Structure and Evolution of Codebase Network over a period of time
Kunal Sood – Project Report
ksood@stanford.edu

Abstract – In this paper, I have focused on the evolution of codebases in PHP and C# .Net over a period of time as a network and in turn gained information about the networks. I have tried to fit the hierarchal model to the networks and I am also evaluating network properties such as Strongly Connected Component Size, Clustering Coefficient, amongst others, to gain insight into the code networks. I have applied the PageRank algorithm to ascertain which functions in the code base are most used functions. Additionally, I have analyzed centrality measures such as degree, closeness, betweenness to identify important nodes (functions) and then contextualize the results through the underlying knowledge of the functions. I have also conducted Robustness Analysis and compared the PHP and C# code networks. Finally, I have extrapolated the results in the context of code quality and performance of the code base and found meaningful use in our day to day operations.

1. INTRODUCTION

With the growth of software as an industry and corresponding rise of development firms, and in turn; developers, quality control has now become more important than ever. Thus, my project focuses on the evolution of the code network in PHP and C# .Net codebase over a period of time and in turn gain insight into the code quality and performance. In our organization, those are the key components we measure about enterprise level code bases. I will attempt to fit the hierarchal model to the network and present the data. I intend to use a code network structure in conjunction with network properties such as clustering coefficient, edges and nodes over a period to deduce what constitutes ‘good’ code based on my criteria listed above.

2. RELATED WORK

While my project idea seems to be something new and specific to coding languages such as PHP and C#, I was able to find a few examples of similar work.

Kumar’s paper [4] studies two online social networks to learn about the structure and evolution of these networks over time. The author has presented a possible structure consisting of three main parts – singletons, giant component and middle region. We look at the properties of each of the structures on the real network and studies the formation and merge of these components. The key finding in this paper is that the Giant Component in the online communities is found to be much smaller than expected.

In [5], the authors talk about using networks to describe or quantify large amounts of data. The authors also report on topological properties such as degree distribution, clustering coefficients and short path lengths. They mainly emphasize the importance of hierarchical organization in order to understand various complexities of a network.

In [6], the authors dive into the structural inferences of hierarchies in networks and describe a statistically approached way to gather the set of hierarchical features that most plausibly explain a particular real-world network. The authors propose Markov chain Monte Carlo (MCMC) algorithm for sampling the space of dendrogram hierarchies in complex networks, focusing the emergence of the scale-free and the hierarchical architecture. The key take away is that most networks have a modular topology, quantified by the high clustering coefficient they display. While the small nodes are part of highly cohesive, densely interlinked clusters, the hubs are not, as their neighbors have a small chance of linking to each other. Therefore, the hubs play the important role of bridging the many small communities of clusters into a single, integrated network.

3. DATASET

For the dataset, I have used the code base available to me from internal company projects. We have enterprise level code bases that I have used to generate a network of approximately 14,500 functions in the PHP code base and 39,500 in the C# codebase. Thus, I was able to do analysis on a large number of nodes. The snapshots of the codebases have been taken monthly starting from December 2013 to August 2014.

Dataset Pre- processing - Custom Parsers

In order to use the PHP and C# data, I have written custom parsers for both languages that help parse the code to extract functions. The syntax of PHP and C# code varies, thus, I have written custom parsers for both the languages. In these, I have removed the common language functions such as _constructors and exit() statements since they do not provide much valuable information about the codebase and are common language constructs used in development.

Removing these functions enables us to dive deeper into the business level functions of the codebase and in turn gain greater insight about the business level code functions and about their performance and code quality.

After parsing, I was able to establish the following categories for the functions in both codebases.

Constructs: this includes constructors, destructors and functions such as init; which are going to be called many times in the codebase but do not give us much information about the code quality. Thus, I have omitted the nodes of this category for the purpose of my analysis and reporting.

CRUD: as the name states, these are basic operation functions and hence have been omitted for the purpose of
my analysis and reporting. CRUD stands for Create, Read, Update, Delete.

**Business Layer** – contains the business logic functions and are a key area for code performance since it contains a lot of heavy business logic.

