ICIC Express Letters
Volume **10**, Number **2**, February **2016**

ICIC International ©2016 ISSN 1881-803X
pp. **485–492**

A NOVEL APPROACH TO IDENTIFY INFLUENTIAL FUNCTIONS IN COMPLEX SOFTWARE NETWORK BASED ON COMPLEX NETWORK

Guoyan Huang[1,2], Xiaojuan Chen[1,2,*], Hongfei Wu[1,2]
Peng Zhang[1,2] and Jiadong Ren[1,2]

[1]College of Information Science and Engineering
[2]Computer Virtual Technology and System Integration Laboratory of Hebei Province
Yanshan University
No. 438, West Hebei Ave., Qinhuangdao 066004, P. R. China
{ hgy; jdren }@ysu.edu.cn; *Corresponding author: xjchen1990@foxmail.com

Received August 2015; accepted November 2015

Abstract. *Identifying influential nodes is an important issue in understanding the process of information diffusion in complex software networks. Researchers generally define functions as nodes, and relationship of function calls as edges to map software into complex network. This paper proposes a novel approach to identify the influential nodes in complex software network based on complex network. Firstly, we map the function and relationship of function calling or dependency to a directed weighted call network (DWCN). Secondly, we define NodeScore (NS) to evaluate the importance of nodes in network. It has higher influence when the node has lager value of NS. The NS of each node is calculated by two parts in our opinion, Direct Dependency Nodes (DDN) and Indirect Dependency Nodes (IDN). Thirdly, algorithm ComputeNodeScore is proposed to calculate NS of each node. Finally, we mine top-k nodes based on the NS. Experimental results of various versions about software Tar and cflow show the approach is effective for identifying the influential nodes.*
Keywords: Complex software network, Function calls, Influential nodes

1. **Introduction.** The understanding of software network's structure has attracted much attention recently, especially identifying the influential nodes. These influential nodes play an important part in ensuring reliability and stability of software. Once we have identified these nodes, we can pay more attention to them in software version updating and software maintenance [1, 2], even software refactoring [3]. Therefore, identifying influential nodes in software network has become an important task in software engineering and became more and more important.

A lot of work has been done in identifying influential nodes in complex network, but less in software network. Mapping software structure to complex network is important from different perspectives by different methods. Thus, the research approaches of complex network can be applied to the research field of software network. Wang et al. [4] proposed an approach to study the evolution of special software kernel components, which adopted the theory of complex networks. They also proposed a generic method to find major structural changes that happened during the evolution of software systems. Li et al. [5] proposed a modular attachment mechanism of software network evolution. Their approach treated object-oriented software system as a modular network, which was more realistic. A new definition of asymmetric probabilities was given to acquire links in directed networks when new nodes are attached to the existing network. Valverde and Solé [6] presented a complex network approach to the study of software engineering. They found universal network patterns in a large collection of object-oriented (OO) software systems. All the systems analyzed here displayed the small-world effects. It was the first time to study

the software as a complex software network. The classes were treated as nodes and the relationships among classes were treated as directed edges. Inspired by the surprising discovery of several recurring structures in various complex networks, a number of works treated software systems as complex networks and indicated that software systems expose the small-world effects and follow scale-free degree distributions. In addition, Cai and Yin [7] treated software excution process as an evolving complex network for the first time. They examined there exist invariant patterns in the dynamic behavior of software systems. The concept of software mirror graph was introduced as a new model of complex network to reflect the software behavior information.

The mostly used measurements to measure the centrality of nodes are degree centrality (DC), betweenness centrality (BC), and closeness centrality (CC) [8]. Following, many improved methods were proposed. An approach named Semi-local centrality measure [9] was proposed. It considered both the nearest and the next nearest neighbors. The approach was a local centrality measure as a tradeoff between low-relevant degree centrality and other time-consuming measures. Kitsak et al. [10] realized that the position of nodes is important in global network and proposed k-shell decomposition analysis to obtain ranking index of nodes. It considered the position of nodes in global network and illustrated more accurate results than degree and betweenness. However, the k-shell decomposition fails to implement the ranking of spreaders in the same k-shell index. Bae and Kim [11] proposed a novel measure called coreness centrality to estimate the spreading influence of a node in a network, which used the k-shell indices of its neighbors. The approach was based on the idea that a powerful spreader has more connections to other nodes which reside in the core of network. The approach also pointed out that the number of a node's neighbors has a large influence to its spreading ability. Also, Liu et al. [12] presented an improved method to generate the ranking list to evaluate the node spreading influence, which took account of the shortest distance between a target node and node set which has the highest k-core value. They turned to a new perspective to understand the relationship between not only the k-shell location, but also the nodes' shortest distance to the network core. It is helpful for us to understand the importance of nodes in a network.

