International Journal of Innovative Computing, Information and Control
Volume **12**, Number **5**, October **2016**

ICIC International ©2016 ISSN 1349-4198
pp. 1665–1677

APPROACH FOR MINING SOFTWARE EVOLUTIONARY COMMUNITY OUTLIERS BASED ON COMMUNITY-MATCHING

Guoyan Huang[1,2], Jiale Wang[1,2,*], Xiaojuan Chen[1,2], Hao Wang[1,2] and Jiadong Ren[1,2]

[1]College of Information Science and Engineering
Yanshan University
[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
No. 438, Hebei Ave., Qinhuangdao 066004, P. R. China
{ gyhuang; jdren }@ysu.edu.cn; *Corresponding author: jellywang92@163.com
xjchen1990@foxmail.com; 1324512861@qq.com

Received April 2016; revised August 2016

Abstract. *Study on detecting a tiny fraction of influential nodes in software evolution is very significant for predicting software updating trends, facilitating software development and version refactoring. We exploit recent advances in mining functions with obvious change to better understand the feature of software evolution. In this paper, detecting software evolutionary community outliers is our main work. Firstly, algorithm Depth-First-Search Weight (DFS-Weight) is proposed to construct the Weighted Function Dependency Network (WFDN). Secondly, we use key-nodes based approach Function Belongingness Matrix Generating (FBM-Gen) to detect the community structure of WFDN, and then find the probability distribution of function nodes in each community. Thirdly, community-matching based algorithm Software Evolutionary Community Outliers Detection (SECO-Detection) is put forward. It generates software evolutionary community outliers which evolve in a different way relative to other community members. Finally, experimental results on both real and synthetic datasets show that the proposed approach is highly effective in discovering interesting evolutionary community outliers.*

Keywords: Complex software network, Software evolution, Community matching, Evolutionary outliers

1. Introduction.

With the increase of software system complexity, the overall structure of the system is becoming more and more complicated, and the structure which is inherited from former must be changed to meet the new application environment. However, managing evolving, collaborative software system is an intricate and expensive process, which still cannot ensure software reliability [1,2]. Hence, the research on software evolution is becoming significant and important especially [3,4]. To better analyze the software evolution features, mining the key function outliers in evolution process is desirable. These outliers significantly affect the quality of new software versions.

Using complex network analysis concepts can open many actionable avenues in software evolution research and practice [5]. Given a complex network, one can unleash a variety of techniques to discover software patterns and communities, detect abnormalities and outliers in evolution process. As a consequence, research on software networks reveals a significant community structure, with similar properties as observed for other complex networks [6]. Authors have observed different phenomena that could promote the emergence of community structure in software networks [7], and have discussed possible applications within software engineering and other sciences [3]. Otherwise, several approaches of network evolution have already been proposed to explain the emergence of local structural modules (community) in different software networks [8].

By analyzing the communities of software evolution process, we can observe that these communities evolve, often contracting, expanding, splitting or merging with each other. Most of the functions within a community follow similar evolution trends and their average defines the evolution trend of the community. However, evolutionary behavior of certain functions is quite different from the average evolutionary behavior of its community. As is well known, a function outlier can highly affect many mechanisms such as cascading, spreading, and synchronizing in software structure [9,10]. Detecting and protecting these functions can effectively identify hidden software defects or abnormalities. To better reveal the trend of software evolution, it is crucial to find outliers evolve in a different way relative to other community members in the software evolution process [11].

