PEOPLE, PROCESSES, AND PRODUCTS: CASE STUDIES IN OPEN-SOURCE SOFTWARE USING COMPLEX NETWORKS

by

Jian James Ma

A Dissertation Submitted to the Faculty of the

Committee On Business Administration

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY
WITH A MAJOR IN MANAGEMENT

In the Graduate College

THE UNIVERSITY OF ARIZONA

2011

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Jian James Ma entitled People, Processes, and Products: Case Studies in Open-Source Software Using Complex Networks and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

____ Date: 11/30/2011
Daniel Zeng

____ Date: 11/30/2011
David E. Pingry

____ Date: 11/30/2011
Zhu Zhang

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

____ Date: 11/30/2011
Dissertation Director: Daniel Zeng

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the author.

SIGNED: Jian James Ma

ACKNOWLEDGEMENTS

First and foremost I want to thank my advisor, Professor Daniel Zeng. He is the mentor of me in both academic work and personal life. Together we have experienced struggles and cheers. His countless and endless support helped me overcome a great number of critical moments. This dissertation would not have been possible without the guide and trust of my advisor. It has been a great honor to study in his research group. His profound intelligence and continuous dedication will be remembered forever.

I am grateful for Professor David E. Pingry's participations in my dissertation committee. He has made a number of useful comments on and suggestions to this research. I would also like to thank Professor Zhu Zhang for his help to not only my dissertation but also the project that we have worked together. His kindness and openness leave me a great memory in my research career.

My gratitude also extends to Professor Wei H. Lin, my minor advisor in the SIE department, who taught me Simulation Modeling and Analysis. I would like to show my gratitude to other MIS faculty and staff members at the University of Arizona for their support during my studies. Cinda Van Winkle and Mona Lopez have provided me tremendous help to a number of administrative matters.

I also own everyone in our research group a big debt: Aaron Runpu Sun and Ping Yan for that they are always there to offer helps and assistance.

Lastly, I owe my deepest gratitude to my wife, Xiaofei Annie Wen. It was she who helped me go through the most difficult period of my life. It was she who always believed me even when I had doubt about myself. It is she who has completed my life in every aspect. I owe everything including my life to her.

I would also like to dedicate my dissertation to my daughter Helen Tee-Tee Ma, my father Xiurang Ma, and to my always beloved mother Lianzhen Kong, may her sole rest in peace forever.

DEDICATION

To

My dearest lifelong influences:

Mother, Father,

Annie,

And Tee-Tee

TABLE OF CONTENTS

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Open-source software becomes increasingly popular nowadays. Many startup companies and small business owners choose to adopt open source software packages to meet their daily office computing needs or to build their IT infrastructure. Unlike proprietary software systems, open source software systems usually have a loosely-organized developer collaboration structure. Developers work on their "assignments" on a voluntary basis. Many developers do not physically meet their "co-workers." This unique developer collaboration pattern leads to unique software development process, and hence unique structure of software products. It is those unique characteristics of open source software that motivate this dissertation study. Our research follows the framework of the four key elements of software engineering: Project, People, Process and Product (Jacobson, Booch et al. 1999). This dissertation studies three of the four P's: People, Process and Product.

Due to the large sizes and high complexities of many open source software packages, the traditional analysis methods and measures in software engineering can not be readily leveraged to analyze those software packages. In this dissertation, we adopt complex network theory to perform our analysis on open source software packages, software development process, and the collaboration among software developers. We intend to discover some common characteristics that are shared by different open source

software packages, and provide a possible explanation of the development process of those software products. Specifically we represent real world entities, such as open source software source code or developer collaborations, with networks composed of inter-connected vertices. We then leverage the topological metrics that have been established in complex network theory to analyze those networks. We also propose our own random network growth model to illustrate open source software development processes. Our research results can be potentially used by software practitioners who are interested to develop high quality software products and reduce the risks in the development process.

Chapter 1 is an introduction of the dissertation's structure and research scope. We aim at studying open source software with complex networks. The details of the 4-P framework will be introduced in that chapter.

Chapter 2 analyzes five C-language based open source software packages by leveraging function dependency networks. That chapter calculates the topological measures of the dependency networks extracted from software source code.

Chapter 3 analyzes the collaborative relationship among open source software developers. We extract developer's co-working data out of two software bug fixing data sets. Again by leveraging complex network theory, we find out a number of topological characteristics of the software developer networks, such as the scale-free property. We also realize the topological differences between from the bug side and from the developer side for the extracted bipartite networks.

Chapter 4 is to compare two widely adopted clustering coefficient definitions, the one proposed by Watts and Strogatz, the other by Newman. The analytical similarities and differences between the two clustering coefficient definitions provide useful guidance to the proposal of the random network growth model that is presented in the next chapter.

Chapter 5 aims to characterize the open source software development process. We propose a two-phase network growth model to illustrate the software development process. Our model describes how different software source code units interconnect as the size of the software grows. A case study was performed by using the same five open source software packages that have been adopted in Chapter 2. The empirical results demonstrate that our model provides a possible explanation on the process of how open source software products are developed.

Chapter 6 concludes the dissertation and highlights the possible future research directions.

CHAPTER 1 INTRODUCTION

Open source software has been playing an increasingly important role in modern companies especially for startup companies. Unlike its proprietary alternatives, open source software packages do not charge any fees for firms and individuals to use. Moreover all the source code is accessible to the public and allows any user to customize based on the user's special requirements (Johnson 2006; Colazo 2010). The developers of open source software usually do not have a rigid management structure since most developers choose to join and leave a project on a voluntary basis (Hahn, Moon et al. 2008; Singh 2011). Very few developers make their commitment choices based on short term financial rewards. Developers usually join an open source software project because of personal interest in coding. The loosely managed developer structure results in the different software development process and different fashion of project management compared with those of proprietary software. Those unique natures that the open source software systems possess are likely to lead to different structures of software source code, the most important part of the software product.

As open source software packages are expected to fulfill more and more business needs, the size of those software packages increases dramatically and their software structures become much more complicated over the years. Quantitative evaluation of software architecture with large size is a complicated and often difficult task. Traditional analysis methods and measures that have been utilized in software engineering field are no longer suitable to analyze those complicated software packages. Many readily-available evaluation metrics such as the number of files in a package, total lines of code, or the number of developers for a given project are not sufficiently descriptive. Even measures such as complexity (Harrison, Samaraweera et al. 1996; Kang and Bieman 1999), maintainability (Li and Henry 1993), and cohesion (Etzkorn, Davis et al. 1998) often fail to fully capture the nature of software systems which are becoming increasingly complex.

Thus, in order to gain meaningful insight into the structure of software systems, software developers and analysts have to search for more powerful, more detailed and more informative tools to serve their needs. Recently random network theory becomes increasingly popular in analyzing complex systems. Topological measures from random network theory and the related modeling techniques can afford a deeper level of understandings regarding the formation and evolution of code-based software structures and the processes governing the development of software systems.

Since its proposal by Erdos and Renyi (Erdos and Renyi 1959; Erdos and Renyi 1960; Erdos and Renyi 1961) (ER) in 1959, random network theory has been applied to the study of complex systems across a wide variety of domains (Barabasi and Albert 1999; Bilke and Peterson 2001; Barabasi, Jeong et al. 2002; Newman, Forrest et al. 2002; Ancel Meyers, Newman et al. 2003; Shaw 2003; Wu and Kshemkalyani 2008; Zheng, Zeng et al. 2008; Simpson, Hayasaka et al. 2011). Our research is aimed at extending this emerging line of work by focusing on identifying unique characteristics of the structure of complex software systems and the structure of software developers, and more importantly, developing models that can explain those structural characteristics.

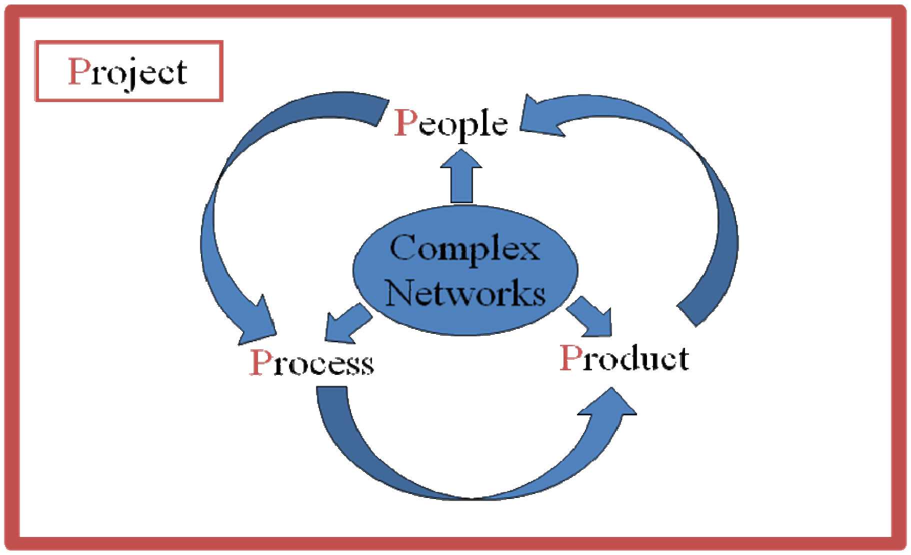


Fig. 1.1 4 P's in the software engineering field.

We intend to study the open source software by following a four P's framework in software engineering field. There are four P's that are considered the key elements in software engineering field: namely Project, People, Process and Product. Fig. 1.1 shows the four P's and their relations.

Project: the discipline of planning, organizing, securing, and managing resources to achieve software development goals. Within the four P's, project defines the overall scope of the software development goals, procedure, milestones, cost, resources, and etc.

People: the prime movers in a software project including architects, developers, testers, and their supporting management, plus users, customers, and other stakeholders.

Process: a flow of control that can execute concurrently with other processes, usually including requirement analysis, architectural and detailed design, implementation, testing, deployment and maintenance.

Product: the artifacts of development, such as models, code, documentations, and work plans. Product is final deliverable and thus the goal of a software project.

We choose to start our research by studying the final goal of the software project, the Product. Chapter 2 presents a descriptive work on the structures of open source software products. We choose five C-based open source software packages with different package size and from different application domains. All of the packages are widely adopted in their domain and hence are considered "successful" products. We extract the function dependency networks from the software source code by representing a function with a vertex and a function call with an edge. By leveraging the topological measures that have been established in random network theory, we realize the five function dependency networks possess the same three features: (1) average degree is independent of network size, (2) clustering coefficient, in either of two definitions proposed by Watts and Strogatz, and by Newman, is independent of network size, and (3) the network is scale-free. Our empirical findings show that there are common structural characteristics that are shared by open source software products across different size and domain.

In Chapter 3, we extend our empirical study by analyzing another P in the Four P framework, the People. Specifically we examine software developer's collaboration networks that are extracted from two real world bug fixing data sets. The bug fixing data sets contain information on which developer has fixed which bug at what time. We extract two bug-developer bipartite networks, and then derive a developer-developer collaboration network for each data set. In the bug-developer bipartite network, each bug and each developer are represented by a vertex. An edge can only exist between a bug and a developer, which indicates the developer has been working on solving that bug. By examining the average degrees of the bug set and the developer set, we find the average number of bugs that developers have been involved in changes as the size of the open source software changes. In the developer-developer collaboration network, a vertex represents a unique developer, and an edge connecting two vertices indicates that the two developers represented by those two vertices have been working on at least one same bug. Our analysis shows that both data sets' collaboration networks are scale-free.

Since our empirical study involves clustering coefficient, one of the most useful topological measures in random network theory, we perform an analytical study on that measure in Chapter 4. We compare two widely adopted clustering coefficient definitions, one proposed by Watts and Strogatz, the other by Newman. Our analytical study shows that Watts-Strogatz's definition is the mean of the ratio of local clustering factors, and Newman's definition is the ratio of the means of local clustering factors. We also examine the lower bounds and upper bounds of those two definitions, and the conditions to meet those extreme bounds. Our further analysis shows that the extremely popular vertices have little impact on Watts-Strogatz's clustering coefficient definition, whereas those popular vertices are the dominating factors for Newman's definition. The analytical results in Chapter 4, e.g. the impact factors of the two clustering coefficient definitions, are used as part of the motivations when our network growth model is proposed in Chapter 5.

We examine the third P in the Four P framework, the Process, in Chapter 5. We aim at providing an explanation on the development process of open source software packages. Specifically, we propose a two-phase network growth model to illustrate the software development process. As the size of the software increases, the software growth experiences two different growth mechanisms, namely two phases. The first phase follows the hierarchical network model of Ravasz and Barabási (Ravasz and Barabasi 2003). The second phase starts when the size of the software grows larger than a threshold. In the second phase, software modules are connected by limited and random inter-module links. The second phase strives to minimize the coupling across software modules.

To validate the two-phase network growth model, we reuse the empirical results from Chapter 2, the five function dependency networks extracted from the C Based open source software packages, and the three common topological features that are shared by those five function dependency networks. Both analytical and numerical studies show that the proposed model reproduces the topological features observed in real-world software packages. We then conclude that our model can be a reasonable explanation of the open source software development process.

Chapter 6 concludes our research findings and presents some possible future study directions.

Our study will provide useful guidance for software engineering researchers and practitioners in order to develop high quality open source software products, and reduce risks and costs in the open source software development process.

CHAPTER 2 ANALYZING COMPLEX SOFTWARE PRODUCTS WITH FUNCTION DEPENDENCY NETWORKS

2.1 Introduction

High-quality architectures can offer software development efforts considerable benefits, including improved productivity during development and maintenance cycles, reduced vulnerability to attacks and system failures, and increased understandability and extensibility. However, evaluating the architectures of software systems, one of the most complex man-made artifacts ever created, is a complicated and often difficult task (Harrison, Samaraweera et al. 1996; Kang and Bieman 1999; Kanmani, Uthariaraj et al. 2004). Simple evaluation metrics, such as the number of files, functions, lines of code, or developers, and more sophisticated metrics measuring (Chhabra and Gupta 2010) such properties as complexity (Etzkorn, Davis et al. 1998; Wang, Chung et al. 2000; Fothi, Nyeky-Gaizler et al. 2003; Ma, He et al. 2010; Shin, Meneely et al. 2011), maintainability (Li and Henry 1993; Bagheri and Gasevic 2011), quality (Sarkar, Kak et al. 2008; Eichinger, Kramer et al. 2010), coupling (Li and Li 2011) and cohesion (Ott and Bieman 1998) currently available in the literature are deemed not sufficiently informative (Wang, Chung et al. 2000; Taherkhani, Korhonen et al. 2011). There is still a great need for methodologies that can help us gain deeper insights into the structures of software systems and the processes governing their development.

In recent years, there have been a large number of studies devoted to characterizing and explaining a wide variety of complex networks, such as the World-Wide-Web, the Internet, movie actor collaboration networks (Watts and Strogatz 1998), science collaboration networks (Newman 2001; Newman 2001; Newman 2001), and cellular machinery networks (Milgram 1967; Barabasi and Albert 1999; Watts 1999; Amaral, Scala et al. 2000; Albert and Barabasi 2002; Barabasi, Jeong et al. 2002; Newman, Watts et al. 2002; Newman 2003; Goodreau, Kitts et al. 2009; Li and Niu 2011; Simpson, Hayasaka et al. 2011). In this chapter, we apply and extend the rich constructs and models produced by this stream of research in analyzing software systems. We model a software package as a network (also called a *graph*), in which a node (also called a *vertex*) represents a function in the package and an edge (also called an arc) connecting two nodes reflects the existence of dependency (i.e., function call) between the functions represented by the nodes. This function dependency network provides a macroscopic view of software structure while obviating the minutiae of source code particulars. As software systems are typically composed of numerous functions, with interactions among them directly reflecting the design and execution of the systems, studying this network can lead to valuable understanding of the underlying systems.

The rest of the chapter is organized as follows. We first introduce some background information of software engineering, open source software packages and random graph theory in Section 2. In Section 3, we present five widely-adopted C-based open source software systems as our empirical study subjects. Section 4 explains how we construct the function dependency networks for the five real world software packages. In Section 5, we introduce several important topological measures that are widely leveraged in random network analysis. Specifically we introduce two different definitions of clustering coefficient. Our empirical findings are summarized in Section 6. The analysis reveals a set of interesting features exhibited by the function dependency networks constructed after the packages. Finally, we conclude the chapter with a summary of contributions and a discussion of possible future research directions in Section 7.

2.2 Research backgrounds

Quantitative evaluation of software architecture is a complicated and often difficult task. Many readily-available evaluation metrics such as the number of files in a package, total lines of code, or the number of developers for a given project are not sufficiently descriptive. Even measures such as complexity (Etzkorn, Davis et al. 1998; Wang, Chung et al. 2000), maintainability (Li and Henry 1993), and cohesion (Ott and Bieman 1998) often fail to fully capture the nature of software systems which are becoming increasingly complex (Wang, Chung et al. 2000). Thus, in order to gain meaningful insight into the structure of software, systems developers and analysts must turn to more detailed indicators. To this end we argue that topological measures from random graph theory and the related modeling techniques can afford a deeper level of understandings regarding the formation and evolution of code-based software structures and the processes governing the development of software systems.

Since its proposal by Erdos and Renyi (Erdos and Renyi 1959) (ER) in 1959, random graph theory has been applied to the study of complex systems across a wide variety of domains (Albert and Barabasi 2002; Newman 2003). The first step of adopting random graph theory is to represent a real complex system by defining a graph composed of weightless and size-less vertices and the edges connecting those vertices. A vertex usually represents a true entity that exists in the real complex system. An edge is to describe a relationship between two vertices. The relationship between two vertices captures the atomic dependency between those two entities that are represented by the two vertices. Sometimes, along with the connection of those two vertices, the edge can also define the degree and the direction of the inter-entity relationship. For example, if a Web page is represented by a vertex, then the edge between two vertices can be leveraged to represent a hyper textual link between two Web pages. Moreover, the edge may have its direction which determines the hyper link direction between the two Web pages. And the edge may also bear a weight that describes the level of connections, namely the number of hyperlinks, between those two Web pages. Since the entities usually contain rather simple internal structures, it is intuitively true that describing the atomic relationship between two entities is relatively straightforward.

Once the atomic inter-entity relationship is defined, assuming the homogeneity of those relationships, researchers can obtain the system level characteristics by aggregating the effect of a large number of atomic relationships. Random graph researchers have developed a number of useful metrics and methods to analyze the aggregated effect of the relationships. Detailed information can be found in following sections. Those metrics are believed to capture the system level characteristics of the complex system which in general sense are difficult to obtain.

Despite the breadth of random graph analysis for complex systems, however, very few studies (e.g., (Potanin, Noble et al. 2005)) have sought to use this framework to analyze software systems. Our research is aimed at extending this emerging line of work in random graph theory to describe the architectural structure of software packages while keep the main characteristics at the detailed design level. Our research can provide an infrastructure to further examine the software structure and hopefully provide explanatory evidences to decipher the software development process in the near future. As is common in the random graph literature, we use the terms "graph" and "network" interchangeably in our research.

2.3 Empirical analysis of real-world software packages

We have identified five widely-adopted open-source software packages—OpenSSH (a secure communication client), Httpd (Apache Web server), Gaim (a multi-protocol instant messaging client), MySQL (a database management system), and GIMP (GNU Image Manipulation Program)—as the focus of our inquiry. We specifically chose open-source software packages because of the following two reasons. First of all, open source software packages bear the nature of sharing their source code with the public users. Hence it is fairly easy to achieve their source code without too much hassle and liability concerns. Second of all, the success of an open-source software package is largely determined by its qualities whereas a proprietary software package may achieve its wide deployment by the financial advantage or readily available marketing channels of the software's owner company. Thus we may conservatively conclude that a popular or widely adopted open source software package reflects the good quality of that software package. At this stage of our research, we are more motivated to reveal the characteristics of good quality software packages than bad or mediocre ones. All of the five software packages that we chose to analyze had proven records of shining popularity and high rate of being downloaded based on voluntary basis. All of those packages are developed using the C programming language, which continues to be one of the most important languages in system software development. We chose C-based software packages due to C language's relatively simple architectural structure. Namely C-based software packages have easily identifiable function calls among C functions. And those function calls are the clear signals of function dependency which is the key point of a software package's architectural complexity, reliability, and extensibility. Table 2.1 contains some basic information about the packages.

While each individual software package may bear its unique micro structure, its overall architecture shares some common characteristics as a result of commonly adopted open source software development principals. For example, open-source software packages are usually developed on a volunteer basis. The development management is executed in a loosely controlled manner. These similarities intrigue us to hypothesize that there are some common properties universally possessed by different open-source software packages across a broad range of purpose, size, and complexity. We are more interested in such universally possessed properties than in the specifics of a particular software package.

Table 2.1 Basic information about five open-source software packages

Package	Version	Size (KB)	Number of files	Total number of lines	Number of lines of source code
OpenSSH	4.0p1	1,978	273	74,087	40,817
Httpd	2.0.54	3,717	302	116,698	60,927
Gaim	1.3.1	6,386	477	229,481	137,030
MySql	4.1.12	11,715	750	371,748	188,099
GIMP	2.2.8	24,611	2,120	759,056	488,503

2.4 Network construction

In order to leverage random network theory to analyze complex software packages, we have to firstly abstract the software package to edge-connected vertices as described in the above section. From the software engineering point of view, naturally, we choose the source code of the software packages as our target object. The source code of a complex software package usually possesses a hierarchical structure. For example, a Java based software project usually contains a number of Java packages, each of which contains a number of Java files. Any Java file may contain one or more Java classes which in turn contain a number of parameters and/or methods. Each Java method contains a number of variables and a number of lines of code. Any level of self-contained source code group may be represented as a vertex in the network which brings in more flexibility and, on the other hand, more difficulty in choosing a reasonable entity as the network vertex.

Since we choose C programming language based software packages as the study subjects, and fortunately, C-based software packages carry a rather flat architectural structure. Specifically, the building block of a C-based package is functions by nature. Each function must have a unique function name to distinguish itself throughout the entire package. As function is the smallest self-contained operational unit of a language C-based software system, we focus on analyzing the dependency relationships among the functions in a package. For each package, we construct a function dependency network representing each unique function in a unique source file with a vertex, and indicating the existence of dependency between two functions (i.e., at least one of the two functions calls the other) with an edge between the corresponding nodes. At this stage of our research, we are not aimed at revealing the internal structure of a function. We do not differentiate function “importance” by reading the code inside a function. Rather we would like to study how functions are connected with one another, and how those connections will affect the architectural characteristics of the software packages. For example, a heavily connected vertex indicates an important function; and a heavily connected network implies high coupling in general, etc.

Note that the function dependency networks we extract from software packages are un-directed rather than directed. That is, we intentionally ignore the direction of function calls when we construct the networks. We choose to adopt this strategy for two main reasons. First, the two definitions of clustering coefficient we use in this paper are mostly applied to undirected graphs. Similar approaches have been adopted in other studies. For example, Barabasi (Barabasi and Albert 1999; Barabasi, Albert et al. 1999) applied undirected networks to analyze the structure of the World-Wide-Web, where vertices represent Web pages and edges represent hyperlinks between Web pages, although the hyperlinks are indeed directed. Ignoring the edge direction and weight allows us to study network properties such as clustering coefficient, going beyond basic degree distributions. No widely-accepted definitions of clustering coefficient for directed, weighted networks are yet available. Second, we are particularly interested in whether two functions are related (connected) than in the direction of their connection in this study. The fact that two functions are related is by itself worth examining and can have important software engineering implications. For example, if a few functions are highly connected to each another, and thus form a complete or semi clique, regardless of direction of function calls, one may suspect that these functions are written and maintained by the same team of software developers. As such, functional relations may provide useful insights for developer social network study.

Furthermore, we ignore the weight attribute on the edges when constructing the function dependency networks (Yook, Jeong et al. 2001). Every edge bears the same weight. That is, we do not differentiate connected function pairs based on the number of function calls between them. In software engineering field, low coupling is an important pursuit that every software developer has to consider. Coupling is defined by a "whether or not connected" relation which means the coupling situation between any pair of connected functions remain the same regardless how strong the connection is between the function pair. To better analyze how coupling is formed among functions, we choose to focus on whether two functions are connected rather than how they are connected.

In a C-based system, we need to treat self loops carefully. In a network, a self loop denotes the situation that a vertex has an edge connecting itself. In a function dependency network, a self loop means a function calls itself, which, in software engineering field, is referred to as the recursion. While recursion is meaningful and can often significantly reduce the length of the code, it is not directly related to our research objectives. Since we are more interested in analyzing coupling among functions, a recursive function does not contribute to the coupling in our research domain. Thus, we specifically remove the self loops when constructing the function dependency networks.

2.5 Network topological measures

After constructing the function dependency networks for the real world software packages, we aggregate the atomic inter-function relationships to the system level metrics. We leverage several topological measures that are widely adopted in random network theory. Table 2.2 summarizes some symbols used in the rest of the paper.

