Copyright ⓒ 2024 KSII

Software Key Node Recognition Algorithm for Defect Detection based on Node Expansion Degree and Improved K-shell Position

Wanchang Jiang[1], and Zhipeng Liu[1*]
[1] School of Computer Science, Northeast Electric Power University
Jilin 132012, China
[e-mail: jwchang84@163.com, liuzp0526@163.com]
[*]Corresponding author: Zhipeng Liu

Received January 3, 2024; revised May 14, 2024; accepted June 18, 2024; published July 31, 2024

Abstract

To solve the problem of insufficient recognition of key nodes in the existing software defect detection process, this paper proposes a key node recognition algorithm based on node expansion degree and improved K-shell position, shortened as SDD_KNR. Firstly, the calculation formula of node expansion degree is designed to improve the degree that can measure the local defect propagation capability of nodes in the software network. Secondly, the concept of improved K-shell position of node is proposed to obtain the improved K-shell position of each node. Finally, the measurement of node defect propagation capability is defined, and the key node recognition algorithm is designed to identify the key function nodes with large defect impact range in the process of software defect detection. Using real software systems such as Nano, Cflow and Tar to design three sets of experiments. The corresponding directed weighted software function invoke networks are built to simulate intentional attack and defect source infection. The proposed SDD_KNR algorithm is compared with the BC algorithm, K-shell algorithm, KNMWSG algorithm and NMNC algorithm. The changing trend of network efficiency and the strength of node propagation force are analyzed to verify the effectiveness of the proposed SDD_KNR algorithm.

Keywords: Software Defect Detection, Node Expansion Degree, K-shell Position, Key Node Recognition

http://doi.org/10.3837/tiis.2024.07.006 **ISSN : 1976-7277**

1. Introduction

With the advancement of digital intelligence, software systems are becoming increasingly intricate, leading to a higher likelihood of software failure [1]. In the process of software development, it is inevitable that modules have defects [2]. Once these defects are exposed during the formal use of the software, they will affect the operation of the software and even cause the cascade collapse of the entire system [3]. Similar to the spread of infectious diseases in the population, the defects in the software system will spread to other modules without defects with the dependencies between software system modules, such as method calls and parameter passing, resulting in other modules cascading failures [4]. In the dynamic execution state of software, the destruction of a small number of key modules with defects in the software system will have a greater impact on the software system, and most other software defects only have a limited impact on the software system [5]. Therefore, if we can accurately detect the modules that have a greater impact on defects, and pay attention to these key modules, it will have important reference value for increasing the stability and reliability of software systems.

Complex network analysis provides a new perspective for analyzing software systems [6,7]. When the software system structure is represented as a network, entities can be extracted from different granularities such as packages, classes, methods, and attributes as nodes, and the dependencies between them can be regarded as edges. Through the combination of these nodes and edges, a software network is constructed [8]. From a network perspective, the identification of critical nodes in software system defect detection begins at the class granularity level. With the deepening of research, scholars have observed that by considering the function in the software system as the fundamental unit of analysis for identifying critical nodes in software defect detection, it is possible to pinpoint the causes of software defects at a more precise level. From the perspective of local node measurement, Dong Jun et al. [9] utilized the typical degree centrality algorithm in complex networks to pinpoint pivotal nodes within the network. Wu Hongfei et al. [10] established a directed weighted software network model and proposed a key node identification algorithm. By considering the influence of neighbor nodes and secondary neighbor nodes on the node, the algorithm makes the node have a better distinction in the local scope of the software network. However, these algorithms ignore the influence of the global information of the software network on the importance of nodes. Wang Qian et al. [11] proposed the concept of structural entropy and utilized node structural entropy to assess the significance of nodes. Employing a global measurement approach, the algorithm discerns function nodes with diminished local significance yet substantial global impact within the network. Xu et al. [12] used the K-shell position of the node to calculate the influence value of the node in the global range of the network, and used the influence value of the node neighbor and the secondary neighbor node to calculate the comprehensive influence value of the node in the local range of the network to obtain the key nodes in the network. The algorithm enhances the precision with which key nodes are identified. Existing algorithms measure the importance of nodes from the perspective of local and global measurement of nodes in the network, but there are still some shortcomings: 1) The algorithm regards the influence of neighbor nodes and secondary neighbor nodes of nodes in the network as the same position; 2) The recognition of key nodes in the network is not enough.

To solve the above problems, this paper proposed a key node recognition algorithm for software defect detection based on node expansion degree and improved K-shell position. The node defect propagation capability measure is defined from the perspective of node defect

propagation, and the key nodes in the software defect detection process are identified according to the measured value. Firstly, the concept of expansion degree is proposed, and the influence coefficient that can be dynamically adjusted is set to balance the different influence degrees of the out-edge neighbors and the out-edge secondary neighbors of the nodes. According to the influence coefficient, the expansion degree of each node is obtained, and the influence of the node on the local structure of the network is measured by the node expansion degree. Secondly, the K-shell algorithm is used to stratify the network, and the improved K-shell position of each node is obtained. The influence of the global structure of the network is measured by the improved K-shell position of the node. Finally, the measurement of node defect propagation capability is defined, and the measurement value of node defect propagation capability is obtained by combining the node expansion degree with the node improved K-shell position. The experimental results show that in the process of simulating software defect detection, the proposed algorithm can better identify the key function nodes in the software network.

2. Directed Weighted Software Function Invoke Network

In this section, the dependencies among function granularity units in a software system are extracted dynamically. The function entity is regarded as a node in the network, and the invoke relationship between functions is regarded as a directed edge in the network, and the direction of the directed edge is consistent with the invoke relationship between the corresponding nodes. In addition, the degree of defect propagation between functions is regarded as the edge weight coefficient for the directed edge [10]. As a result, the directed weighted software function invoke network is built.

Definition 2.1. Directed software function invoke network (DFIN). It is denoted by a two-tuples $DFIN=(V,E)$. $V=\{v_i \mid i=1,2,...,N\}$ is the set of N nodes, where node v_i represents the function entity numbered i in the network, and N is the number of nodes in the network. $E=\left\{e_{kj} \middle| e_{kj}=\left(v_k,v_j\right),\ v_k\in V,\ v_j\in V\right\}$ is the set of directed edge, where e_{kj} is a directed edge formed by a pair of ordered function nodes $\left(v_k,v_j\right)$ in the dynamic execution of software, $e_{kj}\neq e_{jk}$.

And the function entity numbered k calls the function entity numbered j , node v_k is the calling function node of e_{kj} , node v_{j} is the modulated function node of e_{kj} . In particular, if the directed edge set E does not have any edge with function entity i as the calling function node, then the node v_i is called a leaf function node. And the node set of all leaf function nodes in V that satisfy this condition is represented as V_n .

Definition 2.2. Function call chain (FCC). In every software system, a solitary entry function initiates execution, sequentially invoking multiple subordinate functions until culminating in a leaf function that marks the sequence's termination. This ordered series of function invocations, encompassing both the initiating entry and the concluding leaf function, is collectively termed a "function call chain". Suppose the entry function node is $v_a(v_a\in V)$, then a function call chain from node v_a to node $v_{\mathrm{z}}(v_{\mathrm{z}}\in V_n)$ is represented as

$f_{az}=(v_a,...,v_k,v_j,...,v_z)$, for any two adjacent nodes v_k and v_j in the above function call chain, there is $e_{kj}\in E$, and this function call chain f_{az} contains e_{kj}. The chain set of all function call chains from the entry function node and terminating at each leaf function node is represented as $C=\{f_{az}\mid f_{az}=(v_a,...,v_k,v_j,...,v_z),v_a\in V,v_K\in V,v_j\in V,v_z\in V_n\}$.

