Hindawi
Mathematical Problems in Engineering
Volume 2022, Article ID 4283995, 11 pages
https://doi.org/10.1155/2022/4283995

Research Article

Topological Structure Analysis of Software Using Complex Network Theory

Xinxin Xu,[1] **Zengyou Zhang**,[1] **Yan Liang**,[2] **and Liya Wang**[1]

[1]*College of Artificial Intelligence, Zhejiang Industry & Trade Vocational College, Zhejiang 325003, China*
[2]*School of Business, Shanghai Jianqiao University, Shanghai 201306, China*

Correspondence should be addressed to Yan Liang; irisly1020@163.com

Received 24 January 2022; Revised 12 April 2022; Accepted 13 May 2022; Published 27 May 2022

Academic Editor: Chunlai Chai

Copyright © 2022 Xinxin Xu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Due to people's increasing dependence on software, the emergence of software defects will lead to serious consequences. And the essential cause of software defects is the increasing complexity of software. The premise of reducing software defects is to understand the software topology to ensure software quality. The software topology refers to the connection between the internal elements of the software, and it has become an important factor affecting the quality of the software. In this paper, we use complex network theory as a tool to analyze the software topology. Firstly, we extract the software structural information from the source code of the software system and abstract the extracted software structural information with software network theory. Secondly, the metrics widely used in complex networks are introduced to analyze the built software network. When tracking the values of these metrics in the software system, we have a deeper understanding of the software topology. These results provide a different dimension to understanding the software topology, which has important guiding significance for the subsequent understanding of the software and is also very useful for reducing software defects and ensuring software quality.

1. Introduction

Large-scale software systems are composed of countless small elements (class, process, method, etc.), and every tiny error may lead to catastrophic consequences, especially for projects with extremely high software reliability requirements. When the software system becomes more and more complex, how to recognize and measure the software system has become a matter of constant concern and urgent solution.

Some researchers have proposed to study existing software systems from the perspective of software structure. Software structure analysis [1–4] can help us understand the specific situation of the software system to carry out corresponding maintenance and upgrades according to the characteristics. At present, many achievements in the field of software structure analysis have been published. The main software structure analysis approaches are divided into traditional software structure measurement approaches and software structure measurement approaches based on complex networks.

Traditional software structure analysis focuses on analysis from a single module. For example, the McCabe method [5], the Halstead method, the C&K metrics proposed by Chidanber and Kemerer and the MOOD method proposed by Brito all describe the complexity of the software structure from different aspects but focus on the analysis of the local structure and properties of functional individuals of the software system. Therefore, traditional analysis approaches lack the measurement of the overall software structure. As a kind of complex system, the overall structure of the computer software system has a huge impact on its function, performance, and quality [6].

However, some researchers have introduced the theory of complex networks into software research [7–10]. By constructing software networks from software source code, they can use complex networks to understand, analyze, and control the system from a global perspective, rather than from a local perspective. Complex network theory provides us with a new way to understand the structure of software systems.

Based on a weighted network, Wang and Xiao [11] used the theory and technology of complex networks to explore the execution process of Linux. Trindade et al. [3] represented class-level software as Little House.

At present, there is still little research on software networks, and the existing research still has the following shortcomings: (1) Existing research is not accurate enough for the construction of a software network model. (2) In the existing research, the metrics used in the software network analysis and the data sets used in the experiments are not comprehensive enough. In this paper, we adopt the idea of interdisciplinary and propose the software topology analysis approach. Firstly, we extract the software structural information from the source code of the software system and abstract the extracted software structural information with software network theory. Secondly, the metrics widely used in complex networks are introduced to analyze the built software network. By analyzing a series of metrics widely used in complex networks, we can discover the underlying laws that, in the software system, only a few classes contain more important information and have a strong influence, while most other nodes have little influence. That is, developers can quickly locate software defects by finding key classes in the software system.

The rest of this paper is organized as follows. Section 2 introduces some preliminary knowledge with a focus on the framework of the proposed software topology analysis approach and the formal definition of the software network model. Section 3 illustrates our approach by analyzing the experimental results of 6 subject software systems. And we conclude this paper in Section 4.

2. Related Work

Many results of software system research have been reported in the past few years. These studies can be roughly classified into two groups, that is, approaches based on traditional software metrics and approaches based on complex networks. For the approaches based on traditional software metrics, they pay more attention to analyzing software from a single module. There are mainly the following approaches. The McCabe method [5] is mainly based on graph theory and program structure control theory and uses directed graphs to represent program control flow, thereby representing the complexity of the network according to the cyclic complexity in the graph. The programming complexity measured using McCabe's method mainly depends on the complexity of the structural control flow. The Halstead method measures the complexity of the software system by counting the number of operators and operands in the program. However, this method only considers the program data flow but does not consider the control flow, so it can not reflect the complexity of the program fundamentally. The C&K metric proposed by Chidanber and Kemerer is based on object-oriented metric theory, including six metrics: (1) the number of subclasses (the number of direct subclasses of a class); the number of weighted methods of the class; (2) the depth of the inheritance tree (if it is multiple inheritances, calculate the maximum depth from the node to the root of the tree); (3) the number of weighted methods of the class; (4) the degree of coupling between objects (when a class uses member variables or methods of other classes, the two classes are said to be coupled); (5) the number of responses of the class (the total number of out-of-class methods called by all methods in the class); (6) the lack of cohesion in the class method. The MOOD method proposed by Brito indirectly measures the inheritance, encapsulation, polymorphism, and coupling of object-oriented software systems. The traditional software structure measurement method describes the complexity of the software structure from different aspects, but it focuses on analyzing the local structure and properties of functional individuals (classes, procedures, methods, etc.) in the software system. Therefore, the traditional analysis methods lack the overall software structure measurement. However, the measurement method based on a single module cannot understand the software system from the perspective of the overall structure.