In recent years, many studies have concentrated on outlier detection. Chandola et al. [12] and Hodge and Austin [13] provided extensive overview of outlier detection techniques. Four main types of outliers studied in literature are point outliers, contextual outliers, collective outliers and evolutionary outliers. Community Outliers Detection [14] and Association-Based [15] methods have been proposed. However, they work on only single snapshot data and hence cannot detect temporal changes. These methods cannot be applied to mine outliers in different software versions. Traditional time series literature [16] defined relative distance based on Hausdorff Distance associated with an individual object across time, ignoring the community aspect completely. Recent work on outlier detection on data streams has focused on distance-based local outliers [17] or on graph outliers [18], while we focus on outliers in the community context. In general, existing work ignores time or community information in outlier detection, and thus the outliers detected traditionally are not evolutionary community outliers proposed in this paper. The problem studied in this paper has connections with integrating community matching and outlier detection algorithm [19] in the sense that we are trying to search evolutionary community outliers in software evolution.

To address the issue, a novel approach based on community matching to mine software evolutionary community outliers is proposed which considers both time and community information. Designers or developers should pay more attention to software evolutionary community outliers with high abnormity against other community members to enhance the quality of new software versions.

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

- The key-nodes based approach is put forward to get function belongingness matrix, and the notion of software evolutionary community outliers is proposed.
- Method which considers both time and community information is applied to detect software evolutionary outliers. The algorithm SECO-Detection is presented to perform community matching and evolutionary outlier detection simultaneously.
- The experiment on multiple real and synthetic datasets shows the interesting and meaningful outliers by the algorithm SECO-Detection.

The rest of this paper is organized as follows. Section 2 contains some basic definitions. Section 3 is a detailed description of our approach. The experimental results on real and synthetic datasets are shown in Section 4. Conclusion and future work of our research are mentioned in Section 5.

2. Problem Statement and Preliminaries.

Software can be considered as a Function Dependency Network (FDN) $< N, E >$, in which functions are represented by nodes N and the collaborations or calling relationships between functions are abstracted as directed edges E. Here, we will use X_1 and X_2 to denote the two versions of the software respectively. A Function Call Sequence (*FCS*) is an ordered list of functions, denoted as $\{f_1, f_2, \cdots, f_n\}$ where $f_i \in N$ and f_1 is a root node and f_n is leaf node in *FDN*. Here, we use the frequency of *FCSs* to compute the weight. A Weighted Function Dependency Network (*WFDN*) can be described as $< N, E, W >$, W is an adjacency list storing the weight of each directed edge in WFDN. X_1 and X_2 both can be constructed as WFDN. To mine outliers in the evolutionary WFDN, following definitions are given to help formulizing the problem.

Definition 2.1. *Software Community. A community is a probabilistic collection of similar objects, such that similarity between objects within the community is higher than the similarity between objects in different communities.*

Here, software community is a collection of similar functions which have close call relationship. We will use K_1 and K_2 to denote the number of communities in X_1 and X_2.

Definition 2.2. *Function Belongingness Matrix. Each entry in the function belongingness matrix corresponds to the probability with which a function n belongs to a community c. The rows of the matrix correspond to functions while the columns correspond to communities.*

Let us denote the belongingness matrices for the N functions in X_1 and X_2 by P and Q respectively. Thus, $P \in [0,1]^{N \times K_1}$, $Q \in [0,1]^{N \times K_2}$, $\sum_{i=1}^{K_1} p_{ni} = 1$ and $\sum_{i=1}^{K_1} q_{ni} = 1$ for every function n.

Definition 2.3. *Community Correspondence Matrix S. The match between two clusterings may be formulated as a matrix called as the community correspondence matrix* $S^{K_1 \times K_2}$. *Also,* $\sum_{j=1}^{K_2} s_{ij} = 1$ $(\forall 1, \cdots, K_1)$.

Here, we denote soft correspondence to match communities across software versions. Soft correspondence means that a community of a given software version corresponds to every community in another version with different matching degree.

Definition 2.4. *Outlierness Matrix A. We denote the outlierness matrix by* $A^{N \times K_2}$. a_{nj} *represents the outlierness score for the (function, community) entry (n, j).*

Definition 2.5. *Software Evolutionary Community Outliers (SECO). A (function, community) pair (n, j) is an SECO if change in* p_{ni} *to* q_{nj} *is quite different from the average change trend for community i in* X_1 *and j in* X_2. *A function can be considered as an outlier if the change in its probability distribution with respect to community belongingness is quite different from that of its* X_1 *community members.*

Given two different software versions X_1 and X_2, our problem is to estimate S and A and thereby derive SECO with respect to the two software versions.

