IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 17, 2018, accepted May 17, 2018, date of publication May 29, 2018, date of current version June 29, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2840838

Approach to Mine the Modularity of Software Network Based on the Most Vital Nodes

BING ZHANG[1], GUOYAN HUANG[1], ZHANGQI ZHENG[1], JIADONG REN[1], AND CHANGZHEN HU[2]
[1]School of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China
[2]Beijing Key Laboratory of Software Security Engineering Technique, Beijing Institute of Technology, Beijing 100081, China

Corresponding author: Guoyan Huang (hgy@ysu.edu.cn)

This work was supported in part by the National Key R&D Program of China under Grant 2016YFB0800700, in part by the National Natural Science Foundation of China under Grants 61772451, 61772449, 61472341, and 61572420, in part by the Natural Science Foundation of Hebei Province, China, under Grant F2016203330, and in part by the Doctoral Foundation Program of Yanshan University under Grant BL18012.

ABSTRACT Analysis on the modularity of software network plays a critical role in the detection of software vulnerabilities and in the improvement of software stability, reliability, and robustness. This paper intends to propose a method based on the most vital nodes to analyze the modularity of software of different versions. To this end, it first tracked software dynamic execution traces to build a *dynamic software network model*, and then it mapped these traces to a complex network of a *dynamic invoke software network*. Second, it identified the most vital nodes in two steps, namely the *calInitialWeightForNode* and *calFinalWeightForNode* in order to compute the initial weights and the final weights of nodes iteratively. Third, it mined *top-k* nodes as the original communities to create a framework for detecting new community, and expanded these nodes to the community by the algorithm *expandTheCommunity* and evaluated the results with criterion Q. Finally, it calculated the modularity of software of different versions. Experimental results show that the most vital nodes are reasonable to be taken in comparison with other ranking measures, and that the analysis approach to the modularity of software network is effective in evaluating software community structure, and can help assist the developer to refactor the software and improve the software quality.

INDEX TERMS Software network, vital nodes, power-law, modularity.

I. INTRODUCTION

As software functional design is diversifying, software structure is also becoming more and more complex and difficult to be understood [1]. It is a significant research challenge to understand software evolution models [2], [3] and to improve software maintenance [4]. On the basis of these, the understanding of software networks structure has become increasingly important recently, especially internal characteristics of software.

The characteristics of software shown in the software execution process [5], [6] can help us understand its structure. During the execution process, functions carry most of the feature characteristics and the topology information, which play critical roles in maintaining the stability and robustness of the software system. Zhou and Wang [7] and Zemanová *et al.* [8] pointed out that mechanisms such as cascading and spreading could be highly affected by a tiny fraction of key nodes. Freeman [9] adopted betweenness to evaluate the importance of nodes, and pointed out that nodes with larger betweenness may be more important than others in software network. However, it should be noted that it is time-consuming to calculate the betweenness of each node. Callaw *et al.* [10] took node degree to measure the importance of nodes and those with larger degree are regarded as key nodes. Although the node which connects two communities in a network does not have a large degree, it is still an important node because if it is removed the communities will not get connected. In other words, the method is less relevant to global structure information. Kitsak *et al.* [11] realized that the position of nodes is important in global network, and they proposed k-shell decomposition analysis to obtain the ranking index of nodes. Although the k-shell decomposition presents more accurate results than betweenness and degree, it fails to rank the spreaders in the same k-shell index. Bae and Kim [12] proposed a novel measure called coreness centrality to estimate the spreading influence of a node in a network in use of the k-shell indices of its neighbors. The approach was based on the idea that a powerful spreader has more connections to other nodes in the core of network. It pointed out that the number of a node's neighbors has a great influence on its spreading ability. Also, Liu *et al.* [13] presented an improved method to generate the ranking list to evaluate the node spreading influence, taking into account the shortest distance between a target node and node set of the highest k-core value. In addition, they also took a new perspective to understand the relationship between the k-shell location and the nodes shortest distance to the network core. All this research shows that modeling functions as nodes is attracting attention to the analysis of network. However, it is unreasonable to just regard one node as an individual unit despite of the relationship between nodes and the global network. It is to say that network should be considered as a whole in practical application because nodes may interact with each other.

In recent years, the detection and analysis of community structure in software networks have become a research hotspot in many applications [14]. It is thus necessary to understand the global structure of software network. Generally speaking, communities are groups of nodes that are densely interconnected but only sparsely connected with the rest of the network [15]. So far, many algorithms have been proposed to detect these communities, of which the two classic algorithms, namely the spectral bisection algorithm and the Kernighan-Lin algorithm, have improved the initial division of network by optimizing the number of edges in communities in use of a greedy strategy [16]. Newman [17] provided a method by adapting the Laplacian spectral partitioning method to perform community inference and proved it was better than the previous methods. Lin *et al.* [18] studied the problem in detecting communities in incomplete information network with missing edges. They first adopted a distance metric to reproduce the link-based distance between nodes from the observed edges in local information regions. Then they used the distance metric to estimate the distance between any pair of nodes in the network. In recent years, various community detecting algorithms have been proposed on the basis of modularity [19]. Kim *et al.* [20] proposed a generalized form of modularity to identify community structure in directed networks and also provided a model as a benchmark structure to testify its feasibility.

By means of complex network analysis, there were several discoveries in the node-based researches on the community structure of software network in the past years. Zanjani and Darooneh [21] presented an alternate method in order to find the communities in a complex network. They introduced two concepts in complex networks, namely the seed of the community and the absorption power of the seed. They discovered the seeds and developed them by their absorption power to achieve the communities. It was proved that this algorithm was faster and more efficient than some recent algorithms. Chen *et al.* [22] proposed a community detecting algorithm of which the main strategy was to find an initial partial community from a node with maximal node strength and to add tight nodes to expand the partial community. However, most of the researches ignored the dependency among nodes in the process of software dynamic execution. In most cases, studies on static software structure cannot reflect well the interrelation of software network.

Based on the researches above, we examined specifically the information of the dynamic execution process and the evolution of community in software network based on the most vital nodes. According to the information of multiple executions, we built the DSNM to show the structure of software. We put forward CIWN and CFMN methods to mine the most vital nodes according to its initial weight and the construction of its neighbors in the mapped network of DISN. Further, we proposed an algorithm ETC to detect the community of software based on the most vital nodes and analyze the evolution of software accordingly.

The primary contributions in this paper can be summarized as follows.

• A novel software modeling method DSNM was proposed based on the execution file which generates by tracking the execution process of software, which was mapped to the complex network DISN.

• Novel methods CIWN and CFWN were proposed to identify the most vital nodes in software network.

• The Node Final Weight(NFW) of all nodes with different ranking in software network obey the power-law distribution.

