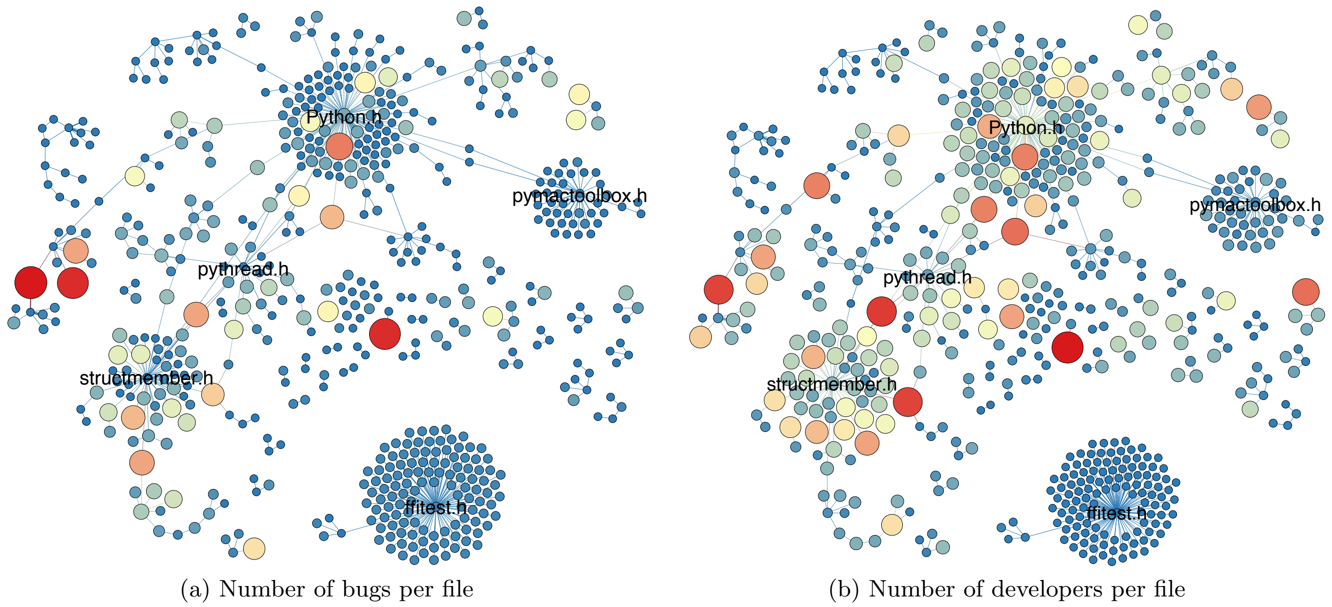


Figure 1: Source Code Dependency Network



(a) Number of bugs per file

(b) Number of developers per file

Figure 2: Bug and Developer Data. A heat-map for the color of the nodes as well as their size represent the numbers of bugs/file and developers/file.

4.2 Developer Network

The developer network is an undirected projection of the bipartite network in which nodes represent developers in the community and edges represent collaborations; an edge was drawn between two nodes if both developers had worked on at least one common file. A weight was then assigned to each of these edges based on the number of common files on which the two developers had worked, thus, strong and weak collaborative relationships could be easily distinguished.

The developer network is in essence an indirect social network—indirect because developers are not required to know each other in the real world to essentially be collaborators in the software development project. However the network topological characteristics are not true for social networks since they are just projections of the real (bipartite) network. For instance, the developer network does not have a strong power-law degree distribution. This might also happen due to the unique fact that Python is an open-source project where contributors (165 in our case) are inclined to look at most files and hence be connected to most other developers in this projection. Some of these links however are sporadic. If we consider the distribution of weights as part of a node's properties (as argued by Opsahl *et al.* [17]) the network behaves as many real networks do and a power law is, in this case, observed.

In order to evaluate the importance of a developer I have then looked at the size of the largest connected component in the network (aka Giant Component). The premise here is that if a user is important she should be missed in the network. This "missing" factor occurs if the removal of the developer from the network causes the network to become more disconnected (size of the Giant Component decreases significantly).

Figure 3 is used to demonstrate that metrics based on the individual are not very realistic in identifying relevant developers. To demonstrate this fact I have removed developers from the network randomly and by the number of commits the developer had to the project (a commit occurs when a developer has produced a new version of the source code; a good approximation for SLOC). Figure 3 shows that the random removal and commit-count removal (highest commit-count first) causes a similar effect to the giant component—it decreases in size somewhat linearly. This result demonstrates that commit-count is as good (or as bad) as a metric to identify talent in a software development process as a random identification. Figure 3 also shows our proposed approach in which talented developers are removed according to centrality measures in the network. Note that they all provide a significantly better measure of talent since the network degenerates faster after about 30% of the nodes are removed. This means that once the developer network is formed one can decide on talent based on the centrality, closeness or degree of the developer.