In the directed software function invoke network, some directed edges are used by more function call chains, and some directed edges are used by less function call chains. This indicates that during the execution of software functions, the directed edge e_{kj} can make different contributions. By calculating the number of function call chains containing the directed edge e_{kj} in C, different weights are given to the directed edge e_{kj}. The directed edge with higher weight indicates that it makes more contribution in the process of software executing functions. The set of chains consisting of chains of function calls in C containing e_{kj} is denoted as C_{kj}, use the number of elements in C_{kj} chain set $|C_{kj}|$ to calculate the weight w_{kj} of e_{kj}, as shown in equation (1).

$$w_{kj}=|C_{kj}| \tag{1}$$

Definition 2.3. Directed weighted software function invoke network (DWFIN). Each directed edge in $DFIN$ is weighted, and a triple $DWFIN=(V,E,W)$ is used to represent the directed weighted software function invoke network. Edge weight coefficient sets $W=\{w_{kj}\mid w_{kj}=|C_{kj}|,e_{kj}\in E\}$.

The edge weights establish a one-to-one correspondence between the elements of set w_{kj} and set e_{kj}, ensuring that both sets contain an identical number of elements.

3. Software Key Node Recognition Algorithm for Defect Detection

3.1 Node Expansion Degree

In software networks, node degree is a property that can measure the ability of nodes to propagate defects locally in the network. It has been widely used in various key node identification algorithms in software defect detection [10]. However, the ability of nodes to propagate defects is not only related to themselves, but also related to the ability of neighbor nodes to propagate defects. If a node itself has a weak ability to spread defects, but its neighbor nodes have a strong ability to spread defects, then according to the neighborhood principle, it is considered that the node has a strong ability to spread defects [12]. The degree of neighbor nodes in the network topology essentially characterizes the ability of the secondary neighbor nodes to propagate defects. Therefore, this subsection sets a dynamically adjustable influence coefficient μ_{v_i} to balance the influence degree of outgoing edge neighbor nodes and outgoing edge secondary neighbor nodes on the defect propagation ability of the specified node in the local scope. The node expansion degree is designed according to the influence coefficient μ_{v_i} to measure the ability of a node in the directed weighted function call network to locally propagate defects in the network.

Definition 3.1. Node nearest neighbors set on out-direction (NNOD). In DWFIN, there is edge e_{kj}, and the modulated function node v_j is the outgoing neighbor of the calling function node v_k. Meet with the calling function of node v_k all the callback function of node v_j node set $\{v_j\}$ for the node v_k nearest neighbors set on out-direction, counted as V_o^k, $|V_o^k| = K(v_k)$ for the node number v_k nearest neighbors set on out-direction. If there is no edge in the edge set E that takes node v_k as the calling function, then nearest neighbors set on out-direction of node v_k is an empty set, and $V_o^k = \{\}$. The weight of the nearest neighbors set on out-direction of node v_k is equal to the sum of the weights of all edges with node v_k as the calling function, that is, $w_o^k = \sum_{v_j \in V_o^k} w_{kj}$.

Definition 3.2. Node next nearest neighbors set on out-direction (NNNOD). In DWFIN, for $\forall v_m \in V_o^k$, the nearest neighbors set on out-direction of node v_m is denoted as V_o^m. Define node v_k next nearest neighbors set on out-direction $V_{oo}^k = \{v_m^k \mid v_m \in V_o^k \text{ and } v_m^k \in V_o^m\}$, $|V_{oo}^k| = D(v_k)$ for the node number v_k next nearest neighbors set on out-direction. If there is no edge in the edge set E with node v_m as the calling function or the nearest neighbors set on out-direction of node v_k is an empty set, then next nearest neighbors set on out-direction of node v_k is an empty set, and $V_{oo}^k = \{\}$. The weight of the next nearest neighbors set on out-direction of node v_k is equal to the sum of the weights of all edges with node $v_m (v_m \in V_o^k)$ as the calling function, that is, $w_{oo}^k = \sum_{v_j \in V_{oo}^k} w_{mj}$.

During the execution of a function within software, the set of direct neighbors (i.e., nodes that are immediately invoked as function nodes) exerts a more significant influence on the proper functioning of the software than the set of indirect neighbors (i.e., nodes that are invoked through secondary connections). To address this disparity, we have devised a dynamic adjustment mechanism for the impact factors, which allows for the balanced assessment of the influence exerted by both directly and indirectly invoked function nodes on the node in question. The node expansion degree is defined accordingly.

Definition 3.3. Node expansion degree (NED). The node expansion degree $NED(v_i)$ of the node v_i in the directed weighted software function invoke network is defined to measure the ability of the node v_i to propagate defects locally in the network. The calculation is shown in equation (2).

$$NED(v_i) = K(v_i) + u_{v_i} D(v_i) \tag{2}$$

Among them, $K(v_i)$ is the number of nearest neighbors set on out-direction of node v_i, $D(v_i)$ is the number of next nearest neighbors set on out-direction of node v_i, and u_{v_i} is the influence coefficient of $D(v_i)$. Through the influence coefficient, the influence degree of the

function node directly called by the node v_i and the function node indirectly called on the node is adjusted. The calculation process of μ_{v_i} is shown in equation (3).

$$\mu_{v_i} = \frac{w_o^i}{w_o^i + w_{oo}^i} \tag{3}$$

Among them, w_o^i represents the sum of the weight of the node v_i's nearest neighbors set on out-direction, as derived from Definition 3.1. And w_{oo}^i represents the sum of the weight of the node v_i's next nearest neighbors set on out-direction, obtained through Definition 3.2.

The directed edge weight of nodes in the network represents the degree of defect propagation between nodes. In the local scope, the neighbor node is closer to the specified node and has greater influence on it, while the sub-neighbor node is farther away from the specified node and has less influence on it. Therefore, the influence coefficient u_{v_i}, which can be dynamically adjusted, is calculated to balance the influence of the nearest neighbors on out-direction of node v_i and the next nearest neighbors on out-direction of node v_i on the defect propagation of node v_i.

3.2 Improved K-shell Position of Node

In the network, the node expansion degree measures the ability of node defect propagation to a certain extent, but it is still an attribute characteristic of the node itself. It reflects the local defect propagation ability of nodes in the network, and ignores the global defect propagation ability of nodes. The K-shell decomposition algorithm quickly divides the network from the outside to the inside according to the node location information, and the k-shell position of the node after partition represents the relative position of the node in the network. The more the K-shell position of the node is in the core, the greater the influence of the node on the network [13]. Xu et al. [12] have demonstrated that the K-shell decomposition algorithm effectively quantifies a node's global defect propagation capacity within the network. However, this algorithm operates on a coarse-grained level, often assigning identical K-shell positions to a multitude of nodes. This approach implies a uniform importance among these nodes, which contradicts the inherent variability in the significance of functional entities within software systems. To more precisely distinguish the importance of functional entities, this manuscript employs an improved K-shell position metric to assess a node's global defect propagation capacity within the network. The program instance written in C language is abstracted as a directed weighted software function invoke network. The network constructed by the program instance is shown in **Fig. 1**. It consists of 12 nodes and 14 edges. The K-shell decomposition of the constructed network is performed to calculate the improved K-shell position of the node. The specific process is as follows:

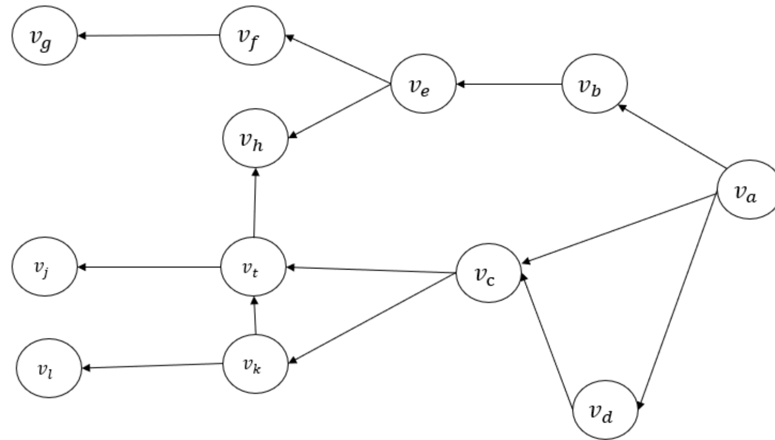


Fig. 1. Directed weighted software function invoke network

(a) The directed weighted software function invoke network in **Fig. 1** is regarded as the corresponding undirected weighted software function invoke network, and the nodes with degree 1 in the corresponding undirected weighted software function call network are deleted. The first deleted nodes are v_g, v_j, v_l. Then, the deletion operation is repeated for the nodes with degree of 1 in the remaining network after removing the nodes, and the second deleted node is v_f. At this time, the degree of nodes in the remaining network is at least 2, so the K-shell position of all nodes deleted above is 1.

