Research Article

Using Software Dependency to Bug Prediction

Peng He,¹ Bing Li,² Yutao Ma,² and Lulu He²

¹ State Key Lab of Software Engineering, School of Computer, Wuhan University, Wuhan 430072, China

² State Key Lab of Software Engineering, School of Computer, Research Center of Complex Network, Wuhan University, Wuhan 430072, China

Correspondence should be addressed to Peng He; penghe@whu.edu.cn

Received 12 March 2013; Revised 16 August 2013; Accepted 16 August 2013

Academic Editor: Chuandong Li

Copyright © 2013 Peng He et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Software maintenance, especially bug prediction, plays an important role in evaluating software quality and balancing development costs. This study attempts to use several quantitative network metrics to explore their relationships with bug prediction in terms of software dependency. Our work consists of four main steps. First, we constructed software dependency networks regarding five dependency scenes at the class-level granularity. Second, we used a set of nine representative and commonly used metrics—namely, centrality, degree, PageRank, and HITS, as well as modularity—to quantify the importance of each class. Third, we identified how these metrics were related to the proneness and severity of fixed bugs in Tomcat and Ant and determined the extent to which they were related. Finally, the significant metrics were considered as predictors for bug proneness and severity. The result suggests that there is a statistically significant relationship between class's importance and bug prediction. Furthermore, betweenness centrality and out-degree metric yield an impressive accuracy for bug prediction and test prioritization. The best accuracy of our prediction for bug proneness and bug severity is up to 54.7% and 66.7% (top 50, Tomcat) and 63.8% and 48.7% (top 100, Ant), respectively, within these two cases.

1. Introduction

During the software development and maintenance procedures, bug (defect) is one of the most important forces to drive the improvement of software quality. It is well-known that software engineering is a systematic and disciplined approach to developing software. In detail, it applies computer science, engineering principles and practices to the generation, operation, and maintenance of software systems. There are many key processes in software engineering. In particular, Software maintenance and upgrade play a vital role in software engineering. It has many advantages, such as improving the efficiency of programming, reducing the cost of maintenance, and promoting the development of software systems. In fact, most development efforts and expenditures are allocated to this stage. We know that the majority of software projects today are becoming increasingly large and complex. While maintenance is considered as an ongoing process throughout the software life cycle, according to maintenance activities account for over 70% of the total life cycle costs of a software product. Large development project

incurs a sizable number of bug reports every day, more and more efforts should be involved to resolve these burdened problems. Thus the effective and efficient solutions to bug prediction and test prioritization are urgent for both open source software and proprietary software.

Due to notable discoveries in the fast evolving field of complex networks and the dramatic increase of scale and complexity of real-world software systems, more and more researches in software engineering have also focused on representing the topology of software systems with network theory. Software represents one of the most diverse and sophisticated human made systems; however, little is known about the actual structure and quantitative properties of (large) software systems. In the context of combining complex networks theories with software engineering practices, the research for bug prediction has already made several discoveries over the past years. In this paper we propose to use some quantitative metrics from network sciences to actualize bug prediction of complex systems and further enhance software engineering practices.

Given the advantages of open source, (i.e., the openness of source codes and the available data repositories), the following work is conducted on open-source software. Open-source software usually maintains a bug repository (e.g., Bugzilla (<http://www.bugzilla.org/>), GNANTS (<http://www.gnu.org/software/gnats/>), and JIRA (<http://www.atlassian.com/software/jira/>)). With the increasing scale of software project, new features are added and more bugs are led into the system. According to the statistics [1], Eclipse on average 37 bugs was submitted, and more than 3 person-hours are spent on handling it per day; Mozilla discovers more than 300 bugs. However, potential bugs are often far more than that have been submitted in a system, and more serious. How to quickly retrieve much more bugs or serious bugs is the overarching goal of this work, specifically which metrics can be adapted to resolve this problem.

The rest of this paper is organized as follows. Section 2 is a review of related work. In Section 3, the preliminary theories of software networks and network metrics are hold and the research questions are presented, and Section 4 focuses on the research approach and the construction of software network. Section 5 shows the whole progress of our experiment. After that, we discuss the empirical results and several limitations. In Section 7, a conclusion for practice and research is made.

2. Related Work

Defect prediction models are used to support software inspection and to improve software quality by locating possible defects. For software bug prediction, many software metrics have been proposed. The most commonly-used ones are CK Object-oriented metrics, also traditional (e.g., LOC), and process (e.g., code churn) metrics are often used in some literature [2]. With the increasing maturity of complex network theory, network metrics are of more concerns, such as centrality and are used to defect prediction [3–5].

2.1. Software Network. Many real-world software systems are extremely complex and regarded as complex systems, in which software entities (e.g., objects, classes, and packages) are abstracted as nodes and the dependencies between two nodes as links or edges. Dependencies essentially represent the information flow within a software system and exert at least some influences on the overall success and quality of the product [6].

Characterizing large or ultralarge scale (ULS) software system as a complex networks is rational, and a great quantity of literature [7–12] has already demonstrated some software networks characteristics, like small-world phenomenon and scale-free distribution of degree. Likewise, with regard to software being built up out of many interacting units and subsystems (software entities) at multiple granularity, different levels of granularity software networks (component, feature, class, method etc.) have been constructed to study by researchers [13].