• ETC Method is presented to expand community based on the most vital nodes so as to analyze the modularity of different versions of software and to assist the developer to refactor the software.

The rest of this paper is organized as follows. Section 2 introduces some preparation works and basic definitions. Section 3 describes in detail the identifying of the most vital nodes. The algorithm of detecting Community in the network is shown in Section 4. The experiment results are given in Section 5. Conclusions are drawn in Section 6.

II. PRELIMINARIES

A. TRACKING THE EXECUTION PROCESS OF SOFTWARE

Under Linux environment, we track the execution process of software. The framework of the tracking process is shown in Fig. 1.

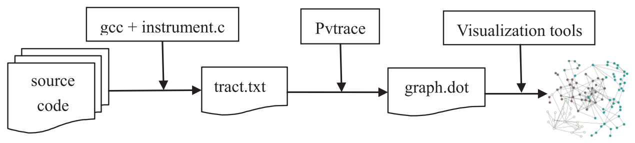


FIGURE 1. Framework of the tracking process of software execution.

(1) First, we compiled the instrument code(*instrument.c*) together with the software code, so as to track the relationship of function calls when software was compiled and installed, and generated the result file *trace.txt* which records addresses of functions.

(2) Second, we undertook the tool *Pvtrace* to analyze trace.txt file and to generate document of graph.dot which records names of functions and relationships of function calls.

(3) Finally, we used the tool *Gephi* to display visualization of call relationship among functions.

B. CREATING A DYNAMIC SOFTWARE NETWORK MODEL

In order to reflect the topological structure and the behavioral characteristics of software accurately, we created a dynamic software network model(DSNM) described by a triple{F, R, W}, among which, F is a set of functions in the network, like $\{f_1, f_2, \ldots, f_i, \ldots\}$, R is a set of call relationships $\{<f_1, f_2>, <f_2, f_3>, \ldots, <f_i, f_j>, \ldots\}$, and W is the weight of each call relationship described as follows and showed in Fig. 2(a).

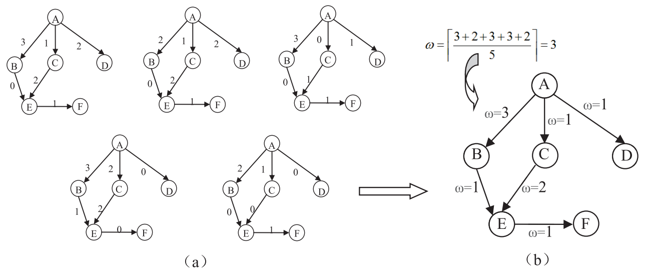


FIGURE 2. A simple illustration of the model creation.

Software information in one execute process could not cover the characteristics of all functions. To reflect the potential characteristics of the software system more accurately, the most comprehensive information is needed to display the call process of the software execution. We got different execution traces through different test cases executing the software system. Then we merged these multiple execution results to build the dynamic execution network software.

Based on some software execution results, we proposed a novel method to eliminate noise as far as possible in the process of execution. First, we counted the number of calls in the process of software execution for several times and took the average value to avoid randomness and uncertainty. Then taking the accidental factor of execution into account, we referred it to 1 when the result is less than 1. By doing so, we took the situation into considering as long as the function or relationship between functions had occurred. We used ω_{ij} to represent the number of call times of function f_i calling function f_j in the multiple executions.

$$\omega_{ij} = \ulcorner \frac{\sum_1^\tau c_{(ij)}}{\tau} \urcorner \tag{1}$$

Where τ is the number of multiple executions and $c_{(ij)}$ is the call number of function f_i calling function f_j in each execution of software. The value of ω_{ij} is rounded to 1 when less than 1.

Fig. 2 is an illustration of the model creation.

In Fig 2, we executed the software for 5 times though 5 different test cases and got different *DSNMs* mapped by various execution traces. Fig 2(a) refers to the function relationship and call times in $n^{th}(1 < n < \tau)$ execution process, while Fig 2(b) describes the final network and call times got from Eq. (1).

C. RELEVANT DEFINITIONS

Definition 1: Dynamic invoke software network (*DISN*). In DISN, nodes represent functions and directed edges represent the call relationship among functions. We use a triple to describe the DISN, as it were DISN = {V, E, W}.

Where V is a set of nodes in the network, like $\{v_1, v_2, \ldots, v_i, \ldots\}$, E is a set of directed edges $\{<v_1, v_2>, <v_2, v_3>, \ldots, <v_i, v_j>, \ldots\}$ and W the weight of each edge which is defined as follows:

$$W_{ij} = \frac{CT_{ij}}{\sum CT_i} * \frac{beCT_{ji}}{\sum beCT_j} \tag{2}$$

Where $\sum CT_{ij}$ is the number of call times(ω) of node v_i calling node v_j, $\sum CT_i$ is the total numbers of node v_i calling others. Likewise, $beCT_{ji}$ is the number of call times of node v_j being called by node v_i, and $\sum beCT_j$ is the total numbers of node v_j being called by other nodes. For example, W_{AC} in Fig 2(b) is equal to $1/(3+1+1) * 1/1 = 1/5$.

Definition 2: Node Initial Weight (*NIW*). NIW is defined as follows to measure the initial weight of node v_i.

$$NIW(v_i) = \sum_{j=1}^{n} W_{ij} \tag{3}$$

Where W_{ij} is the weight of edge $<v_i, v_j>$, NIW of node v_i is the sum of all weights of edges which connect v_i to its neighbors.

Definition 3: Node Final Weight (*NFW*). NFW is defined as follows to measure the final weight of the node v_i.

$$\begin{aligned} NFW(v_i) &= NIW(v_i) + \sum_{j=1}^{n} W_{ij} * NIW(v_j) \\ &= \sum_{j=1}^{n} W_{ij} * [1 + NIW(v_j)] \end{aligned} \tag{4}$$

In Eq. (4) where $NFW(v_i)$ is the final weight of node v_i. v_j represents the neighbors of node v_i. The subnodes behind v_j are all indirect invocation nodes for the v_i. The NFW value of the node not only considers the influence of its direct neighbor nodes, but also depends on the influence of nodes indirectly called until their out-degree is 0. As the instance in Fig 2(b), the NIW of D and F are 0, then the NFW of A can be computed by $NFW(V_A) = W_{AB} * [1 + NFW(V_B)] + W_{AC} * [1 + NFW(V_C)] + W_{AD}$.

III. IDENTIFYING THE MOST VITAL NODES

It is in our opinion that the weight of node v_i is based on its initial value and the contribution of its neighbors. In order to measure the most vital nodes, there were two steps needed to be taken. We first computed the NIW of node v_i in algorithm 1, and then its NFW accumulated in the process of iteration in algorithm 2.