(b) Repeat the process in (a), find the nodes with degree of 2 in the remaining network for deletion, and the nodes deleted for the third time are v_b, v_d, v_e, v_h, v_k. Then, the nodes with a degree of 2 in the remaining network after removing the nodes are deleted repeatedly, and the fourth deleted nodes are v_c, v_a, v_t. At this time, all nodes in the network have been deleted, so the K-shell position of all nodes deleted above is 2.

From the view of the network topology in **Fig. 1**, node v_f and node v_g are obviously different in importance. However, due to the coarse-grained problem of the K-shell decomposition algorithm, node v_f and node v_g have the same K-shell position, which means that the K-shell decomposition algorithm considers node v_f and node v_g to have the same influence on the global network structure. Obviously, this is inconsistent with the reality. It can be seen from the decomposition process (a) (b) of K-shell algorithm that the number of iterative layers of node v_f and node v_g for deletion operation in K-shell decomposition process is different. If the number of iterative layers when nodes are deleted is used as improved K-shell position of node, then the importance of nodes can be further distinguished.

The directed weighted software function invoke network removes the function nodes according to the above K-shell algorithm decomposition process (a) and process (b). The number of iterations when removing the function node v_i is called improved K-shell position of the node v_i, which is denoted as $NIKP(v_i)$. That is, if in the directed weighted software function invoke network, the number of iterations that the function node v_i is removed is q, then the improved K-shell position of the function node v_i can be expressed as $NIKP(v_i) = q$.

In this way, the improved K-shell position of each node in the directed weighted software function invoke network in **Fig. 1** is shown in **Table 1**.

Table 1. Improved K-shell position of all nodes

Improved K-shell position of node	Delete nodes	K-shell position of node
1	v_g、v_j、v_l	1
2	v_f	1
3	v_b、v_d、v_e、v_h、v_k	2
4	v_c、v_a、v_t	2

For the directed weighted software function invoke network node v_f and node v_g shown in **Fig. 1**, the K-shell position of node is 1, the number of iterations when the K-shell decomposition algorithm deletes the v_f node is 2, and the number of iterations when deleting the v_g node is 1. Therefore, the improved K-shell position of the node v_f is 2, and the improved K-shell position of the node v_g is 1. From the perspective of network topology, the relative position of node v_f in the network is closer to the root node than that of node v_g, and it should have a stronger influence in the global structure of the network, indicating that the improved K-shell position of node can more accurately represent the importance of nodes in the network.

3.3 Measurement of Node Defect Propagation Capability

When a node in the directed weighted software function invoke network has a defect, the defect may be propagated to the neighboring nodes through the node invocation relationship in the network, so that the neighboring nodes also have defects. However, due to the relative position of the node in the network, the range of defect propagation may be limited. Therefore, assessing the defect propagation solely within the local or global context of a node is a limited approach. It does not accurately capture the node's true defect propagation potential within the network. Deng et al. [14] proposed a node importance recognition algorithm by combining the node degree with the node K-shell position. The algorithm has achieved good results in the identification of node importance in different complex networks. The basic idea of the algorithm is that the nodes with great influence in the local range and close to the core of the network should have greater influence. This is consistent with the characteristics of functional entity defect propagation in software systems. In the software system, if a function entity has an important position in its own module, and the module to which the function entity belongs to the core module of the software system, then when the function entity defects, the defects will have a great impact on the entire software system with the invoke between functions. In this section, a measure of node defect propagation capability is proposed to analyze the degree of node defect propagation by considering node expansion degree and improved K-shell position of node.

Definition 3.4. Node defect propagation capability (NDPC). The local defect propagation capability of the node v_i in the directed weighted software function invoke network can be measured by the node expansion degree. The global defect propagation capability of the node v_i can be measured by the improved K-shell position measurement of the node. The node

defect propagation capability measure $NDPC(v_i)$ of the node v_i is defined to measure the capability of the node v_i to propagate its own defects to other nodes in the network. The calculation is shown in equation (4).

$$NDPC(v_i) = NED(v_i) * NIKP(v_i) \quad (4)$$

Among them, $NDPC(v_i)$ represents the capability of node v_i to propagate its own defects in the network. The larger the NDPC value of node v_i is, the greater the capability of node v_i to propagate defects, and the more likely errors occur, resulting in software system crash.

Definition 3.5. key nodes (KN). The key nodes in DWFIN are the set of nodes v_i in the network which are sorted from large to small according to the NDPC value of node defect propagation ability, and the top P nodes. Expressed as $KN = \{v_{i_j} \mid j = 1, 2, ..., P\}$, where v_{i_j} is the node whose defect propagation ability ranks j.

3.4 Software Key Node Recognition Algorithm for Defect Detection based on Node Expansion Degree and Improved K-shell Position

Using the software network node defect propagation capability measure NDPC proposed in Section 3.3, a software defect detection key node recognition algorithm based on node expansion degree and improved K-shell position (SDD_KNR) is designed to identify the key nodes in the software defect detection process. The key node recognition algorithm SDD_KNR is divided into four main stages. Firstly, the expansion degree NED of each node is calculated on the directed weighted software function invoke network. The node expansion degree calculates the capability of the node to propagate defects in the local range of the network by balancing the influence of the node neighbor node and the secondary neighbor node on it. Secondly, the K-shell decomposition of the directed weighted software function invoke network is performed to obtain the improved K-shell position NIKP that can characterize the defect propagation capability of the node in the global range of the network. Then, the node defect propagation capability NDPC is obtained by combining the node expansion degree NED with the improved K-shell position of node NIKP. Finally, the node set composed of P nodes with the highest NDPC value of node defect propagation capability is the key node of software defect detection.

The algorithm described as follows:

Input： $DWFIN = (V, E, W)$

Output： KN

Main program： $getNDPC(v_i)$

1) Initialize set $Set = NULL$， $KN = NULL$
2) Calculate the $NIKP(v_i)$ for each node $v_i (v_i \in V)$ // The second stage
3) For each $v_i (v_i \in V)$ do
4) $NDPC(v_i) = NED(v_i) * NIKP(v_i)$
5) Add $NDPC(v_i)$ to Set // The third stage
6) Sort the values in Set in descending order
7) Put the nodes corresponding to the first Top-P values in Set into KN // The fourth stage
8) Return KN

Subroutine： $getNED(v_i)$ // The first stage

1) Get V_o^i of node v_i and calculate the number of $V_o^i K(v_i)$
2) Calculate the w_o^i of node v_i
3) Initialize $D(v_i)=0, w_{oo}^i=0$
4) For each $v_j(v_j \in V_o^i)$ do
5) Calculate the V_o^j of node v_j
6) Calculate the number of $V_o^j K(v_j)$
7) Calculate the w_o^j of node v_j
8) $D(v_i)=D(v_i)+K(v_j)$
9) $w_{oo}^i = w_{oo}^i + w_o^j$
10) Calculate $u_{v_i} = w_o^i / (w_o^i + w_{oo}^i)$
11) Calculate $NED(v_i)=K(v_i)+\mu_{v_i} * D(v_i)$
12) Return $NED(v_i)$

The complexity of the SDD_KNR algorithm is primarily concentrated in two critical processes: firstly, the preprocessing phase, which involves constructing a directed, weighted network of software function calls. During the encoding phase, this study utilizes a depth-first search (DFS) strategy to enumerate all function call chains within the network, facilitating the subsequent computation of the directed edge weights. The most extensive search scenario entails traversing every node and edge, yielding a time complexity of $O(N+E)$, where N signifies the node count, and E represents the edge count. Secondly, we calculate the defect propagation capacity of nodes, a process that primarily involves assessing both the expansion degree and the improved K-shell position of each node. In the most exhaustive scenario for calculating the expansion degree, the search encompasses all nodes and edges within the network, resulting in a time complexity of $O(N)$. For determining the improved K-shell position, the most extensive search scenario involves iterating through all network nodes, leading to a time complexity of $O(N+E)$. Consequently, the overall time complexity for this process is $O(N)+O(N+E)$. Considering the two aforementioned processes, the overall time complexity of our algorithm is $2*O(N+E)+O(N)$, which is not excessively high. This moderate complexity renders the algorithm suitable for the majority of large-scale software systems.