Table 2.2 Symbols of basic network measures

Symbol	**Measure**
N	Number of nodes in the network, referred to as the network size
M	Number of edges in the network
k_i	Degree of node i, i.e., the number of edges connected to the node
$\bar{k}$	Average degree. $\bar{k} = \frac{1}{N}\sum_{i=1}^{N} k_i = \frac{2M}{N}$.
$p(k)$	The fraction of nodes in the network that have degree k, or equivalently, the probability that a node chosen uniformly at random has degree k
$C^{(1)}$	Clustering coefficient in the Watts-Strogatz definition
$C^{(2)}$	Clustering coefficient in the Newman definition
C^{Rand}	Expected clustering coefficient of a purely random graph

2.5.1 Network size

N is the number of vertices, namely functions, in the network. It reflects the size of the network. This variable is a direct indicator on the network complexity and system run time. The higher the N, the more complicated the network is. Since in a C-based system, every function has a unique function name, the computation of N is quite straightforward.

2.5.2 Network connectivity

M is the number of edges, namely function calls, in the network. This variable reflects the system connectivity of the network. When the size of network, N, is the same, the higher the M, the more heavily connected the network is. As described in the previous section, M only counts the unique pair of connected vertices. That is, any unique connected pair of functions counts one regardless how many times those two functions call each other. Also as we described before, self loops are removed due to the scope of our research. The number of edges also counts the connections between two different vertices.

2.5.3 Vertex degree

The degree of node i, k_i, is the number of edges connected to that node. The variable k_i reflects the connectivity around an individual vertex. Thus the variable can be leveraged as an indicator of the importance or popularity of that individual vertex. If a vertex has a significantly high degree comparing with other vertices in the network, it usually implies that vertex represents an important function which calls or is called by many other functions. Since the function dependency network is undirected and weightless, k_i refers to the unique number of vertices that are connected to node *i*, regardless which function is the caller and which one is the callee. Also since self loops are removed, those connected vertices that are being counted into the degree can not be vertex *i* itself.

2.5.4 Average degree

The average degree $\overline{k}$ is the arithmetic mean of the degree values of all the vertices in the network. Due to the impact of the network size N, M itself can not be used to describe the level of connectivity of the network. Thus the average degree $\overline{k}$ is usually leveraged to describe the overall connectivity of the network. On a normal case, the higher the $\overline{k}$, the more connected the network is. Note since the network is undirected, every edge is counted twice when the average degree is computed. The reason is because every edge is used to compute the value of degree for both vertices that that edge connects. If the network were constructed as a directed network, every edge should be counted only once.

2.5.5 Degree distribution

Degree distribution, the distribution of $p(k)$, is also very useful in charactering a network. By definition, $p(k)$ is the fraction of nodes in the network that have degree k, or equivalently, the probability that a node chosen uniformly at random has degree k. In our research, $p(k)$ is calculated as counting the number of vertices that have the same degree value. The average degree $\bar{k}$ may be too abstract to catch the detailed information of the network connectivity. Specifically the impact of a heavily connected vertex may be overwhelmed by a large number of vertices that have low degree values. Thus the distribution of $p(k)$ is adopted in order to reveal more detailed insight of the network connectivity. The degree distribution may imply the variance of vertex degree values.

The display of the distribution of $p(k)$ is usually performed by plotting $p(k)$ against the sorted k values. In most random network research works, the logarithmic values of both $p(k)$ and k are used to display the plot.

While the degree distribution of a random graph is binomial or Poisson in the limit of large graph size, real-world complex networks have been found to exhibit very different degree distributions, indicating that they are not purely random and may have formed following particular philosophies. In particular, networks with power-law degree distributions (i.e., $p(k) \sim k^{-\alpha}$, and $\log p(k)$ is linear with regard to $\log k$) have been the focus of a large number of studies (Barabasi and Albert 1999; Barabasi, Albert et al. 1999; Faloutsos, Faloutsos et al. 1999; Dorogovtsev and Mendes 2001; Albert and

Barabasi 2002; Barabasi, Jeong et al. 2002). Such networks are referred to as *scale-free* networks (Barabasi and Bonabeau 2003). Many real-world networks, including the World-Wide-Web, the Internet, the actor collaboration network, the power grid network, and science citation networks, have been found to be scale-free (Watts and Strogatz 1998; Barabasi and Albert 1999; Newman 2003). Some of them, such as the actor collaboration network and the power grid network, indeed only partially exhibit power-law degree distributions (Barabasi and Albert 1999). In these networks, the frequencies of low-degree nodes are lower than what are expected from power-law distributions. The $\log p(k)$-vs-$\log k$ curves of the networks have a hockey stick shape with a straight line in the middle and a bended head at the beginning. Moreover, there is a considerable amount of noise on the tail, indicating the existence of large variations in the frequencies of nodes with very high degrees. Despite such deviations, for model building purposes and simplicity, in many cases researchers still classified such networks as scale-free, as the majority of their nodes follow power-law distributions (Barabasi and Albert 1999).

2.5.6 Clustering coefficient

Clustering coefficient measures the extent to which being a neighbor is a transitive property. Clustering coefficient captures the level of connectivity of a local community within a network. The higher the clustering coefficient, the more connected the community is. A special case of the heavily connected group is “clique” which includes a group of vertices that every vertex is connected to each other within the clique.

A high clustering coefficient in the function dependency network implies that a group of functions are highly connected to one another. That group of highly connected functions is likely to represent a self-contained software module, or subsystem within the software system. The heavy connections within that module indicate those functions are strongly related and thus are strongly alike from the functionality standpoint. In other words, a highly connected software module indicates the high cohesion within that module.

Cohesion is a measure of how strongly-related the functionality expressed by the source code of a software module is. In a system with high cohesion, the source code readability and extensibility are usually high. Like low coupling, high cohesion is another important software engineer incentive that software developers would like to chase. We hope clustering coefficient can provide a quantitative measure of the level of software cohesion.

Clustering coefficient has two commonly used definitions (Watts and Strogatz 1998; Newman 2003). Watts and Strogatz (Watts and Strogatz 1998; Watts 1999) define a clustering coefficient for any node i that has at least two neighbors (the clustering coefficient of a node with degree zero or one is defined as zero):

$$C_i^{(1)} = \frac{a_i}{k_i(k_i - 1)/2}, \tag{2.1}$$

where a_i is the number of edges among the neighbors of node i. This is equivalent to the following more graphical formulation

$$C_i^{(1)} = \frac{\text{Number of triangles connected to node } i}{\text{Number of connected triples centered on node } i}, \tag{2.2}$$

where a connected triple means a single node connected to an unordered pair of others.

The clustering coefficient for the entire network, is then defined as the average

$$C^{(1)} = \frac{1}{N}\sum_{i=1}^{N} C_i^{(1)} \tag{2.3}$$

Another definition of clustering coefficient introduced by Newman (Newman, Strogatz et al. 2001; Newman, Watts et al. 2002; Newman 2003) is

$$C^{(2)} = \frac{3\times \text{ Number of triangles in the network}}{\text{Number of connected triples in the network}} \tag{2.4}$$

The constant three is used to normalize $C^{(2)}$ into the $[0,1]$ range, as each triangle contributes to three connected triples centered on different nodes. The two definitions are similar in that $C^{(1)}$ calculates the mean of ratios while $C^{(2)}$ the ratio of means. However, they can give quite different results, as $C^{(1)}$ weights the contributions of low-degree nodes more heavily while $C^{(2)}$ treats all nodes equally.

Clustering coefficient also helps to tell whether a network is purely random. A random graph, defined by Erdős and Rényi (Erdos and Renyi 1959; Erdos and Renyi 1960; Erdos and Renyi 1961), consists of N nodes connected by M edges chosen randomly from $N(N-1)/2$ possible edges. If define C^{Rand} as the expected clustering coefficient of a purely random graph, the clustering coefficient of a random graph has an expected value of $C^{Rand} = \bar{k} / N$.

2.6 Empirical findings

By adopting the topological measures in the previous section, we perform our analysis on the five real world software packages. We present some observations concerning the properties of the function dependency networks below, focusing on three topological measures—average degree, clustering coefficient, and degree distribution—which are considered particularly informative (Albert and Barabasi 2002; Newman 2003).

Table 2.3 Topological measures of five function dependency networks

Network	N	M	$\bar{k}$	$C^{(1)}$	$C^{(2)}$	C^{Rand}
OpenSSH	1,221	5,436	8.90	0.160	0.038	0.00729
Httpd	2,061	5,005	4.86	0.108	0.028	0.00236
Gaim	5,181	15,009	5.79	0.084	0.030	0.00112
MySql	5,024	19,745	7.86	0.158	0.034	0.00156
GIMP	14,380	45,224	6.29	0.132	0.023	0.00044

Table 2.3 summarizes several topological measures. We refer to the function dependency network of a software package by the package name. For example, the function dependency network of the MySql package is referred to as the MySql network. The data support the intuition that a larger software package also has a larger function dependency network. Minor exceptions exist. For example, the size of the MySql package is approximately twice of that of the Gaim package, but the size of the MySql network is slightly smaller than that of the Gaim network. Like package size, network size spans a wide range, from about a thousand to over ten thousand. Fig. 2.1 to 2.5 shows the networks drawn with Pajek (http://vlado.fmf.uni-lj.si/pub/networks/pajek/).

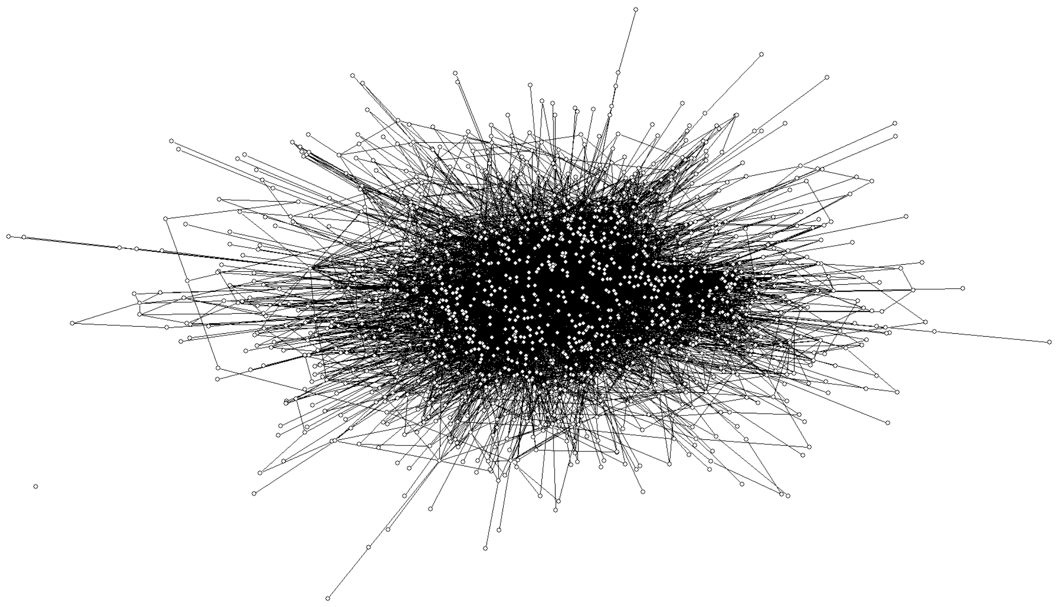


Fig. 2.1 OpenSSH software function dependency network.

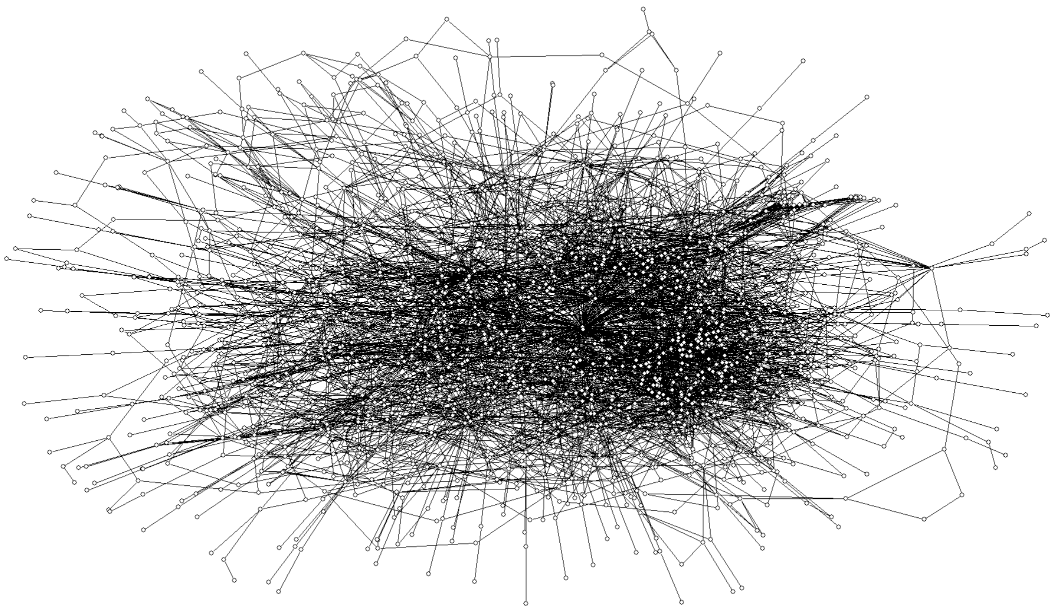


Fig. 2.2 Httpd software function dependency network.

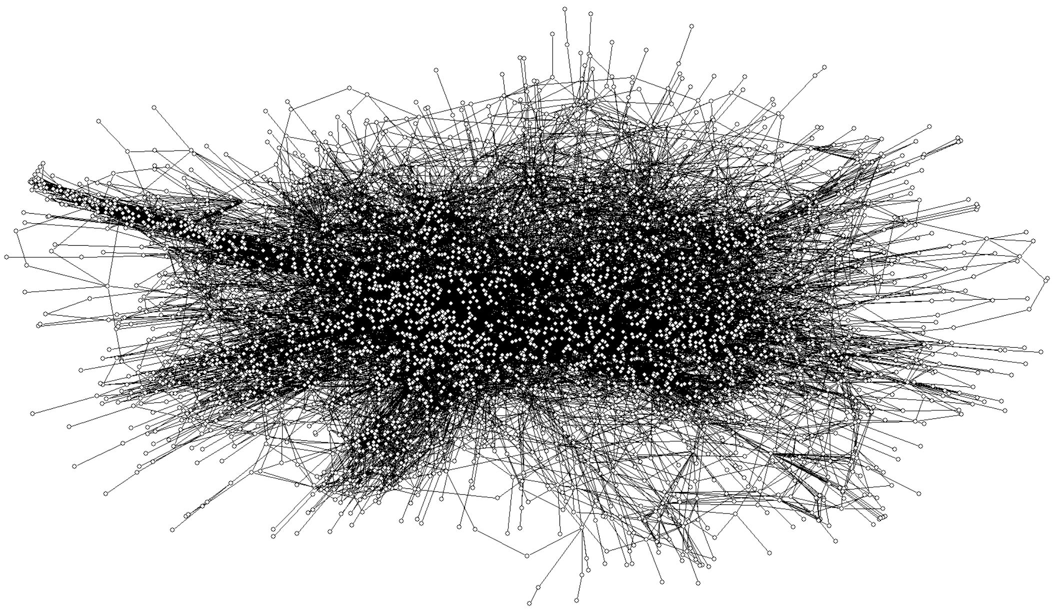


Fig. 2.3 Gaim software function dependency network.

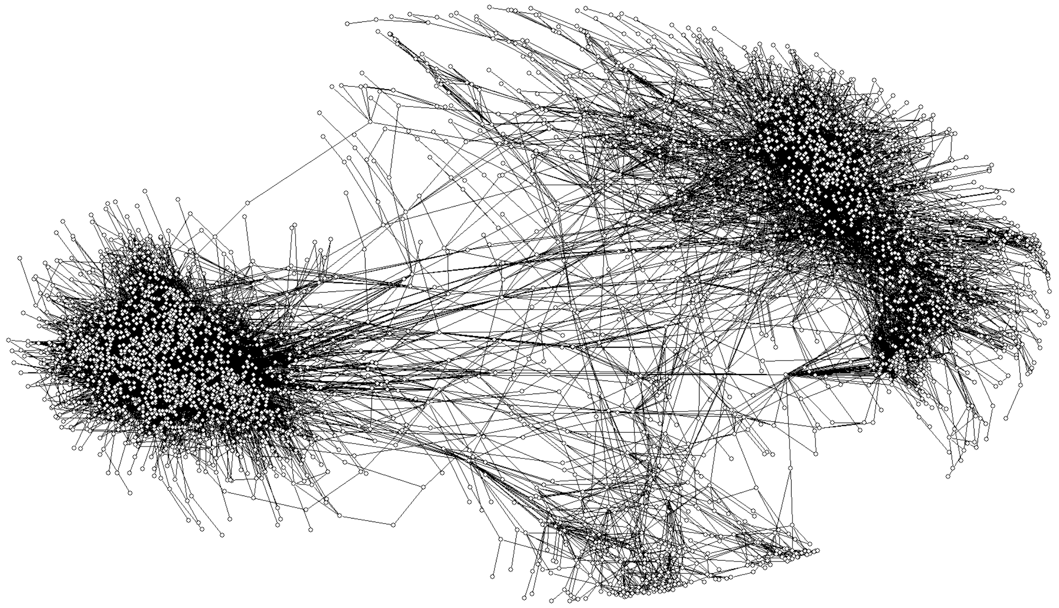


Fig. 2.4 MySql software function dependency network.

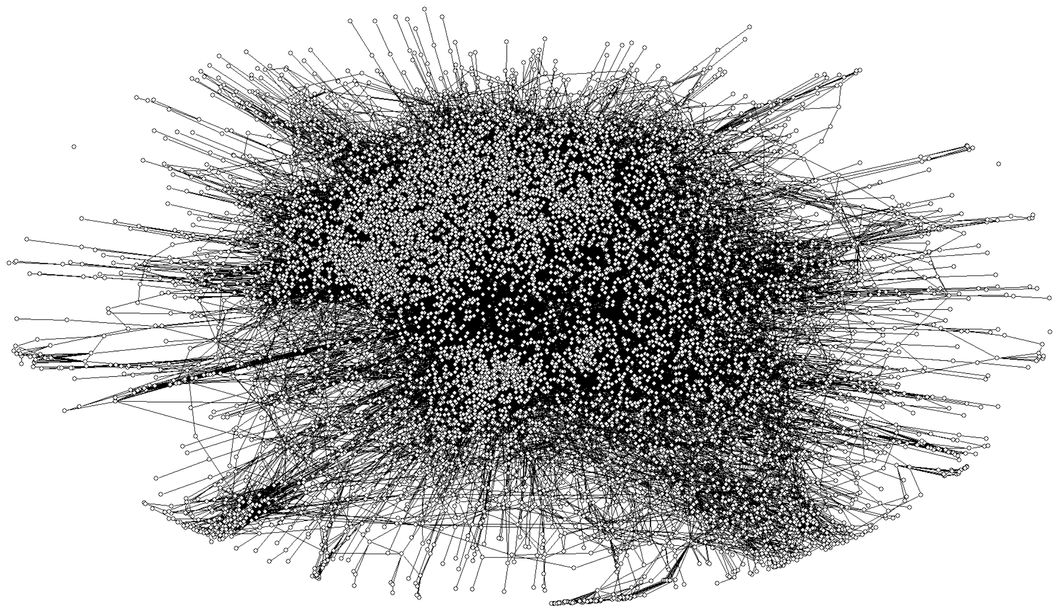


Fig. 2.5 GIMP software function dependency network.

2.6.1 Average degree

While network size spans a wide range, average degree varies only slightly, ranging approximately from five to nine. There is also no clear trend of variation on average degree as the network size increases. This seems to imply that average degree is independent of network size, although, strictly speaking, the five networks are distinct software packages, rather than different versions of the same software package. This finding provides some support to our hypothesis that there are some common topological properties shared by different software packages over a wide range of purpose, size, and complexity.

2.6.2 Clustering coefficient

Table 2.3 lists the clustering coefficient measures of the five real world software packages based on three definitions: Watts and Strogatz's definition $C^{(1)}$, Newman's definition $C^{(2)}$, and Erdős and Rényi's definition C^{Rand}.

Clustering coefficient is a useful measure in charactering a network. It helps to tell whether a network is purely random. Based on Erdős and Rényi's model, the clustering coefficient of a random graph has an expected value of $C^{Rand} = \bar{k} / N$ with N being the number of nodes and M being the number of edges in the random graph.

The clustering coefficient of any of the five networks (Table 2.3) is much larger than that of a random graph with the same N and M. The networks are far from random graphs. The five networks also appear to have very similar clustering coefficients, no matter which of the two definitions is used, although their sizes are largely different. This seems to imply that the clustering coefficient of a software function dependency network, in either of the two definitions, is independent of network size, although the five networks are distinct software packages rather than different versions of the same software package. This finding again provides some support to our hypothesis that there are some common topological properties shared by different software packages over a wide range of purpose, size, and complexity.

2.6.3 Degree distribution

Fig. 2.6 shows the degree distributions, the distribution of $p(k)$, of the five function dependency networks. They all appear to be approximately power-law distributions, as $\log p(k)$ appears to be approximately linear with regard to $\log k$, other than a bended head and some noise on the tail, similar to what have been observed in the actor collaboration network and the power grid network (Barabasi and Albert 1999). To better analyze the degree distributions of the networks, we have applied logarithmic binning (Adamic) to smoothen the original data (Fig. 2.7). The logarithmically binned degree distributions also show that the five real-world networks seem to follow approximately power-law degree distributions. In order to statistically analyze the power-law property, we use linear regression to test the relationship between $\log p(k)$

and $\log k$. As shown in Table 2.4, the coefficient of $\log k$ is statistically significant at the .001 level and the R^2 is over .9 for every network (over .95 for GIMP and Httpd), indicating that the linear model can adequately reflect the relationship between $\log p(k)$ and $\log k$. Thus, the five function dependency networks, similar to the actor collaboration network and the power grid network (Barabasi and Albert 1999), can be considered roughly scale-free and are far from random graphs. The networks also have very similar exponents (α ranging from 1.32 to 1.86), as Potanin et al. (Potanin, Noble et al. 2005) predicted.

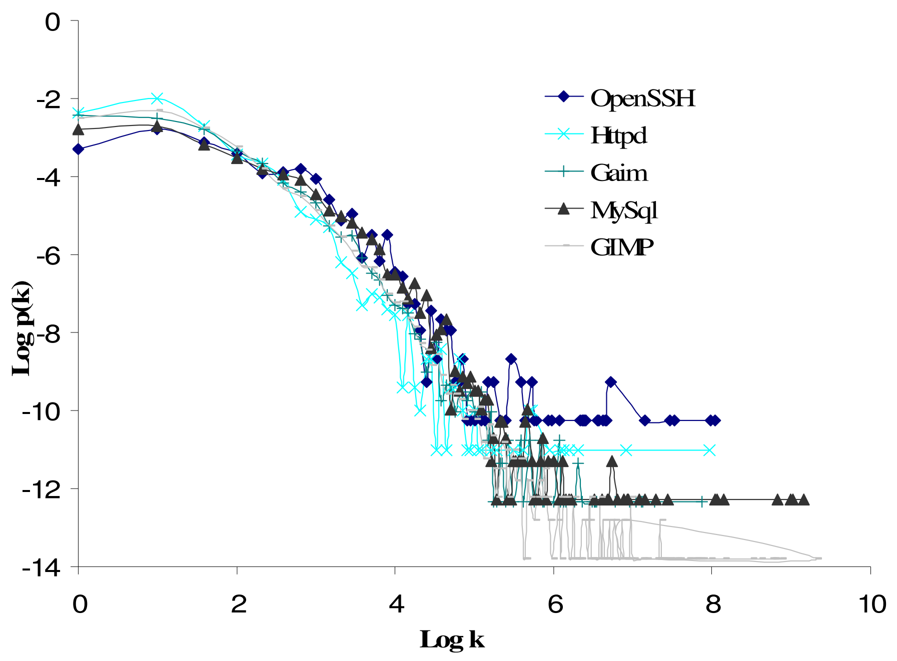


Fig. 2.6 Degree distributions of five software function dependency networks (Logarithm of base two is used throughout this chapter).

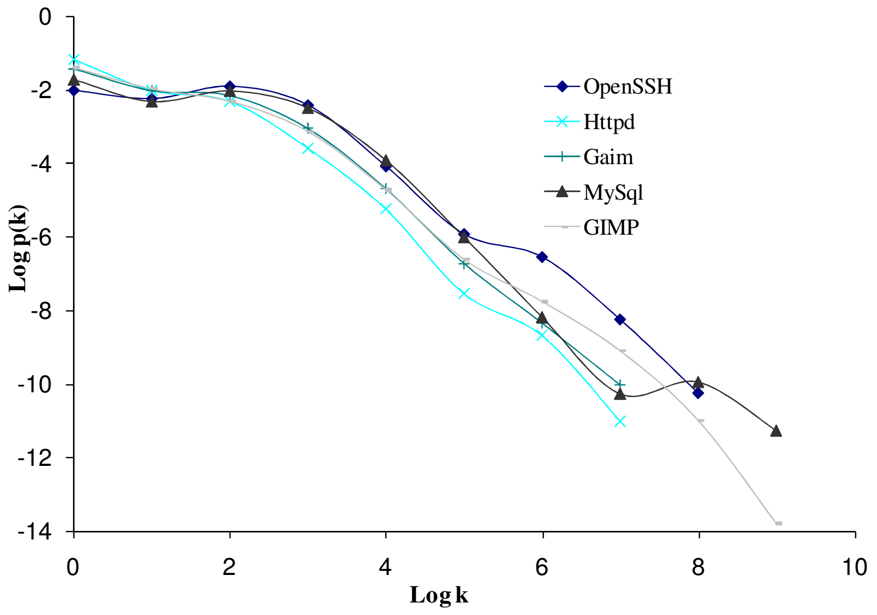


Fig. 2.7 Logarithmically binned degree distributions of five software function dependency networks.