As a kind of complex software system, the overall structure of the system has a great impact on its function, performance, and quality. Therefore, compared with approaches based on traditional software metrics, approaches based on complex networks have great application potential. In this work, we mainly discuss research based on complex network analysis. Based on a weighted network, Wang and Xiao [11] used the theory and technology of complex networks to explore the execution process of Linux. They found that the weight distribution obeys the power-law distribution, and the process management component of Linux plays the most important role. Trindade et al. [3] represented class-level software as Little House. Based on Little House, they analyzed 81 versions of 6 software systems and found some software evolution patterns. Šubelj and Bajec [12] used an Associative Software Graph (ASG) to represent a class-level software system, where nodes represent classes and edges represent "inheritance," "composition," and "dependency" relationships between classes. Based on ASG, they calculated the number of communities, the modularity of the software network, and other network metrics such as clustering coefficient, average path length, and average degree. They then analyzed the correlation between these indicators and the number of defects in the software. They found that medium-sized systems with a community structure tended to have a greater probability of defects. Yang et al. [10] proposed an internal class network of the software system to represent class-level software systems. In a software network, a class is a node, and the calling relationship between the methods contained in each pair of classes constitutes an edge. Based on the software network, they propose a set of metrics to characterize the software network structure and use some machine learning algorithms to build a defect prediction model, and their final results are encouraging. Zakari et al. [13] proposed a software network at the statement level, where statements are nodes and execution trajectories between statements are edges. They calculated two centrality metrics (i.e., degree centrality and closeness centrality) for defect diagnosis based on a software network.

3. Preliminaries

In this section, we show the framework of our software topology analysis approach (see Figure 1). It mainly consists of four parts, ① to ④. The first two parts ① to ② are detailed in Sections 3.1 and 3.2, and the two parts ③ to ④ will be explained in Section 3.

3.1. Data Collection. Data collection is the first step in our approach. For the reliability of the results, we will select software that is widely used in software structure-related research. Thus, we conducted our study on 6 well-known open-source software written in Java from different fields and different scales: Ant https://ant.apache.org/, GWT Portlets http://code.google.com/p/gwtportlets/, jEdit http://jedit.org/index.php, JHotDraw https://sourceforge.net/projects/jhotdraw/, Maze https://sourceforge.net/projects/maze/, and Wro4j https://github.com/wro4j/wro4j. Ant is a tool that provides software automation construction functions; GWT Portlets is an open-source web framework for developing GWT (Google Web Toolkit) applications; jEdit is an open-source text editor written in Java; JHotDraw is an open-source drawing program developed based on Java; Maze is an open-source network file system; Wro4j is a web resource optimization tool.

Table 1 shows the description of the relevant metrics of the subject software system, such as the number of lines of code (LOC), the number of packages (#P), the number of classes (#C), the number of methods (#M), and the number of attributes (#A). These values are calculated based on the Java code listed in the "Directory" column, not the entire distribution of the corresponding software.

3.2. Software Network Model. After data collection, software structure extraction [14–16] is the next step in the construction of software network models. This step aims to extract various software elements (classes, interfaces, attributes, methods, local variables, etc.) and interactions (class inheritance, interface implementation, method calls, etc.).

Based on the results of software structure extraction, this paper introduces the Unweighted Directed Class Coupling Network (UDCCN). In this network, nodes represent class-level elements (classes, interfaces, etc.) in the software system, edges represent the coupling relationship between elements, and the direction of the edges represents the coupling direction between elements. In UDCCN, we considered 7 coupling types:

(i) Inheritance relationship (INR): if class A inherits from another class B by using the keyword "extends."

(ii) Implementation relationship (IMR): if class A implements interface B by using the keyword "implements."

(iii) Parameter relationship (PAR): if one of the methods of class A has at least one parameter of class B type.

(iv) Global variable relation (GVR): if class A has at least one attribute with the type of class B.

(v) Local variable relationship (LVR): if a local variable with the type of class B is declared in a method of class A.

(vi) Method call relationship (MCR): if one of the methods of class A calls a method on an object of class B.

(vii) Return type relationship (RTR): if one of the methods of class A has a return type of class B.

If the above seven relationships exist between elements, we will generate a directed edge in the UDCCN network to describe this coupling relationship. UDCCN is an unweighted directed graph, which is defined as follows:

$$\text{UDCCD} = (V, L), \quad n \in V, l \in L, l = \langle n_i, n_j \rangle, \tag{1}$$

where n represents the class or interface in the software system and l represents the coupling between the node (class i) and the node (class j). And the adjacency matrix ψ_{ij} of UDCCN encodes the coupling between every pair of classes:

$$\psi_{ij} = \begin{cases} 1, & \langle n_i, n_j \rangle \in L, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

That is a $|V| \times |V|$ matrix, where $|V|$ returns the number of classes. ψ_{ij} is the weight assigned to the link $\langle n_i, n_j \rangle \langle n_i, n_j \rangle$; if $\langle n_i, n_j \rangle \in L$, then $\psi_{ij} = 1 \psi_{ij} = 1$; otherwise $\psi_{ij} = 0$.

To explain UDCCN more clearly, Figure 2(a) shows an exemplary Java code snippet. For this code segment, Figure 2(b) shows its corresponding UDCCN. As shown in Figure 2(b), the coupling relationship between classes in the Java code fragment in Figure 2(a) includes inheritance relationship, implementation relationship, parameter relationship, global variable relation, return type relationship, and method call relationship.

3.3. Complex Network Statistical Metrics. We use a software network model to abstract the relationships between elements in the software system, which provides a new perspective for the research of software engineering. Complex networks have gradually become one of the focuses of research. Particularly with the discovery of features such as "small world" and "scale-free," scientists have set off an upsurge in studying complex networks [17–19], covering many fields such as physics, mathematics, and biology. Therefore, we can draw on the abovementioned complex network statistical metrics to reveal the knowledge related to the topology of the software network [20, 21].