Algorithm 1 calInitialWeightOfNode (CIWN)

```
Input: set V = {$v_1$,$v_2$, $\cdots$ ,$v_i$, $\cdots$},
set E = {<$v_{s1}$, $v_{e2}$>,<$v_{s2}$, $v_{e3}$>, $\cdots$ ,<$v_{si}$, $v_{ej}$>, $\cdots$},
outDegreeList[$v_i$],
Output: NIW($v_i$)
1. for( $v_i \in$ V )
2.   if(outDegreeList[$v_i$] != null
     and outDegreeList[$v_i$].size() != 0)
3.     for (child : outDegreeList[$v_i$])
4.       for (child1 : outDegreeList[$v_i$])
5.         outd += child1.getNodeWeight();
6.       p1 = child.getNodeWeight()/outd;
7.       for (edge e : E)
8.         if(e.getEndNode() == child.getNodeIndex())
9.           ind += e.getEdgeWeight(); //
             the number of being called
10.        p2 = child.getNodeWeight()/ind;
11.      weight = p1*p2;
12.      NIW($v_i$) += weight;
13. return NIW($v_i$);
```

A. CALCULATING NODE INITIAL WEIGHT

We calculated the NIW of each node in the software network as its initial weight. We used adjacent table to store each node and its outDegreeList (the child nodes of node v_i). In algorithm 1, if node v_i calls node v_j, we would store the call numbers in the edge $< v_i, v_j >$ and if node v_i is called by others, we would store the number of calls in the structure of node v_i.

As shown in algorithm calInitialWeightOfNode, the initial weight of node v_i is consisted of two parts. Line 2 to line 6 is the process of computing p1, which represents the ratio between the call number of node v_i and the v_i' total call numbers. p2 is obtained from line 7 to line 10, which stands for the ratio between the number of node v_j called by node v_i which is the father node of v_j and the number of nodes calling v_j. Line 11 to line 12 counts the initial weight of node v_i.

B. CALCULATING NODE FINAL WEIGHT

As the weight of node v_i is considered to be based on not only the initial weight of itself but also the contribution of its neighbors in each iteration process, we defined contriWeightForNode(CWN) to represent the contribution. We continued to iterate the process until v_i could no more get any contribution from its neighbors.

The neighbors of v_i make contributions to it with a certain probability p2. The CWN of v_i is the accumulation of its neighbors in each iteration process. We updated the CWN of node v_i after each iteration so as to get the weight of node v_i of next iteration in line 17. In the first iteration, we used the NIW of node v_i as is shown from line 3 to line 7, while in other iterations shown in line 8 to line16, we could only obtain part of the values of neighboring nodes for v_i from last iterations. The iteration is stopped until node v_i could not get any contribution from its neighbors as shown in line 18.

Algorithm 2 calFinalWeightOfNode(CFWN)

```
Input: set V = {$v_1$,$v_2$, $\cdots$ ,$v_i$, $\cdots$},
set E = {<$v_{s1}$, $v_{e2}$>,<$v_{s2}$, $v_{e3}$>, $\cdots$ ,<$v_{si}$, $v_{ej}$>, $\cdots$},
outDegreeList[$v_i$],
Output: NFW($v_i$)
1. do
2.   for(k)// the number of iterations
3.     if(k == 0)// in the first iteration
4.       for ($v_i \in$ V)
5.         if (outDegreeList[$v_i$] != null
           and outDegreeList[$v_i$].size() != 0)
6.           for (child : outDegreeList[$v_i$])
7.             CWN($v_i$) += p2 * calInitialWeightOfNode
               ($v_i$);
8.     else
9.       for ($v_i \in$ V)
10.        if (outDegreeList[$v_i$] != null
           and outDegreeList[$v_i$].size() != 0)
11.          for (child : outDegreeList[$v_i$])
12.            for (entry : map.entrySet)
13.              if (child.equals(entry.getKey()))
14.                weight = entry.getValue();
15.                break;
16.            CWN($v_i$) += p2 * weight;
17. while(CWN($v_i$) != 0)
18.   NFW($v_i$) = Σ CWN($v_i$) + NIW($v_i$);
19. return NFW($v_i$);
```

In line 19 we calculated the final weight of v_i in accordance to its original value and value obtained during the iterative process.

IV. DETECTING COMMUNITY STRUCTURE

Based on the software design principle of "high cohesion and low coupling", the relationships among functions were developed to be different in software. Community structures exist in most of the software networks. It should be noted that functions relies on each other tightly in the community structure but loosely among different community structures.

Considering the dependency relationship among the functions, we partitioned the software network from the perspective of community detection. In the process, we detected functions closely related to software execution process. We took top-k nodes as the original community when the NFW of these nodes were larger than that of other nodes. Nodes are considered as original communities, if those NFW are μ times larger than the maximum NFW value of node. Then we considered the top-k nodes as the center so as to expand the original communities and to evaluate software modules.

A. CREATING ORIGINAL COMMUNITY

In algorithm 3, we ranked the NFW of nodes and mined top-k nodes as the original communities. Accordingly, we chose top-k nodes as communities, of which the NFW was larger

Algorithm 3 createOriginalCommunity(COC)

```
Input: set V = {$v_1$,$v_2$, $\cdots$ ,$v_i$, $\cdots$}, outDegreeList[$v_i$],
Output: top-k nodes list finalist // the original community
1. Initialize resultList;
2. for( $v_i \in$ V )
3.   value = NFW($v_i$);
4.   reslutList.add($v_i$, value);
5. resultList.sort();
6. Max = max(value);
7. if(value($v_i$) > $\mu$ *Max) // 0 < $\mu$ <1
8.   finalList.add($v_i$, value);
9. return finalList;
```

than μ (0< μ <1) times of the largest NFW as the original communities.

In Algorithm 3, we initialized resultList as the measurement list for all nodes in line 1. Line 2 to 4 stands for a looping process to store the value of each node. The sorting process is shown in line 5 and the top-k nodes chosen from the result List is presented from line 6 to line 8.

B. EXPANDING THE ORIGINAL COMMUNITY

In this part, we expanded the original community by adding nodes to it. In the first step, directly dealt with nodes which connects to the original communities(top-k nodes). Then we categorized the remained nodes based on their father nodes. Therefore in the second step, we took into account the father nodes of remained nodes. We counted the classes and weights of their father nodes. In the final step, we classified remained nodes accurately by referring to the statistical data got from the last step.