3.5 Case Calculation

The directed weighted software function invoke network is abstracted from the program instance in **Fig. 1**, which is used as a network instance to illustrate the solution process of the SDD_KNR algorithm. The node v_a as an example. The solution process is as follows:

(1) Calculate the weight sum of the nearest neighbors set on out-direction of node v_a. The node set $\{v_b, v_c, v_d\}$ is the nearest neighbors set on out-direction of node v_a. According to the

equation (1), the corresponding weights are {2,5,5}, then the weight sum of the nearest neighbors set on out-direction of node v_a is 12.

(2) Calculate the weight sum of the next nearest neighbors set on out-direction of node v_a. The node set $\{v_e, v_t, v_k, v_c\}$ is the next nearest neighbors set on out-direction of node v_a, and its number is 4. According to equation (1), the corresponding weights are {2, 4, 6, 5}, then the weight sum of the next nearest neighbors set on out-direction of the node v_a is 17.

(3) The NED value of the expansion degree of node v_a is calculated. According to (1), (2) and equation (3), the influence coefficient of node v_a is denoted as μ_{v_a} and its value is 0.4. Then according to equation (2), the expansion degree of v_a can be obtained as 4.6.

(4) The NDPC value of node defect propagation capability of node v_a is calculated. Using improved the K-shell decomposition algorithm, it is easy to get the improved K-shell position of node v_a is 4. According to equation (4), the defect propagation capability of node v_a can be obtained as 18.4.

Similarly, the NDPC value of the node defect propagation capability of other nodes can be obtained. The NMNC algorithm [12] is used to calculate the capability value of node defect propagation of the network in **Fig. 1**, and the obtained results are compared with the results of the SDD_KNR algorithm. The node defect propagation capability values obtained by the two algorithms are shown in **Table 2**.

Table 2. The ranking of values of node defect propagation capability of each node

Rank	1	2	3	4	5	6	7	8	9	10	11	12
SDD_KNR Number	v_a	v_c	v_k	v_e	v_t	v_b	v_d	v_f	v_g	v_h	v_j	v_l
SDD_KNR Value	18.4	14.4	8.4	8.1	8.0	6.0	4.8	2.0	0	0	0	0
NMNC Number	v_c	v_t	v_k	v_a	v_d	v_e	v_h	v_b	v_j	v_l	v_f	v_g
NMNC Value	108.1	102.8	82.9	79.2	64.8	54.2	52.8	45.2	33.2	28.1	26.0	12.8

It can be seen from **Table 2** that the SDD_KNR algorithm in this paper takes the lead in identifying the entry node v_a. From the network, it can be known that the node v_a does occupy a great advantage in structure. Through the node v_a, all nodes in the network can be called. If the node v_a is protected in advance, the normal operation of the program can be effectively protected. Among other nodes in the network, for example, nodes v_b and v_d have the same network structure, they are all the nearest neighbors on out-direction of node v_a. And the nearest neighbors on out-direction of node v_b is v_e, the nearest neighbors on out-direction of node v_d is v_c, and both node v_e and node v_c have two nearest neighbors on out-direction nodes. But nodes v_e, v_f and v_g can only be called by node v_b, and nodes that node v_d can call can be called by node v_c. Therefore, from the perspective of network structure, node v_b is more important than node v_d. In identifying the node defect propagation capability, the recognition result of SDD_KNR algorithm is that the defect propagation capability of node v_b

is greater than that of node v_d . But the recognition result of NMNC algorithm is that the defect propagation capability of node v_d is greater than that of node v_b . The SDD_KNR algorithm is closer to the real situation of the network structure, indicating that the SDD_KNR algorithm can accurately identify the nodes with greater defect propagation capability in the network.

4. Experiments

4.1 Dataset description

Experiments are done on a PC at AMD Ryzen 5 5600H CPU @ 3.3 GHz with 16 GB of RAM.

To evaluate the performance of the SN_KNR algorithm, we conducted experimental analyses utilizing three procedural software systems: Tar, Nano, and Cflow. The three software systems exhibit variations in the number of function entities they encompass, as well as in the range of functionalities these functions can perform. Within the Ubuntu environment, we conducted dynamic execution tracing of the software system using the GCC compiler and the Pvtrace tool. We marked the software functions to capture their dynamic invocation sequences, which were then logged in .dot files. Subsequently, these .dot files were utilized to construct a directed, weighted network of software function calls. **Table 3** presents the statistical data for the three distinct network types under investigation. Among them, |V| represents the number of nodes in the network, |E| represents the number of edges in the network, |K| represents the average degree of the network, <d> represents the average shortest path of the network, C represents the clustering coefficient of the network, M represents the density of the network, and D represents the diameter of the network.

Table 3. Network statistics information of three software systems

Software	|V|	|E|	<K>	<d>	C	M	D
Tar	190	260	2.737	4.864	0.0144	0.0145	11
Nano	103	151	2.932	3.783	0.0423	0.0287	9
Cflow	106	179	3.377	4.791	0.0928	0.0322	11

As illustrated in **Table 3**, the directed, weighted software function invoke networks constructed from the three software systems exhibit distinct statistical characteristics. The Tar software system, which boasts five primary functions including file creation and decompression, exhibits the lowest inter-entity dependency among its function entities and has the smallest clustering coefficient among the three networks analyzed. The Nano software system, featuring three core functionalities such as text editing and saving, demonstrates a moderately higher inter-entity dependency compared to the others, and its clustering coefficient is intermediate among the three networks studied. The primary function of the Cflow software system is to establish call relationships within language programming. Given that the majority of its function entities are dedicated to this purpose, the system exhibits high inter-entity dependency, resulting in the highest clustering coefficient among the three networks analyzed.

4.2 Experiment design

Three groups of experiments are designed to analyze the feasibility and effectiveness of the SDD_KNR algorithm in this paper:

1) Experiment one: In the Tar directed weighted software function invoke network, the key node recognition algorithm based on local centrality [10] (KNMWSG) and SDD_KNR algorithm are applied to obtain the value of node defect propagation capability, and the NED value of node expansion degree and the NIKP value of improved K-shell position of node in the network are calculated. The feasibility of the SDD_KNR algorithm is verified by analyzing the influence of the nodes in the top 10 of the four metrics on the network.

2) Experiment two: Two classical key node recognition algorithms, betweenness centrality algorithm [15] (BC) and K-shell algorithm [15] (K-shell), and two existing key node recognition algorithms, KNMWSG algorithm and node correlation based key node recognition algorithm [12] (NMNC), are selected as comparison algorithms. The experimental results of different algorithms in the three directed weighted software function invoke networks of Nano, Cflow and Tar were recorded. Removing the top 20% of the nodes in the experimental results simulates the situation that the network is deliberately attacked, and analyzes the impact of removing nodes on the network to select the key nodes in the software defect detection process.

3) Experiment three: Using the index of network efficiency [15] and node propagation force [12], the key nodes of Nano, Cflow and Tar obtained by BC, K-Shell, KNMWSG, NMNC and SDD_KNR were compared and evaluated to verify the effectiveness of SDD_KNR algorithm.