Table 2.4 Linear relationship between $\log p(k)$ and $\log k$

Network	R^2	*Sig.*
OpenSSH	0.906	0.0001
Httpd	0.959	0.0001
Gaim	0.940	0.0001
MySql	0.928	0.0001
GIMP	0.958	0.0001

In summary, our empirical analysis reveals the following features of a software function dependency network: (1) average degree is independent of network size, (2) clustering coefficient, in either of two definitions, is independent of network size, and (3) the network is scale-free.

2.7 Concluding remarks

Inspired by the increasing popularity of open source software systems, and a recent growth of random graph theory research, we model the real world software packages with function dependency networks. We obtain the source code of five C-based open source software packages. Then we construct an abstract network for each one of the software packages with the vertex representing a function and the edge as a function call. In order to keep our research focal point concentrated on the software coupling and software cohesion, we ignore the direction and the number of calls for the function dependency. Thus the function dependency networks that we construct are undirected and un-weighted. Driven by the same incentive, we also ignore the self loops which indicate a function call from a function to itself, namely recursive function calls. Once the function dependency networks are constructed, we leverage several widely adopted topological measures to analyze those networks. Those measures are to reveal some system level coupling and cohesion signals. Our empirical analysis shows three features of a software function dependency network: (1) average degree is independent of network size, (2) clustering coefficient, in either of two definitions, is independent of network size, and (3) the network is scale-free. These findings provide some support to our hypothesis that there are some common topological properties shared by different software packages over a wide range of purpose, size, and complexity.

The results of this work can be used as a starting point to quantitatively analyze software architectural structures. They can also be used to evaluate and compare developed packages in terms of such properties as modularity, intra-module cohesion, and inter-module coupling. The usage of our research does not limit to the open source software systems. Software companies can easily adopt our approach as a tool to examine the architectural structure of their software products since they have the full control of their own software source code. Although C-based software packages are used to perform our analysis, a similar framework can be easily created in order to analyze software systems that are built on other programming languages. For example, we can easily construct a software unit dependency network for a Java-based system by defining vertex as a Java object and edge as a Java object reference. Once the dependency network has been constructed, the similar set of topological measures can be calculated to examine the software system.

Our study opens up several avenues for further research.

First, while we have empirically studied several distinct open-source software packages, investigating networks built from historical versions of the same software package may generate useful insights into the ways software grows and changes over time and may be more appropriate in revealing the true scaling properties of the networks.

Second, while we have modeled software packages as un-directed, un-weighted networks of functions, considering the direction and weight of function calls may reveal additional useful information about software packages.

Third, while we have interpreted the degree distribution as being approximately power-law, future research may conduct finer analysis (e.g., fitting the degree distribution with a stretched-exponential) to reduce the statistical noise on the tail.

Fourth, while all five software packages are considered “popular” or “good quality” software systems, comparing the software systems with different popularities in the same domain would be worth more attention. For example, comparing several Instant Messenger (IM) software systems with different download rates will be likely to reveal more insight of successful software packages.

Fifth, while all five software packages investigated in our empirical study are written in C, a procedural language, it would be interesting to examine software packages written in Object-Oriented (OO) languages, such as C++, C#, and Java, and see how OO features, such as encapsulation, inheritance, and polymorphism, affect the structures of software packages and the processes governing their development.

Network-based analysis of software packages and software engineering efforts provides many intriguing possibilities for future research. Social networking theory is likely to provide guidelines on community discovery when analyzing software authorship networks (Watts, Dodds et al. 2002). Bipartite graph analysis may provide a framework for analyzing the relationships between developers and software components. Further research and improved understanding may lead to the development of useful metrics that provide guidance to engineers and analysts in the development of complex software systems.

CHAPTER 3 ANALYZING OPEN SOURCE SOFTWARE DEVELOPER COLLABORATION NETWORKS

3.1 Introduction

As the size and the complexity of software applications grow over the past few decades, the collaboration among software developers becomes increasingly important (Hahn, Moon et al. 2008; Singh 2011). That a small number of developers finish a software application single-handedly is no longer a common scenario. Software developers have to rely on other developers in order to deliver the software products on time and with satisfying quality. Thus analyzing software developer collaborations is of great interest to both researchers and software engineering practitioners. For example, people are interested to know how many members a functional development team should be composed of in order to have the optimal performance, how balanced the work load of each team member should be, and how many tasks the technical leader of the team should have at any given time. The answers to those questions will provide useful insight for software practitioners in order to achieve a better performance in software development process. However the software developer collaboration is usually complicated for large software systems. Specifically how to define a collaborative relation between two developers is hard to be unanimously agreed. Furthermore in commercial companies the software developer structures are usually confidential and not available for the general public to analyze. The complexity of the developer collaboration and the lack of collaboration practice data result in the fact that few research works have been delivered in that domain (Johnson 2006; Colazo 2010). In this chapter, we aim at analyzing the software developer collaborations by leveraging complex network theory. We choose two open source software applications, MediaWiki and Gentoo, for our empirical analysis. We parse through the bug fixing data for those two applications and define the collaborative relation among developers by considering the co-working experience on a same bug. Since the numbers of bugs and developers are quite large, we perform our analysis by adopting graph theory. Specifically we construct the bug-developer bipartite graphs and the developer-developer collaboration graphs out of the real bug fixing data sets. We then calculate a number of topological measures including the degree distributions for those graphs. Our findings on the topological characteristics can motivate further studies on developer collaborations.

The rest of the chapter is organized as follows. We first introduce some background information in Section 2. In Section 3, we briefly describe the two data sets that we choose as our empirical study subjects. Section 4 explains how we construct the bug-developer bipartite networks and the developer-developer collaboration networks for the two real data sets. In Section 5, we introduce several important topological measures that are widely leveraged in network analysis. Our empirical findings are summarized in Section 6. Finally, we conclude the chapter with a summary of contributions and a discussion of possible future research directions in Section 7.

3.2 Background

We anticipate that the bug fixing processes and developer collaborations of different open source software packages share some common underlying properties. For example, it is intuitively correct that a large number of bugs are fixed by a small number of developers who are more experienced. On the other hand, the majority of developers is not as much experienced, and thus is involved in a less number of bugs that are fixed. That intuition follows the same mechanism of the rich get richer model (Barabasi and Albert 1999) which is the most popular generative model for the scale free property.

In complex network theory, a network is described as scale free if the network's degree distribution follows the power law (Newman, Strogatz et al. 2001). A detailed description of the power law and the scale free property has been introduced in Section 2.5.5.

3.3 An Empirical study with software bug-developer data

From the bug repositories, we have downloaded and crawled the bug and developer data of MediaWiki (https://bugzilla.MediaWiki.org/) and Gentoo (http://bugs.gentoo.org). We chose those two data sets because both software systems are considered successful in its application domain. At this stage of our research, we are more interested to analyze the successful software systems than unsuccessful ones. While each software system may bear its unique software architecture, development process and project management method, we hope to demonstrate some common characteristics in bug-fixing and developer collaboration process.

Both data sets have the same data format. For each bug entry, either data set contains the following six data fields: bug id, developer id (denoted by developer's email address), time stamp, subject, removed content, and added content. Table 3.1 shows an example of the data entry.

Table 3.1 Bug developer data entry example

Bug ID	Developer	Time	Subject	Removed	Added
1	joe@hot.com	2006-11-29 21:45:59	Status		NEW
1	joe@hot.com	2006-11-30 23:55:29	Status	NEW	RESOLVED
2	jane@cool.com	2006-11-10 20:18:32	AssignedTo	jim@Jmail.com	jane@cool.com

In Table 3.1, the primary key is the combination of bug id, developer and time. The bug ID alone can not be used as the primary key of that table. As shown in Table 3.1, a bug, identified by an integer, may have multiple entries in a data set. Each entry indicates a status change of the bug. When the “Added” field is “RESOLVED,” it indicates the bug is fixed. Keep in mind a bug may reappear in the table after it is resolved. The reappearing scenario is common in the software engineering field which indicates the same bug is detected again after that bug is considered fixed.

The combination of bug ID and developer can not serve as the primary key either since there are duplicated bug-developer pairs in either one of the data sets. As shown in Table 3.1, the developer joe@hot.com and bug 1 are associated on two different data entries. The differences between those two entries lie in “Time,” “Removed,” and “Added” fields. Intuitively, the first entry indicates that bug 1 was discovered by the developer joe@hot.com. The second entry which was added about a day later reflects the fact that bug 1 was fixed by the same developer.

In this chapter, we will only consider the associative relations between bugs and developers, and will ignore the context of how those bugs and developers are related. Namely, we are only interested at the first two columns, Bug ID and Developer, and not the last four columns in Table 3.1. If we only consider Bug ID and Developer, the first two rows in Table 3.1 demonstrate that the same pair of bug ID and developer, e.g. joe@hot.com and bug 1, can appear on multiple data entries. In this chapter we refer multiple data entries associating with the same pair of bug and developer the duplicated bug-developer pairs.

In the following sections, we will discuss two different analysis methods depending on how we treat bug-developer duplications. In the first analysis method, we allow and consider the bug-developer duplications. Secondly, we ignore the bug-developer duplications, since our main focus of this research work is whether a specific bug and a specific developer are related than how strongly that pair of entities is related.

Table 3.2 Basic statistics about the two bug-developer data sets

Package	Number of Bugs	Number of Developers	Number of Bug Entries	Number of Unique Bug-Developer Pairs
MediaWiki	16,263	2,646	86,265	35,656
Gentoo	218,387	20,322	1,232,735	580,974

Table 3.2 shows the basic statistics of the two real world bug-developer data sets. The numbers of bugs and developers count unique bugs (identified by an integer) and developers (identified by an email), respectively. The number of bug entries counts the total number of entries in the data sets by considering duplicated bug entries or developer entries. For example, in Table 3.1, the number of bug entries is 3, the number of bugs is 2 (bug 1 and bug 2), and the number of developers is 2 (joe@hot.com and jane@cool.com). The number of unique bug-developer pairs excludes the duplications of bug-developer

pairs. In Table 3.1, the first two rows will count 1 since they are both regarding bug 1 and joe@hot.com. Thus the number of unique bug-developer pairs is 2.

3.4 Network construction

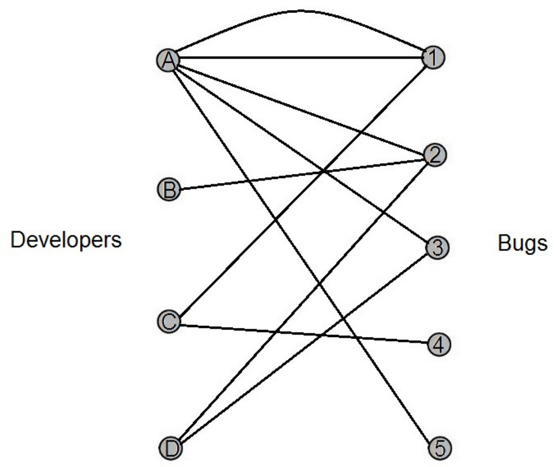
As the sizes of the two bug-developer data sets are very large, we choose a network approach to perform our analysis because network theory barriers the natural strength of analyzing large and complicated systems without loosing the overall system level characteristics.

To start the network analysis, we need to construct a network out of the real world bug-developer data set. We first construct a developer-bug bipartite network. A bipartite network is composed of two disjoint sets of vertices. Every edge connects a vertex in each one of the disjoint sets. There are no edges between any vertices within a set. For our data set, each developer is represented by a vertex. All vertices denoting developers form a set of developer vertices. Similarly we form a set of vertices each of which represents a bug. Since every data entry in the data set indicates a relation between a bug and a developer, naturally that data entry can be represented by an edge connecting a developer vertex and a bug vertex. Since there are no data entries referring two bugs or two developers, it eliminates the possibility of an edge within two developer vertices or two bug vertices. Thus, the developer-bug data set forms a bipartite network by default.

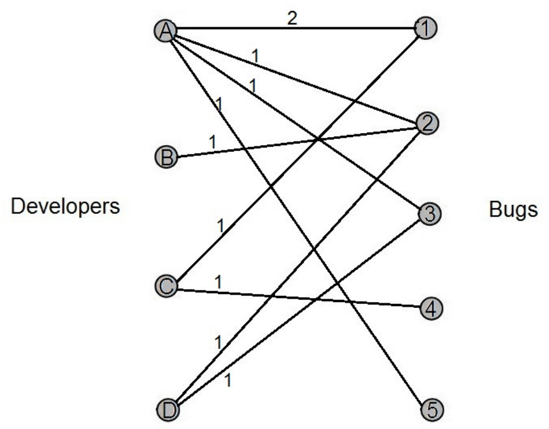
Fig. 3.1(a) illustrates an example of the bipartite network. Specifically there are four developers denoted by four vertices, A, B, C and D. Five bugs are involved whom are represented by five vertices, 1, 2, 3, 4 and 5. Developer vertices and bug vertices form two disjoint sets of vertices. All the edges are between the two disjoint sets and never within a set. For example, an edge B2 indicates a data entry that B changes the status of bug 2.

The bipartite graph is undirected as the direction is meaningless due to the different logic nature of two disjoint sets. The bipartite network could be converted to a weighted network. Note in Fig. 3.1(a) there are two edges between Developer A and Bug 1. Those two edges represent two data entries containing A and 1. This is the same scenario regarding the duplicated bug-developer pairs that we have demonstrated in Table 3.1. If there exists more than one edge connecting two vertices, the network is called a multi network or multi-graph. To ease the analysis of the multi network, we combine all the edges connecting a pair of vertices and assign the number of edges combined to be the weight of that combined edge. Thus the resulting network becomes an undirected, weighted bipartite network as shown in Fig. 3.1(b). This graph represents the scenario that duplicated bug-developer pairs are perceived.

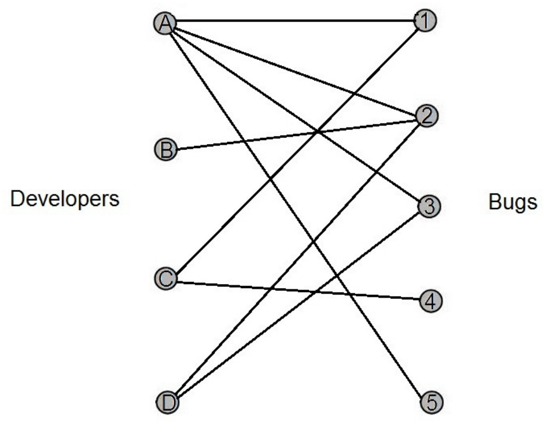
As explained above, we are more interested at whether a specific bug and a specific developer are related than how strongly that pair of entities is related. We then can ignore the duplicated bug-developer pairs observed in Fig. 3.1(a). Namely we can ignore the second edge connecting developer A and bug 1. The resulting graph is shown in Fig. 3.1(c). It is an undirected, un-weighted bipartite network.



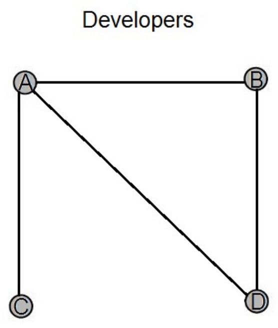
(a)



(b)



(c)



(d)

Fig. 3.1 Bug fixing network construction

(a): Developer-bug bipartite multi-network;

(b): Developer-bug bipartite weighted network;

(c): Developer-bug bipartite un-weighted network;

(d): Developer-developer collaboration un-weighted network

To better understand the collaboration patterns among software developers, we construct a developer-developer collaboration network out of the developer-bug bipartite network. Again we denote a developer with a vertex. Then we define an edge between a pair of vertices to be the collaborative relation between the two developers represented by those two vertices. Specifically, if two developers have worked on the same bug at least once, we would add an edge between those two developer vertices. Fig. 3.1(d) is the developer-developer collaboration network extracted out of the developer-bug bipartite network in Fig. 3.1(c). Note both Fig. 3.1(b) and (c) will yield the same developer-developer collaboration network based on our definition. Our definition only considers whether two developers have worked on the same bug or not. The number of bugs that both developers have worked on together does not have any impact on the resulting developer-developer collaboration network. To keep the network construction logically simple, we choose Fig. 3.1(c) to extract the developer-developer collaboration networks.

As shown in Fig. 3.1(a), both developers A and C have worked on bug 1. Thus there should be an edge connecting developer A and developer C in Fig. 3.1(d). Based on the same logic, the developers A, B and D have worked on bug 2. Thus developers A, B, and D should be connected to each other in the developer-developer collaboration network.

The developer-developer collaboration network we construct is un-directed. At this stage, we do not consider the roles of developers, e.g. who is in charge in the bug-fixing process. Thus the developer-developer collaboration network does not carry the information such like assign-to, which implies that the edges are undirected.

In addition, we ignore the weight in the developer-developer collaboration network. Note in Fig. 3.1(c), the fact that developers A and D have worked on bug 2 adds an edge between A and D in Fig. 3.1(d). Moreover, A and D have also worked on bug 3 together which indicates another edge should be added to connect those two developers. However, in this research work, we are more interested to study whether two developers have collaborated rather than how closely they have collaborated. Therefore, we intentionally ignore the weight of the developer-developer collaboration network (Newman 2004).

Some data edges in the developer-bug network do not play any role in the developer-developer collaboration network. For example, in Fig. 3.1(c), developer A has worked on bug 5, and developer C has worked on bug 4. Since they are the only developers who have worked on those two bugs independently, there is no collaborative work among developers. Therefore those two data entries do not change anything in Fig. 3.1(d).

Lastly, we do not consider self-loops in the developer-developer collaboration network. As shown in Fig. 3.1(a), developer A has two edges connecting bug 1. From the network point of view, the developer A is collaborating with himself which implies a self-loop from vertex A to itself in Fig. 3.1(d). However the scenario that a developer inputs more than one data entry in the bug-developer data set should not lead to the belief that the developer is collaborating with himself. In order to keep our concentration on the subject of developer collaborations, we ignore the self loops in Fig. 3.1(d). That is another reason why we choose Fig. 3.1(c) instead of Fig. 3.1(b) to extract the developer-developer collaboration networks.

In summary, we will analyze three networks for each one of the MediaWiki (Wiki for short) and Gentoo data sets: developer-bug weighted bipartite networks, developer-bug un-weighted bipartite networks, and developer-developer collaboration networks. For each one of the bipartite networks, we will consider two scenarios: bug side and developer side. We will discuss the detail of the two sides of a bipartite network in the following sections.

3.5 Topological metrics

Table 3.3 reuses some notations defined in Table 2.2. Those notations are redefined because, unlike Table 2.2, Table 3.3 defines the symbols that can be used for bipartite networks.

Similar to Table 2.2, the degree of node i, k_i, is the number of edges connected to that vertex even for bipartite networks. As claimed above, we will consider two scenarios for each bipartite network from either the bug side or the developer side.

From the bug side, each vertex denotes a bug, and thus k_i is the number of developers that the bug is associated with. On the other hand, from the developer side, each vertex represents a developer. The value of k_i is the number of bugs that the individual developer has been involved in.

Table 3.3 Symbols of network measures for bipartite networks

Symbol	**Measure**
k_i	Degree of node i, i.e., the number of edges connected to the node
$\bar{k}$	Average degree of all nodes. For bipartite networks, average degree of all nodes in a disjoint vertex set.
$p(k)$	The fraction of nodes in the network that have degree k, or equivalently, the probability that a node chosen uniformly at random has degree k

The average degree $\bar{k}$ is the arithmetic mean of the degree values of all the vertices in the network set. For a bipartite network, the average degree $\bar{k}$ is the arithmetic mean of the degree values of all the vertices in either one of the disjoint set of the network. A developer-bug bipartite network has two average degree values, one from the bug side, the other from the developer side. From the bug side, $\bar{k}$ is the average of degrees of all bugs. Based on the same logic, $\bar{k}$ is the average degree of all developers from the developer side of the bipartite network.

Degree distribution is the distribution of $p(k)$ which is the number of nodes in the network that have degree k. Degree distribution is usually displayed by plotting $p(k)$ against the sorted k values. For a bipartite network, $p(k)$ is calculated separately from either the bug side or the developer side.

3.6 Empirical findings

Leveraging the topological metrics described in the above section, we calculate the topological characteristics for Wiki and Gentoo data sets. As described above, we analyze three networks for each data set: (1) developer-bug bipartite weighted networks, (2) developer-bug bipartite un-weighted networks, and (3) developer-developer collaboration un-weighted networks. For the bipartite networks, we consider two scenarios from either the bug side or the developer side.

By analyzing the above networks, we hope to discover some common properties shared by both data sets. For example, as explained in Section 3.2, a small number of developers are usually involved in a large number of bug fixing processes. We hope that the "rich get richer" mechanism can be supported by the network degree distributions. We also hope to reveal some insights of the bug and developer's relations by comparing the bug side and the developer side of the bipartite networks.

Table 3.4 Bug fixing data degree information

Package	Network	Side (If Bipartite)	Number of Vertices	Number of Edges	Average Degree
Wiki	Weighted Bipartite	Bug	16,263	86,265	5.304
		Developer	2,646	86,265	32.602
	Un-weighted Bipartite	Bug	16,263	35,656	2.192
		Developer	2,646	35,656	13.475
	Dev-Dev	N/A	2,646	101,152	38.228
Gentoo	Weighted Bipartite	Bug	218,387	1,232,735	5.645
		Developer	20,322	1,232,735	60.660
	Un-weighted Bipartite	Bug	218,387	580,974	2.660
		Developer	20,322	580,974	28.588
	Dev-Dev	N/A	20,322	2,360,860	116.172

The numerical results are displayed in Table 3.4. We use Wiki data set as the example to explain the results. Gentoo data set should follow the same logics. In the weighted bipartite network, the number of vertices from the bug side is 16,263. That value indicates there are 16,263 bugs in the data set which matches the results in Table 3.2. The number of edges, 86,265 shows how many developers, duplications acceptable, are associated with those bugs. Note the numbers of edges are the same for the bipartite networks from either the bug side or the developer side. That is understandable if analyzing the examples shown at Fig. 3.1(b) and (c). The average degrees from the bug side and the developer side are 5.304 and 32.602, respectively. The intuition of those two numbers is that, on average, each bug needs about 5 developer's entries to get fixed, and each developer is involved in over 32 bug fixing entries. Note those numbers are for the weighted network meaning duplications of bug-developer pairs are acceptable. If those duplications are ignored, the resulting networks are un-weighted bipartite networks. The average degrees from the bug side and the developer side are 2.192 and 13.475, respectively. That is, out of the 5.304 developer's data entries for each bug, only 2.192 unique developers are involve for that bug. Based on the same logic, each developer has input 32.602 bug data entries, with 13.475 unique bugs. The ratio is 5.304 / 2.192 = 32.602 / 13.475 = 2.419. That means, on average, each developer needs to deal with the same bug 2.419 times before that bug is fixed.

Based on the same logic, we calculate the topological measures for the Gentoo data set. According to the values of the "Number of Vertices" and the "Number of Edges" in Table 3.2, we can easily conclude that Gentoo's bipartite networks are much larger in size than Wiki's networks. After carefully examining the results, we realize the average degrees of those two data sets do not show significant difference from the bug side, namely 5.304 vs. 5.645, and 2.192 vs. 2.660. However, the average degrees of the two data sets are much different from the developer side, namely 32.602 vs. 60.660 and

13.475 vs. 28.588. Gentoo's average degrees from the developer side are about doubled compared with Wiki's for both weighted and un-weighted networks. The intuition behind that interesting finding is even the size, and hence the complexity of the software application increases, each bug's complexity remains about the same. Thus it does not take significantly more time for the developers to fix an individual bug. That intuition leads to the observation that the average degree from the bug side remains about the same regardless the size of the application. On the other hand, as the size of the application increases, each developer needs to be involved in fixing more bugs. Therefore the average degree from the developer side increases as the size of the application increases.

Lastly we examine the developer-developer collaboration networks. As we have explained in the above section, the developer-developer collaboration networks we have constructed are un-weighted, undirected, and simple (no self loops or multi edges) networks. First observation that we can easily recognize is the developer-developer collaboration networks have a lot more edges than the un-weighted bipartite networks, namely 101,152 vs. 35,656 and 2,360,860 vs. 580,974. Note we construct the developer-developer collaboration networks out of the un-weighted bipartite networks, thus we have to use the un-weighted bipartite networks for the comparison instead of the weighted bipartite networks. The explanation of the dramatic increase of the number of edges is based on the so-called "developer cliques (Newman and Girvan 2004)." If a number of developers have worked on the same bug, those developers will form a clique. In a clique, every developer is connected to every other developer within that clique. If a clique is composed of N developers, that clique will yield $N(N-1)/2$ edges in the resulting developer-developer collaboration network.