2.2. Network Metrics. Many metrics have been defined in both complex networks and social network analysis. Centrality [14, 15] was used to assess the relative importance

of nodes in a given network. The simplest one is *degree centrality*, known as the number of connections a node has to other nodes. In a social context, degree centrality can be explained in terms of the potential impact of a node on other nodes. However, degree centrality does not capture the global position of a node in the network in terms of how important a node is to all other nodes, so further measures are proposed such as *closeness centrality*, *betweenness centrality*, and *eigenvector centrality*.

Modularity [15, 16] is a desirable characteristic for software systems. The modularity of software architecture contributes to the sustainability of large scale software projects by fostering the decoupling and cohesiveness of software development. Particularly, as the software evolves over time, modularity might even facilitate its maintainability and expendability. In some literature, besides, in-degree and out-degree [8] presenting the complexity and reuse were used, respectively. PageRank and HITS [17] sorting algorithm were used to handle the directed network.

Additionally, Zimmermann and Nagappan [3] on Windows Server 2003 where the authors leveraged dependency relationships between software entities captured using social network metrics to predict whether they are likely to have defects. They found that network metrics perform significantly better than source code metrics at predicting defects. Premraj and Herzig. [18] presented a replication study based on Z&N work, and found that network metrics undoubtedly perform much better than code metrics in the stratified random sampling method.

2.3. Network Metrics for Application. Meneely et al. [4] examined collaboration structure with the developer network derived from code churn information to predict failures at the file level and developed failure prediction models using test and postrelease failure data. The result indicated that a significant correlation exists between file-based developer network metrics and failures. Crowston et al. [5] identified the core-periphery structure and applied these measures to the interactions around bug fixing for 116 SourceForge projects. Pinzger et al. [19] investigated the relationship between the fragmentation of developer contributions, measured by network centrality metrics and the number of postrelease failure with a developer-module network. Authors claimed that central modules were more likely to be failure prone than those located in surrounding areas of the network. Shin et al. [20] used network analysis to discriminate and predict vulnerable code locations. They evaluated that network metrics can discriminate between vulnerable and neutral files and predict vulnerabilities. Sureka et al. [21] derived a collaboration network from a defect tracking system and applied network analysis to investigate the derived network for the purpose of risk and vulnerability analysis. They demonstrated that important information of risk and vulnerability can be uncovered using network analysis techniques.

Dependencies exist between various pieces of component. Modification to components with little regard to dependencies may have an adverse impact on the quality of the latter. Zimmermann and Nagappan [3] proposed to use

network analysis on software dependency network, helping identify central program units that are more likely to conceal defects. Perin et al. [22] used PageRank for ranking classes of Pharo Smalltalk system based on a dependency graph representing class inheritance and reference. Bhattacharya et al. [23] constructed software networks at source code and module level to capture its evolution and estimate bug severity, prioritize refactoring efforts, and predict defect-prone release. Steidl et al. [24] used different network analysis metrics on dependency network to retrieve central classes and demonstrated that the results can compete with the suggestions of experienced developers. Zanetti et al. [15] studied bug reporter centrality to validate that the centrality of bug reporters is greatly helpful for bug triaging procedures.

To the best of our knowledge, bug prediction is a problem that is still an open question; hence we expect that our study can provide a supplement to the existing solutions. Although the prior researches on using network metrics to predict failure proneness or bug severity, our study leverages more relationships between classes in analyzing the implications of dependencies. Additionally, prior researches did not consider both as is done in this paper. We constructed software network at class-level granularity and then used network metrics to analyze bug proneness and severity. Differing from the existing work, we make some contributions as follows.

- (1) Five dependency scenes are considered in our class-level software network: inheritance, field, method, return, and parameter dependency, yet part of these relationships were taken into account in most of existing researches.
- (2) A comparison between different metrics is made firstly to explore the extent to which they reflect bug quantity and severity, after that to model the predictors with significant metrics to improve software development. Some researchers directly assemble various network metrics without filtration; thus it is inevitable to bring biases so that reducing the accuracy. Our work just bridges this gap by a comparison in the first place.
- (3) We select betweenness centrality and out-degree metrics for bug proneness and severity prediction. Just as 10% efforts are paid, an ideal F -value acquired about bug proneness and severity is up to 54.7% and 66.7% and 63.8% and 48.7%, respectively, which is comparable with the result in [4, 25].

3. Preliminary Study

3.1. Software Networks. There are all kinds of networks around us, such as social networks (e.g., friendship networks and scientific collaboration networks), technological networks (e.g., the Internet and WWW), and biological networks (e.g., neural networks and interaction of proteins in yeast). Surprisingly, the underlying structures of these networks have completely different statistical features than those of regular and random networks. They all belong to complex networks. These discoveries have served to draw together

FIGURE 1: A simple example of software network.

many disparate domains into an emerging science of network science.

With the rapid development of software technology and the pervasive of Internet, software scale and complexity so sharply increase that developers could not control the system in the global perspective; then software plan and quality cannot be guaranteed. In order to overcome the challenges, some pioneers of complex systems try to introduce graph theory that nodes and edges are used to simplify the structure. To our surprise, software network also shows the basic characteristics of complex networks. These exciting discoveries are paid attentions by researchers from software engineering. Through the interdisciplinary between complex network and software engineering, an approach has been emerging by abstracting a software system as a network, that is, software network.