3.3.1. Network Centrality. The metrics of network centrality are mainly to find nodes that have important roles in complex networks and reflect the importance of node locations. These metrics include betweenness centrality, degree centrality, and closeness centrality.

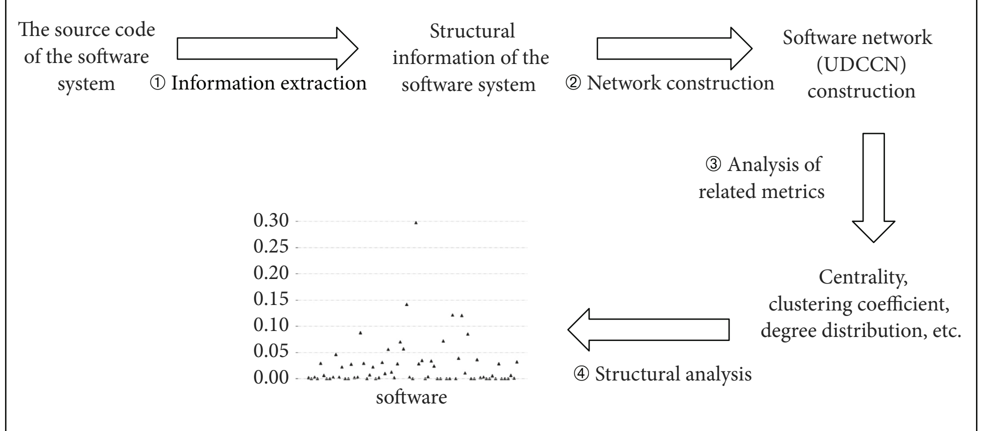

Figure 1: The basic framework of the software topology analysis approach.

Table 1: The description of the subject software system.

System	Version	Directory	LOC	#P	#C(#E)	#M/#A
Ant	1.6.1	Src/main	81515	67	900	7691/4167
GWT Portlets	0.9.5beta	Src	8501	10	145	1145/424
jEdit	5.1.0	Src	112492	41	1082 (9)	7601/4085
JHotDraw	6.0b.1	Src	28330	30	544	5205/865
Maze	1	Src	8881	6	63 (6)	563/284
Wro4j	1.6.3	Src	33736	30	567 (9)	3256/1274

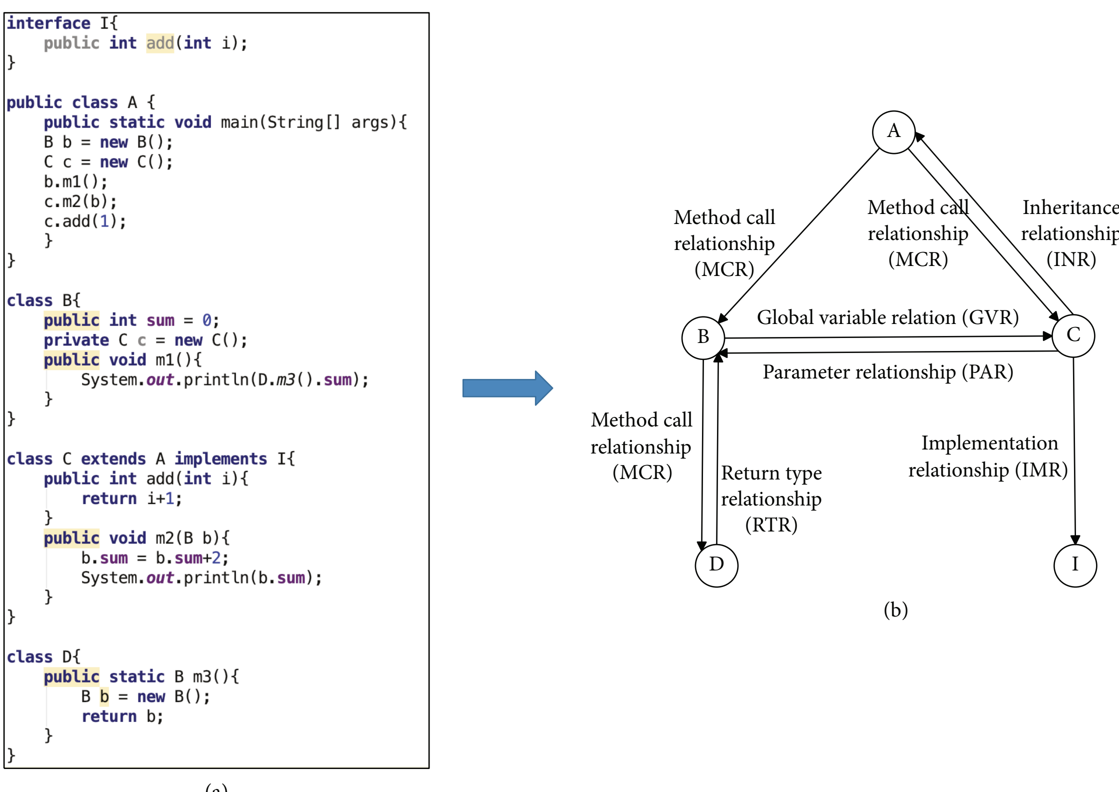

Figure 2: An illustrative exemplary Java code snippet and corresponding UDCCN.

(1) Betweenness Centrality. The betweenness is a parameter that cannot be bypassed when studying complex networks. This parameter reflects the influence and importance of nodes. To understand the definition more intuitively, the following formula is given:

$$g(v) = \sum_{s \neq v \neq t} \frac{\phi_{st}(v)}{\phi_{st}}, \quad (3)$$

where ϕ_{st} represents the number of shortest paths from node s to node t and $\phi_{st}(v)$ represents the number of all paths from node s to node t and through node v in the network. Betweenness centrality [22] reflects the dependence between class nodes. The higher the betweenness centrality of class nodes, the stronger the importance to the software network.