Our second observation regarding the developer-developer collaboration networks is the average degree increases from 38.228 to 116.172 as the size of the network increases. The insight is as the size and complexity of the software application grows, each developer is likely to be involved in fixing more bugs. The observation is intuitively consistent with software development practice.

Fig. 3.2 to Fig. 3.5 are the degree distributions of the bipartite networks: weighted and un-weighted, from the bug side and from the developer side. In order to compare the differences and the similarities between Wiki and Gentoo data sets, we plot the two data sets on every one of the degree distribution graph.

Fig. 3.2 shows the degree distribution results for the bug-developer weighted bipartite networks from the bug side. The majority middle parts of the both curves are close to straight lines; hence possess semi-scale free properties. Both data sets, Wiki and Gentoo, show noticeable dips at the very first points where the degree value $k=1$. That is to say that the number of bugs that have only one bug data entry is less than the number of bugs that have two, or three data entries. After carefully examining the source data, we realize this property is due to the nature and some special rules that the bug fixing procedure caries. Specifically when a bug is found, a bug data entry with "NEW" in the "Added" field will be input to the data set as shown in the example Table 3.1. After the bug is fixed, another data entry with "RESOLVED" in the "Added" field will be input to the data set. Therefore most bugs have at least two data entries in the data sets which results in a limited number of bugs with only one data entry. Note those two data entries, “NEW” and “RESOLVED,” that an individual bug has been involved in may be entered by one developer. If duplications of bug-developer pairs are ignored, the resulting degree distribution will show a different shape as shown in Fig. 3.3.

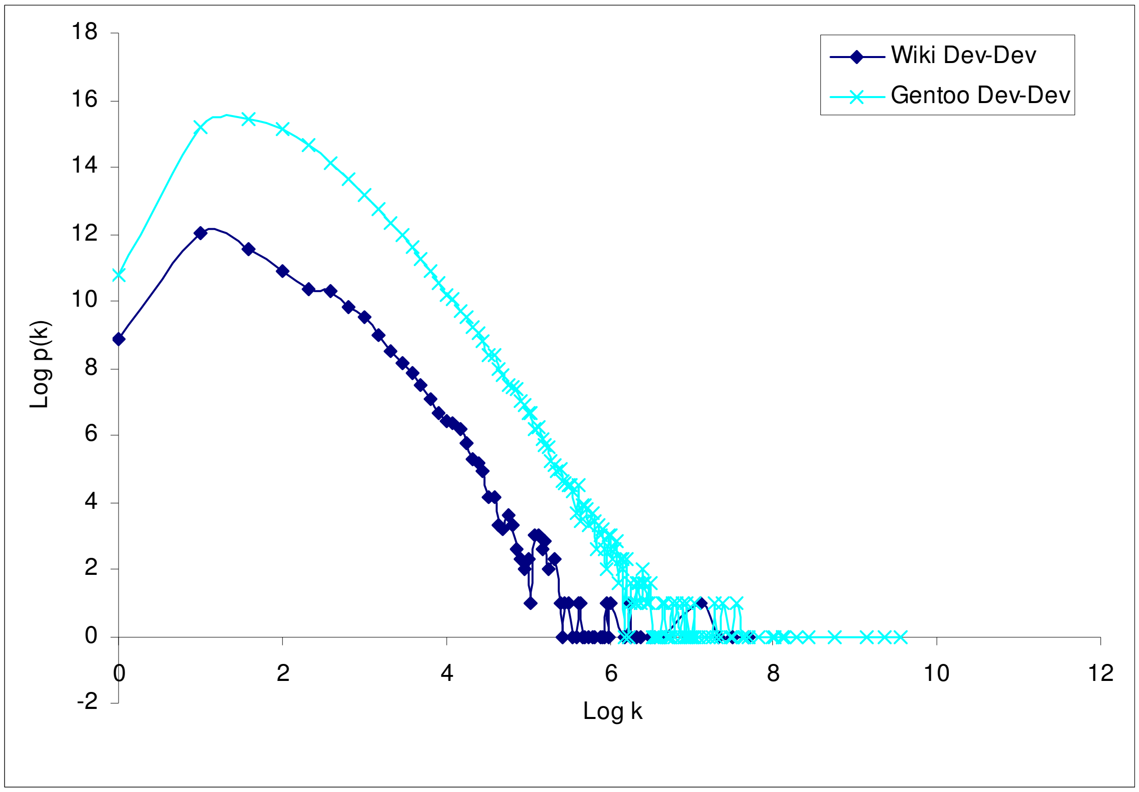


Fig. 3.2 Degree distribution: bug-developer weighted bipartite networks from bug side

Fig. 3.2 also has a long hockey tail for large degree values which indicates that very few bugs with significantly large number of data entries. That is understandable when duplicated bug-developer pairs are allowed.

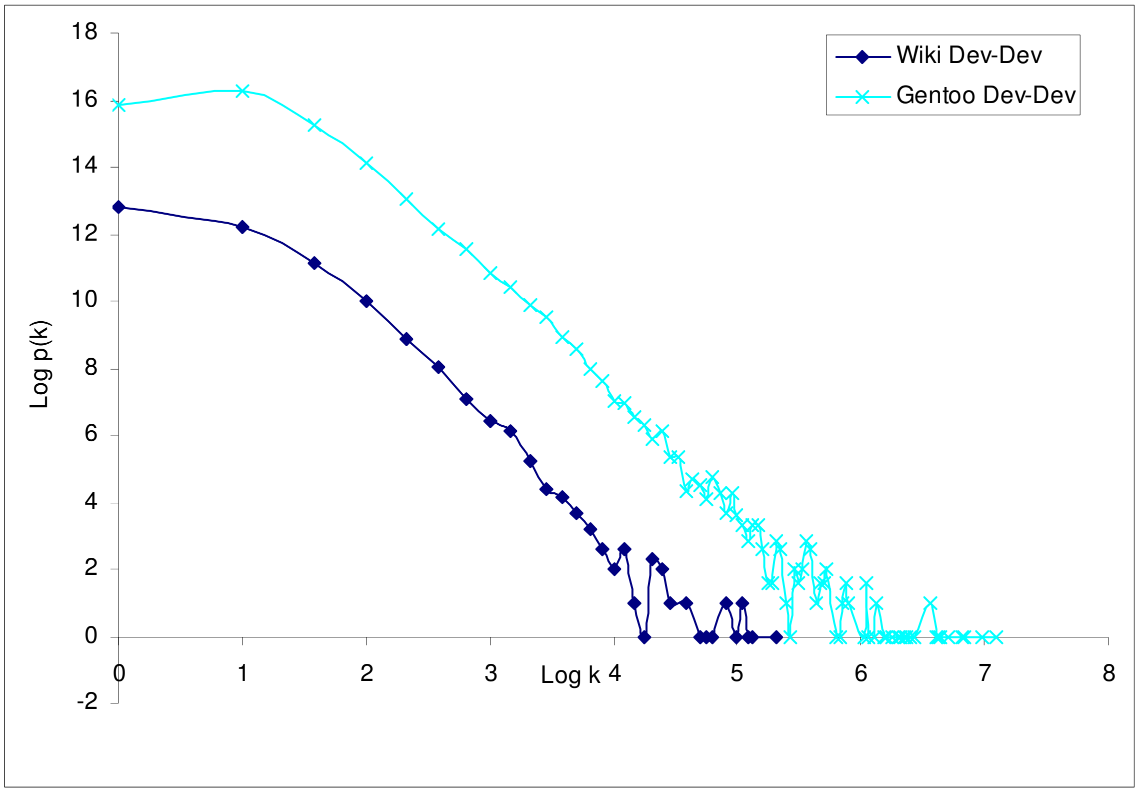


Fig. 3.3 Degree distribution: bug-developer un-weighted bipartite networks from bug side

Comparing Fig. 3.2 and Fig. 3.3 clearly shows the effects of the network weight. Recall the weighted bipartite networks reflect the scenarios that the duplicated bug-developer pairs are allowed. On the other hand, the un-weighted bipartite networks ignore the bug-developer pair duplications. If the bug-developer pairs are ignored, the dips shown in Fig. 3.2 is significantly reduced as shown in Fig. 3.3. As explained in the paragraph above, an individual bug usually has at least two data entries, “NEW” and “RESOLVED.” That is the reason that causes the dips in Fig. 3.2. However if both data entries are entered by the same developer, by the definition of un-weighted bipartite networks, only one edge is considered for that specific bug. Therefore, the number of bugs with only one data entry may not be necessarily limited. That is why the dips are hardly recognizable in Fig. 3.3.

Moreover, because of the same reason, the degree values for bugs with very high degrees are also reduced. Intuitively speaking, a complex bug may be associated with many data entries but many of those data entries may be entered by a limited set of developers. Thus the hockey tail in Fig. 3.3 is dramatically shortened compared with Fig. 3.2. Therefore ignoring the duplications of bug-developer pairs makes the degree distributions more scale free.

Fig. 3.4 and Fig. 3.5 are to show the bipartite networks degree distributions from the developer side. Those two graphs describe developers' involvements with bugs. Both graphs show very strong scale free trends. For weighted networks, the head of the curves have small dips for both data sets. On the other hand, Fig. 3.5, the un-weighted networks do not have dips for both Wiki and Gentoo data sets. The difference on the dips between those two figures can be explained easily by the duplicated bug-developer pairs following the similar logics as shown in the paragraph above.

Unlike the graphs from the bug side (especially Fig. 3.3), both Fig. 3.4 and Fig. 3.5 have long hockey tails. The long hockey tails imply that there exist a small set of developers who are heavily involved in fixing a large number of bugs. Those developers can be considered as the technical leads in the bug fixing process.

From Fig. 3.2 to Fig. 3.5, we observe a common characteristic that occurs in all four graphs: both Wiki and Gentoo networks have similar shapes of degree distributions. Furthermore Gentoo's graphs are always above Wiki's graphs, and they are always in parallel. The two applications are with quite different sizes and in different domains. The similar degree distributions shared between them seem to suggest that there exist common underlying characteristics in the bug fixing process shared by open source software systems across application size and domain.

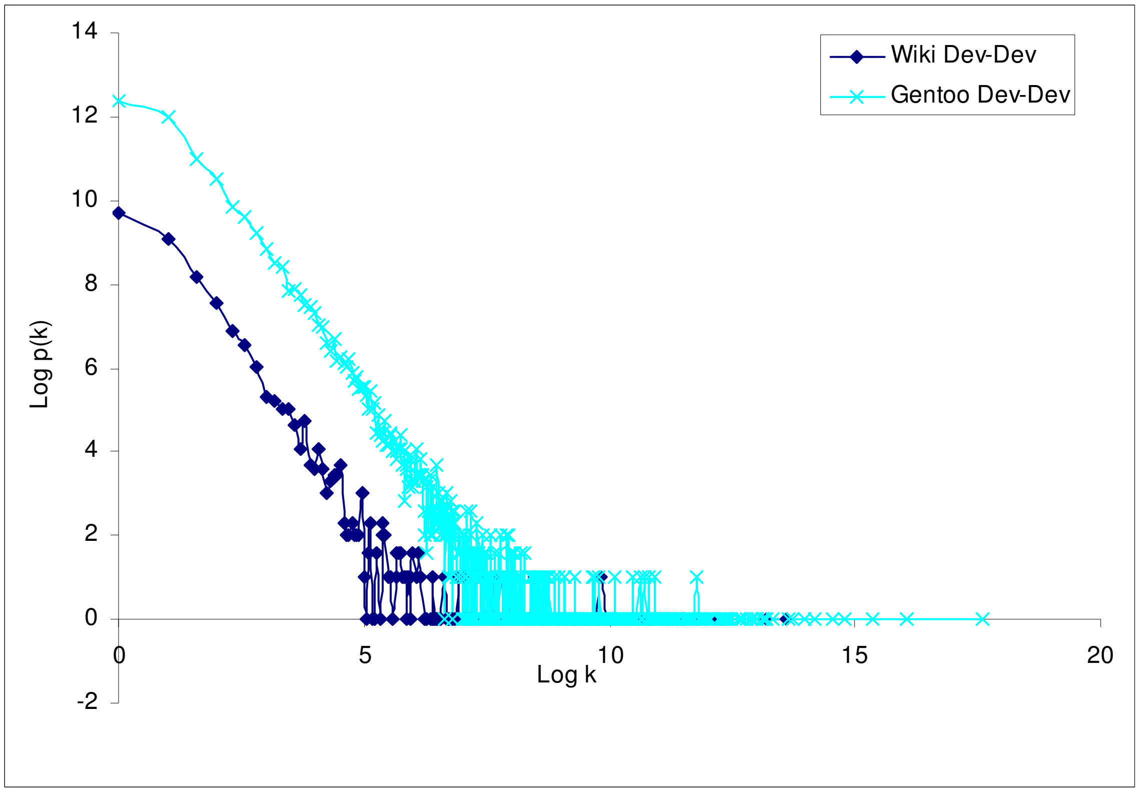


Fig. 3.4 Degree distribution: bug-developer weighted bipartite networks from developer side

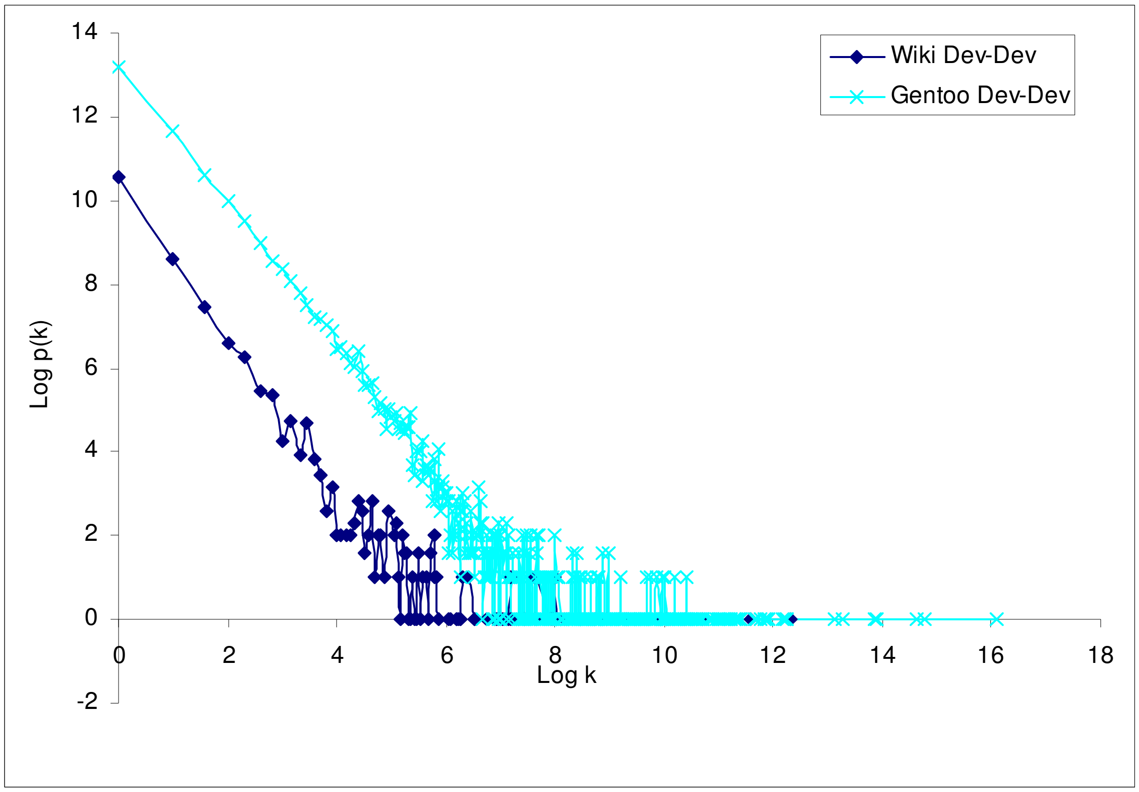


Fig. 3.5 Degree distribution: bug-developer un-weighted bipartite networks from developer side

Fig. 3.6 shows the degree distributions of developer-developer collaboration un-weighted networks. Again, both the Wiki and Gentoo data sets exhibit similar degree distribution curves. The major parts of the two curves form straight lines which indicate both distributions are scale free. The scale free property implies that the more popular a developer is, the more likely other developers are to collaborate with that individual developer. The scale free property makes intuitive sense in software engineering.

Both curves in Fig. 3.6 have heavy hockey tails that imply that there exists a small set of developers with extremely high collaborative connections with other developers. Those developers are usually leaders in the bug fixing process which is consistent with our observations in the previous paragraphs. At the beginning of the two curves, small dips are observed for both data sets. The dips are corresponding to the degree value $k = 1$. The dips suggest that the number of developers that have only one connection to another developer is limited. The observation can be intuitively explained by the fact that developers are usually working with at least two other developers.

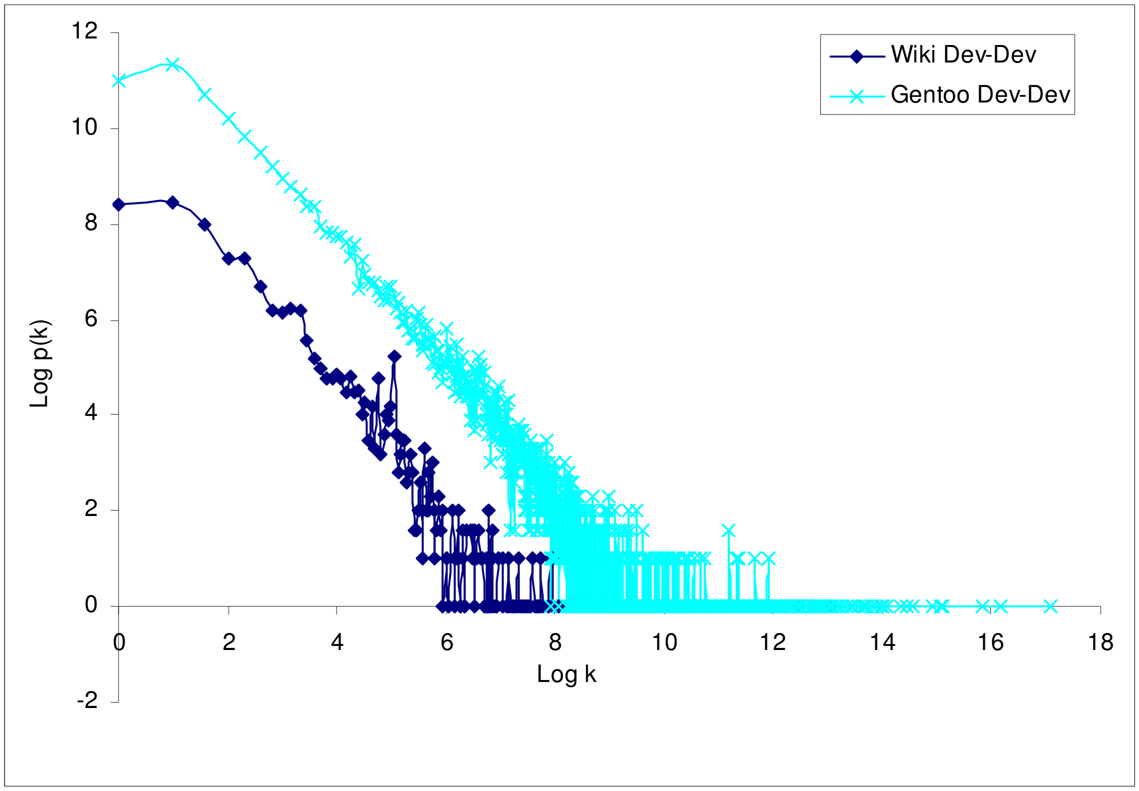


Fig. 3.6 Degree distribution: developer-developer collaboration un-weighted networks

3.7 Conclusions and future study

The software developer collaboration is an important factor in software development process. The quality of the developer collaboration has a direct impact on the overall software development performance, and thus to the quality of the final software product. In our study, we propose a framework to analyze the software developer's collaborative relations by leveraging network theory. Specifically we choose the bug fixing data sets for two real software applications, MediaWiki and Gentoo, to perform the empirical analysis. Two developers are defined as collaborated if they have worked on at least one bug together. Based on that definition, we construct the bug-developer bipartite networks and the developer-developer collaboration networks for those two bug data sets. We then calculate several topological measures for those constructed networks. Our empirical results indicate that all networks possess scale free properties. For the bipartite networks, we find and explain the average degree from the developer side is related to the size of the open source software package. On the other hand, the average degree from the bug side is unrelated to the software package size. Our findings and framework can be leveraged as the infrastructure for later research in the developer collaboration field.

Some limitations of our empirical research lead to some interesting follow-ups in the near future. First of all, the developer-developer collaboration networks that we constructed are un-directed. We do not consider the roles that each developer plays in fixing an individual bug. In the next step of our research, we may consider the heterogeneous roles that developers play for each bug. For example, a developer may assign a bug fixing task to another developer which implies the first developer is the leader of the team and second is the follower. The detailed relation between two developers may be revealed by the timestamp of the bug data entry or the keywords in "Added" and "Removed" fields as shown in Table 3.1. With that detailed relation information, we can construct directed developer-developer collaboration networks where the direction indicates the leadership between those two developers. The directed nature of relations among developers differentiates the bug-fixing developer collaboration from many other collaborative networks, such as scientific collaborations (Newman 2001) and movie actor collaborations (Watts and Strogatz 1998), where relations between two connected people, e.g. scientific scholars or movie actors, do not carry strong directional information.

Second possible follow-up research may extend the current research by considering the weight of the developer-developer collaboration networks. As explained in previous sections, we ignore the weight of the collaboration networks which means we consider whether two developers have collaborated rather than how closely those two developers have collaborated. In reality if two developers have worked together on multiple bugs, the intensive collaborative relation between the two developers should be valued. In the future study, we may extend the network construction by incorporating developer-developer collaboration weight. We hope the more detailed consideration in the network construction will result in more interesting findings to researchers and software practitioners.

CHAPTER 4 COMPARISON OF TWO CLUSTERING COEFFICIENT DEFINITIONS

4.1 Introduction

Complex network analysis has gained more and more attention recently. It is widely adopted in analyzing real world complex systems including software systems (Albert and Barabasi 2002; Newman 2003; Potanin, Noble et al. 2005). One of the most useful topological measures in the network analysis is the clustering coefficient. Clustering coefficient measures the extent to which being a neighbor is a transitive property. Clustering coefficient has two commonly used definitions (Watts and Strogatz 1998; Newman 2001; Newman, Watts et al. 2002; Newman 2003). The first definition is proposed by Watts and Strogatz, and the second is by Newman. Those two definitions share some common considerations and, at the same time, have their own unique angles in measuring the clustering circumstances. However, very few attempts have been proposed in order to compare those two definitions. Most researchers just adopt one of them based on their own needs when conduct their research work. This chapter intends to fill the gap by proposing an analytical comparison between the two clustering coefficient definitions invented by Watts-Strogatz and Newman, respectively. The comparison is performed by analytical derivations showing the mathematical relations between those two definitions. Some numeric properties of the two definitions are presented. Lastly a simulated network example is leveraged to show the impact factors of the clustering coefficient values based on those two definitions. Our research can be potentially leveraged by software practitioners to analyze software product structure, development processes, and engineer collaborations using complex networks.

The rest of the chapter is organized as follows. We first introduce some background information in Section 2. In Section 3, we briefly explain the topological metrics that we will be using in this chapter. Section 4 presents the numeric analysis and the derived formulas of the two clustering coefficient definitions. In Section 5, we introduce several numeric properties of those two definitions including the impact factors of the clustering values for each definition. Finally, we conclude the chapter with a summary of contributions in Section 7.

4.2 Background

As an important topological measure in graph theory (Tutte 1984), clustering coefficient measures the extent to which being a neighbor is a transitive property (Eggemann and Noble 2011). Clustering coefficient has two commonly used definitions (Watts and Strogatz 1998; Newman 2003). We reuse the clustering coefficient definitions and descriptions in Section 2.5.6 to perform our analysis. Equations 2.1 to 2.4 will be reused as the starting point to present our derivations.

4.3 Topological metrics introduction

We reuse the notations in Table 2.2 to perform the analysis for this chapter. Moreover, we present some additional symbols in Table 4.1 that we will be using for the rest of this chapter. As explained above, the networks that I analyze are un-directed and un-weighted networks. The networks are simple networks meaning no self loops or multiple edges connecting two vertices are allowed.