Software network is an interdisciplinary outcome based on the combination of network science theory and software engineering practice. When constructing a software network, the source code should be handled in reverse engineering methods, (i.e., compiled the codes into a xml file, then derived the topology structure from the xml file). Figure 1 is an example of the fragments of software networks.

3.2. Network Metrics. Network metrics treat software entities as nodes in a graph and characterize them on the basis of their dependencies with other entities. As opposed to code metrics, network metrics take into account the interactions between entities, thus modelling the flow of software information. In this paper, node centrality, PageRank and HITS, in-/out-and total degrees, and modularity ratio metrics are involved, then those significant metrics were used to predict bug proneness and severity.

3.2.1. Eigenvector Centrality (EC). It is a measure of the influence of a node in a network. It assigns relative scores to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of current node than equal connections to low-scoring nodes and are computed by the adjacency matrix. For a given network (graph) $G = (V, E)$ with $|V|$ number of nodes, let $A = (a_{ij})$

be the adjacency matrix; if vertex i is linked to j , so $a_{ij} = 1$, and $a_{ij} = 0$ otherwise. The centrality score of vertex i can be defined as

$$x_i = \frac{1}{\lambda} \sum_{j \in M(i)} x_j = \frac{1}{\lambda} \sum_{j \in G} a_{ij} x_j, \quad (1)$$

where $M(i)$ is a set of the neighbors of i and λ is a constant. Only the greatest eigenvalue results in the desired centrality measure, the i th component of the eigenvector, then gives the centrality score of the vertex i in the network [26].

3.2.2. Betweenness Centrality (BC). It is based on the total number of shortest paths between all possible pairs of nodes that pass through a node [27]. It quantifies frequencies that a node acts as a bridge along the short path between two other nodes. If $g_i^{(st)}$ are all shortest path node s and t which pass through node i and $n^{(st)}$ is the shortest path between s and t , n is the total of nodes:

$$BC_i = \frac{\sum_{s < t} g_i^{(st)} / n^{(st)}}{(n-1)(n-2)}. \quad (2)$$

3.2.3. Closeness Centrality (CC). It concerned the farness to all other nodes. Thus, it is defined as the inverse of the sum of all distances to all other nodes [28]. If $d(i, j)$ is the distance between node i and j , so i 's centrality is

$$CC_i = \frac{1}{\sum_{j=1}^V d(i, j)}. \quad (3)$$

3.2.4. PageRank (PR). It is a variant of the Eigenvector centrality measure, which is a probability distribution used to represent the likelihood that while a person randomly clicking on links will arrive at a particular page. The rank value indicates the importance of a page, which is used here to denote the importance of a class:

$$PR_i = \frac{1-q}{N} + q \sum_{j \in V(i)} \frac{PR(j)}{\text{out}D(j)}, \quad (4)$$

where q is a damping factor ($q = 0.85$), $V(i)$ is the set of nodes that link to i , $\text{out}D(j)$ is the number of outgoing edges on node j , and N is the total number of nodes in a network.

3.2.5. HITS. It is also originally designed to rank web pages, which calculates two scores: authority and hub scores. The authority score is computed as the sum of the scaled hub scores that point to that page. The hub score is the sum of the scaled authority scores of the pages it points to. Here the authority score as a experimental metric is used.

3.2.6. Degree (D). In the case of directed network, it is meaningful to distinguish in-degree (the number of incoming links) from out-degree (the number of outgoing links). For software networks, in-degree is a representation of code reuse, and out-degree is design complexity. Nodes with large

in-degree are reused and with large out-degree are more complex in some contexts. In our paper, we use $\text{in}D(i)$, $\text{out}D(i)$, and D representing in-degree, out-degree, and total degree of a node, respectively.

3.2.7. Modularity Ratio (MR). Modularity is a metric used to evaluate the quality of partitioning community in a network by Newman and Girvan [29]. A system consists of many packages, and each package has a large number of classes or subpackages. For a given definition of modules or clusters and their underlying network structure, its respective degree of modularity is defined by

$$Q = \frac{\sum_i^n e_{ii} - \sum_i^n a_i b_i}{1 - \sum_i^n a_i b_i}, \quad (5)$$

where e_{ij} is the fraction of all edges that link nodes in module i to nodes in module j , $a_i = \sum_j^n e_{ij}$, $b_i = \sum_j^n e_{ji}$ (the sum of column and row resp.), while n is the total number of existing modules. According to this equation, we define a MR of modular i as follows:

$$MR_i = e_{ii} - a_i b_i. \quad (6)$$

3.3. Network Metrics for Application. Before presenting the research questions and the details of our approach, we give two terminologies used henceforth in this paper, which indicate the practical application of our work.

- (i) *Bug proneness*: Bug proneness is treated as a quality measure of software entries in this paper. An intuitive understanding is that it represents the probability that an entry (i.e., class) will become buggy in the process of debugging. The higher the bug proneness of a class is, the more likely it will be given priority to testing. For our purpose, we valueate it via the number of bugs and fixed classes.
- (ii) *Bug severity*: This terminology describes the impact of a bug on the entire system. Bug tracking systems classify the severity into several levels, such as critical, major, minor, and trivial. However, severity and priority is not the same concept. In some cases, a bug of critical severity but the priority to resolve it is low as this will happen rarely in real scenario. Here we would like to investigate that the importance of a class in the software network is related to the severity of bug caused by it.