As shown in algorithm 4, in process 1, we considered each node v_i which connects to the original community node directly in line 2. If node v_i had not been classified before, the classification was the same with its father node in line 9 to line 11. If node v_i happened to be an original community node, we would compare the values between the node and the core nodes, and put the node into the community with the maximal value from line 3 to line 6. However If we couldn't distinguish the class for the reason that node v_i has the same value with different core nodes, we would put it back to the source set V again in line 8. Process 2 counts the father nodes of the rest nodes, in which we would judge whether they had been classified or not first in line 13. A list *currentList* is then defined to store them and their information in line 14. If the father node of node v_i had been classified, the weight and size of this class would be added up in line 15 to line 18. Or we would store the new class in the *currentList* in line 20, and add 1 to the size of the class in line 21. Finally in process 24, the final class of node v_i was determined by the weight or the size which had been counted in process 23. When the ratios of community of father nodes and all in-degree nodes are more than 50%, we would consider that the community has an absolute advantage and we would put the node into this community. If we failed to distinguish the community by the

Algorithm 4 expandTheCommunity (ETC)

```
Input: top-k nodes list finalList, set V = {$v_1$,$v_2$, $\cdots$ ,$v_i$, $\cdots$}, outDegreeList[$v_i$],inDegreeList[$v_i$]
Output: class list resultList
Process1 dealing the nodes connecting to the original community directly
1. for(n $\in$ finalList)
2.   for( $v_i \in$ outDegreeList[n])
3.     if ($v_i \in$ finalList) //$v_i$ has been classified
4.       currentNode = finalList.get($v_i$.getIndex());
5.       if (currentNode.getWeight() < $v_i$.getWeight()
         currentNode.classification () != $v_i$.classification ())
6.         store ($v_i$.getIndex(), $v_i$.classification(),
           $v_i$.getWeight();
7.       else if (currentNode.getWeight() == $v_i$.getWeight())
8.         V.add($v_i$); //could not distinguish and put it into
           set V
9.     else
10.      store ($v_i$.getIndex(), n.classification(),
         $v_i$.getWeight());
11.      V.remove($v_i$);
Process2 counting the number of class and weight of the father nodes of remained nodes
12. for(remained nodes $v_i \in$ V)
13.   for( each father $\in$ InDegreeList[$v_i$])
14.     Initialize currentList; //storing the remained nodes
15.     if (father $\in$ resultList) //father node has been
        classified
16.       if (currentList.exist(father.classification()))
17.         currentList.weight += father.weight();
18.         currentList.classification.size += 1;
19.       else
20.         store (father.classification(),
            father.getWeight(), size = 1);
21.       currentList.classification.size = 1;
Process3 classing the remained nodes
22. if(classification.size())//comparing the size
23.   if( classification.size() > 0.5*inDegreeList.size())
24.     finalClass = classification;
25. else //comparing the weight
26.   finalClass = Max(weight).getIndex.classification();
```

size in line 25 to line 26, we would put the node into the community with the maximum value of weight.

And the detail detection strategies in algorithm 4 can be explained by three situations in Fig. 3-5 below.

In Fig. 3-5, the gray node *b* and *e* are initialized as original community structures. In Fig. 3, node *a,c,f* are directly connected with community structure *b* or *e*(called "*b*" or "*e*"), the they can be put into *b* or *e*, like the Fig. 3(a). But for node *d*, there are two judgements to make it belong to its corresponding community structure.

1) Judged by connections. In Fig. 3, there are 3 edges for node *d* connecting with "*b*", larger than 2 edges connecting

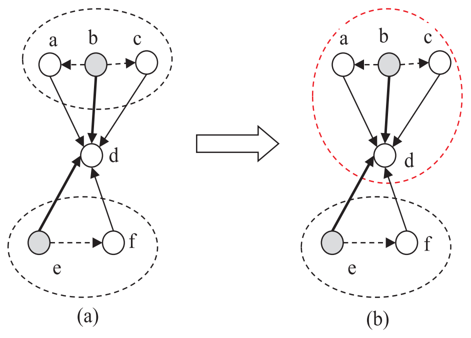


FIGURE 3. Partitioning strategy - situation 1.

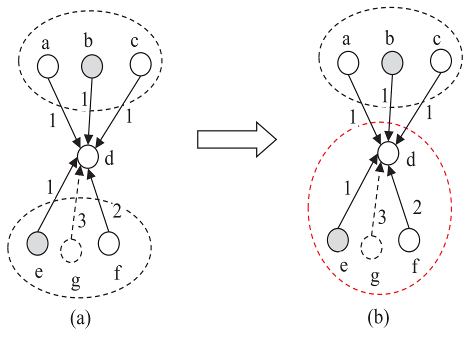


FIGURE 4. Partitioning strategy - situation 2.

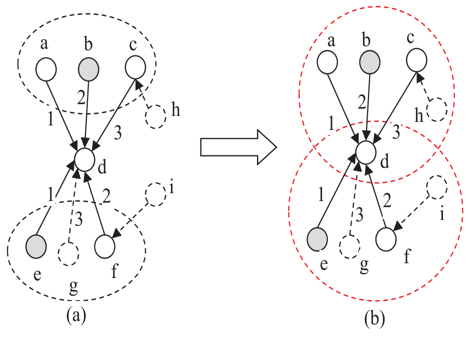


FIGURE 5. Partitioning strategy - situation 3.

with "*e*", so node *d* is put into "*b*", which is shown in Fig. 3(b);

2) Judged by connection weights. From Fig. 4(a), the connection edges with "*b*" and "*e*" for node *d* are the same, in this case, the connection weight for node *d* with "*b*" is $1 + 1 + 1 = 3$, smaller than that$(1 + 3 + 2 = 6)$ with "*e*", so node *d* is assigned to "*e*" in Fig. 4(b).

As is derived in Fig. 5, once there are the same connections or connection weights, the node can be belong to all the communities, which will form the overlapping community. This is another topic discussion in our future research.

V. EXPERIMENTS AND ANALYSIS

In this section, approaches mentioned above will be tested by three open source software *Tar*, *cflow* and *gzip*. *Tar* is a decompression software for Linux, *cflow* an analysis tool for program C to extract the relationship of function calls, and *gzip* an compression software for Linux system file. We would choose their own different versions for further experiments.

The experiments are conducted on *64 bit Windows 10 ultimate*, *Intel(R) Core(TM) i7-4710HQ CPU @2.50GHz*, 512GB SSD, *16G* of *RAM* and *Ubuntu14.04*.

Firstly, we executed the software to capture the execution trace under Linux environment. We inserted some track marks in the compilation and installation stages to obtain the execution information of the software, we got the function invoking traces by using the marks during running. After analyzing these traces, we recognized the relationships among the software functions. Regardless of the occasional situation in some special execute process, we obtained the key execution information from the multiple executions in different experiments for each version of the software. Then we created a dynamic software network model (DSNM) to map the execute trace and built a network DISN accordingly. By doing so, we got the accurate and stable network structure for each version of software. Finally we analyzed the evolution of nodes in the software and mine the most vital nodes. Meanwhile, we analyzed another characteristic of software modularity in different versions of software by referring to the most vital nodes.