Based on the researches above, we find that the influence of a function node depends on not only the nodes which it calls directly but also the nodes which it calls indirectly. Firstly we define some concepts to describe our approach, such as DDN (Direct Dependency Nodes) and IDN (Indirect Dependency Nodes). Then an algorithm ComputeNodeScore is proposed to measure the influence of each node in the network. We rank the influence of each node to mine the top-k nodes. These function nodes have played an important part in ensuring software reliability and stability. So they should be paid more attention to in the process of software updating and software maintenance.

The rest of this paper is organized as follows. In Section 2 some definitions are proposed to describe the problem. Then, algorithm ComputeNodeScore used to evaluate each node is given in Section 3. Experimental results in Section 4 show performances of the algorithm. Finally, the conclusions and future work of the paper are presented in Section 5.

2. **Preliminaries.**

Definition 2.1. DWCN (Directed Weighted Call Network). *In DWCN, nodes represent functions and directed edges represent the relationship among functions. We can use a triple to describe the DWCN.*

$$DWCN = \{V, E, AL\} \tag{1}$$

where V is a set of nodes in the network, like $\{v_1, v_2, \cdots, v_i, \cdots\}$. E is a set of directed edges, $\{< v_1, v_2 >, < v_2, v_3 >, \cdots, < v_i, v_j >, \cdots\}$. AL is an adjacency list stored the nodes, edges and the score of each node in the network.

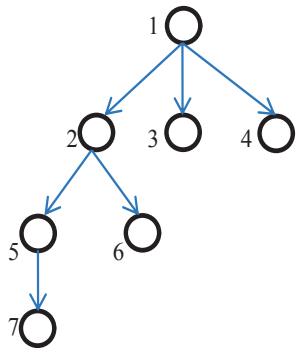

FIGURE 1. Illustration of a simple DWCN

Figure 1 is the illustration of a simple DWCN.

In this illustration, $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, $E = \{< v_1, v_2 >, < v_1, v_3 >, < v_1, v_4 >, < v_2, v_5 >, < v_2, v_6 >, < v_5, v_7 >\}$. Each node and the relations with other nodes in the network have a record in the adjacency list. In addition, the score of each node in the network is different. It is depended on the nodes which it calls directly and others which it calls indirectly. For example in Figure 1, node v_1 calls nodes v_2, v_3, v_4 by edges $< v_1, v_2 >$, $< v_1, v_3 >$, $< v_1, v_4 >$. In our approach nodes v_2, v_3 and v_4 have influence on the importance of node v_1. And other nodes like v_5, v_6 which v_1 can reach by edges $< v_2, v_5 >$ and $< v_2, v_6 >$ also have influence on node v_1.

Definition 2.2. DDN (Direct Dependency Nodes). *For a node v_i, DDN is a set of functions which are called by node v_i directly. The DDN of node v_i is gotten by only one call step.*

As shown in Figure 1, $DDN(v_1) = \{v_2, v_3, v_4\}$.

We have mentioned the influence of node v_i is based on the nodes it can reach directly and indirectly.

Definition 2.3. IDN (Indirect Dependency Nodes). *IDN is a set of nodes that node v_i can reach indirectly.*

In Figure 1, $IDN(v_1) = \{v_5, v_6, v_7\}$.

Definition 2.4. NS (NodeScore). *NS is defined to measure the influence of the node v_i. The NS is given as follows:*

$$NS(v) = p \times |DDN| + \sum p \times NS(u), \quad u \in IDN(v) \tag{2}$$

It has higher influence when v_i has larger value of NS.

$$p = \frac{c_{ij}}{d_i^{out}} \tag{3}$$

where p is the probability of node v_i calling node $v_j \in DDN$. The numerator c_{ij} is a constant to represent whether there is an edge between nodes. If there is an edge $< v_i, v_j >$, c_{ij} is 1; otherwise, c_{ij} is 0. The denominator is the out-degree of node v_i.