With these metrics and application scenes, the following four research questions are proposed to guide the design of the case study. Questions 1–3 investigate the properties of software networks, whereas question 4 predicts the bug proneness and severity with the significant metrics.

- RQ1: Is the position of classes in the software network related to the bug proneness?
- RQ2: Is the position of classes in the software network related to the bug severity?
- RQ3: If so, which metrics are more significant?

RQ4: How well does the approach to predicting bug proneness/severity with significant metrics work?

4. Approach

In our approach, we mainly consider two Open Source projects—Tomcat and Ant—written in Java as our research subjects in the Apache Software Foundation (ASF) (<http://ant.apache.org/>) and (<http://tomcat.apache.org/>). The rationale is threefold. (1) They are well-known and stable projects; each has undergone a number of major release cycles and is still under active development. (2) The source code revision archives and SVN commit-log, dating back several years, and bug reports are available; it is a good chance for rewarding experience in Open Source projects. (3) The choice of Java programming language is limited by the tools developed to construct software networks, and we are interested in understanding open source software written in Java.

We observed the information provided by Bug tracking systems (Bugzilla and JIRA) and SVN commits; found that most bugs have given out the detailed classes and some have already posted the modified fragment of source codes. This information determines the feasibility of our approach. Besides, the dependencies between classes which are directed should not be ignored. Therefore, we decide to abstract the directed but unweighted software networks at the class level.

Let us assume that $G = (V, E)$ is a software network definition, $V = \{v_i\}$ is the set of all classes, and $E = \{e_{ij}\}$ is the set of all dependencies. We distinguish between different kinds of dependencies as follows.

- (i) v_i implementing/extending the interface/class v_j is *inheritance dependency*.
- (ii) v_i having a field of type v_j is *field dependency*.
- (iii) v_i calling a method of v_j is *method dependency*.
- (iv) A method of v_i returning an object of type v_j is *return dependency*.
- (v) A method of v_i taking an object of v_j as a parameter is *parameter dependency*.

A bug report yet offers many fields to help developers understand this issue; one of which is severity assigned based on how severely a bug affects the program by administrators. Table 1 shows eight levels of bug severity and their ranks in the Bugzilla. An objective for software providers is not only to minimize the total number of bugs but to ensure that bugs' severity as low as possible [23]. The approach mainly consists of four phases: (1) compiling the source code files to extract the class-level directed software networks regarding the dependency requirements; (2) exporting SVN commits and integrating them with bug reports, to obtain our necessary bug-class association relationships; (3) calculating a set of network metrics in the established software networks, then acquire their relationships with the number of bugs and bug severity; (4) finally, applying the significant metrics to the analysis of bug prediction. The framework is shown in Figure 2.

TABLE 1: Levels of bug severity and their ranks in the Bugzilla bug tracking system.

Bug severity	Rank
Blocker	8
Critical	7
Regression	6
Major	5
Normal	4
Minor	3
Trivial	2
Enhancement	1

FIGURE 2: The framework of our approach.

5. Case Study

This section presents the subject projects used for our study and the relevant data we collected. With the experimental results, the proposed questions are answered in turn.

5.1. Data. For our analysis, we collect a number of types of data. We gather the open source software data from source code repositories and bug database and determine the rank of each bug. In this paper, we choose Tomcat 7 and Ant 1.8.4 as our experimental subjects. As a successful open source project, both Tomcat and Ant maintain are a publicly available source. To construct expected software network, source codes are compiled to xml file; then software network is established by parsing the xml file. In the subsequent part, we simplified Tomcat software network as TSN and Ant software network as ASN. There are 35 versions for Tomcat 7. The latest version is continuously renovating so far, thereby only the last 34 versions are adopted in our work. But few differences in the number of nodes/edges between each version. A likely explanation for this is that Tomcat is a

TABLE 2: The statistics: The number of bugs and fixed classes nodes/edges of software network.

	Tomcat	Ant
#bugs	475	480
#classes	422(21%)	389(29%)
#SN_nodes	2015	1345
#SN_edges	9573	3937
#community	19	20

relatively mature open source software project with decade development history. Of course, not all projects keep this phenomenon. As we known that some projects have a nearly super linear growth [13]. At last, stable Tomcat 7.0.29 version is used to analyze, consists of 2015 nodes and 9573 edges, 19 communities. 1345 nodes and 3937 edges, 20 communities for Ant.