N_0 is the number of the vertices in the network which is commonly referred to as the size of the network. Unlike in Table 2.2, in Table 4.1 N is the number of vertices whose degrees are greater than 1. N is useful when we calculate the clustering coefficient. On the other hand, N_0 is not directly usable for calculating the clustering coefficient because the vertices with degree 1 do not have any effect on the clustering coefficient value. The variables k_i and a_i have the same meanings as in the previous sections. In order to study the detailed information of an individual vertex' connections, we define two additional variables, $T_a^{(i)}$ and $T_p^{(i)}$. $T_a^{(i)}$ is the number of triangles around vertex i. A triangle is a group of three vertices that connect to each other. $T_p^{(i)}$ is the number of triples centered at vertex i. A triple centered at vertex i is a group of three vertices that vertex i is connected to the other two vertices. T_a and T_p are aggregated variables that count the total numbers of triangles and triples, respectively, in the entire network. $C_{WS}^{(i)}$ is the clustering coefficient in the Watts-Strogatz definition for vertex i that is connected to at least two other vertices. Finally C_{WS} and C_{NW}

are the dependent variables that are the clustering coefficient in the Watts-Strogatz definition and Newman definition, respectively, for the entire network.

Table 4.1 Symbols of network measures

Symbol	**Measure**
N_0	Number of vertices in the network, referred to as the network size
N	Number of vertices in the network whose degrees are greater than 1
k_i	Degree of vertex i, i.e., the number of edges connected to the vertex
a_i	Number of edges among the neighbors of vertex i
$T_a^{(i)}$	Number of triangles around vertex i. A triangle is a group of three vertices that connect to each other.
$T_p^{(i)}$	Number of triples centered at vertex i. A triple means a single vertex connected to two other vertices.
T_a	Total number of triangles in the network
T_p	Total number of triples in the network
$C_{WS}^{(i)}$	Clustering coefficient in the Watts-Strogatz definition for vertex i with the degree value greater than 1
C_{WS}	Clustering coefficient in the Watts-Strogatz definition for the network
C_{NW}	Clustering coefficient in the Newman definition for the network

4.4 Numeric analysis

Using the symbols that we defined in Table 4.1, we present the formulas to calculate C_{WS} and C_{NW}. Based on equations 2.1 to 2.4, we present C_{WS} and C_{NW}'s calculations in the following three equations.

$$C_{WS}^{(i)} = \frac{a_i}{k_i(k_i - 1)/2} \tag{4.1}$$

$$C_{WS} = \frac{1}{N}\sum_{i=1}^{N} C_{WS}^{(i)} \tag{4.2}$$

$$C_{NW} = \frac{3 \times T_a}{T_p} \tag{4.3}$$

Based on the definition, a_i is the number of edges among the neighbors of vertex i. Since every neighbor is connected to vertex i by definition, every edge among vertex i's neighbors corresponds to a triangle around vertex i. On the other hand, every triangle around vertex i must correspond to an edge connecting a pair of vertex i's neighbors. Thus,

$$T_a^{(i)} = a_i \tag{4.4}$$

$T_p^{(i)}$ is the number of triples centered at vertex i. A triple centered at vertex i is a group of three vertices that vertex i is connected to the other two vertices. Every triple centered at vertex i corresponds to an unordered pair of vertex i's neighbors. Thus the total number of triples centered at vertex i is the total number of different combinations of vertex i's un-ordered neighbors which is $k_i(k_i - 1)/2$.

$$T_p^{(i)} = k_i(k_i - 1)/2 \tag{4.5}$$

Implanting equations 4.4 and 4.5 to equation 4.1 leads to a new formula to calculate $C_{WS}^{(i)}$ and then C_{WS}.

$$C_{WS}^{(i)} = \frac{T_a^{(i)}}{T_p^{(i)}} \tag{4.6}$$

$$C_{WS} = \frac{1}{N}\sum_{i=1}^{N}\frac{T_a^{(i)}}{T_p^{(i)}} \tag{4.7}$$

We now consider the formula for C_{NW}. The following two equations are quite straightforward. T_a and T_p are aggregated variables of $T_a^{(i)}$ and $T_p^{(i)}$, respectively. Since every triangle is counted three times when considering each vertex i, the total number of triangles in the network, T_a, should be the summation of $T_a^{(i)}$ divided by 3. Moreover $T_a^{(i)} = T_p^{(i)} = 0$, if $k_i \le 1, \forall i = 1,..N_0$. Thus,

$$T_a = \frac{1}{3}\sum_{i=1}^{N_0} T_a^{(i)} = \frac{1}{3}\sum_{i=1}^{N} T_a^{(i)} \tag{4.8}$$

$$T_p = \sum_{i=1}^{N_0} T_p^{(i)} = \sum_{i=1}^{N} T_p^{(i)} \tag{4.9}$$

We then implant equations 4.8 and 4.9 to equation 4.3, and obtain the formula for C_{NW} as shown in equation 4.10.

$$C_{NW} = \frac{\sum_{i=1}^{N} T_a^{(i)}}{\sum_{i=1}^{N} T_p^{(i)}} \tag{4.10}$$

We will leverage equations 4.7 and 4.10 to compare the two clustering coefficient definitions from now on.

Starting from equation 4.7, $C_{WS} = \frac{1}{N}\sum_{i=1}^{N}\frac{T_a^{(i)}}{T_p^{(i)}} = \text{Mean of } \frac{T_a^{(i)}}{T_p^{(i)}}$. Starting from equation 4.10, $C_{NW} = \frac{\frac{1}{N}\sum_{i=1}^{N} T_a^{(i)}}{\frac{1}{N}\sum_{i=1}^{N} T_p^{(i)}} = \frac{\text{Mean of } T_a^{(i)}}{\text{Mean of } T_p^{(i)}}$. Thus we can say C_{WS} is the mean of the raio $\frac{T_a^{(i)}}{T_p^{(i)}}$, and C_{NW} is the raio of the mean of $T_a^{(i)}$ and the mean of $T_p^{(i)}$.

4.5 Numeric properties

Lower bound. Since $T_a^{(i)} \geq 0, \forall i = 1,2,...N$, then both C_{WS} and C_{NW} are non-negative. Thus the minimum of the values for both C_{WS} and C_{NW} variables may be zero. The minimum is zero if and only if $T_a^{(i)} = 0, \forall i = 1,2,...N$. A formal description is listed below.

$$C_{WS} \geq 0$$

$$C_{NW} \geq 0$$

$$C_{WS} = C_{NW} = 0 \Leftrightarrow T_a^{(i)} = 0, \forall i = 1,2,...N$$

In order to satisfy $T_a^{(i)} \geq 0, \forall i = 1,2,...N$, the network can be a tree where no cycle exists, or a cycle with more than 3 vertices, etc. Fig. 4.1 shows an example for a tree with 8 vertices and a cycle with 5 vertices, respectively. In conclusion the lower bound of C_{WS} and C_{NW} is met when there does not exist three vertices that are connected to each other in the network.

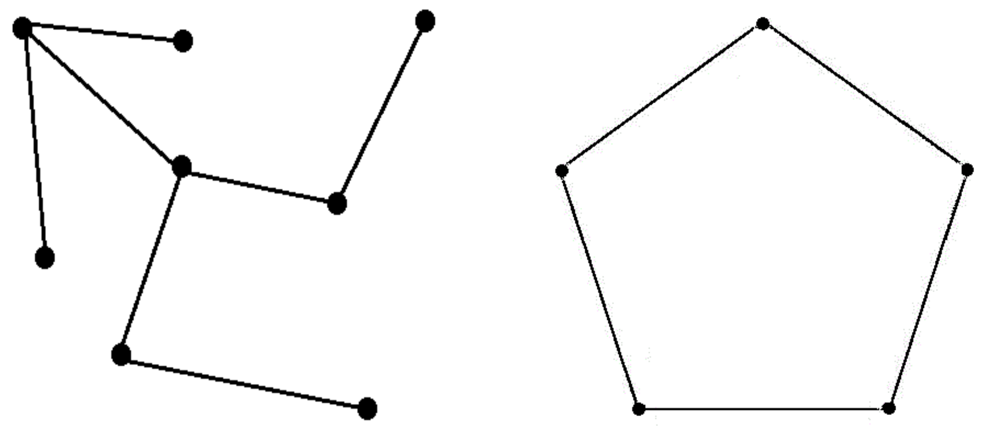


Fig. 4.1 Examples of both C_{WS} and C_{NW} are 0

(a): Tree with 8 vertices; (b): Cycle with 5 vertices.

Upper bound. Since $T_a^{(i)} \le T_p^{(i)}, \forall i = 1,2,...N$, then both C_{WS} and C_{NW} are less than or equal to 1. Thus the maximum of the values for both C_{WS} and C_{NW} variables may be 1. The maximum is 1 if and only if $T_a^{(i)} = T_p^{(i)}, \forall i = 1,2,...N$. A formal description is listed below.

$$C_{WS} \le 1$$

$$C_{NW} \le 1$$

$$C_{WS} = C_{NW} = 1 \Leftrightarrow T_a^{(i)} = T_p^{(i)}, \forall i = 1,2,...N$$

In order to satisfy $T_a^{(i)} = T_p^{(i)}, \forall i = 1,2,...N$, the network has to be a complete graph where every vertex is connected to every other vertices. Thus C_{WS} and C_{NW} values meet the upper bound only when the network is a complete graph. Fig. 4.2 shows two complete graph examples with 5 vertices and with 7 vertices, respectively.

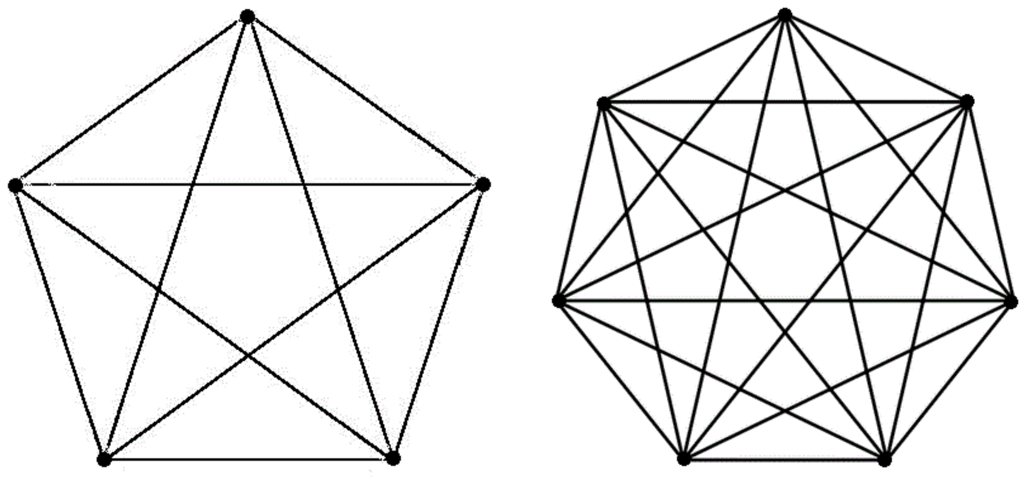


Fig. 4.2 Examples of both C_{WS} and C_{NW} are 1

(a): Complete graph with 5 vertices; (b): Complete graph with 7 vertices.

Equality. We are interested to explore the conditions where $C_{WS} = C_{NW}$. If $T_p^{(i)} = T_p^{(j)}, \forall i, j = 1, 2, \ldots N$, then

$$C_{WS} = \frac{1}{N}\sum_{i=1}^{N}\frac{T_a^{(i)}}{T_p^{(i)}} = \frac{1}{N \cdot T_p^{(1)}}\sum_{i=1}^{N} T_a^{(i)}$$

$$C_{NW} = \frac{\sum_{i=1}^{N} T_a^{(i)}}{\sum_{i=1}^{N} T_p^{(i)}} = \frac{\sum_{i=1}^{N} T_a^{(i)}}{\sum_{i=1}^{N} T_p^{(1)}} = \frac{1}{N \cdot T_p^{(1)}}\sum_{i=1}^{N} T_a^{(i)} = C_{WS}.$$

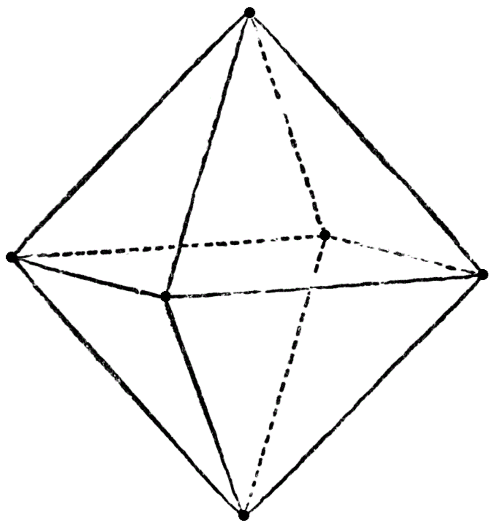
Since $T_p^{(i)} = k_i(k_i - 1)/2$, thus $T_p^{(i)} = T_p^{(j)}, \forall i, j = 1, 2, \ldots N$ is equivalent to $k_i = k_j, \forall i, j = 1, 2, \ldots N$. Therefore,

$$C_{WS} = C_{NW} \Leftarrow k_i = k_j, \forall i, j = 1, 2, \ldots N$$

Note $k_i = k_j, \forall i, j = 1, 2, \ldots N$ is the sufficient condition but not a necessary condition. That is to say, if the degrees of all vertices in a network are the same, then $C_{WS} = C_{NW}$.

Fig. 4.3 shows two graph examples that all vertices in the graph have the same degrees. Fig. 4.3(a) is an octahedron with 6 vertices. Each vertex has a degree 4 and each vertex has 4 triangles around it. Thus $C_{WS} = C_{NW} = \dfrac{4}{4(4-1)/2} = 2/3$.

Fig. 4.3(b) is a cube with 8 vertices, each of which has a degree 3. Note each vertex has 0 triangle around it which results in $C_{WS} = C_{NW} = 0$. This example not only exhibits the equality between C_{WS} and C_{NW}, but also fits in the lower bound scenario as well.


(a)

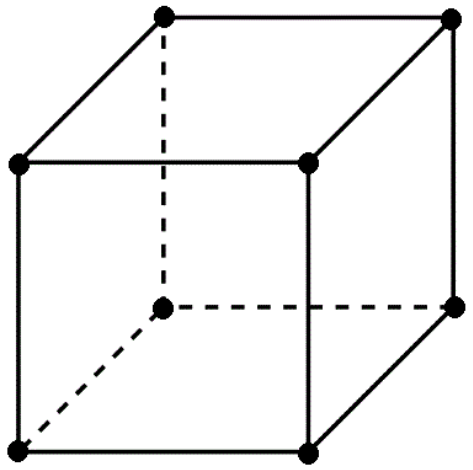

(b)

Fig. 4.3 Examples of $C_{WS} = C_{NW}$

(a): Octahedron; (b): Cube.

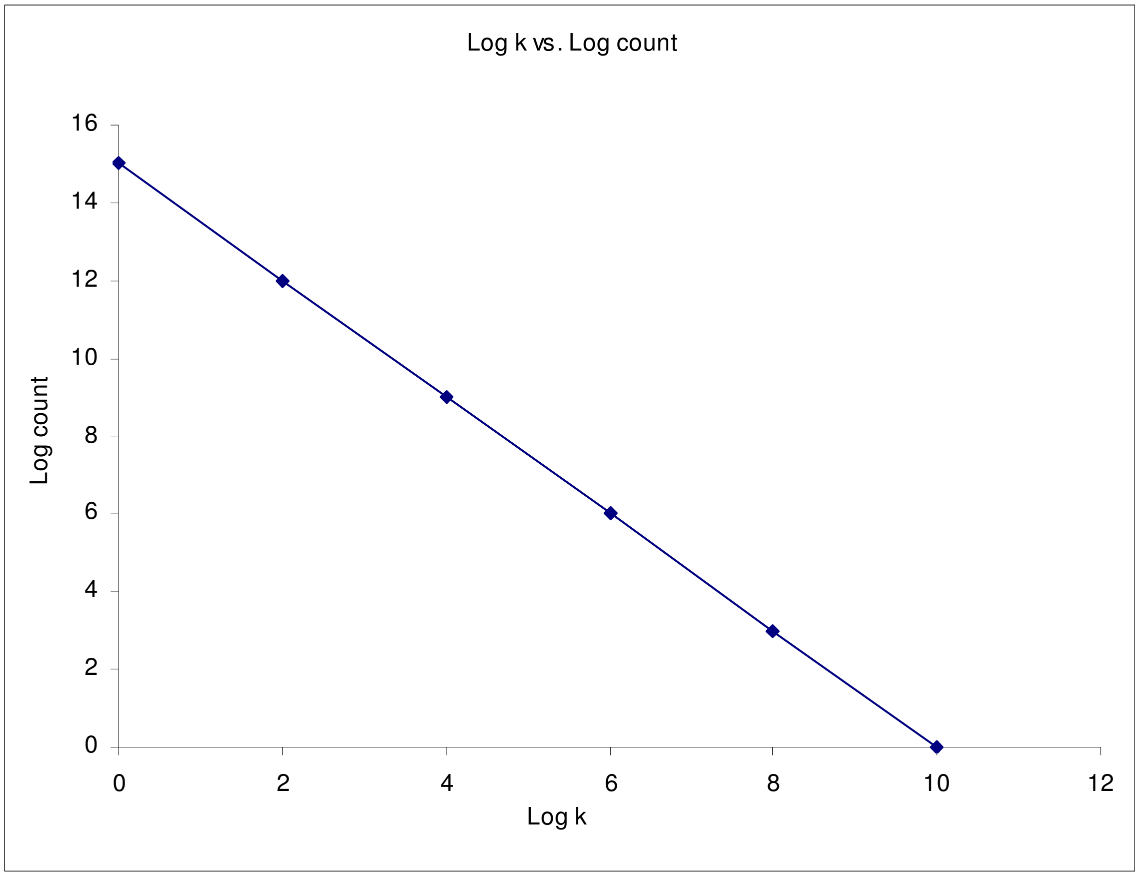


Fig. 4.4 Log degree k vs. log count p(k)

Impact of vertices with large degrees. Recent research findings show that lots of real world complex networks possess the scale-free properties (Newman and Watts 1999; Albert and Barabasi 2000; Albert and Barabasi 2000; Goh, Kahng et al. 2001; Goh, Kahng et al. 2001; Cohen, Ben-Avraham et al. 2002; Goh, Oh et al. 2002; Schwartz, Cohen et al. 2002; Cohen and Havlin 2003; Goh, Lee et al. 2003; Kim, Goh et al. 2003; Vazquez, Boguna et al. 2003) whose degree distributions follow the power law. Specifically the logarithmic values of degrees and the logarithmic values of the number of vertices with the same degrees form a decreasing straight line. Intuitively speaking, there are very few vertices with very large degrees, and most vertices have low degrees. Fig. 4.4 presents a simulated example of the scale-free network.

We leverage the above simulated network to further compare the two clustering coefficient definitions by Watts-Strogatz and Newman. Table 4.2 lists some calculated results derived from the degree distribution. The values of Log k and Log Count form a decreasing straight line that is presented in Fig. 4.4. From the logarithmic values, we can compute k and count values. The last column, k(k-1)/2 * count, will be used later to analyze C_{NW}.

Table 4.2 Simulated degree distribution

Log k	Log Count	k	Count	k(k-1)/2 * Count
0	15	1	32,768	0
2	12	4	4,096	24,576
4	9	16	512	61,440
6	6	64	64	129,024
8	3	256	8	261,120
10	0	1,024	1	523,776

From the previous section, we know $C_{WS} = \frac{1}{N}\sum_{i=1}^{N}\frac{T_a^{(i)}}{T_p^{(i)}} = \text{Mean of } \frac{T_a^{(i)}}{T_p^{(i)}}$. The value of C_{WS} is an average of ratio $\frac{T_a^{(i)}}{T_p^{(i)}}$. And $0 \le \frac{T_a^{(i)}}{T_p^{(i)}} \le 1, \forall i = 1,..N$, the values of $\frac{T_a^{(i)}}{T_p^{(i)}}$ are within a limited range from 0 to 1. If we assume the vertices with the same degrees have the similar a_i values, then the value of C_{WS} is largely depends on the majority of vertices who have the same degree values. Specifically, there are 32,768 vertices with the same degree 1. Those vertices do not have any impact on the clustering coefficient based on our definition, so that we can simply ignore those vertices. There are 4,096 vertices with degree 4. Those 4,096 vertices will yield a dominating factor to the overall value of C_{WS} since C_{WS} is the average of the 4,681 vertices whose degrees are greater than 1. On the other hand, although the one vertex with the degree 1024 is the most "popular" vertex of the network, its $C_{WS}^{(i)}$ only accounts for 1/4681 to the overall C_{WS} value. The impact of the popular vertices is smothered by that of the un-popular vertices which are dominating by the vertex count.

However C_{NW} is totally different from C_{WS} in that the few popular vertices are the dominating factors to the final value of C_{NW}. Equation 4.10 states $C_{NW} = \frac{\sum_{i=1}^{N} T_a^{(i)}}{\sum_{i=1}^{N} T_p^{(i)}}$. Let us consider $T_p^{(i)}$ whose value is $k_i(k_i - 1)/2$ per equation 4.5. To analyze the impact of the popular and un-popular vertices to the mean of $T_p^{(i)}$, we aggregate the impact of $T_p^{(i)}$ for vertices who have the same degree values. The last column in Table

4.2 indicates which set of vertices has the dominating factor. Although the un-popular vertices, with degree 4, have the count advantage (count equals 4,096), the overall summation of $T_p^{(i)}$ for the un-popular vertices is only 24,576. On the other hand, the one popular vertex with degree 1,024 has a huge impact since its $k_i(k_i - 1)/2$ value is 523,776. In the overall $\sum_{i=1}^{N} T_p^{(i)}$ value, the one popular vertex has a dominating factor, and the 4,096 unpopular vertices do not play an important role. Fig. 4.5 plots the degree k versus k(k-1)/2 * count.

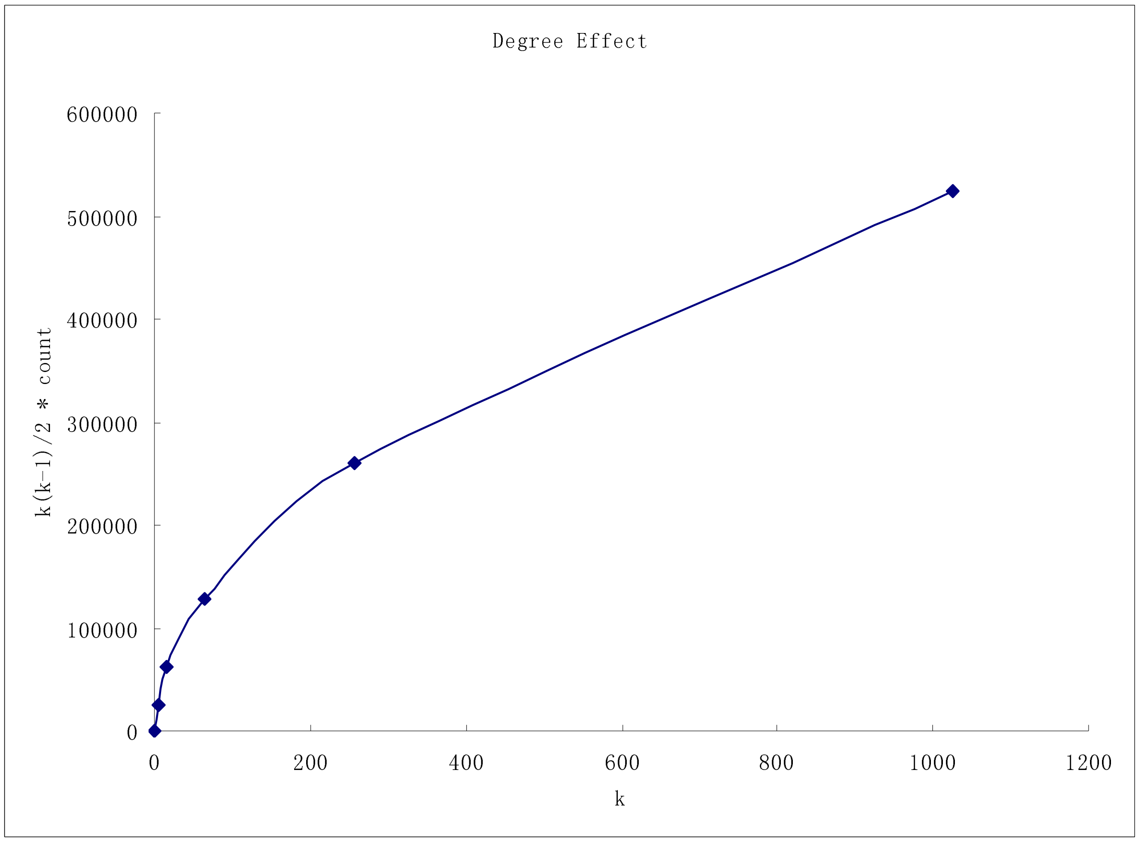


Fig. 4.5 Degree effect: degree k vs. k(k-1)/2 * count

In conclusion the two clustering coefficient definitions, C_{WS} and C_{NW}, has different impact factors. The value of C_{WS} is dominated by the un-popular vertices which have low degree values, and usually the vertex count advantage. The few popular vertices with extremely high degrees do not have much impact on C_{WS}. On the contrary, those few popular vertices play the most important role in calculating C_{NW}. The low-degree vertices are not as important as the few extremely popular vertices.