A. ANALYZING THE NIW OF NODES

We run the algorithm CIWN on each version of *Tar*, *cflow* and *gzip*. By the algorithm CIWN, we calculated the NIW of each function node. Fig. 6, Fig. 7 and Fig. 8, which show the analysis results of different versions of software.

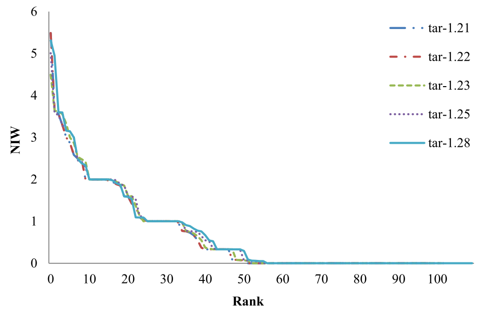


FIGURE 6. NIW value distribution of *Tar*.

As shown in Fig. 6, the NIW distribution of software *Tar* are very similar to each other in its five versions. With the ranking of nodes, the value of NIW shows a trend of decrease. The higher NIW ranges from 4 to 6. Most nodes' NIW are around 1 or 2 but others close to 0. Different versions of software follows the same laws, that is, the node's NIW of a certain ranking remains stable and the NIW distribution of different software version is nearly the same. We can then predict the trends of future versions based on this. Fig. 7 shows the NIW distribution of software *cflow*, and the curve of each version has the same tendency. The higher NIW ranges from 5 to 7 and most nodes' score range from 0 to 2.

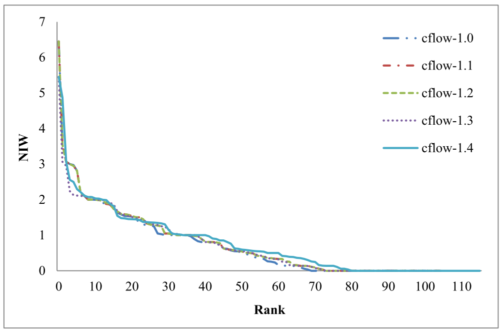


FIGURE 7. NIW value distribution of *cflow*.

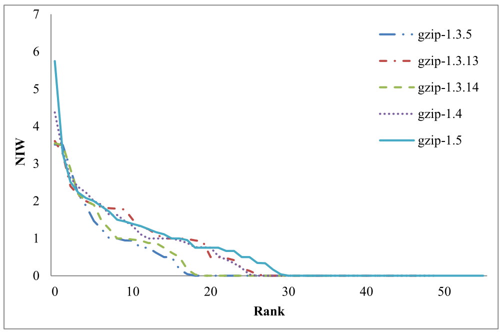


FIGURE 8. NIW value distribution of *gzip*.

The NIW distributions of software *cflow* in different versions follow the same trend, so do the distributions of other two software *gzip* and *tar*.

B. ANALYZING THE NFW OF NODES

By the algorithm CFWN, we calculated the NFW of each function node. The iterator process of the algorithm 2 ended from 6 to 30 times. The NFW of the nodes with same ranking in different versions of software have a small fluctuation, which are shown in Fig. 9, Fig. 10 and Fig. 11.

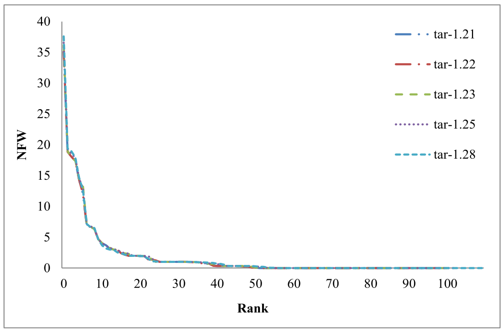


FIGURE 9. NFW value distribution of *Tar*.

Fig. 9, Fig. 10 and Fig. 11 show the NFW value distributions of software *Tar*, *cflow* and *gzip*. From the figures, we can see that only few nodes have great weights while most nodes have small ones. In addition, we can also get some implicit information from the three figures.

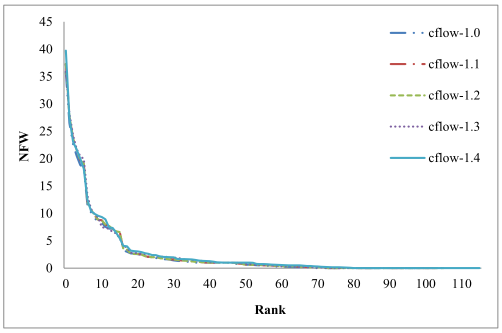


FIGURE 10. NFW value distribution of *cflow*.

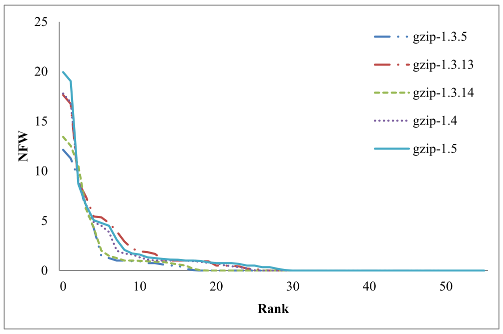


FIGURE 11. NFW value distribution of *gzip*.

- The NFW values of the nodes obey the power-law distribution in the software. We chose software *Tar-1.21*, *cflow-1.2* and *gzip-1.4* in order to describe the distribution separately, and enumerated the fitted results of software *Tar-1.21*, *cflow-1.2* and *gzip-1.4* in Fig. 12, Fig. 13 and Fig. 14 respectively.

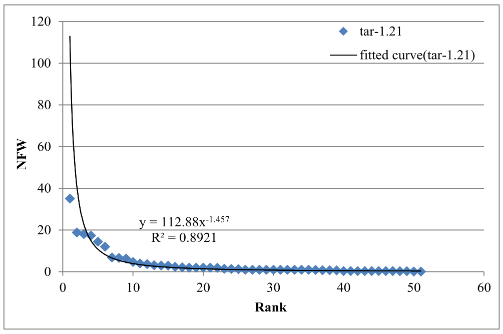


FIGURE 12. NFW value distribution of *Tar-1.21*.

- The NFW distributions of the software are much similar in the five versions. With the nodes ranking, the NFW of each node shows a decrease trend.
- The development of versions follows the same laws, the nodes' NFW of a certain ranking remains stable. Also, the NFW distributions of different software versions are nearly the same. The higher NFW ranges from 15 to 40, but the scores of most nodes are around 5 or 10, others close

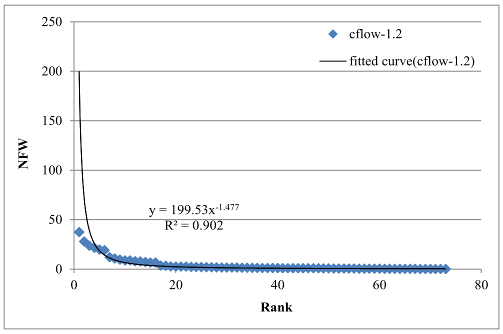


FIGURE 13. NFW value distribution of *cflow-1.2*.