The bug reports are comprised of two sets, one used for the experiment and the other for prediction. That is, the first thirty versions of Tomcat are utilized for progress, (2) and (3), yet the last four versions for prediction. Owing to Ant data refers to only one version; eighty percent of Ant data is used for experiment and the remaining twenty percent for prediction. Note that studies of software defect rely on links between bug databases and commit changes, and this information plays an important role in measuring quality and predicting defects. Some prior researches suggested that there are some missing links between bugs and bug-fix commit logs. They proposed automatic approaches to recover missing links [30, 31] and found that the missing links lead to biased defect information, and it affects prediction performance. However, some authors argued that a bug may not be a bug but a feature or an enhancement [32, 33]. This misclassification also introduces bias in bug prediction models. While considering the missing links, the false position of bug classification increases. Hence only bugs always reported as a bug in the bug tracker are taken into account. Once effective bugs are identified, one checks the change logs by searching for bug ID and calculating the similarity of summary text. The reason for using heuristics is that most of missing links are developed due to misclassification in our datasets. The statistics of the experimental data are shown in Table 2. There are some test classes that are not taken into account because they are duplicate. In bug 53062, for example, both class *org.apache.catalina.connector.TestResponse* and *org.apache.catalina.connector.Response* appear, but they represent the same object in our study. Note that less than thirty percent of the classes are involved; it is confirmed that most of the bugs occur in a small number of classes. So all we should do is to have a guiding detection in a system.

5.2. Results

RQ1: Is the Position of Classes in the Software Network Related to the Bug Proneness? To answer this research question, first of all, the nodes are divided into different groups by metric value in descending order, and each group keeps the same

number of nodes. The reason for doing this is to facilitate the exploration for relations between the metrics and bugs. Owing to the last 15 nodes are never fixed in TSN; thus they are excluded to keep each group of 200 nodes. For ASN, in which nodes are divided into eleven groups evenly. The number of fixed classes and involved bugs in each group are recorded. With this information, the answer to Q1 is not hard to hold.

The number of fixed classes and involved bugs for different metrics in each group are illustrated in Figures 3 and 4. In Figure 3, EC, inD, D, and HITS have a significantly negative correlation (the bigger the average metric value of a class, the more bug-prone); BC and outD show a skew distribution; see Figures 3(a), 3(b), 3(e), 3(f), 3(g), and 3(h). There are some differences in Figure 4; BC, outD, and D show a significant correlation but do not an obviously skew distribution; see Figures 4(b), 4(f), and 4(g). The results of other metrics are fluctuated, especially in groups six, eight, and nine.

MR shows the same relationship with fixed classes and bugs. A few differences from the above metrics are that the distribution of bugs is higher than classes' and more fluctuated. In subfigure (i), the x -axis is the rank of modular (community); the larger modularity ratio, the higher its rank is. An overall positive trend is that the larger metric value of a class is, the more it will be bug-prone, and the more it is likely to be modified is shown, especially, BC, outD, and D metrics.

RQ2: Is the Position of Classes in the Software Network Related to the Bug Severity? Understanding and characterizing the relationships between network metrics and bug severity in open-source software projects is also a very interesting problem. Although the value of metric is positively related to bug proneness, whether it is related to bug severity, the answer may be not, because there exist a lot of bugs in a class, but all of them are not serious. We conduct the next experiment to validate this research question with the severity rank given in Table 1. The results in Figure 5 show that the relationship deviates what was expected. CC keeps stable in both projects; the overall distribution trend is increased first and then decreased. The top row depicts the severity of Tomcat, and the below row does Ant.

Rank 5 and 7 are two prominent breakpoints marked by the dotted line in each subfigure; they refer to the major and critical bugs, respectively. Bugs with rank 8 have the large metric value in TSN, on the contrary, this kind of bugs have the relatively low value in ASN. Blocker bugs should be avoided as much as possible in software engineering practices. Once this kind of bug occurs and is found the origin is where, it may trigger that a lot of other parts are modified, or the problem is only a small mistake but affects an vital node.

RQ3: If So, Which Metrics Are More Significant? Different metrics measure the importance of a node from different aspects in a network. According to the results obtained from RQ1, RQ2, we know that not all the metrics have the same expected relationship; some metrics may be more suitable for bug proneness prediction and others for severity prediction. In this section, another work is conducted to analyze how

FIGURE 3: Tomcat (the x -axis is the group ID, and y -axis is the number of classes and bugs).

significant these metrics are and which metrics are better. Three typical correlation analysis methods are used to resolve this problem: Pearson, Kendall, and Spearman (Sig. $P < 0.01$). Pearson is widely used in statistics to measure the degree of the relationship between linear related variables, but both variables should be normally distributed before using it. Kendall and Spearman are nonparametric tests that does not assume any assumptions related to certain distribution. But the former basically gives the impression of being a permutation test of sorts; the latter is essentially based on the principle of least squares.

Table 3 gives the correlation coefficient between metric value and bug proneness on eight metrics. For Tomcat, the coefficient of outD is minimum with 0.552, and D is maximum with 0.967, yet CC and PR metrics are not correlated. Although the correlations with outD in three methods are minimum, the skew distribution curve in Figure 3(f)

represented that most of the bugs and classes are fixed in the first two groups. On the other hand, as a result of no skew phenomenon for Ant, outD is more significant than other metrics in three methods. However, CC, EC, inD, PR, and HITS are all not significant. The maximum of outD is up to 0.952. The differences mentioned above suggest that treating these metrics should be unequally and with caution. Straightforward to apply all metrics will likely mislead interpretations. Consequently, BC and outD are the suitable metrics to represent the bug proneness and severity. Both of them will be used for the next prediction.