(2) Degree Centrality. In complex network analysis, degree centrality is the most direct metric to describe the importance of nodes. The higher the degree centrality of a node is, the more important the node is in the network. Conversely, if the degree centrality of a node in the network is closer to 0, it means that the node has less contact with other nodes.

(3) Closeness Centrality. In complex network metrics, closeness centrality refers to how close a node in the network is to other nodes. If a node's closeness centrality is higher, then it is closer to other nodes. The closeness centrality of a node is the reciprocal of the average value of the shortest path length between the node and all other nodes in the network, which can be defined as

$$C(i) = \frac{n}{\sum_j d(j,i)}, \quad (4)$$

where $d(j,i)$ represents the distance from node i to node j and n represents the number of nodes.

3.3.2. Clustering Coefficient. In graph theory, the clustering coefficient is used to measure the degree of clustering of nodes in the graph. It is often used to describe the clustering characteristics of the network, indicating the closeness of a node with surrounding nodes [23]. The clustering coefficient of nodes in the network mainly refers to the ratio of the number of connections between the node and adjacent nodes to the maximum number of edges that can be connected between these adjacent nodes. The clustering coefficient C_i of the node i can be defined as

$$C_i = \frac{2e_i}{k_i(k_i - 1)} = \frac{\sum_{j,m} a_{ij} a_{im} a_{mj}}{k_i(k_i - 1)}, \quad (5)$$

where e_i indicates that the value of the clustering coefficient C_i of the node i is equal to the number of edges connected by the neighbor node and $k_i(k_i - 1)/2$ represents the maximum number of edges that may exist. The clustering coefficient of the network is the average of the clustering coefficients of all nodes in the network, which is

$$C = \langle C_i \rangle = \frac{1}{N} \sum_{i \in V} C_i, \quad (6)$$

where N is the number of nodes in the network, which indicates the aggregation trend of nodes in the network and reflects the local characteristics of the network.

3.3.3. Degree Distribution. The degree distribution reflects the most basic characteristics of the complex network topology. The degree of a node in the network refers to the number of nodes adjacent to the node, that is, the number of edges connecting the node. The greater the degree of the node, the more the connections between the nodes and the more important the node in the network. The degree distribution $P(k)$ refers to the probability that the degree of an arbitrarily selected node in the network is exactly k. When the degree distribution of the network satisfies the power rate distribution, it can be defined as $P(k) \sim k^{-r}$, and then the network is a scale-free network.

3.3.4. Average Shortest Path Length. The average shortest path length of the network [24] is defined as the average of the shortest path length between any two nodes in the network. The average shortest path length of the network can be defined as

$$L = \frac{2}{(N(N-1))} \sum_{i \neq j} d_{ij}, \quad (7)$$

where d_{ij} represents the number of edges on the shortest path connecting two nodes i and j in the network and N represents the number of nodes in the network.

3.3.5. Assortativity Coefficient. It is found that many observable networks have mixing patterns in degree, that is, assortative mixing or disassortative mixing. The so-called assortative mixing means that nodes with high degrees are often connected with other nodes with high degrees, and nodes with low degrees are likely to be connected with other nodes with low degrees. Disassortative mixing means that low-degree vertices are more likely to be connected to high-degree vertices, and vice versa.

The assortativity coefficient is often used to quantify the degree of assortative mixing, it is a degree-based Pearson correlation coefficient, and the calculation formula can be expressed as

$$ac = \frac{\sum_{y,z} yz\left(e_{yz} = m_y n_z\right)}{\sigma_y \sigma_z}, \quad (8)$$

where e_{yz} represents the ratio of the node with a degree value of y in the network and the number of its edges to the total number of all edges, $m_y = \sum_y e_{xy}$, $n_z = \sum_z e_{yz}$, $\sigma_y = \sqrt{E(y^2) - E^2(y)}$, and $\sigma_z = \sqrt{E(z^2) - E^2(z)}$. If ac is less than 0, it means that the network is disassortative, while ac being greater than 0 denotes an assortative mixing network.

3.3.6. Structural Holes. Structural hole theory [25] is a new theory in interpersonal network theory, which mainly describes the gaps in social networks. In the social network, an

individual directly finds contact with some individuals but does not have direct contact with other individuals. That is, there are holes in this social network.

If there is no direct connection between the two and the connection can only be formed through a third party, then the acting third party occupies a structural hole in the relationship network. The structural hole is for the third party. If there are structural holes in the network, the third party that connects two actors that are not directly connected has an information advantage and control advantage.

Generally, the effective size metric in structural hole theory is used to measure the network. This metric mainly describes the effectiveness of the node's self-network. Formally, the effective size of a node, expressed as $e(u)$, is defined as follows:

$$e(u) = \sum_{v \in N(u)/\{u\}} \left(1 - \sum_{w \in N(v)} P_{uw} m_{vw}\right), \tag{9}$$

where $N(u)$ is the set of neighbor nodes of u, P_{uw} is the normalized mutual weight of the (directed or undirected) edge connecting u and v, and m_{vw} is the mutual weight of the connecting node v to the node w divided by the v node's maximum connection edge weight with its neighbor nodes. Mutual weight refers to the sum of edge weights connecting node u and node v (in the case of a weightless network, the default edge weight is 1).

4. Topological Structure Analysis

In this section, for the illustration purpose, complex network statistical metrics mentioned above are used to study the software network topology of the subject software.

4.1. Topological Structure Analysis of Network Centrality

4.1.1. Betweenness Centrality. Betweenness centrality is a measure of graph centrality based on the shortest path, generally used to check whether a node is in an important position in the graph. As shown in Figure 3, we found that, in the software network of almost all subject software systems, the betweenness centrality of nearly 90% of the classes is distributed below 0.05, indicating that only 5% of the classes are in an important position in the software system, which has a strong impact on the realization of the software system function, and most other nodes have little influence.