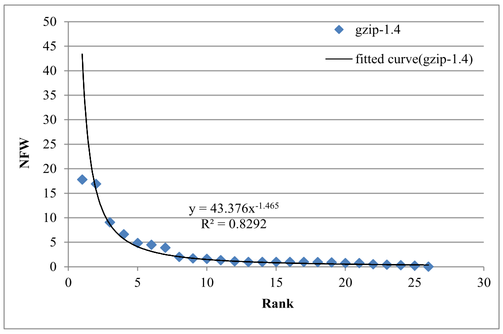


FIGURE 14. NFW value distribution of *gzip-1.4*.

to 0. Seen from the above, we can predict the trends of future versions.

As shown in the Figures, we chose *Tar-1.21*, *cflow-1.2* and *gzip-1.4* as representation of each software. As we can see in these figures, most nodes with small NFW are ordinary functions. In other words, it is not necessary to emphasize them. Unlike, nodes with the NFW around 20 are minorities in each version, but they play important roles software updating and maintenance.

The distributions of different software are shown in Fig. 12, Fig. 13 and Fig. 14. The image feature can be divided into two parts. In Eq. (4), NFW(v_i) is the final weight of node v_i, and the value accumulated indirectly in the path is gradually increased. Observed from the front part in the curve, node with larger values are not located on the curve exactly, this suggested that these nodes have the biggest NFW and have a strong connection with other nodes (such as bigger out/in degree),which is suitable for Scale-free network. But the NFW of all nodes with different ranking obey the *power-law* distribution from the whole tendency. To verify them, we have tested the correctness of the results with *Kolmogorov-Smirnov(KS)* test [23] which is described in detail as follows.

Given an observed distribution $P(x)$, we firstly assume that it obeys a certain form $F(x; a_1, a_2, \ldots, a_l)$, with a set of parameters $a_1, a_2, \ldots, a_l$, whose values are estimated by using the maximum likelihood method [24]. The standard KS distance is defined as the maximal distance between the cumulative density functions of the observed data $P^c(x)$ and the fitting curve $F^c(x)$, namely $D^{KS}_{real} = max_x|P^c(x) - F^c(x)|$. The original hypothesises of KS are the two data distribution being consistent or the data obeying the theoretical distribution. First, the significance level α is set, and it is called the first type of error which is the probability of rejection as the original hypothesis is correct in statistics. By the engineering experience, it is usually set as 0.05, namely $\alpha = 0.05$. Assuming that there is a checksum F, then the value of another index f can be calculated by putting the sample data into F, so the p value in KS is the probability when $F > f$ under the condition of the original hypothesis. If p is less than the given significant level α, we would reject the original hypothesis, vice versa. According to the data provided in this paper, the corresponding p-value and D can be calculated as shown in Table 1.

TABLE 1. The corresponding value of p, D and $D_{0.05}$.

software	Function expression	p	D	$D_{0.05}$
tar-1.21	$y = 112.88x^{-1.457}$	0.1469	0.22039	0.26932
gzip-1.4	$y = 43.376x^{-1.465}$	0.5256	0.22308	0.37719
cflow-1.2	$y = 199.53x^{-1.477}$	0.2044	0.16686	0.2251

According to the experimental results, the p-value in each group experiment is greater than that of α (0.05), and value of D in all groups are all smaller than $D_{0.05}$ in the table of critical value of D in two sample test, so the original data are consistent with *power-law* distribution. Because the NFW of nodes in software networks obey power-law distribution. In other words, a small number of nodes in the software network has a larger weight, a few of the nodes are in control of the entire network, while the NFW of most nodes are small, this phenomenon in the software network proves that the software has scale-free properties.

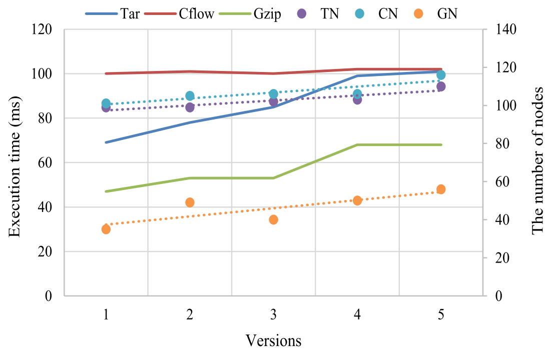


FIGURE 15. Execution time in each version of software.

Also, we listed the execution time of different versions of software as shown in Fig. 15.

As shown in Fig. 15, we listed execution time of the three software in each version. TN, CN and GN represent the number of nodes in the three software network. With the update of software version, the complexity of software is increased and the execution time increased steadily. It is worth mentioning that the more complex of the software, the more execution

TABLE 2. Top-10 nodes of software *Tar-1.25*.

Rank	NFW	CC	BC	OD
main	1	4	54	5
dump_file	2	6	4	11
update_archive	3	13	9	2
dump_file0	4	14	1	1
dump_regular_file	5	15	3	6
create_archive	6	9	14	4
find_next_block	7	11	5	35
start_header	8	10	6	3
flush_archive	9	16	2	20
open_archive	10	3	16	14

time among the three software, and its execution time is also the longest, such as *cflow*, the number of nodes is the largest, so the corresponding execution time is the longest.

In software engineering, although important functions will be invoked rarely, they may have great influences to others. Unlike, simple functions will be reused frequently. In other words, the *out-degree* represents the complexity of a function and the *in-degree* of a function the high reusability.

We mined top-10 nodes in software based on the NFW and we listed *Tar-1.25*, *cflow-1.1* and *gzip-1.4* in Table 2, Table 3 and Table 4 respectively. In order to ensure that the method CFWN is effective in mining most vital nodes, we compared the results with the results got from those ranking measures widely used in network, such as *closeness centrality*(CC), *betweenness centrality*(BC) [25] and *out-degree*(OD). It needs to be pointed out that *Closeness* of node is defined as the reciprocal of the sum of geodesic distances to all other nodes, and betweenness, a centrality measure of a node in a network, is defined as the fraction of the shortest paths between node pairs that pass through the node of interest.

TABLE 3. Top-10 nodes of software *cflow-1.1*.