We have prioritized all nodes in the network based on the computed results from our key node identification algorithm, ranking them in descending order. Subsequently, the top K nodes are excised from the network, after which we calculate the efficiency of the remaining network and the size of its largest connected component. A lower network efficiency score, coupled with a reduced number of nodes in the largest connected component, indicates a diminished capacity for interconnectivity among the remaining nodes. This reduction signifies a more severe disruption to the network's integrity, thereby highlighting the significant influence of the removed node set on the network's overall robustness.

4.3 Algorithm feasibility analysis

The SDD_KNR algorithm is used to obtain the defect propagation capability value NDPC of each node in the directed weighted software function invoke network of Nano, Cflow and Tar. The node NDPC value distribution of the directed weighted software function invoke network of the three software is shown in **Fig. 2**. In the **Fig. 2**, the abscissa represents the node ranking of the three networks, and the ordinate represents the NDPC value corresponding to each node.

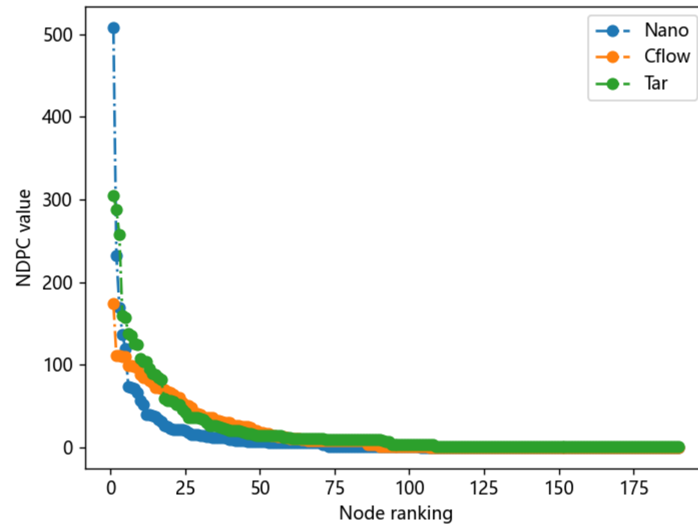


Fig. 2. Directed weighted software function invoke network node NDPC value distribution

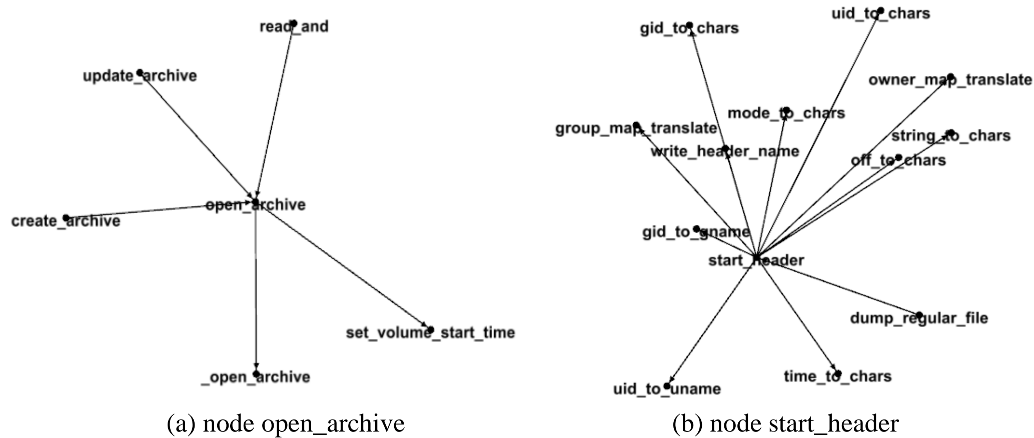
In the **Fig. 2**, the NDPC value curves of the three network nodes as a whole show the characteristics that the NDPC value of the node decreases with the increase of node ranking. Especially at the beginning of the curve, the NDPC value of the node shows a cliff-like decline, that is, the NDPC value of the node decreases significantly compared with the NDPC value of the previous node, indicating that there are indeed a small number of key nodes in the network. These nodes have great capability of defect propagation and should be paid attention to in the process of software defect detection.

In order to further verify the feasibility of the SDD_KNR algorithm, the directed weighted software function invoke network established by Tar software is selected. The KNMWSG algorithm and the SDD_KNR algorithm are used to obtain the node defect propagation capability value. At the same time, the network node expansion degree NED value and the node improved K-shell position NIKP value are calculated. The top 10 network nodes of these four metrics are shown in **Table 4**.

Table 4. The top 10 nodes for each of the four metrics

Rank	KNMWSG	Node expansion degree	Improved K-shell position of node	SDD_KNR
1	main	main	close_archive	update_archive
2	update_archive	update_archive	read_and	read_and
3	create_archive	read_and	update_archive	main
4	read_and	dump_file0	write_eot	extract_archive
5	_open_archive	extract_archive	set_next_block_after	dump_file0
6	open_archive	start_header	tar_stat_destroy	extract_file
7	dump_regular_file	_open_archive	open_archive	create_archive
8	dump_file	dump_regular_file	skip_file	_open_archive
9	dump_file0	extract_file	create_archive	dump_regular_file
10	extract_file	decode_header	find_next_block	dump_file

As shown in Table 4, the nodes with the top 10 node defect propagation capability values obtained by the KNMWSG algorithm are different from the nodes with the top 10 node NED metrics. Among them, the KNMWSG algorithm identifies that the open_archive node ranked 6 is not in the top 10 node sequence of the NED metric, and the start_header node ranked 6 of the NED metric is not in the top 10 node sequence identified by the KNMWSG algorithm. Fig. 3 is the local network diagram of node open_archive and node start_header in Tar network.

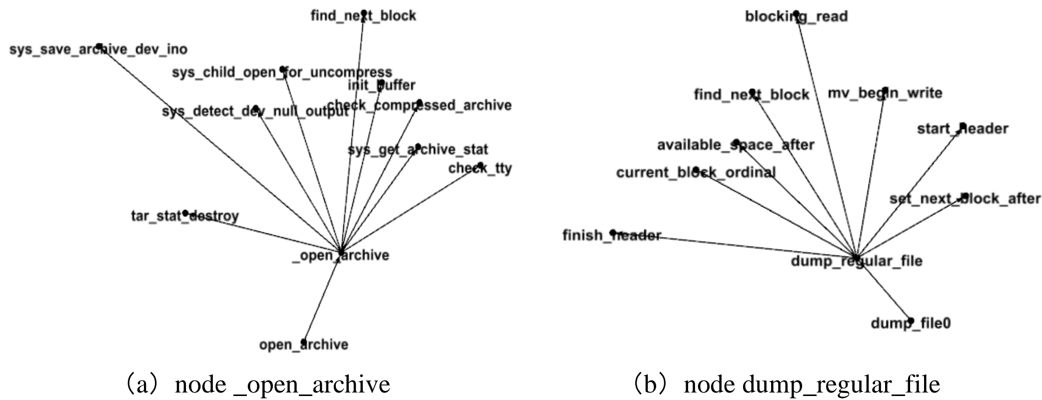


(a) node open_archive (b) node start_header

Fig. 3. Local network graph of open_archive and start_header nodes in Tar network

From Fig. 3, it can be seen that the node start_header has a more complex structure than the node open_archive in the local network. That is to say, when the node open_archive and the node start_header call a defect, it is more likely to infect the defect than the node open_archive due to the more complexity of the node start_header. The node open_archive and the node start_header are used to simulate the degree of damage to the network when the software cannot run due to defects. The network connectivity rate decreases to 0.2381 after the node open_archive is removed, and the network connectivity rate decreases to 0.2323 after the node start_header is removed. It shows that the node start_header is more important than the network, and the node expansion degree can measure the importance of the node to a certain extent.