RQ4: How Well Does the Approach to Predicting Bug Proneness/Severity with Significant Metrics Work? Lastly, once we have understood and characterized the relationships between network metrics and bug proneness or severity in two open source projects, what are the effects, if any, on software

FIGURE 4: Ant (the x -axis is the group ID, and y -axis is the number of classes and bugs).

TABLE 3: Bug proneness: the correlation coefficient with eight network metrics.

	Bug proneness (Sig. $P < 0.01$)							
	BC	CC	D	EC	inD	outD	PR	HITS
Pearson-TSN	-0.804	—	-0.903	-0.852	-0.852	-0.752	-0.728	-0.933
Kendall-TSN	-0.764	—	-0.899	-0.689	-0.689	-0.552	—	-0.733
Spearman-TSN	-0.888	—	-0.967	-0.879	-0.879	-0.656	—	-0.903
Pearson-ASN	-0.891	—	-0.895	—	—	-0.915	—	—
Kendall-ASN	-0.782	—	-0.807	—	—	-0.844	—	—
Spearman-ASN	-0.909	—	-0.916	—	—	-0.952	—	—

The bold value refers to the maximum correlation coefficient of different analysis methods per project by comparing the eight metrics.

quality? Or what are the benefits to software engineering practices?

We have learnt that BC and outD metrics are remarkable to represent bug proneness and severity with the above experiments. Whether this conclusion is workable or not, we should validate it through bug proneness and severity

prediction with the test data. There are 67 effective bugs in test data, and 87 fixed classes related to these bugs for Tomcat and 135 bugs and 149 fixed classes for Ant. Given the experimental requirements, bug severity is divided into two categories, one is slight category in which bugs' rank is less than 4, but rather the other is severe category. Table 4 shows the resulting

predictive F -value, from Top 50 to 200 for Tomcat and from Top 50 to 125 for Ant, using selected metrics on software networks. The reason for top k within a range of 200 and 125 for these two cases is to ensure the selected nodes from the first group. F -value is calculated by integrating precise and recall as

$$F = \frac{2 * \text{precise} * \text{recall}}{\text{precise} + \text{recall}}. \quad (7)$$

The BCD metric is a combination of BC and outD, used to compare with each of them and follows the rule of $BCD = BC + outD$. For Tomcat, BCD metric performs better on fixed classes prediction than the others, and the maximum is 0.3316 when returning the top 100 nodes. While all the nodes in group $ID = 1$ are selected, three metrics obtain the same accuracy 0.2857. When it comes to the number of bugs, the maximum is 0.547 when using outD metric, and just top 50 classes are needed to check. In this column, outD and BCD metrics are better than BC metric in all cases. The last column is the results of bug severity. The finding shows outD performs far better than the other metrics, and the F -value is nearly more than twice. The best case is 0.6667 at top of 50 nodes.

For Ant, BCD metric performs better than the other two on fixed classes and bugs when returning top 50 classes. Except this case, outD has the highest accuracy at top 100 and 125 and obtains the best result when returning top 100 classes; the maximum of classes, bugs, and severity are 0.3534, 0.6383, and 0.4865, respectively. Throughout the prediction, outD is the most suitable metric for predicting the bug proneness and bug severity in a directed class-level software network.

Finally, few concentrations are devoted to analyze the average human consumptions and needed comments when developer wants to overcome one kind of bug. In Figure 6 one observes that more serious bugs need more people to participate, and these people do more discussions before these bugs are identified as fixed, which is consistent with the common experience except rank = 1. A reasonable explanation for this exception is that the enhancement bug has the lowest difficulty that many developers are able to find the problem and offer their own suggestions or revisions.

6. Discussion

RQ1: From the perspective of node importance, the more important a node is, the greater its probability of being modified is, and more bugs would be involved during the software development process. The central classes (nodes) form the skeleton of the entire system, especially the nodes with large betweenness centrality (BC) and outgoing links (outD). The former category of nodes plays an important role in bridging the different modules to ensure the normal execution of the entire system. Such nodes are also named “broker” in network science. The latter nodes are more complex for they aggregate behaviors from many other nodes. They give some implications to software testing. Nodes with high BC mean that they are more important in many contexts and have significant external responsibility. Bugs occurred in these nodes will influence the reachability between modules. On the other hand, nodes with a high outD mean that

they aggregate many behaviors of other nodes and have a significant internal complexity. They are more likely to be fixed and contain bugs. In this part, PR and CC metrics are not significant. In summary, our results are consistent with the work in [34]; some metrics are related to bug proneness, and the others are not.

We conclude that the importance of different metrics of a node plays an inconsistent role in the analysis of bug proneness. It is clear that BC and outD are suitable to test prioritization when the developers want to know which classes should be paid more attentions. In some special scenarios, for example, the node importance and out-degree can also be applied to bug location with regard to the relationship between them.

RQ2, RQ3: the hypothesis that a class would be apt to high bug proneness for great metric value, and the bugs would be more serious in this class, is not always true. Fortunately, Table 3 shows the average of BC, D, and outD metrics has a significant correlation with software bug proneness. The most likely explanation for strong correlations among some metrics and bug proneness is that the classes were divided into different groups having the same number of classes in descending order during the experiment. This treatment can also represent that the average metric value of preceding group is larger than the succeeding one and is corresponding to the top- k recommendation in the subsequent section. The results show an overall trend between groups so that stronger than the ones reported generally.