In the actual development process, the calls between classes are usually a call chain, and important classes frequently call other classes or are frequently called by other classes. For example, the key class is usually called frequently by other classes in the software system to complete the corresponding function. Therefore, analyzing the betweenness centrality can provide greater help in identifying the key classes of the software system.

4.1.2. Degree Centrality. In complex network analysis, degree centrality is the most direct metric to describe the importance of nodes. The higher the degree centrality of a node is, the more important the node is in the network. Conversely, if the degree centrality of a node in the network is closer to 0, it means that the node has less contact with other nodes.

As shown in Figure 4, the degree centrality of the class nodes in the software network of the six subject software systems is mostly close to 0, while a few are between 0.01 and 0.05. It shows that only a small number of classes are closely connected with other classes and have a relatively strong influence, while most of the classes are not very influential.

In the actual software system, only a few classes will frequently call other classes or be frequently called by other classes. Usually in software development, if this class frequently calls other classes or is frequently called by other classes, it means that this class has a higher status in the software system, that is, the key class. How to find the key classes is of great importance to software cost prediction. If we ignore the importance of key classes, we will underestimate the complexity and cost of the software system to be developed, which may cause great losses to the company.

4.1.3. Closeness Centrality. Closeness centrality reflects the closeness between a node and other nodes in the network. If a node is very close to other nodes, then it does not need to rely on other nodes when transmitting information, indicating that this node is very important. When we calculated the closeness centrality of the six subject software systems, we found that, in the four software systems of Ant, jEdit, JHotDraw, and Wro4j, the closeness centrality of most nodes is close to 0. In the software systems of GWT Portlets, the closeness centrality of nodes is almost evenly distributed between 0 and 0.25. And in the software system Maze, the closeness centrality of most nodes is between 0.1 and 0.2 (see Figure 5).

If the closeness centrality of the node is 0, it means that there are a few isolated nodes in the software system, and these isolated nodes do not have any connection with other nodes. And the closer the value is to 1.0, the higher the closeness of the node is. Therefore, the greater the closeness centrality of a class node is, the closer the node is likely to be connected with all other class nodes. It also shows that the location of these class nodes has the best view of the network and can perceive the dynamics of the entire software network and the direction of information circulation. From the perspective of the structure of the software network, in general, the key classes are closely related to other class nodes; that is, the key class can usually get a higher value of closeness centrality.

4.2. Topological Structure Analysis of Clustering Coefficient. The clustering coefficient of a node indicates how interconnected its adjacent nodes are. The clustering coefficient distribution of each node in the software network of the six subject software systems is shown in Figure 6. The clustering coefficients of most class nodes in Ant, jEdit, JHotDraw, and Wro4j are less than 0.5, and only a few nodes have high clustering coefficients, which are nodes with high clustering degrees in the software network. For the software GWT

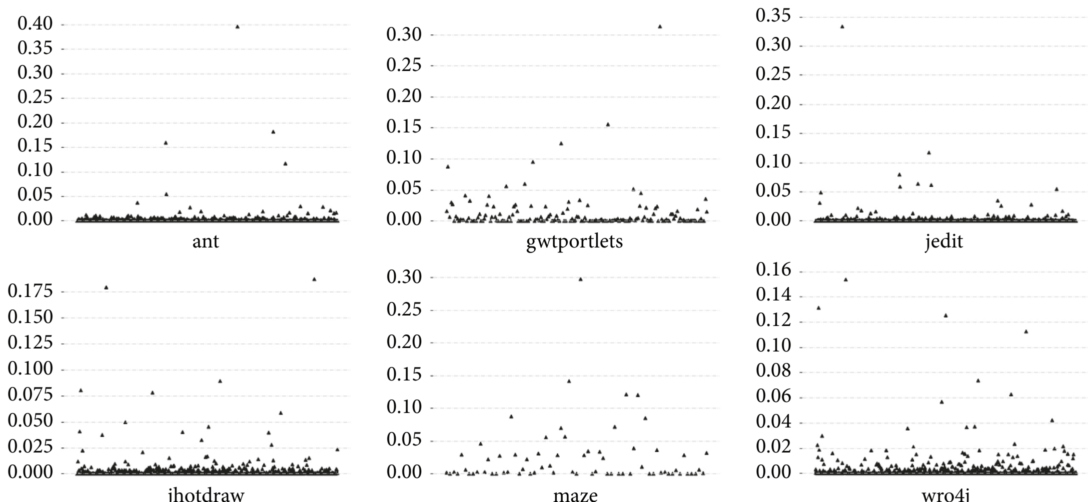

Figure 3: The distribution of betweenness centrality.

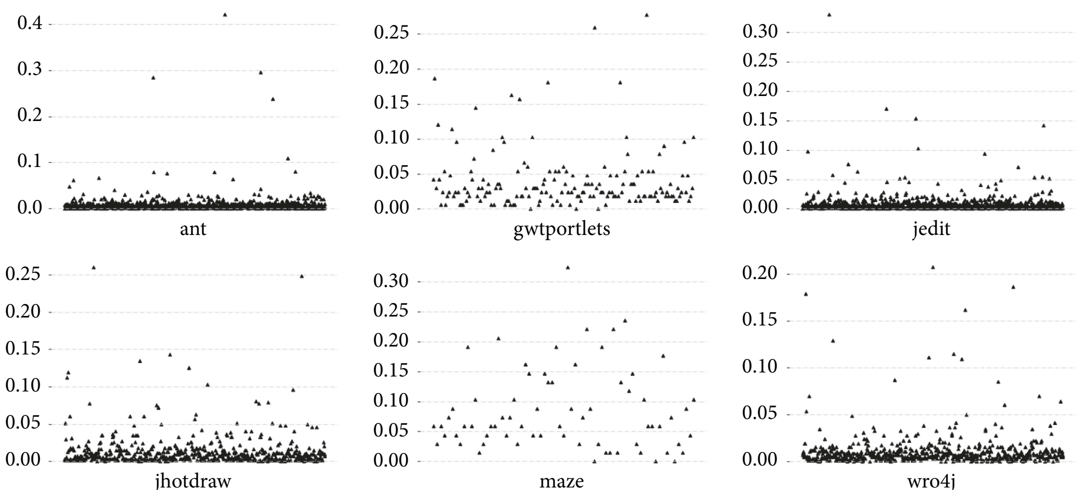

Figure 4: The distribution of degree centrality.