4.6 Conclusions

Network analysis becomes an important method to study complex systems, and the clustering coefficient remains one of the most useful measures in examining network characteristics. This chapter aims to provide an analytical comparison between two widely adopted clustering coefficient definitions, C_{WS} and C_{NW}, proposed by Watts-Strogatz and Newman, respectively. Mathematical derivations are presented to compare the similarities and the differences between those two definitions. Our findings show that the two definitions both depend on $T_a^{(i)}$ and $T_p^{(i)}$, the number of triangles and triples, respectively, around vertex i. The difference between those two definitions lies in that C_{WS} is the mean of the ratio $T_a^{(i)}$ and $T_p^{(i)}$, and C_{NW} is the ratio of the two means of $T_a^{(i)}$ and $T_p^{(i)}$. We also examine the lower bounds and upper bounds of those two definitions, and the conditions to meet those extreme bounds. Our further analysis shows the impact factors of C_{WS} and C_{NW} values. Using a simulated network which is scale-free, we find that the extremely popular vertices have little impact on C_{WS} due to the limited number of those popular vertices. Whereas those popular vertices are the dominating factors in determining the value of C_{NW}.

Our research findings show detailed properties of the two clustering coefficient definitions, C_{WS} and C_{NW}. It provides researchers more insights when conducting network analysis research. Our findings provide the guidelines on which clustering coefficient definition should be used when analyzing a network. Moreover our results give researchers usable hints when a random network model is needed to explain the topological measures found in real world complex systems. Specifically in software engineering, software practitioners can leverage our research results to analyze complex software products, engineer collaborations, and product development processes if complex networks are chosen to conduct the analysis.

CHAPTER 5 MODELING DEVELOPMENT PROCESS OF COMPLEX SOFTWARE PRODUCTS

5.1 Introduction

As the development expenses of software products increases dramatically over the years, more and more software engineering practitioners are motivated to discover the optimal or semi-optimal software development patterns which can assure a good quality software product in the end (Jacobson, Booch et al. 1999; Sarkar, Kak et al. 2008; Chhabra and Gupta 2010; Eichinger, Kramer et al. 2010; Ma, He et al. 2010). A good software development pattern can save a great deal of resource waste during the development process, and significantly reduce the risk of reworking or overhauling a developed software product (Rine and Sonnemann 1998; Frakes and Succi 2001). Since most software practitioners are overwhelmingly result driven (Sarkar, Kak et al. 2008; Shin, Meneely et al. 2011; Taherkhani, Korhonen et al. 2011), the final software products remain the major or, in some cases, the only focal point in the software lifecycle which results in the lack of data and effort for software development analysis. Very little research or analysis effort has been performed in exploring the insight of the software development process (Cimitile and Decarlini 1991; Jacobson, Booch et al. 1999).

Since its proposal by Erdos and Renyi (Erdos and Renyi 1959) (ER) in 1959, random graph theory has been applied to the study of complex systems across a wide variety of domains (Albert and Barabasi 2002; Newman 2003; Goodreau, Kitts et al. 2009; Durrett 2010; Gondal 2011; Simpson, Hayasaka et al. 2011). Despite this breadth of analysis, however, very few studies (e.g., (Potanin, Noble et al. 2005)) have sought to use this framework to analyze software systems. Very recently, researchers have started to analyze software systems from the perspective of complex networks (Potanin, Noble et al. 2005), but the preliminary studies reported in the literature so far have been limited to observations of a restricted set of topological measures and do not provide further explanations of the formation and evolution of software structures yet. These few existing studies are limited as they tend to be purely descriptive.

Our research presented in this chapter explicitly takes into account the particular characteristics of software systems and strive to better explain the development and the resulting structures of software systems. Specifically we revisit the function dependency networks extracted from the five widely-adopted C-based open source software packages as described in Chapter 2. By leveraging several network topological measures from Chapter 2, we reuse those common characteristics that are shared by those real world software packages. Driven by some fundamental incentives in the software engineering field and in network growth analysis (Krapivsky, Redner et al. 2000; Jin, Girvan et al. 2001; Milo, Shen-Orr et al. 2002; Goh, Oh et al. 2003; Milo, Itzkovitz et al. 2004), and incorporating the analytical results of Chapter 4, e.g. the impact factors of the clustering coefficient, we propose a two-phase network growth model to simulate the development process of the software products. Our analysis shows that our model can successfully explain all the characteristics that are obtained from the real world software packages. To the best of our knowledge, our model is the only one that can explain all those characteristics. We then conclude that our model can be a reasonable explanation of the software product formation and development process.

The rest of the chapter is organized as follows. We first introduce some background information in Section 1. In Section 2, we perform an empirical analysis of five widely-adopted open-source software packages by exhibiting their topological measures. Section 3 shows the fitting of some existing models. We present our two-phase network growth in Section 4 in detail. Specially we explain the rational behind our model, and present our model with textual description and a formal mathematical description. In Section 5, we compute the topological measures of the networks that are generated by following our network growth model. In additional to the analytical results, we also perform the numeric study in Section 6 to show some measures that are not readily calculated by analytical approach. Finally, we conclude the chapter with a summary of contributions and a discussion of possible future research directions in Section 7.

5.2 An empirical study of open source packages

An empirical analysis is performed to identify the topological properties of real software structures. To perform our empirical study, we have downloaded the source code of five widely-adopted open-source software packages: OpenSSH (a secure communication client), Httpd (Apache Web server), Gaim (a multi-protocol instant messaging client), MySQL (a database management system), and GIMP (GNU Image Manipulation Program). All of the packages are using C as the programming language. These applications vary widely in terms of the size of the package source code, allowing us to gain insights into the design of software systems across a range of sizes and complexity.

We define each node to be a function within a source file because functions are the smallest self-contained operational component of software systems. An edge between two vertices corresponds to a function call relationship between the functions represented by the vertices. As is common in complex systems analysis (Barabasi and Albert 1999; Ravasz and Barabasi 2003), at this stage of our inquiry we consider the edges to be weightless and directionless. Using the above definitions, we extract the function call graphs from the software source code, and compute the topological measures of each graph. The graph extraction is accomplished by using Imagix 4D (Murphy, Notkin et al. 1998), a commercial application intended to help developers model and analyze complex software systems. After constructing the function dependency networks for the real world software packages, we aggregate the atomic inter-function relationships to the system level topological metrics. The topological measures are computed by a home-grown Java-based application. The results are presented in Fig. 5.1 and Table 5.1.

.

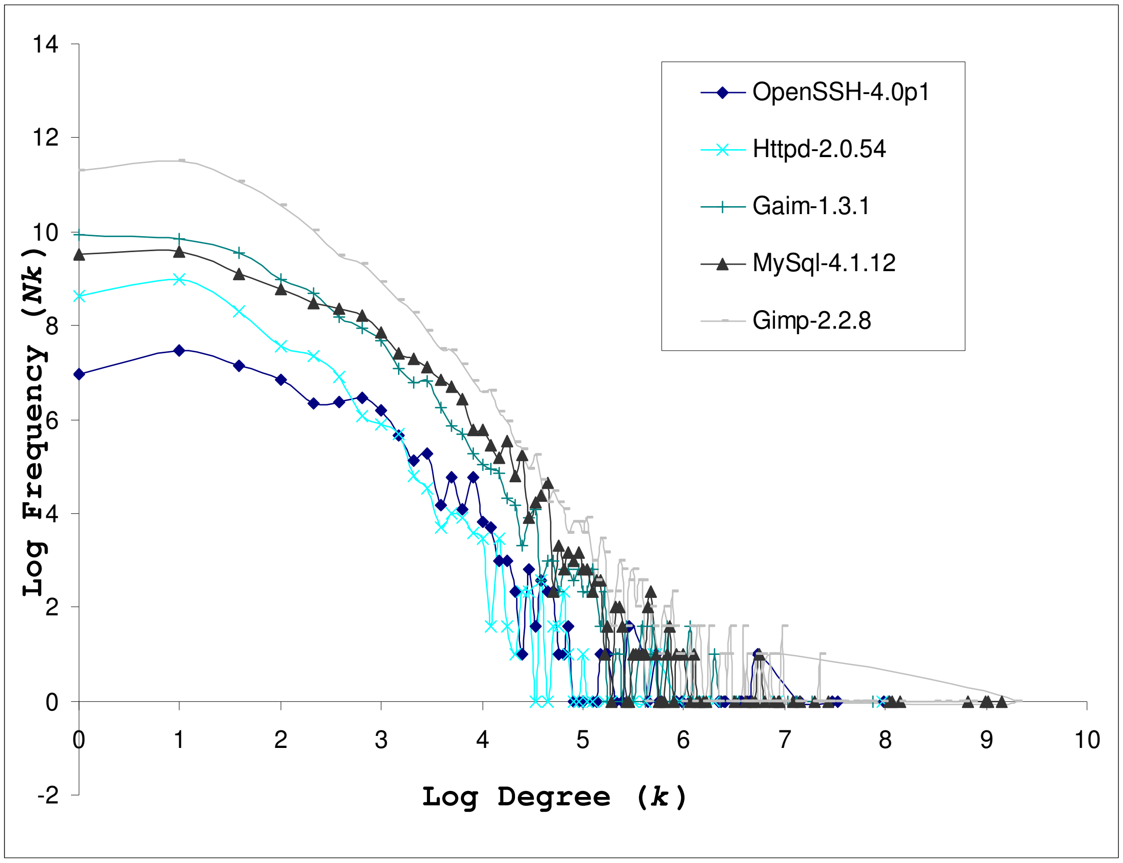


Fig. 5.1 Degree distributions of five software function dependency networks (Logarithm of base two is used throughout this chapter).

Four properties are observed in our empirical study. 1) Figure 5.1 shows the degree distributions of the function call graphs have scale-free property. 2) The distributions of the five graphs have similar slopes, and thus are parallel. The intercept increases on the Log Frequency axis as the network size increases. 3) The clustering using Watts-Strogatz definition remains invariant as the network sizes increases. 4) The clustering using Newman definition also remains invariant of the network sizes.

Table 5.1 Topological measures of five function dependency networks

Network	N	M	$\bar{k}$	$C^{(1)}$	$C^{(2)}$	C^{Rand}
OpenSSH	1,221	5,436	8.90	0.160	0.038	0.00729
Httpd	2,061	5,005	4.86	0.108	0.028	0.00236
Gaim	5,181	15,009	5.79	0.084	0.030	0.00112
MySql	5,024	19,745	7.86	0.158	0.034	0.00156
GIMP	14,380	45,224	6.29	0.132	0.023	0.00044

5.3 Review of existing models

We examine whether existing network models can closely reproduce the observed topological features of the function dependency networks and thus provide possible explanations for the formation and evolution of such networks. A large number of network models exist in the literature. Having observed inadmissible gaps between the Erdős-Rényi random graph model (Erdos and Renyi 1959; Erdos and Renyi 1960; Erdos and Renyi 1961) and the observed networks, we now focus on two additional influential models we feel most relevant to this study, i.e., the Barabási-Albert (BA) network growth model (Barabasi and Albert 1999) and the Ravasz-Barabási (RB) hierarchical network model (Ravasz and Barabasi 2003).

In the BA model, a network starts with a small set of nodes and continuously expands with the addition of new nodes, which are attached preferentially to existing nodes that are already well connected. Specifically, the probability that an existing node receives a new edge is proportional to the degree of the node. The resulting network is scale-free and has a power-law degree distribution with a fixed exponent. However, the clustering coefficient decreases as the network grows (Albert and Barabasi 2002). Using the Watts-Strogatz definition, it follows approximately a power law $C^{(1)} \sim N^{-0.75}$ and tends to zero in the limit of large network size. This apparently disagrees with the observations on the software function dependency networks.

In the RB model, small groups of nodes organize in a hierarchical manner into increasingly larger groups. During the network growth, sub-networks are replicated. A particular node serves as the hub of the network and is connected to all newly incorporated sub-networks. Ravasz and Barabási have demonstrated that this model approximately retains the scale-free property (Ravasz and Barabasi 2003). The RB model also leads to invariant clustering coefficient in the Watts-Strogatz definition (Watts and Strogatz 1998). However, the clustering coefficient in the Newman definition decreases substantially as network size increases (Newman 2003) (see Fig. 5.2). Hence, this model is also inadequate in explaining software function dependency networks. In the next section, we propose a new model, which extends the RB model and leads to invariant clustering coefficient in either of the two definitions.

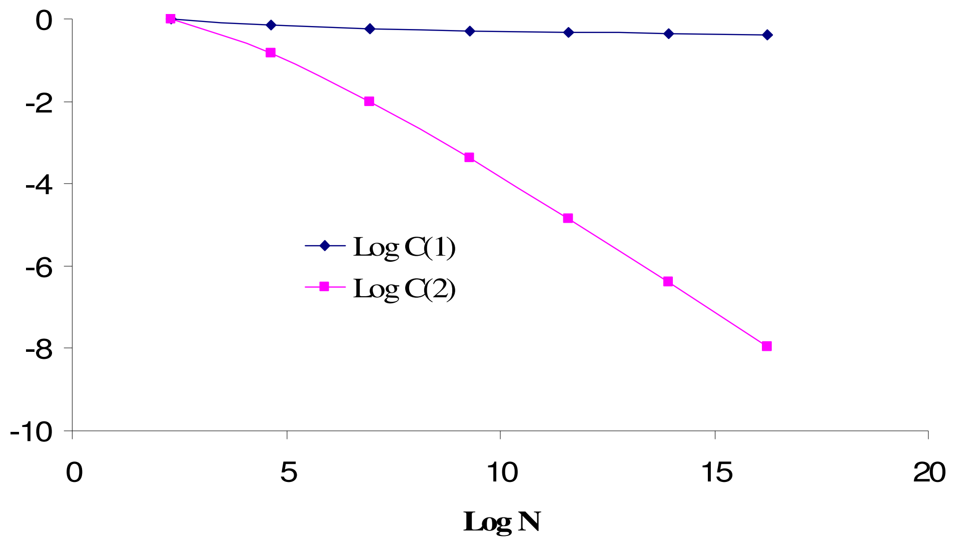


Fig. 5.2 Clustering coefficients of networks generated by the RB model

5.4 Proposed two-phase network growth model

In our proposed model, network growth experiences two phases. In the first phase, the network expands following the RB hierarchical network model. When the size of the network reaches a particular threshold, the network growth switches to the second phase, in which the sub-networks are replicated and linked to each other with sparse connections. Major differences of the second phase from the first one are that the inter-sub-network connections are sparse and that there is no particular node serving as a hub to all sub-networks. In contrast to generic network models, such as the BA model and the RB model, our proposed model is specifically motivated by software development principals. We first briefly discuss the rationales underlying the model and then present the model itself both descriptively and formally.

5.4.1 Rationales

Most scale-free networks that have been studied in the literature (e.g. the World-Wide-Web, the Internet, citation networks, and social networks) are gradually built up over time. The size of the growth at each step is relatively trivial compared to the size of the existing network. Connections between new growth and the existing network are not necessarily sparse. The network growth mechanism does not change as the network grows. Most existing network models that generate scale-free networks assume a similar growth process.

We posit that the development of software systems, however, follows a different process, while the resulting function dependency networks also possess the scale-free property. Regarded as a unique practice of complex system generations, software development has its own philosophy. Some generally-advocated software development principals include divide-and-conquer, modularization, high intra-module cohesion, and low inter-module coupling. A large software system is usually divided into several self-contained sub-systems of manageable sizes. The sub-systems, often referred to as modules (Parnas 1972; Parnas 1972), can be developed by one or a few highly-interactive engineering teams. Due to the high level of interaction within the teams, the structures of the modules are highly cohesive, with dense intra-module connections among functional units. On the other hand, to achieve encapsulation and independence of modules, it is desired to keep the inter-module connections sparse so that any change made on a certain module will have minimal impact on others (Dhama 1995; Tonella 2001; Leino and Nelson 2002; Darcy, Kemerer et al. 2005). As software engineers apply these principals in practice, the structures of the software systems they develop will have some special characteristics, which can be reflected by topological measures of the corresponding function dependency networks.

We can also look at software packages from a structure angle. Viewed from the point of high structural levels, software packages are amalgamations of several semi-independent modules, which share similar topological characteristics due to common development practices. The connections among modules occur intermittently and are sparse for the purpose of improving manageability, modularity, and extensibility by maintaining low coupling between functionally distinct modules (Dhama 1995; Tonella 2001; Darcy, Kemerer et al. 2005). There should not be a single function or module that serves as a hub of the entire package in order to avoid system crashes caused by single-point failures (Kennel, Perry et al. 1989; Albert, Jeong et al. 2000; Callaway, Newman et al. 2000; Li, Zou et al. 2004; Hu, Guo et al. 2005). On the other hand, at a much detailed structural level, the functions within a module are well connected to improve intra-module cohesion. There are very likely one or a few largely connected functions serving as local hubs of the module to pass on data and commands. Therefore, we propose that there are two different mechanisms that govern the growth of a software function dependency network at different levels depending on the size of the network.

5.4.2 Model description

We first define two parameters: n, the size of the initial network, and T, the number of steps that the network grows before the growth mechanism migrates from phase one to phase two. It is plausible to assume that $n \geq 3$ and $T \geq 1$ for a large software system.

Phase one

The first phase of our model follows the same rules defined in the RB hierarchical network model (Ravasz and Barabasi 2003). At the starting point, referred to as step one, the network is a small cluster with n nodes where each node is connected to every other node. Of the n nodes, one is defined as the "hub" and the others are called "peripherals". The "hub" of the initial cluster is also referred to as the "center" of the network. Throughout phase one, there is only one center.

At step two, $n-1$ replicas of this small cluster are generated, resulting in an n^2-node cluster. The nodes that originate from the hub become the new hubs. The copies of peripherals become the new peripherals. To connect the original cluster and its replicas, the new peripherals are connected to the center.

Subsequently, the n^2-node cluster is again replicated $n-1$ times, generating an n^3-node cluster. The copies of the hubs and peripherals in the previous step become the new hubs and peripherals, respectively. The new peripherals are then connected to the center.

Following the same procedure, the network generation will be repeated for T steps. The cluster obtained at step T is referred to as a "module".

Phase two

Starting from step $T+1$, the network generation migrates to phase two. The module generated at the end of phase one is replicated. We refer to the original module as module 1 and the replica as module 2. All nodes that originate from hubs and peripherals are again called hubs and peripherals, respectively. In addition, the copy of the center of module 1 is called the center of module 2. Unlike in phase one, the number of centers increases by one at each step in phase two. Two least-connected hubs—one from each module—are connected through an edge. Specifically, the last hub in module 1 is connected to the second last hub in module 2. This newly added edge is referred to as an "inter-module edge". The hubs connecting the modules resemble interfacing "studs" in software modules designed for the purpose of inter-module communications. It is usually desired to keep the linkage between the studs and other parts of a module to a minimum. Such studs are usually developed after the functionalities of the module have been developed. The combination of the two modules and the inter-module edge forms the network of step $T+1$.

At the next step, another replica of the original module, referred to as module 3, is added to the network. Two least-connected hubs—one from module 3 and the other from module 2—are connected through an inter-module edge. Specifically, the last hub in module 2 is connected to the second last hub in module 3. This procedure can be repeated indefinitely.

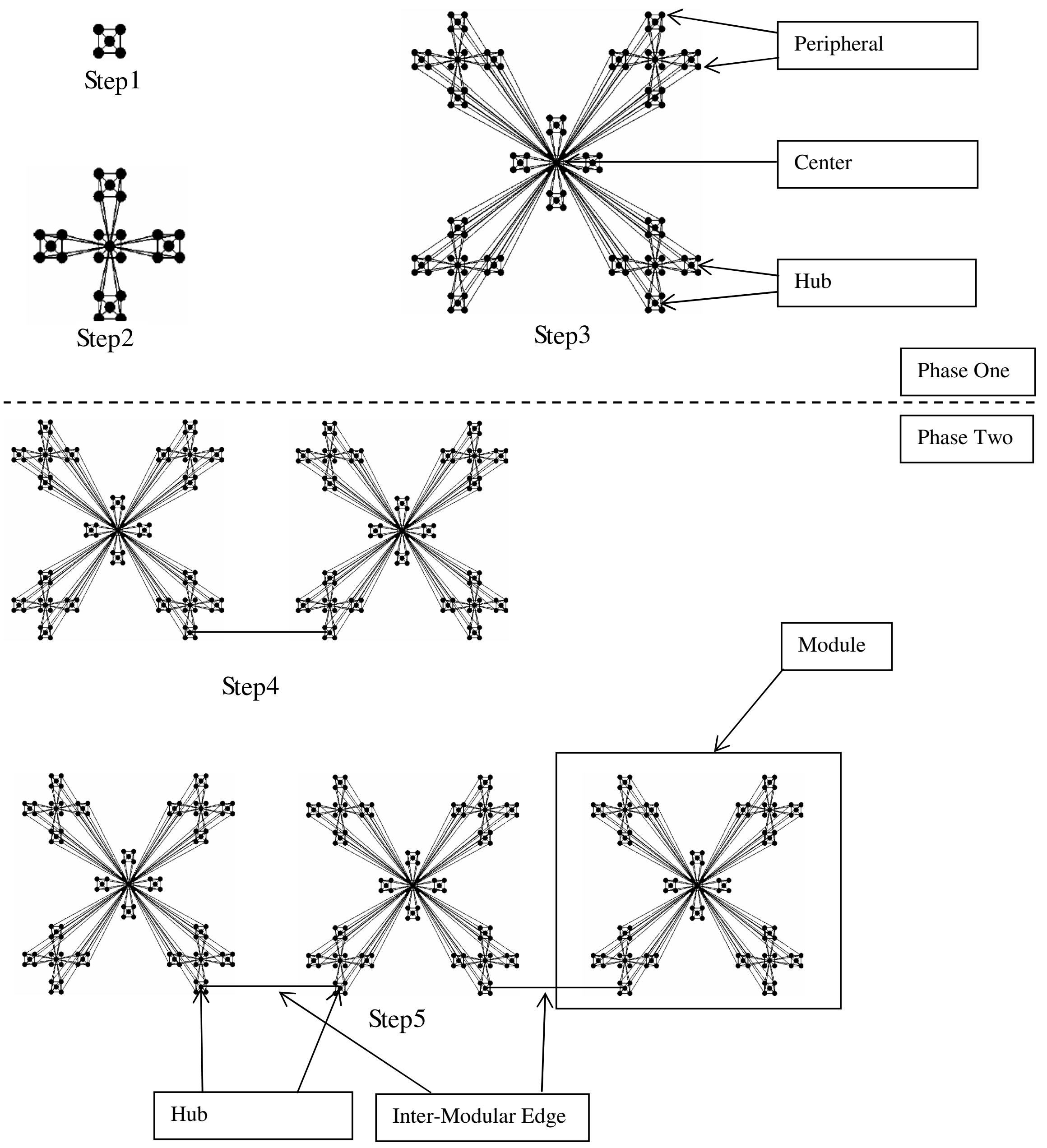


Fig. 5.3 Two-phase network growth model. Note that the initial network is fully connected, although the edges connecting diagonal nodes are not evident.

Fig. 5.3 illustrates the complete procedure using an example where $n = 5$ and $T = 3$. The first three steps follow the first-phase generation rules. Steps 4 and 5 represent the second-phase network generation.

5.4.3 Formal model description

We reuse the symbols listed in Table 2.2, and add a subscript representing the step of network growth. For example, N_t denotes the number of nodes at step t. Table 5.2 lists some additional necessary symbols.

Table 5.2 Additional symbols used in the proposed model

Symbol	Measure
t	Current step of network growth
V_t	Set of nodes at step t
VH_t	Set of hubs at step t
VP_t	Set of peripherals at step t
VC_t	Set of centers at step t
(i, j)	Edge connecting nodes i and j
E_t	Set of edges at step t
EI_t	Set of inter-module edges at step t

Table 5.3 Various aspects of the proposed two-phase network growth model

	$t=1$	$2 \le t \le T$
V_t	$\{1,2,\ldots,n\}$	$V_{t-1} \cup \{i + c \cdot n^{t-1} \mid i \in V_{t-1}; c = 1,2,\ldots,n-1\} \;=\; \{1,2,\ldots,n^t\}$
VH_t	$\{1\}$	$\{i + c \cdot n^{t-1} \mid i \in VH_{t-1}; c = 1,2,\ldots,n-1\}$
VP_t	$\{2,3,\ldots,n\}$	$\{i + c \cdot n^{t-1} \mid i \in VP_{t-1}; c = 1,2,\ldots,n-1\}$
VC_t	$\{1\}$	$\{1\}$
E_t	$\{(i,j) \mid i,j = 1,2,\ldots,n;\ i \ne j\}$	$E_{t-1} \cup \{(i + c \cdot n^{t-1}, j + c \cdot n^{t-1}) \mid (i,j) \in E_{t-1}; c = 1,2,\ldots,n-1\}$ $\cup \{(1,i) \mid i \in VP_t\}$
EI_t	$\varnothing$	$\varnothing$

	$t > T$
V_t	$V_{t-1} \cup \{(t-T)n^T + i \mid i \in V_T\} \;=\; \{1,2,\ldots,(t-T+1)n^T\}$
VH_t	$VH_{t-1} \cup \{(t-T)n^T + i \mid i \in VH_T\}$
VP_t	$VP_{t-1} \cup \{(t-T)n^T + i \mid i \in VP_T\}$
VC_t	$VC_{t-1} \cup \{(t-T)n^T + 1\} \;=\; \{1 + c \cdot n^T \mid c = 0,1,\ldots,t-T\}$
E_t	$E_{t-1} \cup \{((t-T)n^T + i, (t-T)n^T + j) \mid (i,j) \in E_T\} \cup (EI_t \setminus EI_{t-1})$
EI_t	$EI_{t-1} \cup \{((t-T)n^T - n + 1, (t-T+1)n^T - 2n + 1)\} \;=$ $\{(c \cdot n^T - n + 1, (c+1)n^T - 2n + 1) \mid c = 1,2,\ldots t-T\}$

Various aspects of our two-phase network growth model are formally defined in Table 5.3. When $1 \le t \le T$, the network growth is in phase one. When $t > T$, the network growth is in phase two.