In Figure 5, major, regression and critical bugs have a relatively high value in both projects. Moreover, this phenomenon represented by major bug is more obvious. Additionally, Bugs raised from the broker will hinder the transfer of information between modules and function scheduling of entire system. As long as the modules cannot work smoothly, the quality of system cannot be guaranteed. A class with high outgoing links has more potential bugs, but the severity of the bugs is higher. Such classes are usually the central component in a system or the central nodes in a network.

As the members of a project team, who are more concerned about whether they can find serious bugs sometimes. In other words, they hope to find more bugs quickly and efficiently, while more severe bugs in a limited period of time. Our results provide an appropriate method to alleviate this challenge by BC and outD metrics.

RQ4: by the contrast among eight network metrics, BC and outD are selected to predict bug proneness and severity. Overall, compared to other metrics, a significant advantage of outD is shown in our results. It can be applied to identify which classes should be modified prior to others. The result also indicates that severe bugs usually have strong internal complexity.

Based on our work, we acquire some meaningful answers to the proposed four questions in Section 3. However, there are still some potential limitations and threats to the validity of our work.

- (i) All datasets used in our experiments are collected from open source projects Apache Tomcat 7 and Ant. We know that there are many other available

TABLE 4: *F*-value: fixed classes, bugs, and bug severity.

	Classes			Bugs			Severity		
	BC	outD	BCD	BC	outD	BCD	BC	outD	BCD
Top 50	0.2043	0.2920	0.2628	0.3760	0.5470	0.4786	0.2917	0.6667	0.4583
Top 100	0.2567	0.3102	0.3316	0.4072	0.4431	0.4551	0.3288	0.5069	0.3836
Top 150	0.2616	0.2954	0.2954	0.3594	0.3779	0.3687	0.2857	0.4184	0.2959
Top 200	0.2857	0.2857	0.2857	0.3071	0.3071	0.3521	0.2439	0.3740	0.2846
Top 50	0.1307	0.2111	0.2312	0.1946	0.4221	0.4322	0.1481	0.2613	0.2613
Top 100	0.2651	0.3534	0.3213	0.3489	0.6383	0.5957	0.2270	0.4865	0.4649
Top 125	0.3139	0.3358	0.3285	0.4385	0.5923	0.5846	0.3143	0.4381	0.4286

The bold value refers to the best *F*-value, identified by comparing three metrics (BC, outD, and BCD) at the top K, of predicting fixed classes, bugs, and bug severity, respectively.

FIGURE 5: Bug severity (Tomcat-top/Ant-bottom, the *x*-axis is bug rank, and *y*-axis is the mean of metric).

software repositories that are helpful to bug analysis. We would like to improve our approach by combining more software resources, such as mailing-list, forum messages, in the future work, to enrich the findings.

- (ii) A limitation of our work is that we choose projects written in Java, because of the tool used to construct software network can only deal with java source code. Whether our conclusion can explain projects written in other language or not, it still needs to evaluate.
- (iii) Theoretically, software networks will evolve over time; therefore differences are inevitable between multiple

versions. In this paper, we utilize a stable version of each project as the construction standard of software network, instead of considering the structural difference between them. Although such treatment is rough, it will not affect the final experimental results.

- (iv) As mentioned at the beginning, software system can be characterized from multiple granularity, which is a multi-granularity software network. We investigate the bug proneness and severity through network metrics in class level. If the results will be more

FIGURE 6: Human factor (the x-axis is bug rank).

delightful from the other granularity, it is an attractive theme.

7. Conclusion

We constructed a class-level software networks and introduced nine representative and commonly-used network metrics to bug prediction. An empirical study has been conducted on open-source project Apache Tomcat 7 from version 7.0.0 to 7.0.33 and Ant 1.8.4 from Feb 2002 to November 2012. Our analysis demonstrated that there is a statistically significant relation between class's importance and the probability that the class is to be modified and buggy. Also, class's importance is related to the severity of raised bugs. The result also showed that just about ten percent of efforts paid, and the accuracy of our prediction for bug proneness and bug severity can be up to 54.7% and 66.7%, 63.8% and 48.7%, respectively, when successively returning the top 50 and 100 classes within these two cases.

We expect that our findings are insightful and can be used to support the design and development of software, helping engineers in assessing the risk of adding or dropping a feature in the case of supporting existing dependencies between classes. We also believe that our approach can be leveraged in the analysis of bug prediction, test prioritization of others open source softwares. Finally, our findings also provide additional empirical evidence on the importance of dependencies between classes to researchers in the social network analysis domain.

The next work will mainly focus on two aspects. On the one hand, we will collect more open source projects (i.e., Eclipse, Mozilla, or projects deployed on SourceForge) to validate the generality of our approach. On the other hand, we will further take into account the human factors, by

the state of software development, are a process of human participation. An exploration about what impact will be brought by sociotechnical congruence on bug prediction is urgent and meaningful.

Acknowledgment

This work is supported by the National Basic Research Program of China no. 2014CB340401, National Natural Science Foundation of China nos. 61273216, 61272111, 61202048, and 61202032, Wuhan Planning Project of Science and Technology no. 201210621214, the Open Foundation of Jiangsu Provincial Key Laboratory of Electronic Business no. JSEB2012-02, and Zhejiang Provincial Natural Science Foundation of China no. LQ12F02011.