3. **Detecting SECO in Software Evolution.** In this section, we will present the approach of solving SECO-Detection problem in length. In Section 3.1, weighted function dependency network is constructed by algorithm DFS-Weight. Key-nodes based method FBM-Gen is used to generate function belongingness matrix in Section 3.2. Section 3.3 is the formulation derivation of computing the community matching matrix and outlierness score matrix respectively. In Section 3.4, we propose an algorithm called SECO-Detection mining evolutionary community outlier from software evolution. The framework of solving SECO-Detection problem is shown in Figure 1.

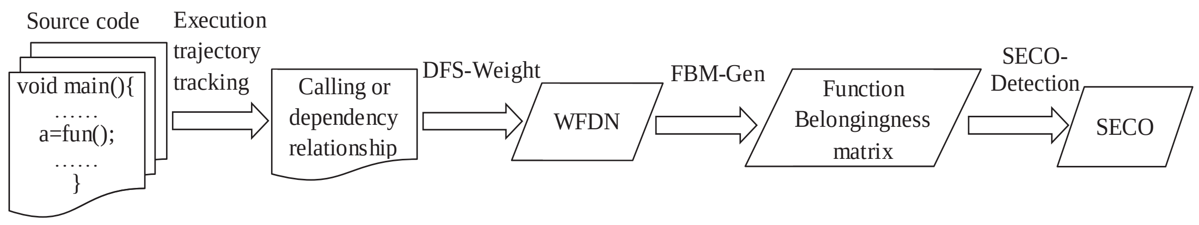
Source code
void main(){
……
a=fun();
……
}
Execution trajectory tracking
Calling or dependency relationship
DFS-Weight
WFDN
FBM-Gen
Function Belongingness matrix
SECO-Detection
SECO

FIGURE 1. Framework of detecting SECO

3.1. **Constructing WFDN.** Normally, functions and relationships of function calling or dependency are extracted from software source code. Then we map them to function dependency network. Finally, algorithm DFS-Weight is used to get the weight of each edge, and the WFDN is constructed.

Algorithm 1 DFS-Weight for adding weight to FDN

```
Input: function dependency network ($FDN$)
Output: weighted function dependency network ($WFDN$)
        Procedure: GetWFDN
 1: run GetFCS-File to get FCS-File; //FCS-File is a data set storing all FCSs
 2: for each directed edge $< v, u >$ in FDN do
 3:     initialize $W = 0$;
 4:     for each FCS in DFS-File do
 5:         if FCS contains edge $< v, u >$ then
 6:             W++;
 7:         end if
 8:     end for
 9:     $W_{v \to u} = W/|FCS\text{-}File|$;
10: end for
11: return WFDN;
        Procedure: GetFCS-File
12: initialize FCS = rootnode;
13: if rootnode.child is not null then
14:     for each child in set of rootnode.child do
15:         FCS = FCS+child;
16:         GetFCS-File (child); //call procedure GetFCS-File iteratively
17:     end for
18:     FCS-File.add(FCS);
19: end if
20: return FCS-File;
```

Algorithm DFS-Weight consists of two procedures. In procedure GetWFDN, GetFCS-File is called to find all FCSs firstly (line 1). Then the weight of each edge is obtained (line 2 to line 10) and a WFDN is constructed (line 11). Procedure GetFCS-File generates all FCSs recursively from FDN based on DFS strategy. A data set FCS-File is used to store all FCSs. Weight is computed in line 9, $|FCS\text{-}File|$ is the number of all FCSs and W is the number of FCSs that contains directed edge $< v, u >$.