Table 5.4 lists some basic measures. Most measures can be straightforwardly derived, except $M_t = |E_t|$, as E_t is only recursively specified. M_t can be derived from the following recursively defined sequence.

$$M_1 = C_n^2 = \frac{n(n-1)}{2} \tag{5.1}$$

$$M_t = n \cdot M_{t-1} + |VP_t| = n \cdot M_{t-1} + (n-1)^t \text{, when } 2 \le t \le T. \tag{5.2}$$

$$M_t = M_{t-1} + M_T + 1 = (t - T + 1)M_T + t - T \text{, when } t > T. \tag{5.3}$$

Table 5.4 Basic measures of the proposed two-phase network growth model

	$1 \le t \le T$	$t > T$
$N_t = \lvert V_t \rvert$	n^t	$(t-T+1)n^T$
$\lvert VH_t \rvert$	$(n-1)^{t-1}$	$(t-T+1)(n-1)^{T-1}$
$\lvert VP_t \rvert$	$(n-1)^t$	$(t-T+1)(n-1)^T$
$\lvert VC_t \rvert$	1	$t-T+1$
$M_t = \lvert E_t \rvert$	$\frac{1}{2}n^{t-1}(3n-2)(n-1)-(n-1)^{t+1}$	$(t-T+1)(\frac{1}{2}n^{T-1}(3n-2)(n-1)-(n-1)^{T+1})+t-T$
$\lvert EI_t \rvert$	0	$t-T$

5.5 Properties of the proposed model

In this section, we derive and discuss some properties of networks generated following the proposed two-phase growth model. The analysis will show that the proposed model reproduces the features observed in the empirical analysis of real-world software packages.

5.5.1 Average degree

Property 1: In the limit of large network size (as $t > T \to \infty$), average degree $\bar{k}_t$ tends to a constant, which is independent of the network size.

Proof:

When $t > T$, $M_t = (t-T+1)(\frac{1}{2}n^{T-1}(3n-2)(n-1)-(n-1)^{T+1})+t-T$, $N_t = (t-T+1)n^T$.

$$\bar{k}_t = \frac{2M_t}{N_t} = \frac{(3n-2)(n-1)}{n} + \frac{2-2(n-1)^{T+1}}{n^T} - \frac{2}{(t-T+1)n^T}. \qquad (5.4)$$

The time-dependent term $\frac{2}{(t-T+1)n^T}$ is negligible in a reasonably-sized software package. For example, when n=5, T=5, and t=10, $\frac{2}{(t-T+1)n^T}$ is about 0.0001 and has little effect on $\bar{k}_t$. As $t \to \infty$, $\bar{k}_t \to \frac{(3n-2)(n-1)}{n} + \frac{2-2(n-1)^{T+1}}{n^T}$, which is constant and is only dependent on the initial cluster size n and the phase migration threshold T. ■

5.5.2 Clustering coefficient

First, we show that in the RB hierarchical network model (corresponding to the situation where $T=\infty$, such that the network growth remains in the first phase, in the proposed model), the clustering coefficient in the Newman definition tends to zero in the limit of large network size. This has also been demonstrated in Fig. 4 and is inconsistent with what we observe in the function dependency networks of real-world software packages.

Property 2: If $T=\infty$ and $t \le T$ always holds, as $t \to \infty$, the clustering coefficient in the Newman definition $C_t^{(2)} \to 0$.

Proof:

If $T=\infty$ and $t \le T$ always holds, the network growth remains in the first phase.

Recall that $C_t^{(2)} = \frac{3 \times p_t}{q_t}$, where p_t is the number of triangles and q_t is the number of connected triples. At $t=1$, as the initial network is fully connected, every triple is also a triangle, each triangle contributes three connected triples centered on different nodes, $q_1 = 3 \times p_1$, and thus $C_1^{(2)} = 1$.

When $2 \le t \le T$, the triangles in the network G_t come from two sources: the replicas of G_{t-1}, and the new triangles formed by the inter-cluster edges connecting the center and the peripherals. Note that a new triangle can only be formed when two inter-cluster edges are connecting the center and two peripherals from the same replica of the original cluster G_1. Therefore,

$$p_t = np_{t-1} + (n-1)^{t-1} C_{n-1}^2 = np_{t-1} + (n-1)^{t-1} \frac{(n-1)(n-2)}{2} \tag{5.5}$$

The connected triples in G_t come from four sources: the replicas of G_{t-1}, the new triples formed by two inter-cluster edges; the new triples formed by one inter-cluster edge and one intra-cluster edge that belongs to a replica of G_{t-1}; the new triples formed by one inter-cluster edge and one intra-cluster edge that belongs to G_{t-1}. Therefore,

$$q_t = nq_{t-1} + (n-1)^t \frac{(n-1)^t - 1}{2} + (n-1)^t (n+t-3) + (n-1)^t \frac{(n-1)^t - (n-1)}{n-2} \tag{5.6}$$

Apparently, q_t (on the order of n^{2t}) increases much faster than p_t (on the order of n^{t+1}). $C_t^{(2)}$ monotonically decreases as t increases. $C_t^{(2)} \to 0$, as $t \to \infty$. ■

We now show that when the proposed two-phase network growth model is indeed in effect (i.e., T is finite), the clustering coefficient in the Newman definition tends to a non-zero constant in the limit of large network size.

Property 3: If T is finite, as $t > T \to \infty$, $C_t^{(2)}$ tends to a non-zero constant.

Proof:

When $t > T$, the new triangles of G_t come only from the addition of a new replica of G_T to G_{t-1}. The single inter-module edge that links the new replica of G_T and G_{t-1} does not introduce any new triangle. Therefore,

$$p_t = p_{t-1} + p_T = (t - T + 1) p_T \tag{5.7}$$

The inter-module edge does introduce new connected triples. The number of such connected triples is twice the degree of the hubs connected by the inter-module edge.

$$q_t = q_{t-1} + q_T + 2(n-1) = (t-T+1)q_T + 2(t-T)(n-1) \tag{5.8}$$

In phase two, we choose to connect hubs instead of peripherals when a new module is added to the network. The reason is that the degree of hubs does not change in phase one. In fact, it remains $n-1$ throughout the first phase. On the other hand, the degrees of peripherals change as the network grows in phase one, as they are used to connect new replicas with the center.

$$C_t^{(2)} = \frac{3 \times p_t}{q_t} = \frac{3(t-T+1)p_T}{(t-T+1)q_T + 2(t-T)(n-1)} = \frac{3p_T}{q_T + 2(n-1) - \dfrac{2(n-1)}{t-T+1}} \tag{5.9}$$

$C_t^{(2)}$ monotonically decreases as t increases. However, the time-dependent term $\frac{2(n-1)}{t-T+1}$ becomes negligible as t is sufficiently large. $C_t^{(2)} \to \frac{3p_T}{q_T + 2(n-1)}$, as $t > T \to \infty$. ■

Finally, while it has been shown that the RB hierarchical network model leads to invariant clustering coefficient in the Watts-Strogatz definition, $C_t^{(1)}$ (Watts and Strogatz 1998), we now show that in the proposed two-phase model, $C_t^{(1)}$ tends to a non-zero constant in the limit of large network size.

Property 4: As $t > T \to \infty$, $C_t^{(1)}$ tends to a non-zero constant.

Proof:

Recall that $C_t^{(1)}$ is defined as the mean of the clustering coefficients of all nodes and $C_{i,t}^{(1)} = \frac{a_{i,t}}{k_{i,t}(k_{i,t}-1)/2}$, where $a_{i,t}$ is the number of edges among the neighbors of node i. When $t > T$, at each step, only the two hubs (denoted j and l) connected by the new inter-module edge experience any degree or edge connection change. Previously (at step $t-1$), all the neighbors of these hubs are connected to each other and $C_{j,t-1}^{(1)} = C_{l,t-1}^{(1)} = 1$. Now (at step t), the degree of these hubs increases by 1 due to the added inter-module edge.

$$C_{j,t}^{(1)} = C_{l,t}^{(1)} = \frac{(n-1)(n-2)/2}{n(n-1)/2} = \frac{n-2}{n} \tag{5.10}$$

$$C_t^{(1)} = \frac{1}{N_t}(N_{t-1}C_{t-1}^{(1)} + N_T C_T^{(1)} + (C_{j,t}^{(1)} - C_{j,t-1}^{(1)}) + (C_{l,t}^{(1)} - C_{l,t-1}^{(1)}))$$

$$= \frac{(t-T)C_{t-1}^{(1)} + C_T^{(1)}}{(t-T+1)} - \frac{4}{(t-T+1)n^{T+1}}$$

$$= C_T^{(1)} - \frac{4(t-T)}{(t-T+1)n^{T+1}} = C_T^{(1)} - \frac{4}{(1+\frac{T}{t-T})n^{T+1}} \tag{5.11}$$

As $t \to \infty$, $C_t^{(1)} \to C_T^{(1)} - \frac{4}{n^{T+1}}$. ■

5.5.3 Degree distribution

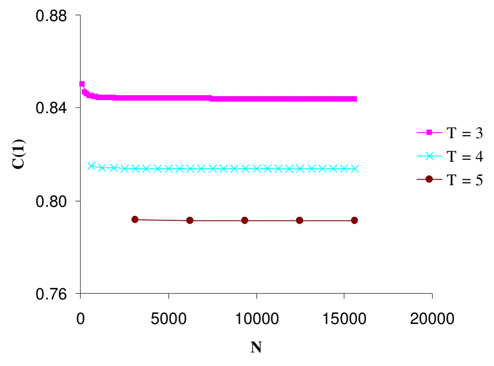
Property 5: A network generated by the two-phase model is approximately scale-free. Its degree distribution approximately follows the power law.

We discuss this property informally here and will demonstrate it through a numeric study in the next section. Ravasz and Barabási have demonstrated that the RB hierarchical network model approximately retains the scale-free property (Ravasz and Barabasi 2003). The degree distribution of a RB hierarchical network approximately follows the power law. In the second phase of the proposed two-phase model, the network generated using the RB model for T steps are replicated. Such replication does not change the degree distribution. The only change is caused by the inter-module edges. At step $t > T$, the proportion of n-degree nodes $p_t(n)$ increases by $\frac{2|EI_t|}{M_t} = \frac{1}{(1+\frac{1}{t-T})(\frac{1}{2}n^{T-1}(3n-2)(n-1)-(n-1)^{T+1})+1}$ and the proportion of ($n-1$)-degree nodes $p_t(n-1)$ decreases by the same amount. This amount of change is negligible in a reasonably-sized network. The degree distribution at any other degree is identical to that of the RB hierarchical network at step T. The network is still approximately scale-free.

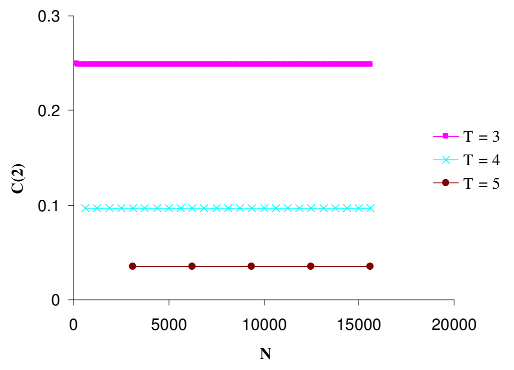
5.6 Numeric study

In this section, we numerically illustrate some properties of the proposed model. We also study some possible variations of the model. In the original model, one inter-module edge connects the hubs of two modules. We now investigate the impacts of the number and location of the inter-module edges between two modules. As the uniqueness of the model is in the second phase, we focus the study on the second phase only.

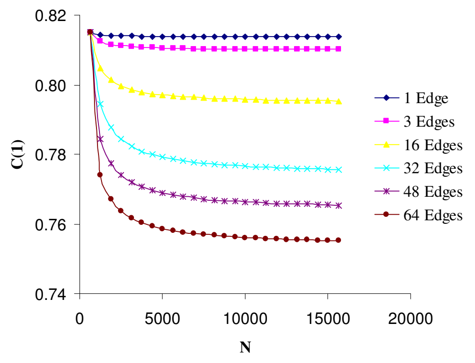
5.6.1 Clustering coefficient



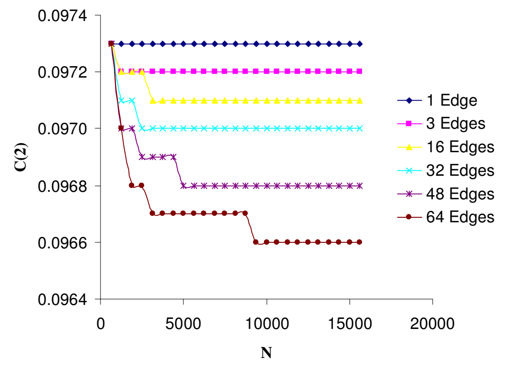
(a)



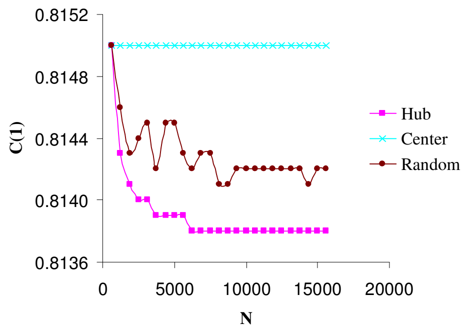
(b)



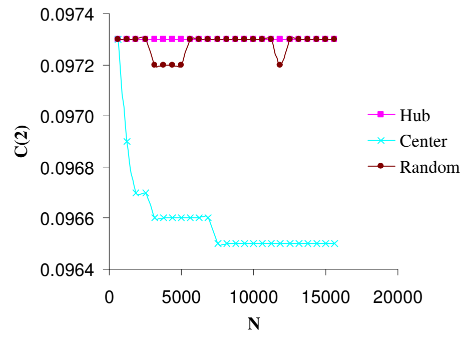
(c)



(d)



(e)



(f)

Fig. 5.4 Clustering coefficients of two-phase network growth model

(a), (b): Original model; (c),(d): Varying the number of inter-module edges (T =4); (e),(f): Varying the locations of inter-module edges (T =4).

Fig. 5.4 shows the clustering coefficients in the Newman and Watts-Strogatz definitions under various settings. We set the size of the initial cluster to n =5 throughout this section. The value of n affects the absolute values, but not the trends, of clustering coefficient and other topological properties examined later.

Figures (a) and (b) show the clustering coefficients of the original model under different values of T, which determines the size of a module generated in phase one. The clustering coefficients by both definitions quickly approach certain lower bounds as the network grows. The larger the modules are, the lower the overall clustering is.

Figures (c) and (d) show the effects of the number of inter-module edges between two modules on clustering. The inter-module edges connect hubs of the modules. For n =5, T =4, each module has 64 hubs. As the number of inter-module edges increases, the lower bounds of the clustering coefficients decrease and it takes longer to approach the lower bounds. This corresponds to the intuition that lowering inter-module coupling helps to improve the overall clustering.

Figures (e) and (f) show the effects of the locations of inter-module edges on clustering. Three conditions are investigated: the inter-module edge between two modules connects (1) hubs, (2) centers, and (3) randomly selected nodes. The first two conditions set the upper and lower bounds of clustering, as the center of a module has the highest degree and the hubs have the lowest degree. Placing the inter-module edge randomly leads to clustering lying between the bounds. The locations of inter-module edges have different effects on the clustering coefficients in different definitions. Connecting hubs gives the upper bound of $C^{(1)}$ and the lower bound of $C^{(2)}$.

5.6.2 Average degree

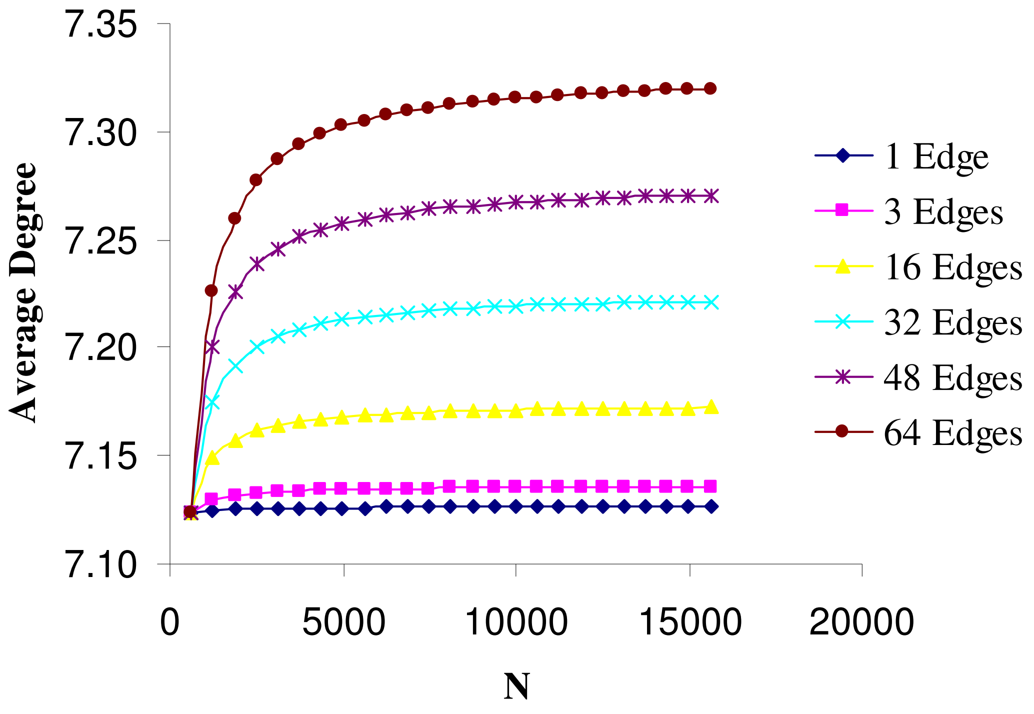
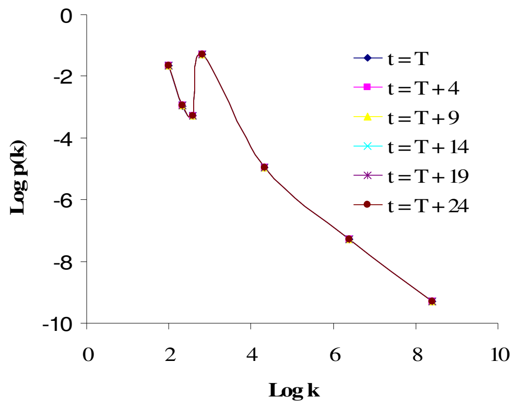


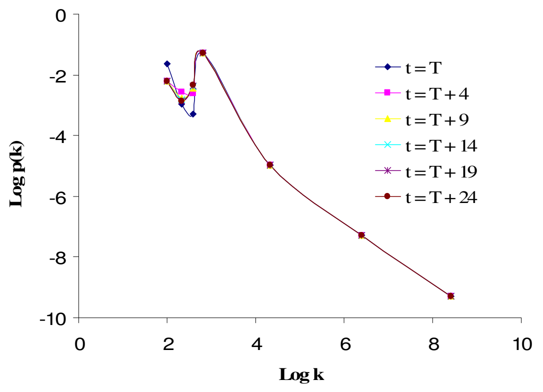
Fig. 5.5 Average degree (T=4) of two-phase network growth model.

Fig. 5.5 shows the average degrees of the original model and its variants with different numbers of inter-module edges. In the original model, average degree quickly approaches an upper bound. As the number of inter-module edges increases, the upper bound also increases and it takes longer to approach the upper bound. The locations of inter-module edges have no effect on average degree.

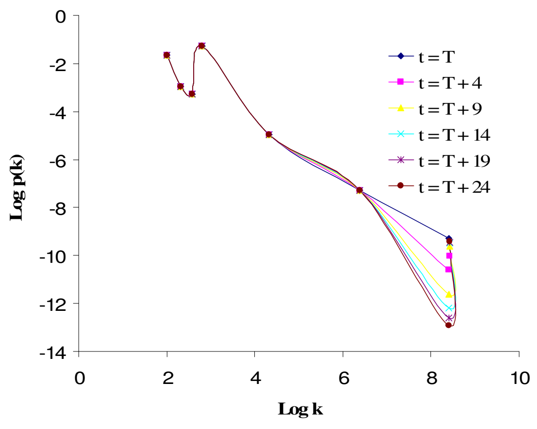
5.6.3 Degree distribution



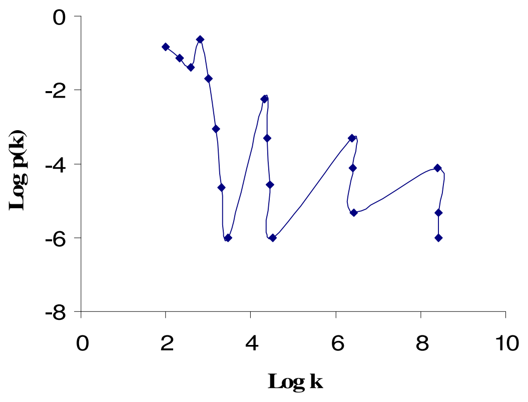
(a)



(b)



(c)



(d)

Fig. 5.6 Degree distribution (T=4) of two-phase network growth model.

(a): Original model; (b): 64 inter-module edges connecting hubs; (c): one inter-module edge connecting centers; (d): 64 inter-module edges connecting randomly-selected nodes ($t = T + 24$)

Fig. 5.6 shows the degree distributions of the original model and some variants. In the original model (see Fig. 5.6(a)), the degree distribution remains almost the same as the network grows. The only changes at degrees n and $n-1$ are negligible. The model basically retains the degree distribution of the RB hierarchical network model (Note that when $t \leq T$, the model is identical to the RB hierarchical network model (Ravasz and Barabasi 2003)). The degree distribution approximately follows the power law (Ravasz and Barabasi 2003).

As the number of inter-module edges increases (see Fig. 5.6(b)), the deviation from the RB hierarchical network model increases. The changes occur mainly at low degrees. The degree distribution still appears to approximately follow the power law.

If the inter-module edge between two modules connects the centers, rather than hubs, of the modules (see Fig. 5.6(c)), the degree distribution deviates from that of the RB model at high degrees. The deviation increases as the network grows. The degree distribution gradually departs from the power law.

If there are many inter-module edges connecting randomly selected nodes (see Fig. 5.6(d)), the degree distribution may severely deviate from the power law over time, thus destroying the scale-free property. Deviation may occur at any degree.

From Fig. 5.6(c) and (d), we observe that the connecting vertices of the inter-module edges play an important role in maintaining the power law property. On the other hand, Fig. 5.6(b) shows that the number of inter-module edges connecting two modules is a less significant factor. The connecting vertices of inter-module edges correspond to the interface functions of the modules in software packages. Our finding is consistent with the common knowledge established in the software engineering field that interface functions are very important in maintaining the software structure as the software size expands. If the interface functions are chosen wisely, a software package can maintain a steady structure even though interface functions are heavily called across modules.

The effects of the number and location of inter-module edges can be summarized as follows. The number of inter-module edges has little effect on degree distribution. However, as this number increases, clustering decreases while average degree increases. The location of inter-module edges has no effect on average degree but affects clustering and degree distribution. Its effect on clustering is different depending on the definition of clustering coefficient. Connecting hubs (centers) of modules provides the upper (lower) bound of $C^{(1)}$ and the lower (upper) bound of $C^{(2)}$. If the inter-module edges connect centers, degree distribution gradually departs from the power law as the network grows. Connecting randomly-selected nodes across modules may destroy the scale-free property.

5.7 Concluding remarks

In this chapter, we propose a new network growth model. Our development of the model is inspired by generally-advocated software engineering principals, such as divide-and-conquer, modularization, high intra-module cohesion, and low inter-module coupling. The model has two phases. The first phase follows the hierarchical network model of Ravasz and Barabási (Ravasz and Barabasi 2003). The second phase strives to minimize the coupling across modules generated following the hierarchical network model. Both analytical and numerical studies show that the proposed model adequately reproduces the topological features observed in real-world software packages.

The results of this work can be used in developing metrics and associated guidelines—complementing existing ones—in CASE tools. Such metrics provide further insights into the overall structure of a software package and the process governing its development. They can be used by software developers and managers to adjust processes and strategies during software development. They can also be used to evaluate and compare developed packages, especially open-source products, in terms of such properties as modularity, intra-module cohesion, and inter-module coupling.