Rank	NFW	CC	BC	OD
main	1	18	69	2
parse_declaration	2	16	4	11
func_body	3	17	14	5
yyparse	4	15	18	7
parse_function_declaration	5	11	22	18
parse_variable_declaration	6	19	5	1
parse_dcl	7	3	7	9
dcl	8	2	8	17
dirdcl	9	7	9	12
maybe_parm_list	10	12	6	19

As shown in the tables, we mined top-10 nodes by the method NFW in the first column and then rankings of NFW, closeness centrality, betweenness centrality, out-degree in turn. There being 80% nodes in the top-10 nodes mined by metric NFW exist in top-20 nodes ranking by other measures. And if more nodes listed, the similarities between the nodes in NFW's result-list and others would be even greater. Especially in software *gzip*, most nodes in NFW's top-10 result-list were almost in the top-10 nodes in other result-lists of measurements.

TABLE 4. Top-10 nodes of software *gzip-1.4*

Rank	NFW	CC	BC	OD
main	1	1	24	4
treat_file	2	2	3	1
zip	3	7	1	2
deflate	4	6	4	5
flush_block	5	12	2	3
unzip	6	3	9	10
inflate	7	8	5	11
inflate_block	8	13	8	18
compress_block	9	5	49	25
build_tree	10	18	6	8

Here we defined a similarity coefficient $\mathcal{C}$ to measure the correlation between NFW ranking and others. The coefficient $\mathcal{C}$ is shown as follows.

$$\mathcal{C} = \frac{|\{NFW_{v_i}\} \bigcap \{otherMetric_{v_i}\}|}{TN} \tag{5}$$

Where $\{NFW(v_i)\}$ is a set of nodes which contains top-k nodes about measure NFW, where $\{otherMetricv_i\}$ is a set of nodes about other measures like *closeness centrality*, *betweenness centrality* and *out-degree*. Here *TN* represents the total number of nodes. The coefficient of similarity is used to measure the degree of similarity between NFW and other measures. A higher value indicates a more accurate ranking of the nodes' NFW.

The similarity coefficient of the top-10 functions between NFW and closeness centrality can be as high as 0.7 in *gzip-1.5* and as low as a half. As for the other measure *betweeness centrality*, the coefficient $\mathcal{C}$ ranges from 0.6 to 0.8 between NFW and this metric. Similarly, nodes of *closeness centrality* and *betweenness centrality*, *out-degree* of nodes shows the same characteristics, in most cases, $\mathcal{C}$ is 0.5 or 0.7 in different software.

Compared NFW with other measures, their betweenness was found to be more complex to compute but their degrees simple but more inaccurate. We can thus infer that the method we proposed is helpful to identify the most vital nodes in the software network, which can be used for further studies.

C. ANALYZING THE MODULARITY OF THE SOFTWARE

Arenas *et al.* [26] proposed a generalization of modularity in directed networks by simply replacing the strength terms into directional ones. The generalized modularity can be described as follows:

$$Q = \frac{1}{M}\sum_{i,j}[w_{ij} - \frac{w_i^{out} * w_j^{in}}{M}]\delta_{c_i,c_j} \tag{6}$$

Where w_{ij} represents the weight of link pointing from node i to node j, $w_i^{out} = \sum_j w_{ij}$ and $w_j^{in} = \sum_i w_{ij}$ are the out-strength and in-strength of node i and node j respectively, and the total strength is M $= \sum_i w_i^{out} = \sum_j w_j^{in} = \sum_{i,j} w_{ij}$.

We adopted modularity Q to evaluate the modularity of software. According to the NFW of each node by Algorithm 2 CFWN, we chose top-k ($\mu = 0.2$ in Algorithm 3) nodes which are larger than others. The top-k nodes range 5 to 14 in each version of different software. The modularity of different versions shows in Fig. 16, Fig. 17 and Fig. 18.

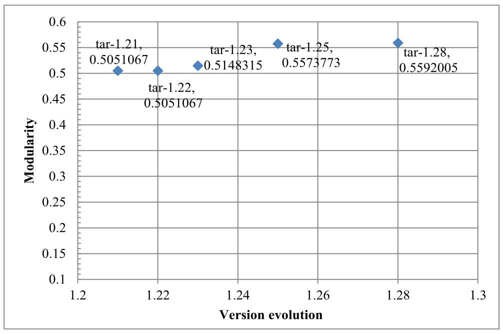


FIGURE 16. **Modularity of software *Tar*.**

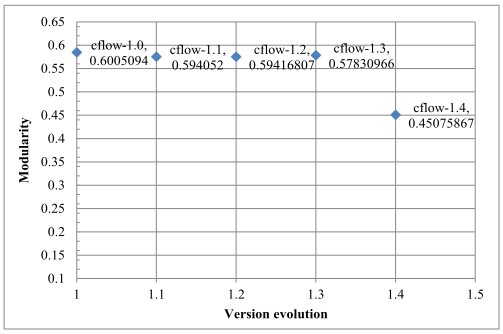


FIGURE 17. **Modularity of software *cflow*.**

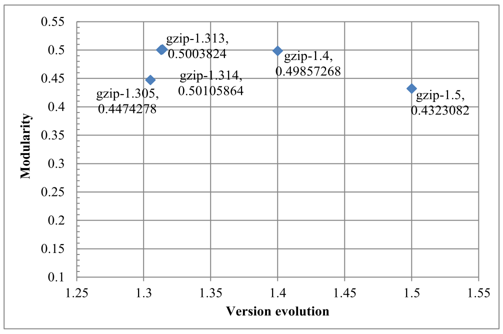


FIGURE 18. **Modularity of software *gzip*.**

As shown in the three figures, we can find some information.

- Modularity of different versions of the software have small wave ranges. For example, in Fig. 16, the modularity ranges from 0.501067 to 0.5592005, and the difference between the two values is less than 0.1. Likewise, there are little differences in various versions of software *cflow* and *gzip*.
- This metric reveals whether the projects follow good software engineering practices or not. In the evolution of software, it has the tendency of high cohesion and low coupling, even if the performance is not good in a certain version. The metric of software *tar* in version *tar-1.22* shows the trend of decreasing, but the value of modularity grows steadily in later versions. It is to say that in later versions, software will be reconstructed to improve its modularity. Similar to the version *tar-1.22*, we need to refactor software in some version. For example, the modularity of *cflow-1.4* is less than its previous versions in Fig. 17, and for software *gzip-1.5* in Fig. 18, its modularity is lower than the modularity of other versions, which implies that the software of these versions should be refactored.
- We can predict the modularity of the future versions based on the existing version. High cohesion and low coupling software make the modularity of software significant to be an new direction of development and research. To put it another way, the modularity will have a gradual increase.

VI. CONCLUSIONS

This paper examined specifically the modularity of software network based on the most vital nodes. A dynamic software network model was constructed by multiple process of execution and then mapped to a directed-weight network, in which the most vital nodes were identified by the NFW in accordance to the call numbers and the contribution of neighbor nodes. Besides, the top-k nodes were mined to be original communities by a parameter μ, which were then expanded by the ETC algorithm. Finally, the relationships among nodes were discussed and analyzed by ETC method, and these nodes were put into the suitable communities. During experiments, values of the NIW and NFW distribution of the nodes in different software versions were analyzed to study the evolution rules of the software. The experimental results show that it is acceptable to use NFM method to identify the most vital nodes and that it is reasonable to adopt criterion Q to evaluate software modularity. Follow-up studies focus on the exploring of more structural features of software execution network.