3. **Method of Identifying Influential Nodes.** Normally, the software which we mapped consisted of functions and function calls among them. Firstly we extracted these from the source code. Then we map them to a DWCN, and we get the adjacency list at the same time. Finally, algorithm ComputeNodeScore is used to identify influential nodes by calculating the NS of each node, as shown in Figure 2.

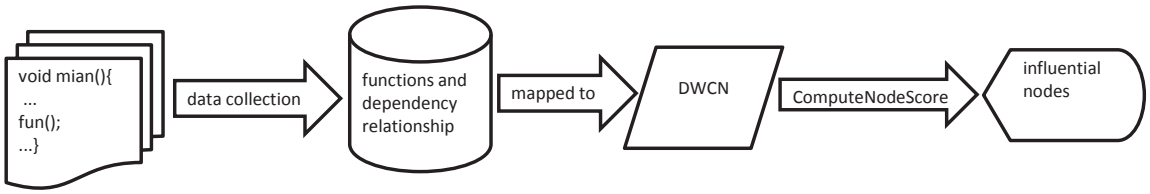

FIGURE 2. Framework of our approach

To measure the NS of node v_i, we build adjacency list based on node set N and edge set E in DWCN. A novel algorithm named ComputeNodeScore is proposed. In this approach, we evaluate the importance of the node v_i iteratively. Finally, we calculate the NS of node v_i.

3.1. **Building adjacency list.** The result of algorithm BuildAdjacencyList is a part of DWCN. We study the software as a complex software network and get two sets which are node set N and edge set E. The process of building adjacency list is based on the two sets. We build a list for each node.

Algorithm 1 BuildAdjacencyList

Inputs: set $V = \{v_1, v_2, \cdots, v_i, \cdots\}$, set $E = \{< v_{s1}, v_{e2} >, < v_{s2}, v_{e3} >, \cdots, < v_{si}, v_{ej} >, \cdots\}$

Output: OutDegreeList(v_i) //the out-degree list of node v_i

```
1: Process:
2: for (each v_i ∈ V){
3:    for(< v_si, v_ej > ∈ E){
4:       if(v_i = v_si)
5:          OutDegreeList(v_i).add(v_ej);
6:    }
7:  Return OutDegreeList(v_i);
8: }
```

As shown in algorithm BuildAdjacencyList, for each node in set N we traverse the edges in set E in line 1 and line 2. We define the nodes of an edge start node v_{si} and end node v_{ej} respectively. In line 3 to line 4 we add the end node v_{ej} of an edge to the OutDegreeList of node v_i, when node v_i equals the start node v_{si} of the edge. Finally, we calculate the OutDegreeList of v_i in line 6.

3.2. **Mining influential nodes.** Based on the out-degree adjacency list of node v_i, we can calculate NS of each node. Algorithm ComputeNodeScore is used to mine influential nodes in the software complex network. We calculate the influence from $IDN(v_i)$ and $DDN(v_i)$, as shown in algorithm ComputeNodeScore.

As shown in algorithm ComputeNodeScore, we evaluate the influence of a node in the network by an iterative process. Firstly, we initialize NS of node v_i as a constant 0. We also build a queue for node v_i which is stored the set $DDN(v_i)$. Line 2 to line 3 is the process to compute the probability of node v_i call node $v_j \in DDN$. If the out-degree of node v_i is not equal to 0, we set the possibility to $1/d_i^{out}$. Otherwise, the possibility is 0. Line 4 to line 6 is the process of enqueueing of v_j when one of the set $DDN(v_i)$ is not in the queue. Then the NS of node v_i in RN is calculated iteratively in line 7. Finally, we calculate NS of v_i in line 10.

Next, we rank the NS of these nodes and mine the top-k influential nodes in the software network.