Similar to some related studies (e.g., (Zheng, Zeng et al. 2008)), we focus on the physical topological properties of a particular type of networks (i.e., software function dependency networks). However, while our model has been developed explicitly to characterize and explain software function dependency networks, it may be applied to study pervasive physical topological properties possessed by other networks that exhibit similar features. It may be especially useful for studying networks that exhibit such features as modularization, high cohesion, and low coupling. For example, a movie actor is usually associated with a certain company or agent. Naturally, actor collaboration networks should exhibit heavy intra-company collaborations and relatively sparse inter-company collaborations. Our two-phase network growth model can potentially be used to analyze such actor collaboration networks.

Our study opens up several avenues for further research. First, investigating networks built from historical versions of the same software package may generate useful insights. Analyze the evolvement of the software unit dependency networks and how software grows and changes over time is likely to provide direct evidence to support our network growth model.

Second, while we have modeled software packages as un-directed, un-weighted networks of functions, considering the direction and weight of function calls may reveal additional useful information about software packages.

Third, while we have focused on seeking a general understanding of software development in this chapter, our model can be extended in the future to accommodate more specifics (e.g., complexity differences across functions and size differences across modules) with parameters (e.g., T) tuned to fit a particular software package of interest.

Further research and improved understanding of random networks may lead to the development of useful metrics that provide guidance to software developers and architects in the development of complex software systems.

CHAPTER 6 CONCLUSIONS AND FUTURE DIRECTIONS

Open source software has gained increasing popularity for the past few years. Lots of businesses choose to adopt open source software due to its cost efficiency and customizable functionalities. However, as open source software packages become increasingly complicated, it is very difficult to analyze the structures of the software products, and predict the quality of the software at the early stage of the development process. In this dissertation, we confront the challenges that exist in open source software engineering. Specifically we examine the collaborative relations among open source software developers, study software development processes, and analyze the structures of open source software packages' source code. Our intentions are to discover some common properties that are shared by different open source software applications, and hopefully to reveal the underlying characteristics that guide the open source software development. Our research follows the 4-P framework which defines the four key elements in software engineering: *Project*, *People*, *Process*, and *Product*. In terms of research methodology, we leverage the topological metrics that have been established in complex network theory, and propose a random network growth model to illustrate open source software development processes.

In Chapter 2, we analyze real world software packages with function dependency networks. Our study subject is the source code of open source software packages, the product of software engineering. We obtain the source code of five C language based open source software packages that have different package sizes, and are from different application domains. All those packages are considered successful in their own application domain. We then construct an undirected and un-weighted network for each one of the software packages with the vertex representing a function and the edge as a function call. Once the function dependency networks are constructed, we leverage several widely adopted topological measures to analyze those networks. Our empirical study indicates three common features that those five function dependency networks possess: (1) average degree is independent of network size, (2) clustering coefficient, in either of two definitions proposed by Watts and Strogatz, and by Newman, is independent of network size, and (3) the networks are scale-free. These findings support our hypothesis that there exist some common topological properties shared by different open source software packages over a wide range of purpose, domain, size, and complexity.

The results of Chapter 2 can be used as a starting point to quantitatively analyze software architectural structures. Although we used open source software packages to perform the case study, the usage of our research is not confined to the scope of open source software applications. Software companies can easily adopt our method to examine the architectural structure of their proprietary software products since they have the full control of their own software source code. Moreover a similar framework can be easily created to analyze software systems that are developed in programming languages other than C language. For example, we can construct a software unit dependency network for a Java-based system by defining vertex as a Java object and edge as a Java object reference. Once the dependency network has been constructed, the similar set of topological measures can be calculated to examine that software package.

Chapter 3 focuses on another important factor in software engineering field, People, more specifically software developers. The collaboration among software developers is an important factor in open source software development as the efficiency of the developer collaboration has a direct impact on the quality of the final software product. In this chapter, we intend to discover some common patterns from different real world software developers' collaborations. Specifically we choose the bug fixing data sets of two real world software applications, MediaWiki and Gentoo, to perform the empirical analysis. We define the fact that two developers have worked to solve at least one bug together as the collaboration between those two developers. Based on that definition, we extract bug-developer bipartite networks from those two bug fixing data sets, and derive developer-developer collaboration networks from the bipartite networks. Following the similar procedure described in Chapter 2, we calculate several topological measures for those constructed bug-developer bipartite networks and developer-developer collaboration networks. The empirical findings show that all networks including the bipartite networks possess the scale-free property. Our empirical findings and research framework can be extended to further examine open source software developer collaborations such as development leadership, task assignments and scheduling, the impact of key developer's sudden departure, etc.

In both Chapter 2 and Chapter 3, the networks that we extract out of real world entities are undirected and un-weighted. In the future research, we may extend our empirical study by considering meaningful directions and weights on the constructed networks. By considering the direction and weight of edges, we will incorporate more usable information of the real world entities, and thus may reveal additional useful findings about software products and developer collaborations.

In complex network theory, clustering coefficient is one of the most informative topological measures. Chapter 4 aims to provide an analytical comparison between two widely adopted clustering coefficient definitions, C_{WS} and C_{NW}, proposed by Watts-Strogatz and Newman, respectively. Mathematical derivations are presented to compare the similarities and differences between those two definitions. Our findings show the two definitions both depend on $T_a^{(i)}$ and $T_p^{(i)}$, the number of triangles and triples, respectively, around vertex i. The difference between the two definitions lies in that C_{WS} is the mean of the ratio $T_a^{(i)}$ and $T_p^{(i)}$, and C_{NW} is the ratio of the two means of $T_a^{(i)}$ and $T_p^{(i)}$. We also examine the lower bounds and upper bounds of those two definitions, and the conditions to meet those extreme bounds. Our further analysis shows the impact factors of C_{WS} and C_{NW} values. Using simulated scale-free networks, we demonstrate that the extremely popular vertices have little impact on C_{WS} due to the limited number of those popular vertices. Whereas those popular vertices are the dominating factors in determining the value of C_{NW}.

Our research results show some useful analytical properties of the two clustering coefficient definitions, C_{WS} and C_{NW}. The results provide complex network researchers more insights when they conduct network analysis research. For example, our findings provide the guidelines on which clustering coefficient definition should be adopted when a complex network needs to be analyzed. Moreover the analytical properties of the two clustering coefficient definitions can be useful when a random network growth model is proposed to explain the formation of a complex network.

Chapter 5 incorporates all the empirical and analytical findings that have been presented in the previous chapters. In this chapter, we intend to discover the underlying characteristics of open source software development processes, and provide a reasonable explanation of the formation of open source software packages. Together with Chapter 2 and Chapter 3, we have studied three of the four P's, Product, People, and Process, in the Four P Software Engineering Framework.

Chapter 5 reuses the empirical findings from Chapter 2, the topological features of the function dependency networks that have been extracted from five real world C-based open source software packages. In addition to the descriptive findings, in Chapter 5 we propose a two-phase network growth model that simulates the development process of open source software products. Our network growth model is inspired by some widely adopted principles in software engineering, e.g. modularization, high intra-module cohesion, low inter-module coupling, etc. The model also leverages the analytical results in Chapter 4 by considering the similarities and differences between C_{WS} and C_{NW}. Our network growth model describes two different growth phases as the size of the software grows. The first phase follows the hierarchical network model proposed by Ravasz and Barabási (Ravasz and Barabasi 2003). In the second phase software modules are connected with sparse inter-module connections. The second phase observes the low inter-module coupling principle. Both analytical and numerical results demonstrate that the proposed two-phase network growth model adequately reproduces the topological properties that have been found in Chapter 2.

The results of Chapter 5 can be leveraged by software developers and managers to adjust processes and strategies during software development in order to reduce the software development costs and risks.

This dissertation opens up several avenues for further research.

First, while we have studied several distinct open source software packages, investigating networks built from historical versions of the same software package is likely to provide direct evidence to support our network growth model. By comparing the networks extracted from different versions of the same software, we will be able to realize how software structures evolve over time. Moreover, if we consider the popularity, such as the download rate and user rating, of each software version, we are likely to discover more insights on how network structure and popularity correlate.

Second, most networks that we have extracted from the real world entities are un-directed and un-weighted. Considering the direction and weight of those networks may reveal additional useful information about open source software. For the C-based software packages, the edge direction can indicate the direction of the function call between the two connected C functions. Similarly, the weight on an edge may represent the number of function calls between those two functions. Note that by considering the direction of the network edge, we have to take into consideration of possible network loops and self loops. Thus the constructed networks will likely have more complicated topological structures.

Third, all of the five real world software packages that we chose for the case studies are considered "popular" or "good quality" software systems. On the other hand, comparing software packages with different popularities, e.g. download rates, and qualities, e.g. user ratings, in the same application domain would be worthwhile. The comparison is likely to reveal the correlation between software structure and software quality. The challenge in this line of research extension lies in two folds. Firstly, the quantitative definition of the so-called software quality is still under intense debate in software engineering field. Secondly, the popularity depends on a much broader range of factors than the structure of the software package. Other related impact factors include the timing of the software release, user accessibility, key developers' name recognition, etc.

Fourth, while all five software packages in this dissertation are written in C which is a procedural language, it would be interesting to examine software packages written in Object-Oriented languages, such as C++, C#, and Java. Lots of modern open source software projects are developed in Java, e.g. Hadoop, Solr, Lucene, etc. The high popularity of Java based open source software projects provides more open source subjects to analyze. Unlike C, Java based software packages have more hierarchical structures. For example, a network vertex can represent a package, a Java file, a class, or a method. How to choose the appropriate level of abstraction is the key challenge when a network is to be constructed out of the software source code.

The results of this dissertation can be used as a starting point to quantitatively analyze open source software architectural structures, and the development process of the software packages. More metrics and associated guidelines can be developed to help software developers adjust software development strategies in order to minimize the risks and costs of software development.

While our research focuses on a particular type of networks, i.e., open source software networks, our research methods, measures and the proposed network growth model may be applied to study other networks that exhibit similar features. For example, movie actor collaboration networks also possess such features as modularization, high cohesion, and low coupling, our research methods and the two-phase network growth model can potentially be used to analyze the actor collaboration networks.

REFERENCES

Adamic, L. A. "Zipf, Power-laws, and Pareto - a ranking tutorial." from http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

Albert, R. and A. L. Barabasi (2000). "Dynamics of complex systems: scaling laws for the period of boolean networks." Phys Rev Lett **84**(24): 5660-3.

Albert, R. and A. L. Barabasi (2000). "Topology of evolving networks: local events and universality." Phys Rev Lett **85**(24): 5234-7.

Albert, R. and A. L. Barabasi (2002). "Statistical mechanics of complex networks." Reviews of Modern Physics **74**(1): 47-97.

Albert, R., H. Jeong, et al. (2000). "Error and attack tolerance of complex networks." Nature **406**(6794): 378-82.

Amaral, L. A. N., A. Scala, et al. (2000). "Classes of small-world networks." Proceedings of the National Academy of Sciences of the United States of America **97**(21): 11149-11152.

Ancel Meyers, L., M. E. Newman, et al. (2003). "Applying network theory to epidemics: control measures for Mycoplasma pneumoniae outbreaks." Emerg Infect Dis **9**(2): 204-10.

Bagheri, E. and D. Gasevic (2011). "Assessing the maintainability of software product line feature models using structural metrics." Software Quality Journal **19**(3): 579-612.

Barabasi, A. L. and R. Albert (1999). "Emergence of scaling in random networks." Science **286**(5439): 509-512.

Barabasi, A. L., R. Albert, et al. (1999). "Mean-field theory for scale-free random networks." Physica A **272**(1-2): 173-187.

Barabasi, A. L. and E. Bonabeau (2003). "Scale-free networks." Sci Am **288**(5): 60-9.

Barabasi, A. L., H. Jeong, et al. (2002). "Evolution of the social network of scientific collaborations." Physica A **311**(3-4): 590-614.

Bilke, S. and C. Peterson (2001). "Topological properties of citation and metabolic networks." Phys Rev E Stat Nonlin Soft Matter Phys **64**(3 Pt 2): 036106.

Callaway, D. S., M. E. Newman, et al. (2000). "Network robustness and fragility: percolation on random graphs." Phys Rev Lett **85**(25): 5468-71.

Chhabra, J. K. and V. Gupta (2010). "A Survey of Dynamic Software Metrics." Journal of Computer Science and Technology **25**(5): 1016-1029.

Cimitile, A. and U. Decarlini (1991). "Reverse Engineering - Algorithms for Program Graph Production." Software-Practice & Experience **21**(5): 519-537.

Cohen, R., D. Ben-Avraham, et al. (2002). "Percolation critical exponents in scale-free networks." Phys Rev E Stat Nonlin Soft Matter Phys **66**(3 Pt 2A): 036113.

Cohen, R. and S. Havlin (2003). "Scale-free networks are ultrasmall." Phys Rev Lett

90(5): 058701.

Colazo, J. A. (2010). "Collaboration structure and performance in new software development: Findings from the study of open source projects." Int. J. Innov. Manage. International Journal of Innovation Management **14**(5): 735-758.

Darcy, D. P., C. F. Kemerer, et al. (2005). "The structural complexity of software: An experimental test." IEEE Transactions on Software Engineering **31**(11): 982-995.

Dhama, H. (1995). "Quantitative Models of Cohesion and Coupling in Software." Journal of Systems and Software **29**(1): 65-74.

Dorogovtsev, S. N. and J. F. F. Mendes (2001). "Scaling properties of scale-free evolving networks: Continuous approach." Physical Review E **6305**(5): -.

Durrett, R. (2010). "Some features of the spread of epidemics and information on a random graph." Proceedings of the National Academy of Sciences of the United States of America **107**(10): 4491-4498.

Eggemann, N. and S. D. Noble (2011). "The clustering coefficient of a scale-free random graph." Discrete Applied Mathematics **159**(10): 953-965.

Eichinger, F., D. Kramer, et al. (2010). "From source code to runtime behaviour: Software metrics help to select the computer architecture." Knowledge-Based Systems **23**(4): 343-349.

Erdos, P. and A. Renyi (1959). "On Random Graphs." Publicationes Mathematicae **6**: 290-297.

Erdos, P. and A. Renyi (1960). "On the Evolution of Random Graphs." Bulletin of the International Statistical Institute **38**(4): 343-347.

Erdos, P. and A. Renyi (1961). "On the strength of connectedness of a random graph." Acta Mathematica Scientia Hungary **12**: 261-267.

Etzkorn, L., C. Davis, et al. (1998). "A practical look at the lack of cohesion in methods metric." Journal of Object-Oriented Programming **11**(5): 27-34.

Faloutsos, C., P. Faloutsos, et al. (1999). On power law relationships of the Internet topology. Proceedings of ACM SIGCOMM. Cambridge, MA. **12:** 251-262.

Fothi, A., J. Nyeky-Gaizler, et al. (2003). "The structured complexity of object-oriented programs." Mathematical and Computer Modelling **38**(7-9): 815-827.

Frakes, W. B. and G. Succi (2001). "An industrial study of reuse, quality, and productivity." Journal of Systems and Software **57**(2): 99-106.

Goh, K. I., B. Kahng, et al. (2001). "Spectra and eigenvectors of scale-free networks." Phys Rev E Stat Nonlin Soft Matter Phys **64**(5 Pt 1): 051903.

Goh, K. I., B. Kahng, et al. (2001). "Universal behavior of load distribution in scale-free networks." Phys Rev Lett **87**(27 Pt 1): 278701.

Goh, K. I., D. S. Lee, et al. (2003). "Sandpile on scale-free networks." Phys Rev Lett **91**(14): 148701.

Goh, K. I., E. Oh, et al. (2002). "Classification of scale-free networks." Proc Natl Acad Sci U S A **99**(20): 12583-8.

Goh, K. I., E. Oh, et al. (2003). "Betweenness centrality correlation in social networks."

Phys Rev E Stat Nonlin Soft Matter Phys **67**(1 Pt 2): 017101.

Gondal, N. (2011). "The local and global structure of knowledge production in an emergent research field: An exponential random graph analysis." Social Networks **33**(1): 20-30.

Goodreau, S. M., J. A. Kitts, et al. (2009). "Birds of a Feather, or Friend of a Friend? Using Exponential Random Graph Models to Investigate Adolescent Social Networks." Demography **46**(1): 103-125.

Hahn, J., J. Y. Moon, et al. (2008). "Emergence of new project teams from open source software developer networks: Impact of prior collaboration ties." Inf. Syst. Res. Information Systems Research **19**(3): 369-391.

Harrison, R., L. G. Samaraweera, et al. (1996). "Comparing programming paradigms: An evaluation of functional and object-oriented programs." Software Engineering Journal **11**(4): 247-254.

Hu, J. Q., C. G. Guo, et al. (2005). "Stratus: A distributed Web Service Discovery Infrastructure based on double-overlay network." Web Technologies Research and Development - Apweb 2005 **3399**: 1027-1032.

Jacobson, I., G. Booch, et al. (1999). The Unified Software Development Process, Edison-Wesley.

Jin, E. M., M. Girvan, et al. (2001). "Structure of growing social networks." Phys Rev E Stat Nonlin Soft Matter Phys **64**(4 Pt 2): 046132.

Johnson, J. P. (2006). "Collaboration, peer review and open source software." Information Economics and Policy **18**(4): 477-497.

Kang, B. K. and J. M. Bieman (1999). "A quantitative framework for software restructuring." Journal of Software Maintenance-Research and Practice **11**(4): 245-284.

Kanmani, S., V. R. Uthariaraj, et al. (2004). "Measuring the Object-Oriented properties in small sized C++ programs - An empirical investigation." Product Focused Software Process Improvement **3009**: 185-202.

Kennel, E. B., M. S. Perry, et al. (1989). "Reliability and Single Point Failure Design Considerations in Thermionic Space Nuclear-Power Systems." Space Power **8**(1-2): 219-223.

Kim, J. H., K. I. Goh, et al. (2003). "Probabilistic prediction in scale-free networks: diameter changes." Phys Rev Lett **91**(5): 058701.

Krapivsky, P. L., S. Redner, et al. (2000). "Connectivity of growing random networks." Physical Review Letters **85**(21): 4629-4632.

Leino, K. R. M. and G. Nelson (2002). "Data abstraction and information hiding." ACM Transactions on Programming Languages and Systems **24**(5): 491-553.

Li, B. C. and D. Niu (2011). "Random Network Coding in Peer-to-Peer Networks: From Theory to Practice." Proceedings of the Ieee **99**(3): 513-523.

Li, H. A. and B. Li (2011). "A Pair of Coupling Metrics for Software Networks." Journal of Systems Science & Complexity **24**(1): 51-60.

Li, W. and S. Henry (1993). "Object-Oriented Metrics That Predict Maintainability." Journal of Systems and Software **23**(2): 111-122.

Li, Y., F. T. Zou, et al. (2004). "PWSD: A scalable Web service discovery architecture based on peer-to-peer overlay network." Advanced Web Technologies and Applications **3007**: 291-300.

Ma, Y. T., K. Q. He, et al. (2010). "A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems." Journal of Computer Science and Technology **25**(6): 1184-1201.

Milgram, S. (1967). "Small-World Problem." Psychology Today **1**(1): 61-67.

Milo, R., S. Itzkovitz, et al. (2004). "Superfamilies of evolved and designed networks." Science **303**(5663): 1538-42.

Milo, R., S. Shen-Orr, et al. (2002). "Network motifs: simple building blocks of complex networks." Science **298**(5594): 824-7.

Murphy, G., D. Notkin, et al. (1998). "An Empirical Study of Static Call Graph Extractors." ACM Transactions on Soft-ware Engineering and Methodology **7**: 158-191.

Newman, M. E. (2001). "Clustering and preferential attachment in growing networks." Phys Rev E Stat Nonlin Soft Matter Phys **64**(2 Pt 2): 025102.

Newman, M. E. (2001). "Scientific collaboration networks. I. Network construction and fundamental results." Phys Rev E Stat Nonlin Soft Matter Phys **64**(1 Pt 2): 016131.

Newman, M. E. (2001). "Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality." Phys Rev E Stat Nonlin Soft Matter Phys **64**(1 Pt 2): 016132.

Newman, M. E. (2001). "The structure of scientific collaboration networks." Proc Natl Acad Sci U S A **98**(2): 404-9.

Newman, M. E., S. Forrest, et al. (2002). "Email networks and the spread of computer viruses." Phys Rev E Stat Nonlin Soft Matter Phys **66**(3 Pt 2A): 035101.

Newman, M. E. and M. Girvan (2004). "Finding and evaluating community structure in networks." Phys Rev E Stat Nonlin Soft Matter Phys **69**(2 Pt 2): 026113.

Newman, M. E., S. H. Strogatz, et al. (2001). "Random graphs with arbitrary degree distributions and their applications." Phys Rev E Stat Nonlin Soft Matter Phys **64**(2 Pt 2): 026118.

Newman, M. E. and D. J. Watts (1999). "Scaling and percolation in the small-world network model." Phys Rev E Stat Phys Plasmas Fluids Relat Interdiscip Topics **60**(6 Pt B): 7332-42.

Newman, M. E., D. J. Watts, et al. (2002). "Random graph models of social networks." Proc Natl Acad Sci U S A **99 Suppl 1**: 2566-72.

Newman, M. E. J. (2001). The structure of scientific collaboration networks. Proceedings of the National Academy of Sciences of the United States of America.

Newman, M. E. J. (2003). "The structure and function of complex networks." SIAM

Review 45(2): 167-256.

Newman, M. E. J. (2004). "Analysis of weighted networks." Phys. Rev. E **70**: 056131.

Newman, M. E. J., S. H. Strogatz, et al. (2001). "Random graphs with arbitrary degree distributions and their applications." Physical Review E **6402**(2): -.

Newman, M. E. J., D. J. Watts, et al. (2002). "Random graph models of social networks." Proceedings of the National Academy of Sciences of the United States of America **99**: 2566-2572.

Ott, L. M. and J. M. Bieman (1998). "Program slices as an abstraction for cohesion measurement." Information and Software Technology **40**(11-12): 691-699.

Parnas, D. L. (1972). "Criteria to Be Used in Decomposing Systems into Modules." Communications of the ACM **15**(12): 1053-&.

Parnas, D. L. (1972). "Technique for Software Module Specification with Examples." Communications of the ACM **15**(5): 330-&.

Potanin, A., J. Noble, et al. (2005). "Scale-free geometry in OO programs." Communications of the ACM **48**: 99-103.

Ravasz, E. and A. L. Barabasi (2003). "Hierarchical organization in complex networks." Physical Review E **67**(2): -.

Rine, D. C. and R. M. Sonnemann (1998). "Investments in reusable software. A study of software reuse investment success factors." Journal of Systems and Software **41**(1): 17-32.

Sarkar, S., A. C. Kak, et al. (2008). "Metrics for measuring the quality of modularization of large-scale object-oriented software." Ieee Transactions on Software Engineering **34**(5): 700-720.

Schwartz, N., R. Cohen, et al. (2002). "Percolation in directed scale-free networks." Phys Rev E Stat Nonlin Soft Matter Phys **66**(1 Pt 2): 015104.

Shaw, S. (2003). Evidence of Scale-Free Topology and Dynamics in Gene Regulatory Networks. ISCA 12th International Conference on Intelligent and Adaptive Systems and Software Engineering.

Shin, Y., A. Meneely, et al. (2011). "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities." Ieee Transactions on Software Engineering **37**(6): 772-787.

Simpson, S. L., S. Hayasaka, et al. (2011). "Exponential Random Graph Modeling for Complex Brain Networks." Plos One **6**(5).

Singh, P. V. (2011). "The Small-World Effect: The Influence of Macro-Level Properties of Developer Collaboration Networks on Open-Source Project Success." ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY **20**(2): Article 6.

Taherkhani, A., A. Korhonen, et al. (2011). "Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms." Computer Journal **54**(7): 1049-1066.

Tonella, P. (2001). "Concept analysis for module restructuring." IEEE Transactions on

Software Engineering **27**(4): 351-363.

Tutte, W. T. (1984). Graph theory, Addison-Wesley Publishing Co.

Vazquez, A., M. Boguna, et al. (2003). "Topology and correlations in structured scale-free networks." Phys Rev E Stat Nonlin Soft Matter Phys **67**(4 Pt 2): 046111.

Wang, Y. H., C. M. Chung, et al. (2000). "The complexity measurement of software through program decomposition." Computer Systems Science and Engineering **15**(2): 127-134.

Watts, D. J. (1999). Small Worlds: The Dynamics of Networks between Order and Randomness. Cambridge, Cambridge Univ. Press.

Watts, D. J., P. S. Dodds, et al. (2002). "Identity and search in social networks." Science **296**(5571): 1302-5.

Watts, D. J. and S. H. Strogatz (1998). "Collective dynamics of 'small-world' networks." Nature **393**(6684): 440-442.

Wu, B. and A. D. Kshemkalyani (2008). "Modeling message propagation in random graph networks." Computer Communications **31**(17): 4138-4148.

Yook, S. H., H. Jeong, et al. (2001). "Weighted evolving networks." Phys Rev Lett **86**(25): 5835-8.

Zheng, X., D. Zeng, et al. (2008). "Analyzing open-source software systems as complex networks." Physica A: Statistical Mechanics and its Applications **387**(24): 6190-6200.