**Code Quality** – includes unit testing functions, as well Quality Assurance testing functions/scripts.

**Code Security** – contains functions that affect the security of the code and is needed to prevent a malicious attack. If we find many security related functions in our centrality and/or PageRank measure top 100 results, then we can infer a high level security in the codebase.

**Codebase – PHP**

This is a paperless transaction management software designed to automate and grow real estate businesses. It enables realtors to manage listings, buyer’s leads, showings and short sales using this web-based system. This codebase is about 10 years old and was written in PHP 4.2 to start off with and is currently running on PHP 5.5, it connects to MySQL 5.1.73 on the backend. The code follows standard MVC architecture and categorization is laid out as follows:

**Business Layer**: The functions are found in the /BusinessLayer and its sub folders. These functions give us key indications about performance and there is heavy interaction with the database (CRUD category as listed above) done thru this layer with the code in the /DataLayer and its sub folders.

**Code Quality**: the functions relevant for code quality can be found in the /BusinessLayer/Test folder. This folder contains the unit tests that have been generated for the entire project till date.

**Code Security**: the functions relating to code security were found in /Commons and /Globals folders. Most generic functions such as build_http_query, httpRequest etc. were found in these folders.

**Codebase – C# .Net**

This is an electronic signature / digital signature software enables users to fill out and sign documents online, from home or office, using any web browser. The same C# codebase serves as a service layer backend for the iPad and Android versions of the application as well. The codebase has been in existence for more than 11 years and is not very well written or structured. It is currently using MVC 3.0 in C# and Kendo UI controls for the presentation layer.

**Business Layer**: The functions that pertain to the business logic of the application are found in the /BLL and its sub-folders of the application

**Code Quality**: the functions relevant for code quality can be found in the /Infrastructure/Quality/ folder and its sub folders. This folder contains the unit tests that have been generated for the entire project till date.

**Code Security**: the functions relating to code security were found in /Helpers and its sub folders. Most generic functions such as GetQueryParamByName, etc. were found in these folders.

Overall, the C# codebase is not well structured and even with the MVC architecture, we see a lot of unstructured code files and classes created unnecessarily. In the analysis below, we will actually see that the PHP code network seems more robust that the C# code network.

### 4. Hierarchical Model

As we see in [14], we can establish a general mechanism for identifying a hierarchical structure from network data. To depict the existence of hierarchy, we can concurrently explain and quantitatively reproduce many commonly seen topological properties of networks, such a right-skewed degree distributions, high clustering coefficients and short path lengths. Our goal is to detect and analyze the hierarchical structure, if one exists, for our PHP and C# code networks.

**Definition**

We now turn to attempting to fit our code network to the Hierarchical Model. In principle, a network consists of nodes and links interconnecting the nodes. A hierarchical network consists of disjoint sets of nodes denoted by clusters. In addition, each cluster contains at least one hub node. The backbone consists of the hub nodes interconnected by backbone links. Similarly, nodes in each cluster are interconnected by cluster links.

![Figure 1 - An example of a Hierarchical Model](image)

In Figure 1, each cluster contains one hub, which is indicated by a filled square. The backbone links are the links interconnecting the hubs and the clusters are the links which interconnect nodes internally in a cluster. Note that there are no links between non-hub nodes in different clusters. The backbone network is a ring, and the cluster networks are examples of four topologies: Fully interconnected, tree, star and mesh. The mesh topology allows for any selection of links and thus other topologies are special types of meshes.

The hierarchical data model organizes data in a tree structure. There is a hierarchy of parent and child data segments. Usually, hierarchical networks are seen in the form of a tree or a dendogram in which the closely associated pairs of vertices have the lowest common ancestors that are lower in the tree than those of more distant related pairs. We can expect the probability of a connection between two vertices to depend on their degree of connectedness. Structures of such kind can be modeled
using a probabilistic approach in which we can assign each internal node \( r \) of the dendogram with a probability \( p_r \) and then connect each pair of vertices for which \( r \) is the lowest common ancestor independently with probability \( p_r \).