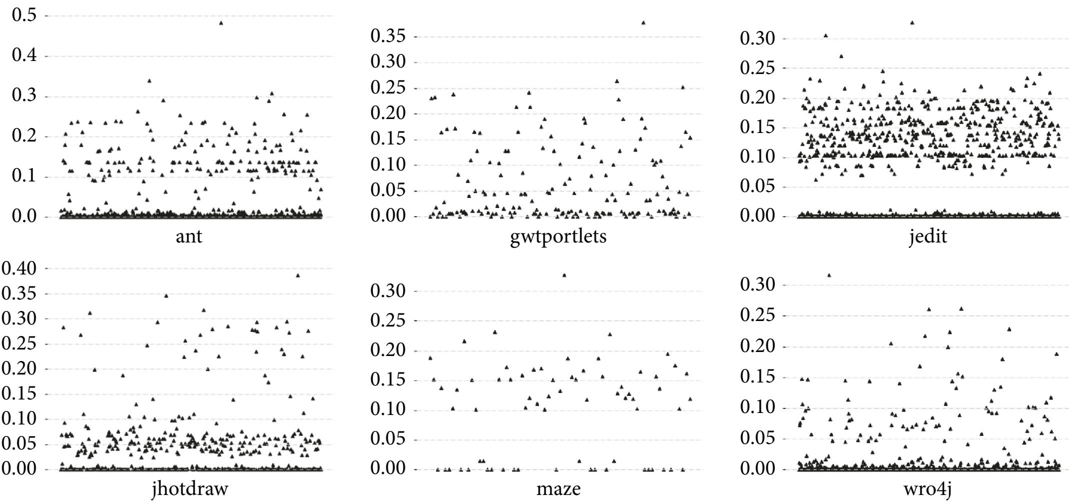

Figure 5: The distribution of closeness centrality.

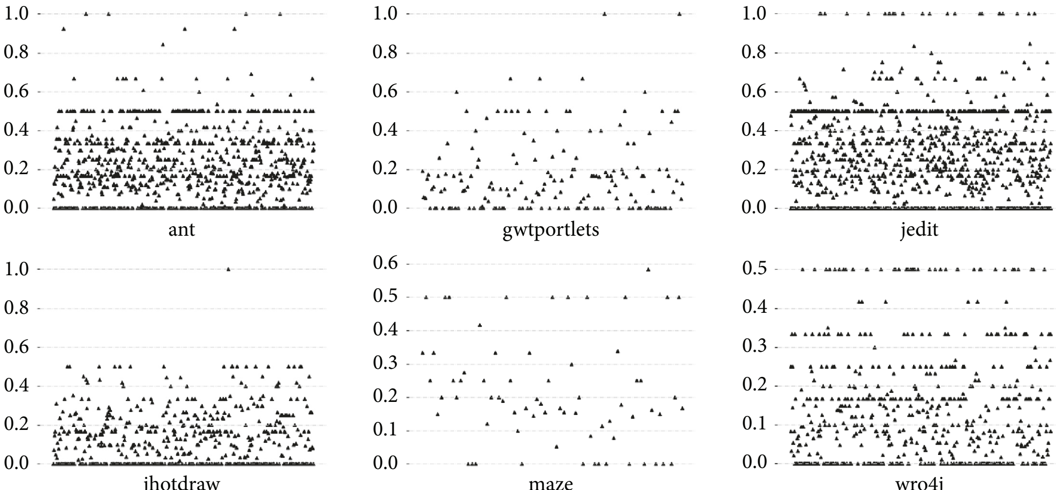

Figure 6: The distribution of clustering coefficient.

Portlets and Maze, although the nodes are relatively evenly distributed in the graph, for the part of the graph with high clustering coefficients, the number of class nodes is still relatively small. The larger the clustering coefficient of a node, the higher the degree of relationship between the nodes around the node and the higher the clustering degree of the group, and the highest value is 1, indicating that all the points around a node are related.

For the actual software system, only a few class nodes will have a relatively high clustering coefficient; that is, only a few classes will use other classes more or are used more by other classes. This is in line with the characteristics of key classes of software systems. In other words, analyzing the clustering coefficient of a software network is also helpful to identify key classes in the software system.

4.3. Topological Structure Analysis of Degree Distribution. In the study of graphs and networks, the degree of a node in a network is the number of connections it has to other nodes, and the degree distribution is the probability distribution of these degrees over the whole network. The degree distribution of nodes in the software network of the six subject software systems is shown in Figure 7. The horizontal axis in the figure is degrees, and the vertical axis is the number of nodes. It can be seen from the figure that as the degree becomes larger, the number of nodes declines. And when the software network has more nodes, this trend becomes more obvious. In software Ant, jEdit, JHotDraw, and Wro4j, this trend is more obvious than in software networks with fewer nodes. It can be observed from the figure that the number of nodes with a degree less than 10 accounts for almost 90% of the nodes in the software network, and the number of nodes with a degree greater than 50 is almost zero.

In a software network, most nodes are only connected to a few nodes, while a few nodes are connected to most of the nodes, which is in line with the typical characteristics of a scale-free network. Therefore, in the software system, we can find that most of the classes only call a few classes or are called by a few classes, and only a few classes call other classes or are called by a large number of classes.