From the last column of Table 3, it can be seen that after the node start_header is combined with its improved K-shell position, its node ranking falls out of the top 10. The previous 7th-ranked _open_archive node and 8th-ranked dump_regular_file node rank 8th and 9th, respectively, when combined with their improved k-shell positions. Fig. 4 is the local network diagram of the node _open_archive and the node dump_regular_file in the Tar network.



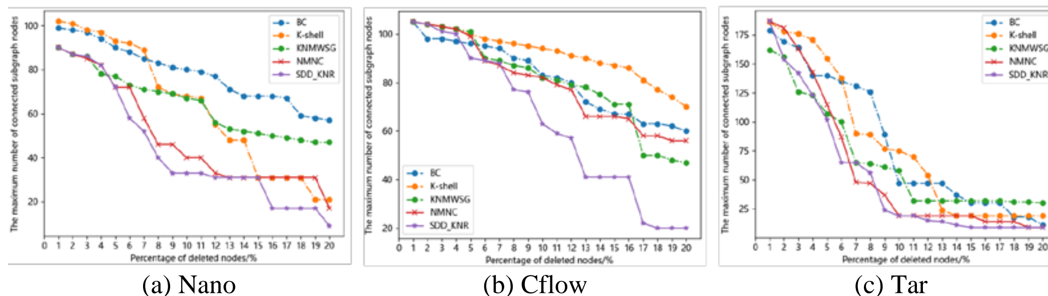
(a) node _open_archive (b) node dump_regular_file

Fig. 4. Local network graphs of _open _ archive and dump_regular_file nodes in Tar network

It can be seen from **Fig. 4** that _open_archive, dump_regular_file nodes and start_header nodes have similar complexity in the Tar local network, so their node expansion metrics are close, but _open_archive and dump_regular_file nodes have greater influence on the Tar overall network than start_header nodes. For example, the dump_regular_file node, which is the in-degree node of the start_header node, means that in the software system, the function dump_regular_file node calls the start_header node. When the start_header node is defective, the function dump_regular_file node is more likely to defect and cannot run. The node _open_archive and the node dump_regular_file are used to simulate the degree of damage to the network when the software cannot run due to defects. The network connectivity rate decreases to 0.2179 after the node _open_archive is removed, and the network connectivity rate decreases to 0.2303 after the dump_regular_file node is removed, which is lower than the network connectivity rate after the node start_header is removed to 0.2323. It shows that the SDD_KNR algorithm combined with the node expansion degree and the node improved K-shell position can be applied to the identification of key nodes in software defect detection.

4.4 Key node of software defect detection

SDD_KNR algorithm, BC algorithm, K-shell algorithm, KNMWSG algorithm and NMNC algorithm are used to obtain five kinds of node sequences sorted by importance from large to small on the directed weighted software function invoke network constructed by Nano, Cflow and Tar software systems. Then, the first 20% nodes of the five node sequences are removed from the network to simulate the deliberate attack on the network, and the maximum number of connected subgraph nodes in the remaining network is calculated. The graph of the maximum number of connected subgraph nodes in the remaining network in the three directed weighted software function invoke networks with the proportion of deleted nodes is shown in **Fig. 5**.



(a) Nano (b) Cflow (c) Tar

Fig. 5. The maximum number of connected subgraph nodes in the remaining network

As shown in **Fig. 5 (a)(b)(c)**, in the process of deleting the first 20 % nodes of the three network node sequences, when the proportion of deleted nodes is 5%, 10%, 15% and 20%, the maximum number of connected subgraph nodes in the remaining network identified by the SDD_KNR algorithm is the least in the process of node deletion. From the perspective of network robustness, the less the maximum number of connected subgraph nodes after deleting nodes, the greater the impact of deleted nodes on the robustness of the network, that is, the greater the importance of these nodes to the network. From the perspective of software defect detection, deleting a node is equivalent to a node that is defective and cannot run. The smaller the number of maximum connected subgraph nodes after deleting a node, the less functions the software system can run, indicating that the deleted nodes have a greater impact on the software system. In this way, these nodes that have a significant impact on the software system should be focused on during software defect detection. These nodes that are focused on are the key nodes for software defect detection. In the related research [16,17,18] network node sequence, the recommended threshold of key nodes is 15%. Therefore, based on the above experimental analysis, the first 15% of the node sequence obtained by the SDD_KNR algorithm is selected as the key node of the software defect detection process for experimental effectiveness analysis.

4.5 Effectiveness Analysis of Key Nodes

A. Analysis of network efficiency

In the directed weighted software function invoke network established by three software systems Nano, Cflow and Tar, the software defect detection key node recognition algorithm is used to identify the key nodes. Five algorithms are used to obtain the node ranking sequence corresponding to the node defect propagation capability value in the network. According to the analysis results in Section 4.4, the first 15% nodes of the node ranking sequence are taken as the key nodes in the software defect detection process. The results are shown in **Table 5**.

Table 5. The key nodes obtained by SDD_KNR and other four key defect node recognition algorithms on three software networks

Nano					
Rank	**SDD_KNR**	**NMNC**	**KNMWSG**	**K-shell**	**BC**
1	main	main	main	make_new_node	display_string
2	open_buffer	open_buffer	open_buffer	read_file	statusline
3	read_file	copy_of	read_file	copy_of	read_file
4	copy_of	read_file	open_file	make_new_buffer	open_buffer
5	nmalloc	nmalloc	statusbar	open_buffer	copy_of
6	display_string	make_new_buffer	edit_refresh	history_init	close_and_go
7	ingraft_buffer	display_string	prepare_for_display	nmalloc	finish
8	statusline	statusline	do_rcfiles	main	update_line
9	do_rcfiles	make_new_node	have_statedir	ingraft_buffer	titlebar
10	make_new_buffer	ingraft_buffer	process_a_keystroke	nrealloc	statusbar
11	make_new_node	history_init	stat_with_alloc	mallocstrcpy	do_exit

12	nrealloc	do_rcfiles	display_string	statusline	process_a_keystroke
13	update_line	nrealloc	parse_kbinput	do_rcfiles	ingraft_buffer
14	history_init	open_file	load_history	get_homedir	parse_kbinput
15	edit_refresh	mallocstrcpy	is_good_file	display_string	get_kbinput

Cflow

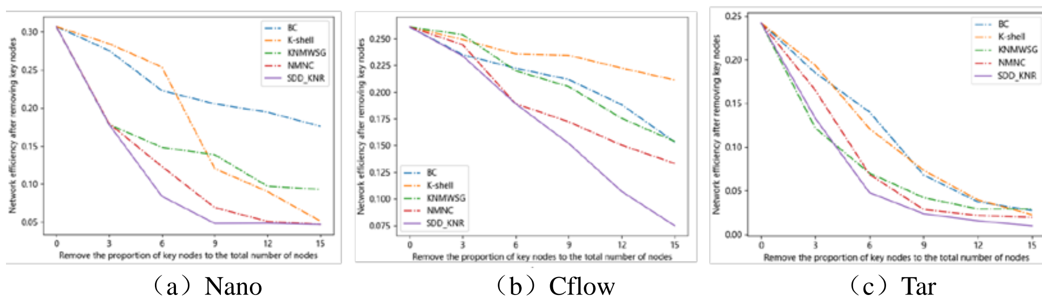
Rank	SDD_KNR	NMNC	KNMWSG	K-shell	BC
1	nexttoken	nexttoken	parse_declaration	dirdcl	nexttoken
2	inverted_tree	expression	parse_variable_declaration	nexttoken	yylex
3	direct_tree	parse_variable_declaration	maybe_parm_list	putback	get_token
4	parse_variable_declaration	inverted_tree	dirdcl	dcl	parse_declaration
5	tree_output	direct_tree	dcl	expression	parse_variable_declaration
6	expression	tree_output	func_body	add_reference	parse_dcl
7	parse_declaration	func_body	parse_typedef	maybe_parm_list	maybe_parm_list
8	func_body	fake_struct	yyparse	parse_variable_declaration	declare
9	parse_dcl	putback	parse_function_declaration	reference	yy_get_next_buffer
10	parse_typedef	parse_typedef	main	get_symbol	yyrestart
11	declare	dirdcl	expression	parse_typedef	dirdcl
12	yyparse	yyparse	nexttoken	linked_list_append	expression
13	main	parse_declaration	get_token	is_function	linked_list_iterate
14	dirdcl	dcl	fake_struct	func_body	func_body
15	fake_struct	maybe_parm_list	declare	is_printable	yyparse