ACKNOWLEDGMENTS

The authors are grateful to the valuable comments and suggestions of the reviewers.

REFERENCES

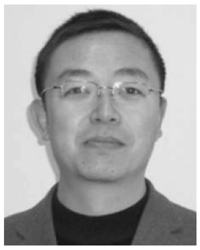
[1] I. Turnu, M. Marchesi, and R. Tonelli, "Entropy of the degree distribution and object-oriented software quality," in *Proc. 3rd Int. Workshop Emerg. Trends Softw. Metrics*. Piscataway, NJ, USA: IEEE Press, Jun. 2012, pp. 77–82.
[2] I. Neamtiu, G. Xie, and J. Chen, "Towards a better understanding of software evolution: An empirical study on open-source software," *J. Softw., Evol. Process.*, vol. 25, no. 3, pp. 193–218, 2013.
[3] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. 34th Int. Conf. Softw. Eng.* Piscataway, NJ, USA: IEEE Press, Jun. 2012, pp. 419–429.
[4] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proc. 35th Int. Conf. Softw. Eng.* Piscataway, NJ, USA: IEEE Press, May 2013, pp. 612–621.
[5] G. Huang, B. Zhang, R. Rong, and R. Jiadong, "An algorithm to find critical execution paths of software based on complex network," *Int. J. Mod. Phys. C*, vol. 26, no. 9, p. 1550101, Sep. 2015.
[6] S. Lamzabi, S. Lazfi, H. Ez-Zahraouy, A. Benyoussef, A. Rachadi, and S. Ziti, "Pair-dependent rejection rate and its impact on traffic flow in a scale-free network," *Int. J. Mod. Phys. C*, vol. 25, no. 7, p. 1450019, 2014.

[7] Z. Tao and W. Bing-Hong, "Catastrophes in scale-free networks," *Chin. Phys. Lett.*, vol. 22, no. 5, pp. 1072–1075, 2005.
[8] L. Zemanová, C. Zhou, and J. Kurths, "Structural and functional clusters of complex brain networks," *Phys. D, Nonlinear Phenomena*, vol. 224, nos. 1–2, pp. 202–212, 2006.
[9] L. C. Freeman, "Centrality in social networks conceptual clarification," *Soc. Netw.*, vol. 1, no. 3, pp. 215–239, 1978.
[10] D. S. Callaway, M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Network robustness and fragility: Percolation on random graphs," *Phys. Rev. Lett.*, vol. 85, no. 25, p. 5468, 2000.
[11] M. Kitsak *et al.*, "Identification of influential spreaders in complex networks," *Nature Phys.*, vol. 6, pp. 888–893, Aug. 2010.
[12] J. Bae and S. Kim, "Identifying and ranking influential spreaders in complex networks by neighborhood coreness," *Phys. A, Statist. Mech. Appl.*, vol. 395, pp. 549–559, Feb. 2014.
[13] J.-G. Liu, Z.-M. Ren, and Q. Guo, "Ranking the spreading influence in complex networks," *Phys. A, Statist. Mech. Appl.*, vol. 392, no. 18, pp. 4154–4159, 2013.
[14] S. Fortunato, "Community detection in graphs," *Phys. Rep.*, vol. 486, nos. 3–5, pp. 75–174, 2010.
[15] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, "Complex networks: Structure and dynamics," *Phys. Rep.*, vol. 424, nos. 4–5, pp. 175–308, 2006.
[16] B. W. Kernighan, and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Techn. J.*, vol. 49, no. 2, pp. 291–307, Feb. 1970.
[17] M. E. J. Newman, "Community detection and graph partitioning," *Europhys. Lett.*, vol. 103, no. 2, p. 28003, 2013.
[18] W. Lin, X. Kong, P. S. Yu, Q. Wu, Y. Jia, and C. Li, "Community detection in incomplete information networks," in *Proc. 21st Int. Conf. World Wide Web*, 2012, pp. 341–350.
[19] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 69, p. 026113, Feb. 2004.
[20] Y. Kim, S.-W. Son, and H. Jeong, "Finding communities in directed networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 81, no. 1, p. 016103, 2010.
[21] A. A. H. Zanjani and A. H. Darooneh, "Finding communities in linear time by developing the seeds," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 84, no. 3, p. 036109, 2011.
[22] D. Chen, M. Shang, Z. Lv, and Y. Fu, "Detecting overlapping communities of weighted networks via a local algorithm," *Phys. A, Statist. Mech. Appl.*, vol. 389, no. 19, pp. 4177–4187, 2010.
[23] X.-Y. Yan, X.-P. Han, B.-H. Wang, and T. Zhou, "Diversity of individual mobility patterns and emergence of aggregated scaling laws," *Sci. Rep.*, vol. 3, Sep. 2013, Art. no. 2678.
[24] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, 2009.
[25] U. Brandes, "A faster algorithm for betweenness centrality," *J. Math. Sociol.*, vol. 25, no. 2, pp. 163–177, 2001.
[26] A. Arenas, J. Duch, A. Fernández, and S. Gómez, "Size reduction of complex networks preserving modularity," *New J. Phys.*, vol. 9, no. 6, p. 176, 2007.

BING ZHANG received the bachelor's degree from the College of Computer and Information Technology, Three Gorges University, China, in 2012, and the Ph.D. degree from the School of Information Science and Engineering, Yanshan University, China, in 2018. He is currently holding a post-doctoral position with the School of Information Science and Engineering, Yanshan University. His research interests include data mining, machine learning, and software security.

GUOYAN HUANG is currently a Professor with the School of Information Science and Engineering, Yanshan University, China. His research interests include data mining, temporal data modeling, and software security. His research has been supported by the National Natural Science Foundation of China and the Science Foundation of Hebei Province.

ZHANGQI ZHENG received the bachelor's degree from the Liren College, Yanshan University, China, where she is currently pursuing the master's degree with the School of Information Science and Engineering. She is currently focusing on a project on software security. Her research interests include data mining and machine learning.



JIADONG REN is currently a Professor with the School of Information Science and Engineering, Yanshan University, China. His research interests include data mining, temporal data modeling, and software security. His research has been supported by the National Natural Science Foundation of China and the Science Foundation of Hebei Province. He is a Senior Member of the Chinese Computer Society, and a member of the IEEE SMC Society and ACM.

CHANGZHEN HU is currently a Professor with the School of Software, Beijing Institute of Technology, China. His research interests include information safety.