4.4. Topological Structure Analysis of Average Shortest Path Length. The average shortest path length is a concept in the network topology that is defined as the average number of steps along the shortest paths for all possible pairs of network nodes. It is a measure of the efficiency of information or mass transport on a network. It can be seen from Table 2 that although the size of the subject software is different, the distance between nodes is stable at about 3. When calculating the average shortest path length, we found that the maximum value is 3.379 and the minimum value is 2.806. Therefore, the software network conforms to the "small world" effect in the complex network. Research shows that, in reality, the number of nodes in many networks is very large, but the average shortest path length of the entire network is relatively small, such as the World Wide Web, so formal networks generally have the characteristics of "small world" in complex networks.

4.5. Topological Structure Analysis of Assortativity Coefficient. Assortative mixing is a preference for a network's nodes to attach to others that are similar in some way. According to the calculation formula (8), it can be found from Table 3 that the calculated assortativity coefficients of 6 subject software systems are all negative, indicating that these software systems have a disassortative mixing network. That is to say, in the software network, nodes with high degrees and nodes with low degrees have a relatively high connection probability. And it means that key classes with a high frequency of use are usually related to classes with a low frequency of use, instead of being related to each other.

4.6. Topological Structure Analysis of Structural Holes. In the structural hole theory, the larger the effective size, the greater the effectiveness of the node. As shown in Figure 8, we can find that, in the software system, the effective size of most

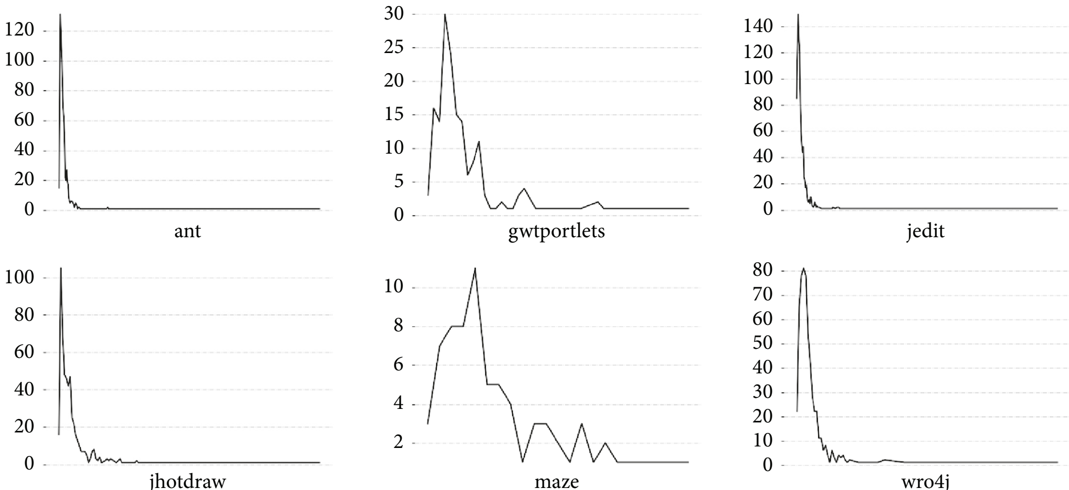

Figure 7: The distribution of degree distribution.

Table 2: Average shortest path length of software network.

Software system	Ant	GWT Portlets	jEdit	Maze	JHotDraw	Wro4j
Average shortest path length	3.178	3.072	3.290	2.806	3.235	3.379

Table 3: Assortativity coefficient of software network.

Software system	Ant	GWT Portlets	jEdit	Maze	JHotDraw	Wro4j
Assortativity coefficient	−0.126	−0.098	−0.152	−0.174	−0.165	−0.055

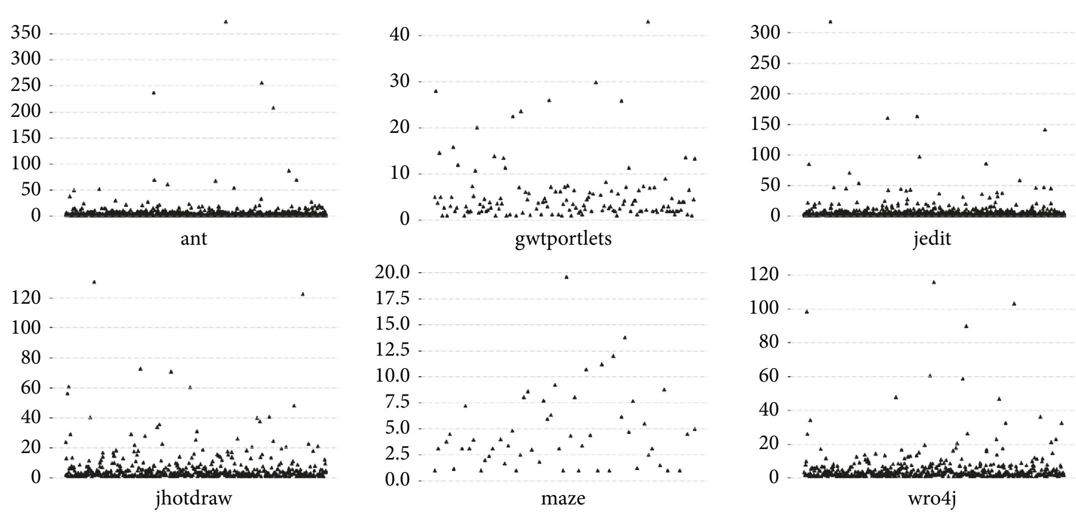

Figure 8: The distribution of effective size in structural hole theory.

nodes is between 0 and 20, while only a few nodes are above 20. Therefore, for the software network, only a few nodes are very important in the software system. Compared with other nodes, these nodes have information advantages and control advantages.

From the perspective of structural hole theory, some key classes in software systems usually act as a bridge in class calls. For example, functional aggregation classes are usually shown as a bridge of some single tool classes in a software network, and there is no direct connection between tool classes. Therefore, in the effective size metric, the value of the key class is larger than that of other common classes.