Tar

Rank	SDD_KNR	NMNC	KNMWSG	K-shell	BC
1	update_archive	update_archive	main	close_archive	flush_archive
2	read_and	read_and	update_archive	read_and	dump_file0
3	main	set_next_block_after	create_archive	update_archive	find_next_block
4	extract_archive	tar_stat_destroy	read_and	write_eot	dump_regular_file
5	dump_file0	create_archive	_open_archive	set_next_block_after	dump_file
6	extract_file	find_next_block	open_archive	tar_stat_destroy	start_header
7	create_archive	extract_file	dump_regular_file	open_archive	_open_archive
8	_open_archive	main	dump_file	skip_file	gnu_flush_write

9	dump_regular_file	close_archive	dump_file0	create_archive	flush_read
10	dump_file	decode_header	extract_file	find_next_block	open_archive
11	find_next_block	extract_archive	extract_archive	extract_file	update_archive
12	close_archive	dump_file0	find_next_block	extract_archive	close_archive
13	open_archive	read_header	flush_archive	skip_member	_gnu_flush_write
14	tar_stat_destroy	write_eot	close_archive	excluded_name	gnu_flush_read
15	set_next_block_after	open_archive	read_header	available_space_after	read_and
16	start_header	dump_regular_file	check_compressed_archive	read_header	extract_archive
17	decode_header	_open_archive	write_eot	_open_archive	decode_header
18	skip_member	dump_file	skip_file	dump_file	extract_file
19	read_header	start_header	write_short_name	main	create_archive
20	write_eot	skip_file	start_header	dump_regular_file	_flush_write
21	assign_string	excluded_name	skip_member	check_compressed_archive	_gnu_flush_read
22	decode_options	skip_member	write_header_name	dump_file0	name_next_elt
23	skip_file	assign_string	_flush_write	assign_string	finish_header
24	apply_nonancestor_delayed_set_stat	name_gather	file_selection_option	tar_checksum	write_short_name
25	to_chars	transform_stat_info	file_count_links	simple_finish_header	tar_stat_destroy
26	flush_archive	names_notfound	name_gather	start_header	set_stat
27	name_next_elt	available_space_after	gnu_flush_write	finish_deferred_unlinks	write_header_name
28	name_gather	finish_deferred_unlinks	buffer_write_global_xheader	mv_begin_read	apply_nonancestor_delayed_set_stat

The initial network efficiency of Nano network is 0.307, the initial network efficiency of Cflow network is 0.261, and the initial network efficiency of Tar network is 0.242. The three networks are sequentially removed according to the order of importance of key nodes in **Table 5** to simulate the deliberate attack on the network. With the removal of key nodes, the overall trend of network efficiency changes is shown in **Fig. 6**.

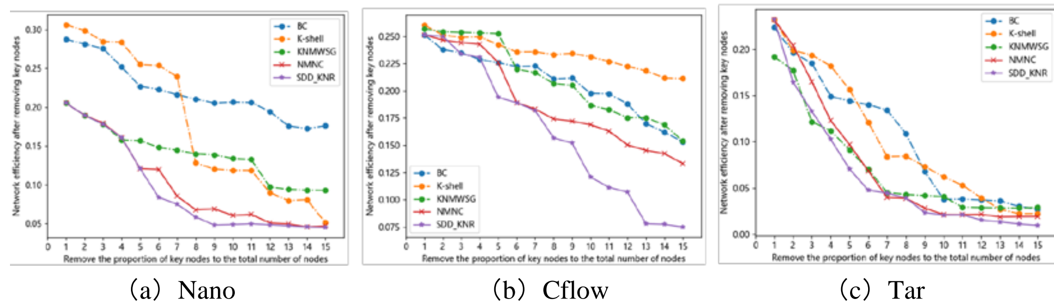


（a） Nano （b） Cflow （c） Tar

Fig. 6. The overall change trend of network efficiency after the key nodes are removed in turn

As shown in **Fig. 6 (a)(b)(c)**, in the process of removing key nodes in the network successively, the network efficiency of the five algorithms on the three networks all showed an obvious downward trend. As a whole, the network efficiency of SDD_KNR declined faster than that of the other four algorithms. After removing all the key nodes, on the Nano network, the network efficiency of algorithm BC, algorithm K-shell, algorithm KNMWSG, algorithm NMNC and algorithm SDD_KNR decreased by 0.131, 0.256, 0.214, 0.260 and 0.262 respectively. Compared with the network decline rate of the other four algorithm networks, the network efficiency decline rate of SDD_KNR Increased by 74.4 %, 11.7 %, 51.6 % and 4.2 % respectively. On the Cflow network, the network efficiency of algorithm BC, algorithm K-shell, algorithm KNMWSG, algorithm NMNC and SDD_KNR decreased by 0.108, 0.05, 0.107, 0.128 and 0.186 respectively. Compared with the network decline rate of the other four algorithms, the network efficiency decline rate of SDD_KNR increased by 50.9 %, 64.4 %, 51.4 % and 43.6 % respectively. On the Tar network, the network efficiency of algorithm BC, algorithm K-shell, algorithm KNMWSG, algorithm NMNC and SDD_KNR decreased by 0.215, 0.220, 0.213, 0.223 and 0.232 respectively. Compared with the network decline rate of the other four algorithm networks, the network efficiency decline rate of SDD_KNR increased by 62.4 %, 53.5 %, 65.4 % and 46.5 % respectively.

After the above key nodes are removed successively, the details of the decline in each part of the network efficiency are shown in **Fig. 7**.



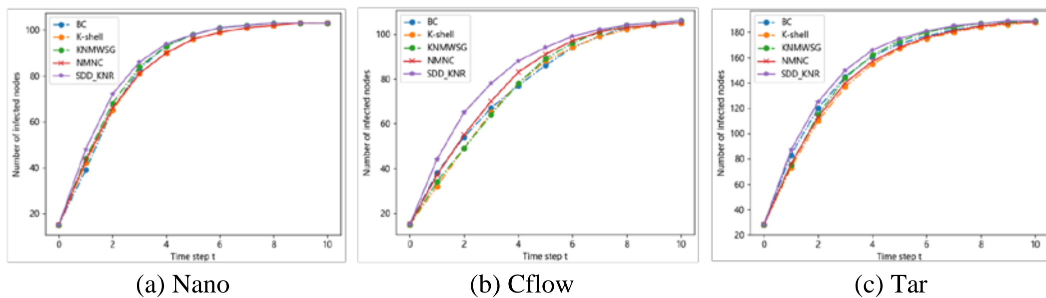
(a) Nano (b) Cflow (c) Tar

Fig. 7. The detail change trend of network efficiency after key nodes are removed in turn

As shown in **Fig. 7 (a)(b)(c)**, in the process of removing key nodes, on Nano network, SDD_KNR performs the best when the proportion of nodes is deleted at 8 places, ranks first with NMNC algorithm when the proportion of nodes is deleted at 5 places, and ranks second when the proportion of nodes is deleted at 2 places. On the Cflow network, SDD_KNR performs best when deleting the proportion of nodes at 13 of them, ranks second when deleting the proportion of nodes at 1, and ranks third when deleting the proportion of nodes at 1. On the Tar network, SDD_KNR performs best when deleting the proportion of nodes at 11 of them, and performs second when deleting the proportion of nodes at 2. On the whole, most of the network efficiency degradation processes of algorithm BC, algorithm K-shell algorithm KNMWSG and algorithm NMNC are worse than SDD_KNR, which indicates that only considering the ability of local or global defect propagation in the network cannot accurately identify the key nodes in the software defect detection process. In summary, in the process of removing key nodes on the three networks, SDD_KNR algorithm is at a low network efficiency in most of the processes, indicating that SDD_KNR algorithm can effectively identify key nodes that may have an important impact on the network efficiency.