**Hierarchical Random Graphs**

Our model for hierarchical organization for the code network can be given as described easily. Let \( G \) be a graph with \( n \) vertices. A dendogram \( D \) can essentially be viewed as a binary tree with \( n \) leaves corresponding to the vertices of \( G \). Each of the \( n-1 \) internal nodes of \( D \) corresponds to a group of vertices that are derived from it. We assign a probability \( P_r \) with each of our internals nodes \( r \). Thus, given the two vertices \( i, j \) of \( G \) with \( r \) as their lowest common ancestor in \( D \), the probability \( p_{ij} \) that they are connected by an edge would be \( p_{ij} = p_r \) where \( r \) would be the lowest common ancestor in \( D \). The combination \( (D, \{p_r\}) \) of the dendogram and the set of probabilities then gives us a hierarchical random graph. We can view the hierarchical random graph as a slight variation of the classical Erdős–Rényi random graph \( G(n,p) \): this is because the presence or absence of an edge between any pair of vertices is independent of the presence or absence of any other edge.

**Fitting the graph to data**

We are tasked to find the hierarchical random graph(s) that will fit best our generated code network \( G \). We will operate under the assumption that all hierarchical graphs are \textit{a priori} equally likely; then we can also assume that the probability that a given model \((D, \{p_r\})\) is the correct explanation of the data is proportional to the posterior probability or likelihood \( L \) with which that model generates the observed network, (we can simply infer this from the Bayes Theorem). Our goal is to maximize \( L \) or more generally, to sample with probability proportional to \( L \) from the space of all models. Let us define the following:

- \( E_r \) – the number of the edges in \( G \) with \( r \) as their lowest common ancestor in \( D \).
- \( L_r \) – number of leaves in the left subtrees with root at \( r \).
- \( R_r \) – number of leaves in the right subtrees with root at \( r \).

The likelihood of the hierarchical random graph, consisting of Dendogram \( D \) and the set of probabilities \( \{p_r\} \) (using the convention \( 0^0 = 1 \)) is:

\[
\mathcal{L}(D, \{p_r\}) = \prod_{r \in D} p_r^{E_r} (1 - p_r)^{L_r R_r - E_r}
\]

**(E1)**

If we further fix the Dendogram \( D \), it is easy to find the probabilities \( \{\overline{p}_r\} \) that will maximize \( L \):

\[
\overline{p}_r = \frac{E_r}{L_r R_r}.
\]

**(E2)**

The likelihood of the dendogram evaluated at the maximum is then:

\[
\mathcal{L}(D) = \prod_{r \in D} \left[ \overline{p}_r^{E_r} (1 - \overline{p}_r)^{1-E_r} \right]^{L_r R_r}.
\]

**(E3)**

Thus, we can finalize the following formula:

\[
\log \mathcal{L}(D) = -\sum_{r \in D} L_r R_r h(\overline{p}_r),
\]

**(E4)**

We now use Markov chain Monte Carlo method to sample dendograms \( D \) with probability proportional to their likelihood \( L(D) \). To create the Markov chain we need to pick a set of transitions between possible dendrograms.
Each step of our Markov chain consists first of choosing an internal node \( r \) uniformly at random (other than the root) and then choosing uniformly at random between the two alternate configurations of the subtrees associated with that node and adopting that configuration. The result is a new dendrogram \( D \). My results of fitting are described in section 6.

5. Metrics and Centrality Measures

Clustering Coefficient

Clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. The local clustering coefficient of a vertex (node) in a graph quantifies how close its neighbors are to being a clique. Duncan J. Watts and Steven Strogatz [16] introduced the measure in 1998 to determine whether a graph is a small-world network. A graph \( G = (V, E) \) formally consists of a set of vertices \( V \) and a set of edges \( E \) between them. An edge \( e_{ij} \) connects vertex \( v_i \) with vertex \( v_j \). The neighborhood \( N_i \) for a vertex \( v_i \) is defined as its immediately connected neighbors as follows:

\[
N_i = \{ v_j : e_{ij} \in E \land e_{ji} \in E \}.
\]

We define \( k_i \) as the number of vertices \( |N_i| \) in the neighborhood \( N_i \) of a vertex.