5. Threats to Validity

In this study, we obtained several important results about the software topology from our experiments. However, potential threats to our jobs remain. In our empirical research, we use 6 software systems of different scales as the research objects, all of which are widely used in the research of software

engineering. However, since the results obtained by these 6 software systems may not be so common, we hope to continue to do empirical research on more software systems to further evaluate their effectiveness.

The software systems we use for empirical research are all developed based on Java. Java is one of the most widely used programming languages. The software developed in Java has a clear structure, and the components in the software system are easier to extract. However, since there is no empirical research on software systems developed in other languages, this may affect the final results. We hope to continue empirical research on software systems developed in other languages for further evaluation of their validity, which will be important work for us in the future.

6. Conclusions and Future Work

In this paper, we proposed an approach to study the topological structure of software using the tool of complex network theory. For illustration, we conducted case studies on 6 software systems. Firstly, the software structure information is extracted from the source code of the software system, and the Unweighted Directed Class Coupling Network model is constructed based on this structure information. Secondly, several aspects of these software networks are studied by using the parameters widely used in complex network theory.

Through the analysis of software structure, we concluded that software network has significant characteristics of "small world" and "scale-free." The important structural features in software network topology help us to provide valuable insights and different dimensions for our understanding of software systems. Through the analysis of the experimental results, we found that only a few classes in the software are key classes, which play a great role in the function realization of the software. After finding the key classes, we can make a series of optimizations, such as the prediction and positioning of software defects.

There are a few areas that could be explored in future research: (1) investigating more software networks to validate the proposed approach, (2) investigating systems written in other languages to validate the proposed approach, and (3) using the parameters in other theories to study the software from different angles.

Data Availability

All data used during the study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] M. Kitsak, L. K. Gallos, S. Havlin et al., "Identification of influential spreaders in complex networks," *Nature Physics*, vol. 6, no. 11, pp. 888–893, 2010.

[2] P. Weifeng, L. Bing, and L. Jing, "Multi-granularity evolution analysis of software using complex network theory," *Journal of Systems Science and Complexity*, vol. 24, no. 6, pp. 1068–1082, 2011.

[3] R. P. F. Trindade, T. S. Orfanó, K. A. M. Ferreira, and F. Elizabeth, "The dance of classes-A stochastic model for software structure evolution," in *Proceedings of the 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, pp. 22–28, IEEE, Buenos Aires, Argentina, May 2017.

[4] P. Weifeng and C. Chunlai, "Measuring software stability based on complex networks in software," *Cluster Computing*, vol. 22, no. s2, pp. 2589–2598, 2019.

[5] T. J. Mccabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 2006.

[6] Y. Kamei, E. Shihab, B. Adams, and E. H. Ahmed, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[7] P. Weifeng, "Applying complex network theory to software structure analysis," *Engineering and Technology*, vol. 60, pp. 1636–1642, World Academy of Science, 2011.

[8] W. Y. Pan, H. Jiang, and Z. Yunfang, "Measuring software modularity based on software networks," *Entropy*, vol. 21, no. 4, 344 pages, 2019.

[9] G. Concas, M. Marchesi, C. Monni, and O. Matteo, "Software quality and community structure in java software networks," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 7, pp. 1063–1096, 2017.

[10] Y. Yang, J. Ai, X. Li, and W. E. Wong, "MHCP model for quality evaluation for software structure based on software complex network," in *Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE 2016)*, pp. 298–308, Ottawa, Canada, October 2016.

[11] H. Wang and G. Xiao, "Analysis of the runtime Linux operating system as a complex weighted network," in *Proceedings of the 2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pp. 7–11, IEEE, Kunming, China, November 2016.

[12] L. Šubelj and M. Bajec, "Software systems through complex networks science: review, analysis and applications," in *Proceedings of the International Workshop on Software Mining*, pp. 9–16, Singapore, September 2012.

[13] A. Zakari, S. P. Lee, and C. Y. Chong, "Simultaneous localization of software faults based on complex network theory," *IEEE Access*, vol. 6, p. 1, 2018.

[14] W. Pan, H. Ming, Z. Yang, and T. Wang, "Comments on "using k-core decomposition on class dependency networks to improve bug prediction model's practical performance"," *IEEE Transactions on Software Engineering*, vol. 1, 2022.

[15] L. Hao, W. Tian, P. Weifeng, C. Pengyu, and W. Jiale, "Mining key classes in java projects by examining a very small number of classes: a complex network-based approach," *IEEE Access*, vol. 9, pp. 28076–28088, 2021.

[16] X. Du, T. Wang, L. Wang et al., "CoreBug: improving effort-aware bug prediction in software systems using generalized *k*-core decomposition in class dependency networks," *Axioms*, vol. 11, no. 5, 205 pages, 2022.

[17] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[18] D. Hylandwood, "Scale-free nature of java software package, class and method collaboration graphs," 2006, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.2085&rep=rep1&type=pdf.

[19] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in OO programs," *Communications of the ACM*, vol. 48, no. 5, pp. 99–103, 2005.

[20] S. H. Strogatz, "Exploring complex networks," *Nature*, vol. 410, no. 6825, pp. 268–276, 2001.

[21] J. Xu, "Topological structure and analysis of interconnection networks[J]," *Springer Berlin*, vol. 7, no. 2-3, pp. 969-970, 2001.

[22] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[23] G. Sabidussi, "The centrality index of a graph," *Psychometrika*, vol. 31, no. 4, pp. 581–603, 1966.

[24] G. Mao and N. Zhang, "Analysis of average shortest-path length of scale-free network," *Journal of Applied Mathematics*, vol. 2013, Article ID 865643, 5 pages, 2013.

[25] S. Goyal and F. Vega-Redondo, "Structural holes in social networks," *Journal of Economic Theory*, vol. 137, no. 1, pp. 460–492, 2007.