B. Analysis of node propagation force

The epidemic model is a common model for the inspection of key nodes in the software defect detection process. In this section, the SI epidemic propagation model is applied. The network key nodes identified by different algorithms are regarded as defect sources to infect their neighbor nodes, and the number of infected nodes in a certain period of time is used as the node propagation force. Under the same conditions, the more the number of infected nodes, the stronger the node propagation force. Five key node recognition algorithms were applied to the directed weighted software function invoke network established by the software packages Nano, Cflow and Tar to obtain the network key node, which was used as the defect source to conduct the infection propagation experiment of the SI model. The total number of infected nodes of each algorithm changed with the time step t, as shown in **Fig. 8**.



(a) Nano (b) Cflow (c) Tar

Fig. 8. Key node propagation experiment

On the three networks in **Fig. 8**, the curve of SDD_KNR is above the other four algorithms, indicating that SDD_KNR is ahead of the other four algorithms in the first propagation speed on the three networks. On Nano network, SDD_KNR has an average increase of 3.9%, 4.3%, 1.8% and 3.7% compared with BC, K-shell, KNMWSG and NMNC respectively. On Cflow network, SDD_KNR has an average increase of 8.9%, 12.2%, 10.9% and 6.4% compared with BC, K-shell, KNMWSG and NMNC respectively. On Tar network, SDD_KNR has an average increase of 2.4%, 6.4%, 3.6% and 5.1% over BC, K-shell, KNMWSG and NMNC respectively.

The key nodes identified by the five algorithms are subjected to propagation experiments. From the experimental results, it can be seen that the key nodes identified by the SDD_KNR algorithm will propagate the defects to most nodes in the network, making most of the other nodes also defective. Therefore, in the software defect detection, if these key nodes can be focused on in advance, the failure of the software system can be prevented.

5. Conclusion

In this paper, a key node recognition algorithm for software defect detection is proposed to solve the problem of insufficient recognition of existing key node recognition algorithms for software defect detection. Identify the key nodes of the software defect detection process. Experiments are carried out on the real software system Tar, and the feasibility of SDD_KNR algorithm is analyzed. The SDD_KNR algorithm and the other four algorithms are applied to the real software systems Nano, Cflow and Tar. The network efficiency and node propagation force index of the key node set identified by the SDD_KNR algorithm are better than the other four algorithms, which verifies the effectiveness of the SDD_KNR algorithm.

While our proposed algorithm demonstrates feasibility and efficacy in identifying critical nodes for defect detection within process-oriented software systems, its applicability within

object-oriented systems remains to be substantiated. In subsequent research, we intend to confirm and refine the algorithm's effectiveness, specifically targeting process-oriented and object-oriented software systems characterized by higher clustering coefficients.

References

[1] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Information and Software Technology*, vol.69, pp.50-70, Jan. 2016. Article(CrossRef Link)

[2] T. Menzies, Z. Milton, B. Turhan et al., "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol.17, pp.375-407, May. 2010. Article (CrossRef Link)

[3] H. He, T. Yin, C. Pei, H. Wu, J. Ren, "Mining weighted frequent traversal pattern from software executing graph," *ICIC Express Letters*, vol.9, no.11, pp.2893-2900, 2015. Article (CrossRef Link)

[4] W-F. Pan, B. Li et al., "Measuring Structural Quality of Object-Oriented Softwares via Bug Propagation Analysis on Weighted Software Networks," *Journal of Computer Science and Technology*, vol.25, no.6, pp.1202-1213, 2010. Article(CrossRef Link)

[5] H. Maggie, G. P. Katerina, "Common Trends in Software Fault and Failure Data," *IEEE Transactions on Software Engineering*, vol.35, no.4, pp.484-496, 2009. Article (CrossRef Link)

[6] J. Y. Dai, B. Wang, J. F. Sheng et al., "Identifying Influential Nodes in Complex Networks based on Local Neighbor Contribution," *IEEE Access*, vol.7, pp.131719-131731, 2019. Article(CrossRef Link)

[7] W. Pan, H. Ming, C. K. Chang, Z. Yang et al., "ElementRank: Ranking Java Software Classes and Packages using a Multilayer Complex Network-Based Approach," *IEEE Transactions on Software Engineering*, vol.47, no.10, pp.2272-2295, Oct. 2021. Article(CrossRef Link)

[8] B. Y. Wang, J. H. Lü, "Software Networks Nodes Impact Analysis of Complex Software Systems," *Journal of Software*, vol.24, no.12, pp.2814-2829, 2013. Article(CrossRef Link)

[9] J. Dong, L. Q. Yang et al., "Identification method of key function node in software network," *Journal of Yanshan University*, vol.42, no.5, pp.434-443, 2018. Article(CrossRef Link)

[10] J. Ren, H. Wu, T. Yin, L. Bai, B. Zhang, "A Novel Approach for Mining Important Nodes in Directed-Weighted Complex Software Network," *Journal of Computational Information Systems*, vol.11, no.8, pp.3059-3071, 2015. Article(CrossRef Link)

[11] Q. Wang, S. W. Hu, J. W. Guo et al., "Structure Entropy of Directed Complex Network Based Key Node Mining Algorithm in Software Dynamic Execution," *Journal of Chinese Computer Systems*, vol.40, no.4, pp.884-889, 2019. Article(CrossRefLink)

[12] F. S. Xu, "Research on key nodes of software network based on static analysis and dynamic tracking," *M.S. thesis, Dept. Electron. Eng., Yanshan Univ., Qinhuangdao, China*, 2021. Article(CrossRef Link)

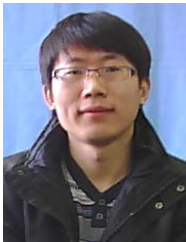
[13] C. Q Xiong, X. H Gu, X. Y Wu, "Evaluation method of node importance in complex networks based on K-shell position and neighborhood within two steps," *Application Research of Computers*, vol.40, no.3, pp.738-742, 2023. Article(CrossRef Link)

[14] K. X. Deng, H. C Chen, R. Y Huang, "Method of Node Importance Ranking Based on Improved K-shell," *Application Research of Computers*, vol.34, no.10, pp.3017-3019, Oct. 2017. Article(CrossRef Link)

[15] J. X. Zhang, K. Song, P. He, B. Li, "Identification of Key Classes in software Systems Based on Graph Neural Networks," *Computer Science*, vol.48, no.12, pp.149-158, 2021. Article(CrossRef Link)

[16] A. Zaidman, S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *Journal of Software Maintenance and Evolution: Research and Practice*, vol.20, no.6, pp.387-417, 2008. Article(CrossRef Link)

[17] I. Şora, C. B. Chirila, "Finding key classes in object-oriented software systems by techniques based on static analysis," *Information and Software Technology*, vol.116, 2019. Article(CrossRef Link)
[18] W. Pan, B. Song, K. Li, K. Zhang, "Identifying key classes in object-oriented software using generalized k-core decomposition," *Future Generation Computer Systems*, vol.81, pp.188-202, 2018. Article(CrossRef Link)



Wanchang Jiang received a B.S. degree from Liaocheng University, in 2005, and M.S. and Ph.D. degrees from Yanshan University, in 2008 and 2017, respectively. Since 2008, he has been a Teacher with the School of Computer Science, Northeast Electric Power University. Currently, he is an Associate Professor. His research interests include data mining and complex networks.

Zhipeng Liu received a B.S. degree from Changzhi University in 2020. He is currently pursuing a master's degree in the School of Computer Science, Northeast Electric Power University. His main research interest is software networks.