An undirected graph has the property that \( e_{ij} \) and \( e_{ji} \) are considered identical. Therefore, if a vertex \( v_i \) has \( k_i \) neighbors, \( \frac{2}{k_i} \) edges could exist among the vertices within the neighborhood. Thus, the local clustering coefficient for undirected graphs can be defined as:

\[
C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}.
\]

Let \( \lambda_G(v) \) be the number of triangles on \( v \in V(G) \) for undirected graph \( G \). That is, \( \lambda_G(v) \) is the number of subgraphs of \( G \) with 3 edges and 3 vertices, one of which is \( v \).

Let \( \tau_G(v) \) be the number of triples on \( v \in G \). That is, \( \tau_G(v) \) is the number of subgraphs with 2 edges and 3 vertices, one of which is \( v \) and such that \( v \) is incident to both edges. Then we can also define the clustering coefficient as:

\[
C_i = \frac{\lambda_G(v)}{\tau_G(v)}.
\]

It is simple to show that the two preceding definitions are the same, since

\[
\tau_G(v) = C(k_i, 2) = \frac{1}{2}k_i(k_i - 1).
\]

These measures are 1 if every neighbor connected to \( v \) is also connected to every other vertex within the neighborhood, and 0 if no vertex that is connected to \( v \) connects to any other vertex that is connected to \( v \).

As an alternative to the global clustering coefficient, the overall level of clustering in a network is measured by Watts and Strogatz as the average of the local clustering coefficients of all the vertices \( n \):

\[
\bar{C} = \frac{1}{n} \sum_{i=1}^{n} C_i.
\]

I have calculated the clustering coefficient using SNAP’s GetClustCf method.

Diameter

The length \( \max_{u, v} d(u, v) \) of the "longest shortest path" (i.e., the longest graph geodesic) between any two graph vertices \( (u, v) \) of a graph, where \( d(u, v) \) is a graph distance. In other words, a graph’s diameter is the largest number of vertices which must be traversed in order to travel from one vertex to another when paths which backtrack, detour, or loop are excluded from consideration. In this report, the diameter has been calculated using SNAP’s GetBfsFullDiam method.

PageRank Measures

PageRank algorithm allocates authority weights to every page of a web graph based search on the web hyperlink structure. The basic premise of the PageRank Algorithm is that a higher authority score should be assigned to a page with higher number of links into it. This was refined by Brin and Page [1998] when they observed that not all links carry the same weight, thus links that came from higher quality pages should translate into higher authority scores.

I have executed the PageRank functions to gain insights on which functions seem to be the most important in the codebase. PageRank is the stationary distribution of a random walk which, at each step, with a certain probability \( \epsilon \) jumps to a random node, and with probability \( 1 - \epsilon \) follows a randomly chosen outgoing edge from the current node. PageRank is based on a random surfer model and may be viewed as a stationary distribution of a Markov chain. The PageRank solution is a principal eigenvector of a linear system that can be found via the power method. PageRank formula is defined as follows:

\[
P_R = (1 - d) \frac{1}{N} + d \sum_{i=1}^{k} \frac{PR(p_i)}{C(p_i)}
\]
Where $N$ is the total number of pages on the Web, $d$ is a damping factor, $C(p)$ is the outdegree of $p$, and $p_i$ denotes the inlinks of $p$.

Centrality Measures

Degree Centrality

Degree centrality of a node is defined as its degree/(N-1), where $N$ is the number of nodes in the network. I have reported my results of degree centrality using SNAP’s GetDegreeCentr method.

Degree centrality is a measure of network connectivity and identifies the most connected node in the network. As such, a node with high Degree centrality will have direct connections to many other nodes in the network.

The degree centrality of a vertex $v$, for a given graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges, is defined as:

$$C_D(v) = \deg(v)$$

The definition of centrality on the function level can be extended to the whole graph. Let $Y$ be the function with highest degree centrality in $G$. Let $X = (Y, Z)$ be the $Y$ function connected graph that maximizes the following quantity (with $Y$ being the function with highest degree centrality in $X$):

$$H = \sum_{j=1}^{(|Y|)} C_D(y^*) - C_D(y_j)$$

Therefore, the degree centrality of the graph $G$ is as follows:

$$C_D(G) = \frac{\sum_{i=1}^{H} [C_D(v^*) - C_D(v_i)]}{H}$$

The value of $H$ is maximized when the graph $X$ contains one central node to which all other nodes are connected, and in this case $H = (n - 1)(n - 2)$.