Algorithm 2 ComputeNodeScore

Inputs: node v_i, OutDegreeList(v_i)
Output: the NS of node v_i

```
Process:
Initialize NS(v_i) = 0
if (OutDegreeList [v_i] != null) {
  p = 1/OutDegreeList.size();
  for (each v_j ∈ OutDegreeList [v_i]) {
    if (v_j is not in Queue(v_i)) {
      Queue(v_j).add(v_j);
      NS(v_i)+ = p* NodeScoreNumber(v_j);
    }
  }
  NS(v_i) = p*OutDegreeList.size() + NS(v_i);
}
else {
  p = 0;
}
```

4. **Experiments and Analysis.** In this section,we choose two open source software *Tar* and *cflow* to validate the algorithm ComputeNodeScore. *Tar* is a decompression software for Linux, and *cflow* is an analysis tool for C program to extract the relationship of function calls (download from the open source software library: Http://sourceforge.net).

Firstly we run the algorithm on each version of *Tar* and *cflow*. By the algorithm ComputeNodeScore, we calculate the NS of each function node. Here we mine top-10 nodes in each version about software *Tar* and *cflow*. It is shown in Table 1 and Table 2 respectively.

As it is shown in Table 1, for version *tar-1.24*, *tar-1.25* and *tar-1.26*, the NodeScore of the top-10 are almost the same. The reason is that the difference among the three versions only reflects on the number of function calls. In other words, there is no change of the component function of these three versions. In the latest two versions, some new functions added into the software results in the ranking variation. For example, in the prior versions node *name_add_name* ranked 9th but it ranked 36th in version *tar-1.27* and *tar-1.28*. Table 2 shows the top-10 influential functions of software *cflow* in different versions. The ranking of a function in each version of *cflow* varies but with little range.

Table 1. Top-10 influential nodes for each version of software *tar*

	tar-1.24	tar-1.25	tar-1.26	tar-1.27	tar-1.28
function_name	rank/NS	rank/NS	rank/NS	rank/NS	rank/NS
write_header_name	1/5.00	1/5.00	1/5.00	1/5.00	1/5.00
find_next_block	2/5.00	2/5.00	2/5.00	2/5.00	2/5.00
write_short_name	3/4.00	3/4.00	3/4.00	3/4.00	3/4.00
flush_archive	4/4.00	4/4.00	4/4.00	4/4.00	4/4.00
gnu_flush_write	5/3.00	5/3.00	5/3.00	5/3.00	5/3.00
write_eot	6/2.67	6/2.67	6/2.67	6/2.67	6/2.67
start_header	7/2.56	7/2.56	7/2.56	7/2.56	7/2.56
create_archive	8/2.16	8/2.16	8/2.16	9/2.33	8/2.16
name_add_name	9/2.00	9/2.00	9/2.00	36/1.00	36/1.00
init_buffer	10/2.00	10/2.00	10/2.00	10/2.00	10/2.00

TABLE 2. Top-10 influential nodes for each version of software *cflow*

	cflow-1.1	cflow-1.2	cflow-1.3	cflow-1.4
function_name	rank/NS	rank/NS	rank/NS	rank/NS
parse_opt	1/4.00	1/4.00	4/3.00	6/3.00
skip_to	2/3.67	2/3.67	1/3.62	1/3.63
get_token	3/3.33	3/3.33	2/3.23	3/3.23
add_reference	4/3.25	4/3.25	29/2.23	30/2.25
parse_knr_dcl	5/3.18	5/3.16	3/3.15	4/3.15
main	6/3.06	6/3.06	11/2.94	5/3.01
add_name	7/3.00	7/3.00	32/2.00	34/2.00
delete_autos	8/3.00	8/3.00	22/2.50	23/2.50
collect_symbols	9/3.00	9/3.00	5/3.00	24/2.50
begin	10/3.00	10/3.00	6/3.00	9/3.00