4.3 Source-Code Network

In this network, nodes represent source code files and weighted edges represent the number of developers that have worked on both files. In this experiment, I measure the robustness of the source-code network after the removal of developers. When a developer is removed from a network (say because she left the company) her contribution to all the edges of the source-code network has to be removed. Hence, I assume here that a developer is important when she is required to maintain the connectivity of the source-code network. If a developer disconnects the source-code network when removed, that developer is important for the flow of knowledge between different parts of the development process and should then be highly valued.

I repeated the removal experiment of Figure 3 but this time measuring the size of the giant component in the source-code network. The effect here is very interesting since I immediately see the importance of a few individuals. When developers are removed randomly or using commit counts the network maintains itself well connected until about 60% of the nodes are removed. Even more interestingly is to observe how poorly commit-count removal performs—my experiments with the Python community demonstrates that commit-counts perform worse the random removal. In practice this experiment says that the random identification of a developer to be rewarded is better than the identification based on how many modifications he performed to the source code in the project. On the other hand, the centrality measures are very effective and perform equally well. The size of the giant component decreased to about half of its size when only 20% of the network was removed.

4.4 Further Tests

In the previous sections I describe the successes I had experimenting with the Python community. The reasons that Python was chosen as my initial case study were detailed in section 4 and it is for those reasons that I include it here as an example of the process I followed in this study. Nevertheless, I did also perform the tests described above on a handful of other communities. Specifically, I ran the same tests on the Linux, Ruby, Ruby on Rails, Django, and Networkx communities. These spanned the spectrum of SCM technology, using not just classical centralized systems such as Subversion, but also newer decentralized systems such as Git and Mercurial and so the communities behind each differed in their behavior with respect to centralization. My tests in each of these cases were not nearly as successful as my initial case. In each of these cases commit count performed roughly equivalently, if not just slightly worse than, network centrality measures in determining the worth of individual developers within a community. The key takeaway here though is that, even though the test results were not quite as striking as those in the first example, they still showed that network measures always performed at least as well as, if not better than, traditional measures making the network a more consistent and reliable tool than traditional metrics for evaluating software developers within an organization.

5. CONCLUSION AND FUTURE WORK

In this paper I propose that current methods for evaluating software developers within an organization are poor indicators of the overall importance of the developer. Metrics that have been in use since the creation of the software engineering discipline are still largely in use today despite their well known inadequacies. As a solution, I propose the use of network measures of centrality as a better indicator of how vital each developer is to the success of their organization.

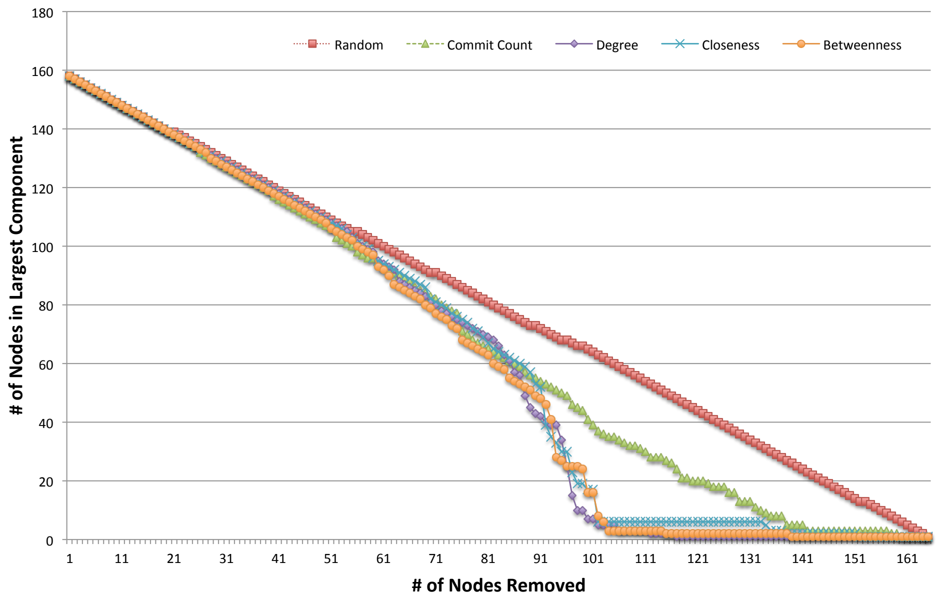


Figure 3: Behavior of the giant component in the developer network as nodes are removed from the network.