Centrality is defined in the context of this report as the functions that seem to be the most widely used in the codebase. Initially when I generated this data, it seemed to contain the constructor as the most widely used function, and logically that made sense but that was not best suited for the purpose of our analysis. Thus, I ran the results again with those functions removed.

Closeness and Betweenness Centrality

Let us consider a graph $G = (V, E)$ where the set $V$ of vertices represents actors and the set $E$ of edges representing sent links between actors. Here, a path is defined from $s \in V$ to $t \in V$ as an alternating sequence of vertices and edges, beginning with $s$ and ending with $t$, and each edge is connecting its preceding and its succeeding vertex. We can designate $d_G(s,t)$ to denote the distance between vertices $s$ and $t$ - the minimum length of any path connecting $s$ and $t$ in $G$. We know that $d_G(s,s) = 0$ and for every $s \in V$ and $d_G(s,t) = d_G(s,t)$ for $s,t \in V$. Therefore, formula to calculate closeness centrality is:

$$C_C(v) = \frac{1}{\sum_{t \in V} d_G(v,t)}$$

Closeness centrality is proposed by Sabidussi [13]. I have reported my results for Closeness centrality using SNAP’s GetClosenessCentr method.

Betweenness Centrality is calculated using the Floyd–Warshall algorithm. Let $\sigma_{st} = \sigma_{ts}$ represent the number of shortest paths from $s \in V$ to $t \in V$, where $\sigma_{ss} = 1$. Let $\sigma_{st}(v)$ denote the number of shortest paths from $s$ to $t$ that has some $v \in V$ lies on. The mathematical formula for it can be given as follows:

$$C_B(v) = \sum_{s\neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

I have reported my results for Betweenness centrality for nodes and edges using SNAP’s GetBetweennessCentr method.

Bonacich Centrality

The key contribution of Ballester, Calv´o-Armengol, and Zenou (2006) was first to relate the Nash equilibrium outcomes of a game to the Bonacich centrality vector, due to Bonacich (1987), defined by

$$b(G, a) = (I - aG)^{-1}1,$$

where $1$ is the n-dimensional vector with all components equal one and $a$ is the attenuation parameter. The Bonacich centrality describes the potential importance, influence, prominence of a function in the network. Since for $a < \frac{1}{\lambda_{max}(G)}$ it holds that:

$$b(G, a) = (I - aG)^{-1}1 = \sum_{k=0}^{+\infty} a^k 1^T G^k,$$

and $(G^k)_{ij}$ counts the total number of walks of length $k$ from $i$ to $j$. The Bonacich centrality of a function $i$ can be interpreted as follows:
Intuitively, the Bonacich centrality of a function \( i \) counts the number of walks emanating from \( i \) discounted by \( a \) to the power of their length. Hence, the attenuation parameter \( a \) captures the decay of influence of distant consumers on a particular consumer Bonacich centrality.

\[
b_i(G, a) = \sum_{k=0}^{\infty} a^k \sum_{j=1}^{n} (G^k)_{ij}.
\]

Figure 5 - PHP Code Network Dendogram

6. EXPERIMENTAL EVALUATION

Hierarchical Model
I was able to establish the best fit value for the PHP and C# code networks. It appears that the PHP code network is more hierarchical as seen from its dendogram (Figure 5), shrinking diameter (Figure 17) and its Clustering coefficient (Figure 7) whereas the C# network does seem to fit the Hierarchical model.