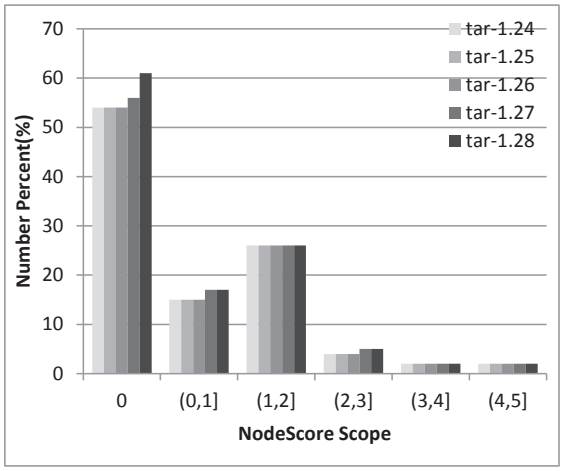

FIGURE 3. Number percentage of nodes in score scope of software *Tar*

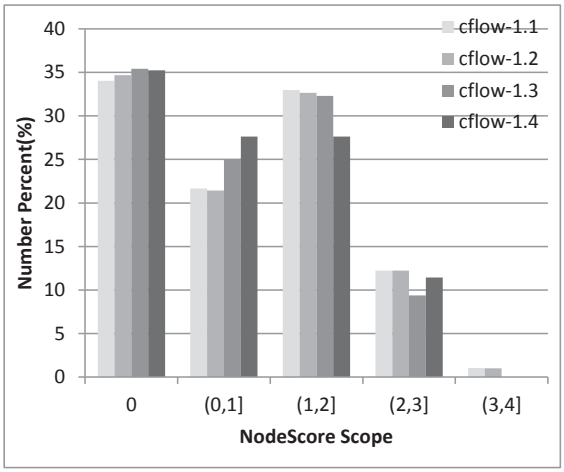

FIGURE 4. Number percentage of nodes in score scope of software *cflow*

For example, function *skip_to*'s ranking ranges from 1 to 2. So we can make a prediction that it may still be more influential than most others in next new version.

In addition, the number of nodes which have high NS is rather small in each version. These high score nodes have taken a great part in ensuring software reliability and stability. It means that there are few functions that should be paid more attention to in software updating and software maintenance. We calculate the percentage of functions' NS that equal 0 and in the range of $(0, 1]$, $(1, 2]$, $(2, 3]$, $(3, 4]$, $(4, 5]$. The results of software *Tar* and *cflow* are shown in Figure 3 and Figure 4 respectively.

As we can see in Figure 3, 50% of functions' NodeScore equals 0. They are ordinary functions. In other words, we would not pay more attentions to them. Meanwhile, the NS of nodes around 5 is less than 2% in each version. These 2% nodes that have high NodeScore should be paid more attention. They play important roles in the process of software updating and software maintenance. For software *cflow*, the number percentage of nodes in each scope of different versions is shown in Figure 4. It has the same characteristic with software *Tar*. The number of nodes with high NS is much less than the number of low NS. By paying more attention to these influential nodes in future versions, we can improve software reliability and stability. Thereby we can greatly reduce the amount of work and improve work efficiency.

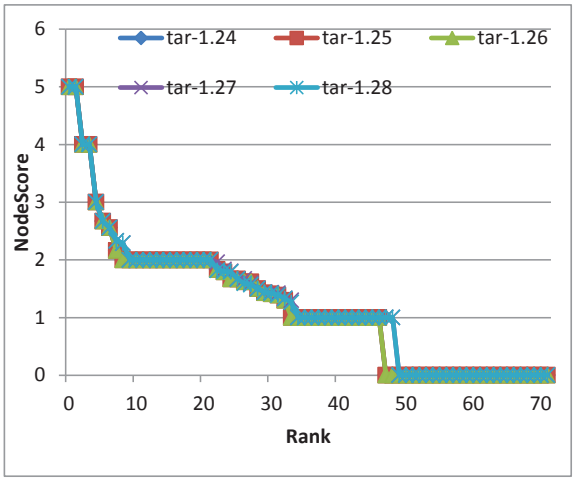

FIGURE 5. NS distribution of *Tar* (top-70)

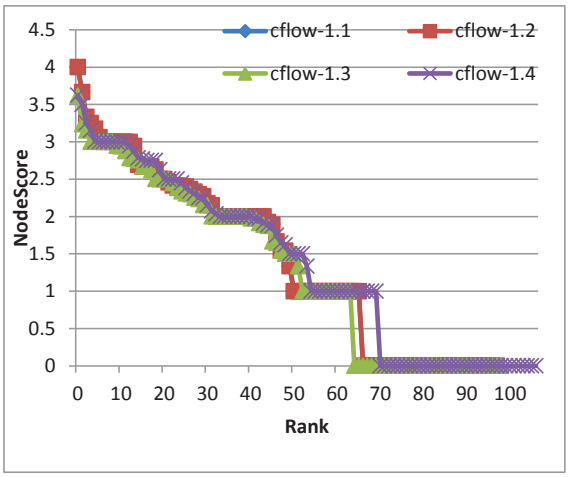

FIGURE 6. NS distribution of *cflow* (top-100)

At the same time, the NodeScore of the same ranking nodes within different versions has slight wave, as show in Figure 5 and Figure 6.