References

- [1] J. Anvik, "Automating bug report assignment," in *Proceedings of the 28th International Conference on Software Engineering 2006 (ICSE '06)*, pp. 937–940, May 2006.
- [2] D. Radjenovica, M. Herickob, R. Torkarc, and A. Živkovičb, "Software fault prediction metrics: a systematic literature review," *Information and Software Technology*, vol. 55, pp. 1397–1418, 2013.
- [3] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th International Conference on Software Engineering 2008 (ICSE '08)*, pp. 531–540, May 2008.
- [4] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2008/FSE '08)*, pp. 13–23, November 2008.

- [5] K. Crowston, K. Wei, Q. Li, and J. Howison, "Core and periphery in Free/Libre and Open Source software team communications," in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS '06)*, p. 118, January 2006.
- [6] T. Zimmerman, N. Nagappan, K. Herzig, R. Premraj, and L. Williams, "An empirical study on the relation between dependency neighborhoods and failures," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation (ICST '11)*, pp. 347–356, March 2011.
- [7] S. Valverde, R. F. Cancho, and R. V. Solé, "Scale-free networks from optimal design," *Europhysics Letters*, vol. 60, no. 4, pp. 512–517, 2002.
- [8] C. R. Myers, "Software systems as complex networks: structure, function, and evolvability of software collaboration graphs," *Physical Review E*, vol. 68, no. 4, Article ID 046116, 15 pages, 2003.
- [9] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 687–708, 2007.
- [10] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, article 2, 2008.
- [11] B. Li, Yutao Ma, and J. Liu, "Advances in the studies on complex networks of software systems," *Advances in Mechanics*, vol. 38, no. 6, pp. 805–814, 2008.
- [12] B. Li, H. Wang, Z. Y. Li, K. Q. He, and D. H. Yu, "Software complexity metrics based on complex networks," *Acta Electronica Sinica*, vol. 34, pp. 2371–2375, 2006.
- [13] W. Pan, B. Li, Y. Ma, and J. Liu, "Multi-granularity evolution analysis of software using complex network theory," *Journal of Systems Science and Complexity*, vol. 24, no. 6, pp. 1068–1082, 2011.
- [14] T. Opsahl, F. Agneessens, and J. Skvoretz, "Node centrality in weighted networks: generalizing degree and shortest paths," *Social Networks*, vol. 32, no. 3, pp. 245–251, 2010.
- [15] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pp. 1032–1041, 2013.
- [16] M. S. Zanetti and F. Schweitzer, "A network perspective on software modularity," in *Proceedings of the International Workshop on Complex Sciences in the Engineering of Computing Systems*, pp. 175–186, 2012.
- [17] C. Ding, X. He, P. Husbands, H. Zha, and H. D. Simon, "PageRank, HITS and a unified framework for link analysis," in *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 353, 354, 2002.
- [18] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: a replication study," in *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*, pp. 215–224, September 2011.
- [19] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2008/FSE '08)*, pp. 2–12, November 2008.
- [20] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
- [21] A. Sureka, A. Goyal, and A. Rastogi, "Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis," in *Proceedings of the 4th India Software Engineering Conference (ISEC '11)*, pp. 195–204, February 2011.
- [22] F. Perin, L. Renggli, and J. Ressa, "Ranking software artifacts," in *Proceedings of the 4th Workshop on FAMIX and Moose in Reengineering (ICSM '10)*, pp. 1–4, 2010.
- [23] P. Bhattacharya, M. Iliofotou, I. Neamtii, and M. Faloutsos, "Graphbased analysis and prediction for software evolution," in *Proceedings of the International Conference on Software Engineering (ICSE '12)*, pp. 419–429, 2012.
- [24] D. Steidl, B. Hummel, and E. Juergens, "Using network analysis for recommendation of central software classes," in *Proceedings of the 19th Working Conference on Reverse Engineering*, pp. 93–102, 2012.
- [25] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis," in *Proceedings of the International Conference on Software Engineering (ICSE '09)*, pp. 1–11, May 2009.
- [26] M. E. J. Newman, *Networks*, Oxford University Press, Oxford, UK, 2010.
- [27] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*, Cambridge University Press, 1994.
- [28] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1978.
- [29] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, Article ID 026113, 15 pages, 2004.
- [30] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '10)*, pp. 97–106, November 2010.
- [31] R. Wu, H. Zhang, S. Kim, and S. C. Cheung, "ReLink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT/FSE '11)*, pp. 15–25, September 2011.
- [32] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. G. Guéhéneuc, "Is it a bug or an enhancement? A text-based approach to classify change requests," in *Proceedings of the Conference of the Center for Advanced Studies (CASCON '08)*, ACM, October 2008.
- [33] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 35 International Conference on Software Engineering*, pp. 392–401, 2013.
- [34] R. Tonelli, G. Concas, M. Marchesi, and A. Murgia, "An analysis of SNA metrics on the Java Qualitas Corpus," in *Proceedings of the 4th India Software Engineering Conference (ISEC '11)*, pp. 205–213, February 2011.