<table>
<thead>
<tr>
<th>Metric</th>
<th>PHP Code Network</th>
<th>C# Code Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Fit L Value</td>
<td>-1279256.317</td>
<td>-17441324.989</td>
</tr>
<tr>
<td>Step number of last best</td>
<td>99994</td>
<td>99897</td>
</tr>
</tbody>
</table>

Table 1 - Fitting Results – I have listed out the results of fitting the PHP and C# code networks to the hierarchical model. The C# network, leads us to believe the C# network does not fit the model. I was able to support this conclusion using sampling. Here, I was generating (100) random instances and comparing the fitted and actual models my metrics. On the other hand, the PHP network seems to loosely fit the hierarchical model and its metrics were more aligned.

Figure 6 - C# Code Network Dendogram

\[
C(K) \quad \text{property of Hierarchical Graph}
\]

The clustering coefficient of real networks is to a high degree independent of the number of nodes in the network. This inherent hierarchy can be categorized in a measureable manner using the recent finding of Dorogovtsev, Goltsev and Mendes that in deterministic scale-free networks the clustering coefficient of a node with \( k \) links follows the scaling law:

\[ C(k) \sim k^{-1} \]

This scaling law quantifies the coexistence of a hierarchy of nodes with different degrees of clustering. To establish if such hierarchical organization is present in the code networks, I measured the \( C(k) \) function for PHP code network and plotted it as follows:

Figure 7 - \( C(k) \) per Node

In Figure 7, it seems that we are able to establish an inverse relationship between the \( C(k) \) per node and the degree to meet the criteria for a Hierarchical model. Thus, the results are as follows:

**PHP Code Network** - \( C(k) \sim k^{-1.8} \)

The PHP network seems to be scale free, power law code network. We find that \( C(k) \) scales as \( k^{-1} \), indicating that the network has a hierarchical topology.

**C# Code Network** - \( C(k) \sim k^{-0.35} \), which we have looks like a loosely Hierarchical model.

7. ROBUSTNESS ANALYSIS

To conduct the Robustness analysis, I have used the fraction of functional nodes. A functional node in this scenario specifies that all of its out degree node edges are not deleted and for this robustness analysis, I have made this graph a directed graph. Additionally, I have also used the fraction of nodes in the largest connected component properties. I have
deleted nodes from the code network and computed a given property of network after each deletion phase. As we did in the homework, I have deleted the nodes in batches of X/100 and X/1000 in order to computationally manage the data (where X=total number of nodes). I have used two deletion policies – *Failure*, where I sample nodes uniformly at random and delete them from the graph and *Attack*, corresponds to a scenario where an adversary selectively targets a specific node. In this case, I have deleted the nodes in decreasing order of their degree, i.e. highest degree node first.

Figure 8 - Change in fraction of functional nodes of different graphs with node removal from 0 - 50%  
Figure 9 - Change in fraction of functional nodes of different graphs with node removal from 0 - 2%  
Figure 10 - Change in size of largest WCC of different graphs with node removal from 0 - 50%. Blue line represents attack and green represents failure.

We can observe that the PHP code network seems to be more robust than the C# code network. We can observe that networks break down fairly rapidly if they are less robust and in turn, more susceptible to attack such as the C# network in this case, as the weaker one. One observation we can make from this finding is that the PHP code seems to have better error handling, while the C# network is less robust and contains fewer functions for error handling.

8. FINDINGS

I was able to obtain the static and dynamic properties of the network. I have assumed for each function in the codebase to represent a node. An edge will be defined as a function calling another function. We are going to consider this as an *undirected graph*. The analysis can be categorized into Static and Dynamic Properties of the networks

**Static Properties**
In order to gain an overall understanding of the network, I have analyzed the static properties. These are listed as follows:

- Strongly Connected Component Size, Clustering Coefficient
- Degree Distribution

### Table 2 – Static Properties’ values

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value (PHP)</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Connected Component Size</td>
<td>14275</td>
<td>38976</td>
</tr>
<tr>
<td>Clustering Coefficient</td>
<td>0.099643</td>
<td>0.077872379</td>
</tr>
</tbody>
</table>

**Degree Distribution**

I have generated an undirected graph, below is the degree distribution. I am using Maximum Likelihood Estimation to fit the distribution, thus I have calculated $\alpha$ to be as follows.