As it is shown in Figure 5, the NS distribution of software *Tar* is similar extremely in the five versions. With the increasing of node ranking, the NodeScore of each node shows a decrease trend. As the lower rank, the score shows a trend of increase. The higher NS ranges from 4 to 5. Most nodes' score are around 1 or 2 and others are close to 0. The development of versions follows the same laws, the node's NodeScore of a certain ranking remain stable and the NS distribution of different software version is nearly the same. So, we can predict the future versions' trends based on this. Meanwhile, Figure 6 shows the NS distribution of software *cflow*, and the curve of each version has the same tendency. The higher NS ranges from 3 to 4 and most nodes' score range from 1 to 3. The NS distribution of software *cflow* in different versions follows the same trend.

5. **Conclusions and Future Work.** To make it more convenient and flexible in software version updating and software maintenance, a novel method was proposed in this paper to mine the influential nodes in software network. The algorithm named ComputeNodeScore was used to calculate the NodeScore and evaluate the importance of each node in software network. The method considered the influence from directly dependence nodes and indirectly dependence nodes. Then, we mine top-k nodes from all nodes by the NS. Experimental results indicate that the proposed approach is effective and can help us to identify influential nodes in software network.

Although the approach we proposed shows some feasibilities in identifying influence nodes in complex software network, the broad validity of our approach should be demonstrated further. Our future work is using more open-source software network to evaluate the validity to improve our approach.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61170190, No. 61472341 and the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2013203324, No. F2014203152.

REFERENCES

[1] P. Bhattacharya, M. Iliofotou, I. Neamtiu et al., Graph-based analysis and prediction for software evolution, *Proc. of the 34th International Conference on Software Engineering*, pp.419-429, 2012.

[2] P. Hosek and C. Cadar, Safe software updates via multi-version execution, *Proc. of the International Conference on Software Engineering*, pp.612-621, 2013.

[3] X. Ge and E. Murphy-Hill, Manual refactoring changes with automated refactoring validation, *Proc. of the 36th International Conference on Software Engineering*, pp.1095-1105, 2014.
[4] L. Wang, P. Yu, Z. Wang et al., On the evolution of linux kernels: A complex network perspective, *Journal of Software: Evolution and Process*, vol.25, no.5, pp.439-458, 2013.
[5] H. Li, H. Zhao, W. Cai et al., A modular attachment mechanism for software network evolution, *Physica A: Statistical Mechanics and Its Applications*, vol.392, no.9, pp.2025-2037, 2013.
[6] S. Valverde and R. V. Solé, Hierarchical small worlds in software architecture, *arXiv Preprint Cond-mat/0307278*, 2003.
[7] K. Y. Cai and B. B. Yin, Software execution processes as an evolving complex network, *Information Sciences*, vol.179, no.12, pp.1903-1928, 2009.
[8] M. Huang, P. Sun and B. Xiao, *Centrality Different Influence Models of Node Centrality in Transactional Community*, 2014.
[9] D. Chen, L. Lv, M. S. Shang et al., Identifying influential nodes in complex networks, *Physica A: Statistical Mechanics and Its Applications*, vol.391, no.4, pp.1777-1787, 2012.
[10] 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.
[11] J. Bae and S. Kim, Identifying and ranking influential spreaders in complex networks by neighborhood coreness, *Physica A: Statistical Mechanics and Its Applications*, vol.395, pp.549-559, 2014.
[12] J. G. Liu, Z. M. Ren and Q. Guo, Ranking the spreading influence in complex networks, *Physica A: Statistical Mechanics and Its Applications*, vol.392, no.18, pp.4154-4159, 2013.