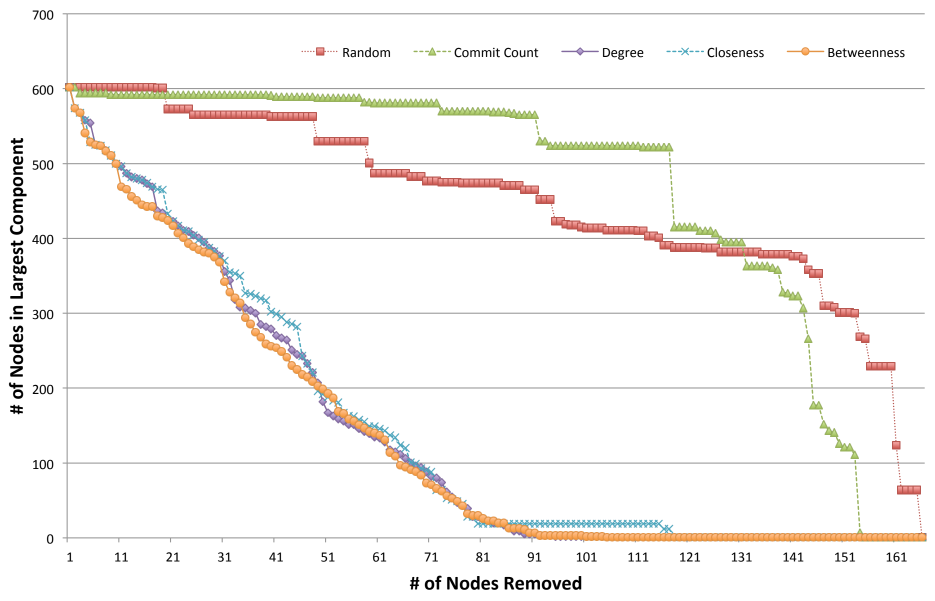


Figure 4: Behavior of the giant component in the Source Code Network as developers are removed from the bipartite network (edges in the Source Code Network).

In this study, I have chosen a large, well-known, open source project with a very active developer community with which to test my hypothesis. The previous section discusses the results of this study in detail and also discusses a handful of secondary tests as well, but in short, I found that network measures were indeed a much more accurate measure of the importance of an individual developer within a development community. In all cases I found that network measures always performed at a level equal to, or better than, traditional metrics.

There are two main areas on which I'd like to concentrate my efforts for future work. The first is simply solidifying the results I have attained thus far by experimenting with even more projects in the Open Source community to make sure that the results found thus far are consistent across a much larger number of different projects and not specific to only the Python community. Testing these measurements on a proprietary project, I feel, would also help to cement the results found in this study and is an option that I am actively pursuing at this time. The second area I'd like to focus on is the dynamic nature of network measurements and how important it may be in determining developer importance. To illustrate this point consider a community of several teams of developers; each team working on a specific area of the software with some overlap between teams. As members of the community come and go for whatever reason, the importance of individuals within the organization changes. For example, if several developers on a single team leave at once, the importance of the remaining members of that team should grow since the loss of these individuals would open up a structural hole in the community. Since network centrality measures take into account the structure of the network, they tend to change as the structure of the network changes, whereas measures such as code and commit count remain constant no matter how much the surrounding environment changes. In this study I focused on measuring individuals importance with respect to the network as it currently exists. For future work Id'd like to examine how changes in the community structure effect the importance of individuals within it and how well this method captures those changes as opposed to the static methods used in current practice.

6. REFERENCES

[1] Peter Naur and Brian Randell. Software engineering: Report of a conference sponsored by the nato science committee. Technical report, North Atlantic Treaty Organization, 1968.

[2] Norman E. Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

[3] Steven D. Sheetz, David Henderson, and Linda Wallace. Understanding developer and manager perceptions of function points and source lines of code. *Journal of Systems and Software*, 82(9):1540–1549, 2009.

[4] S.C Eick, J.L Steffen, and E.E Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[5] Lucian Voinea, Alex Telea, and Jarke Wijk. CVSscan: visualization of code evolution. *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, May 2005.

[6] Tudor Gîrba, A Kuhn, M Seeberger, and S Ducasse. How developers drive software evolution. *Eighth International Workshop on Principles of Software Evolution*, pages 113–122, 2005.

[7] Gregory Madey, Vincent Freeh, and James Howison. The open source software development phenomenon: An analysis based on social network theory. In *AMCIS 2002 Proceedings*, 2002.

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

[9] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[10] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[11] Stanley Wasserman and Katherine Faust, editors. *Models and Methods in Social Network Analysis*, volume 8 of *Structural Analysis in the Social Sciences.* Cambridge Press, 1994.

[12] Peter J. Carrington, John Scott, and Stanley Wasserman, editors. *Models and Methods in Social Network Analysis*, volume 27 of *Structural Analysis in the Social Sciences.* Cambridge Press, 2005.

[13] Pavel Berkhim. A survey on PageRank computing. *Internet Mathematics*, 2(1):73–120, 2005.

[14] Ted G. Lewis, editor. *Network Science: Theory and Applications.* Wiley, 2009.

[15] Romualdo Pastor-Satorras and Alessandro Vespignani. Immunization of complex networks. *Phys. Rev. E*, 65(3):036104, February 2002.

[16] Albert-Laszlo Barabasi and Eric Bonabeau. Scale-free networks. *Scientific American*, pages 50–59, 2003.

[17] Tore Opsahl, Filip Agneessens, and John Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):245–251, 2010.