### Table 3 - Alpha Values for PHP & C#

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP</td>
<td>1.6555971664079352</td>
</tr>
<tr>
<td>C#</td>
<td>2.0064477601849409</td>
</tr>
</tbody>
</table>

**Figure 14 - Degree Distribution**

We can observe from the PHP and C# code networks that they both seem to have fairly similar degree distributions. As we can see, they both follow the power law distribution.

**Dynamic Properties**

I also studied the dynamic properties of the graph over time. These are listed as follows:

- Nodes
- Edges
- Clustering Coefficient
- Average path length (Diameter)
- Triads

<table>
<thead>
<tr>
<th>Property Name</th>
<th>PHP Value</th>
<th>C# Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>Starts at 8187 and ends at 15098 after twelve months</td>
<td>Starts at 37069 and ends at 39460 after twelve months</td>
</tr>
<tr>
<td>Edges</td>
<td>Starts at 65415 and ends at 178175 after twelve months</td>
<td>Starts at 2293399 and ends at 2607827 after twelve months</td>
</tr>
<tr>
<td>Triads</td>
<td>Starts at 86816 and ends at 268787 after twelve months</td>
<td>Starts at 49573830 and ends at 21902194 after twelve months</td>
</tr>
</tbody>
</table>

**Figure 15 - Clustering Coefficient over a period of 12 months**

We can observe that the clustering coefficient values show great variations over a period of twelve months for both PHP and C# networks.

**Figure 16 - Number of Functions (by category) as found in the top 100 results that had the highest degree centrality values**

As we can see from figure above, the number of functions in the code quality category are increasing and that is a very good sign. Upon my discussion with the team, I found that they made a significant push for code quality and standardizing the codebase (towards August 2014) and their overall code quality went up and they received the same feedback from end users as well (fewer bugs made it out to production). Thus, we can stipulate that a ‘good’ quality codebase should contain a larger number of functions in the code quality category, followed by business layer.

Further, a very clear finding from this result will be that the functions that are returned with the highest degree centrality results (in the top 100) should have mandated unit test cases
written with them. We can mandate 100% unit test coverage for these methods since these methods are being used the most in the business layer and are being called the most heavily. Thus, as the codebase is progressing, we need to clearly stipulate 100% coverage for these business layer functions to ensure acceptable levels of code quality. The beauty of this finding is that even if the application code did not have code coverage to begin with; our analysis serves as a great starting point to begin to focus on quality. Further, we can also provide the development team clear criteria to ensure that their code is meeting the quality standards.

The Centrality measures obtained from my results, can be used heavily to judge/establish the code quality and gain insight into the functions most heavily used in the project. To explain with a use case that we face daily in our organization, amongst the 200 or so products we deal with; one of the most common issues is to identify issues in legacy codebases. Since such codebases span over millions of lines of code, it becomes an overwhelming task to identify where it is that the code is faltering and in fact it is like looking for a needle in a haystack. It is in this scenario that we can utilize the centrality measures to identify which functions are used most heavily and used that serves as a starting point. Further, the presence of unit test functions and other test functions provides us very useful insight into the code quality.

**PageRank Measure & Bonacich Centrality**

After looking at the Hubscore data, and it is very interesting to see that the ShowListing function seems to be highest scored. This application is used by real estate agents to sell/show property listings to potential buyers. Thus, it makes sense that the ShowListing function that is used to display the details of the listing has the highest hubscore since that would the function most used/link to in the application.

In the C# code, I found the DrawCaption caption function had one of the highest node PageRank. This finding makes sense for this codebase since the DrawCaption function is used to annotate on documents once a digital signature has been placed. Thus, it would make sense that this method is the most called/link to from the other functions in the application. This function returns a finalized document in which the digital signature is present along with the finalized annotations. Thus, we can see how critical this result is for a digital signature application. After talking to the developers of the team, they indicated that this is a critical method in the application and is very widely used; thus I was able to validate the high PageRank scores for this function.

Thus, we can clearly use PageRank measures to find the functions that are having an effect on application performance – directly or indirectly. Using the above results in the PHP and C# codebases, we were able to identify the functions most widely used, and in turn, the ones that needed to be optimized. After presenting my findings to the development team; they optimized the ShowListing and ResultCustomProgressStep methods for performance and saw a 4% improvement overall in terms of page load. Similarly, the C# team optimized DrawCaption, ReleaseComObject and PropertyChanged methods and saw a 6% improvement in the page load time.

PageRank seemed to be a most appropriate measure to gauge performance since it was able to help us identify functions that were used in the Business Layer, API layer, UI layer and the functions being called by the mobile applications. Thus, we can see that the teleportation vector works and PageRanks results are able to help us identify the functions that have the highest impact on the application performance.

As expected, we observe that the PageRank Node Authorities score results are almost identical to the Bonacich Centrality results. This seems logical since the calculation of Bonacich centrality can be considered almost the same as PageRank calculation with the exception of the teleportation vector being used in PageRank, whereas, it is not there in Bonacich centrality.

httpRequest method has a very high Bonacich Centrality and it seems to be doing a direct CURL request to an API, without any measures of data sanitization, this is indicative of the fact that the code is prone to security/malicious attacks since the user can directly pass data/url to the httpRequest function – this is a serious threat and needs to be fixed immediately. This proved to be an invaluable finding to ensure code security as the application was fairly vulnerable without a proper security fix in place.

**Betweenness and Closeness Centrality**

Using the results of Betweenness, I was able to identify that the ArchiveForLater (nodeId=788) function actually was the cause of major performance issues in our PHP codebase. This function that gets called in the interest of performance for user experience. This function adds items to a queue to be processed at a later time. Thus, this is a live use case of the Betweenness measure, and nobody had manually debugged this function since it wasn’t used or called in the presentation layer of our code. Using the Betweenness results, this function was uncovered and proved to be instrumental for my team.

**Diameter**

![Figure 17 Diameter over a period of 12 months](image-url)
In Figure above, the diameter of both of the code networks has been represented. We can see that the diameter (shortest path length) fluctuated a bit for the C# network but ended up more or less where it got started. For PHP, it is very interesting to see that the diameter has actually decreased over the course of twelve months. This interesting finding of shrinking diameters has been presented in [15] as a power law, with the following relationship:

$$e(t) \propto n(t)^{\alpha},$$

Where \(e(t)\) and \(n(t)\) denote the number of edges and nodes of the graph at time \(t\). \(\alpha\) lies strictly between 1 and 2. We can draw a parallel to the Community Guided Attachment (CGA) described in [15]; Graph Densification is based on a decomposition of the nodes into a nested set of communities such that the difficulty of forming links between communities increases with the community size. As suggested in [15], this model, which creates densifying graphs and can be considered as an example of a hierarchical graph generation model. The linkage probability between nodes is seen to decrease as a function of their relative distance in the hierarchy. We can draw a parallel to communities in our case with code categories – constructors/init, business layer functions, code security and quality functions. It would appear that especially in the PHP codebase, that as more code gets written, linkage amongst other nodes becomes harder and harder.

### 9. Conclusion

To summarize, I was able to perform successful experimental evaluation of algorithms and models on an interesting dataset(s). It was truly interesting to see that I could use my results to define clear quality standards for code for my teams. I was able to stipulate the following very clearly about enterprise level applications in my organization:

- The functions with highest degree centrality should have 100% coverage in unit test cases
- Any security related functions that come up in the top 10 results for PageRank, Bonacich, Degree Centrality need to be immediately optimized for code security. This is because these functions pose a grave threat to the application, due to their heavy usage. If security of these functions is compromised in a malicious attack, then the effects will be catastrophic to the application.
- We can also infer loosely that the absence of a hierarchical model does imply an unstructured codebase since the PHP codebase is well organized and structured whereas the C# code base is not.
- We can also lightly infer that more structured/standardized code bases will tend to be more robust; the underlying assumption is that there is proper structure in place for error handling.

This work can clearly be extended and used to extrapolate further meaningful measures of code quality and performance. These stipulations can be then applied to enterprise level business applications.

### References

[16] http://www.nature.com/nature/journal/v393/n6684/full/393440a0.html
[17] https://github.com/ndronen/PyHRG.