3.2. **Generating function belongingness matrix.** The approach we proposed is based on community matching, so we need construct function belongingness matrix firstly.

We use the key-nodes based approach to divide the WFDN into communities and get the function belongingness matrices. First of all, we rank all functions according to their weights, and find top-k nodes as the community center. For community center functions, the probability respect to their own community is 1. To the contrary, the probability respect to other communities is 0. For other functions, we compute their probability by their weight and the distance to center functions. Intuitively, the probability is high when the relation position of a function node is close to center function or with high weight, the probability is low when relation position of the function is far away from center functions or with low weight. P and Q generate respectively via this approach.

Algorithm 2 FBM-Gen for generating function dependency matrix

```
Input: weighted function dependency network (WFDN)
Output: function dependency matrix $P(Q)$
        Procedure: GetFBM
 1: run GetC to get $C$; //$C$ is a set of community center nodes
 2: for each $node \in C$ do
 3:     initialize $p_{current} = 1$, and $p_{others} = 0$;
 4:     for each node $n \in WFDN/C$ do // WFDN/C is node in WFDN, but not in $C$
 5:         if $n$ is one step node then
 6:             set $p$ as $\alpha$;
 7:         else
 8:             set $p$ as $\beta$;
 9:         end if
10:     end for
11: end for
12: return $P(Q)$;
        Procedure: GetC
13: for each node $n \in WFDN$ do
14:     if n.weight is highest in the remaining nodes then
15:         insert $n$ to list L;
16:     end if
17: end for
18: return top-k node as community center $C$ in L;
```

Algorithm FBM-Gen consists of two procedures. In procedure GetFBM, GetC is called to find community center nodes firstly (line 1). Then probability with respect to communities is obtained (line 2 to line 9) and a function belongingness matrix is constructed (line 10). Procedure GetC generates all community center nodes from WFDN according to their weight.

3.3. Deriving formulation for computing S and A.

In this section, two formulations will be derived. The formulations are iterative update rules for computing Community Correspondence Matrix S and Outlierness Matrix A. Detailed derivation process is followed.

To mine software evolutionary community outlier, first of all, we need to perform community matching between the two software versions to get the most similar communities, which means we need to estimate a correspondence matrix $S^{K_1 \times K_2}$ such that the distance (sum of entrywise squared differences) between the matrices Q and $P \times S$ is minimized. However, such an approach will perform biased community matching if we take account of the contribution from outlier entries too, when estimating the correspondence matrix

S. For higher quality matching, one needs to ignore evolutionary outlier entries. Hence, we need to incorporate the outlierness score matrix A into community matching. In the remainder of this section, we will develop an integrated approach to compute S and A. $\left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2$ denotes the squared error incurred in community matching and $\ln\left(\frac{1}{a_{nj}}\right)$ determines the outlier weight associated with the (n, j)th entry. Note that even if there were no outliers, there would still be some matching error, because each object evolves somewhat differently from the community averages. However, outliers that evolve very differently from community averages are penalized using a higher a_{nj} value. Using $\ln\left(\frac{1}{a_{nj}}\right)$ in the objective function allows us to smooth out outlierness values. The ln function makes sure that the weights for individual entry matching across software versions lie within a small range. The more anomalous a particular $(function, community)$ entry (n, j) is, the higher will be the value of a_{nj} and so $\ln\left(\frac{1}{a_{nj}}\right)$ will be lower. This would mean that lower weight will be associated with the (n, j)th outlier entry when performing community matching. While there could be other ways of formulating the objective function, we use this particular formulation for ease of computation.

$$\min_{S,A} \sum_{n=1}^{N} \sum_{j=1}^{K_2} \ln\left(\frac{1}{a_{nj}}\right) \left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2 \tag{1}$$

This formulation should be subject to the following conditions. Every entry s_{ij} in S is nonnegative, and $\sum_{j=1}^{K_2} s_{ij} = 1$ $(\forall i = 1, \cdots, K_1)$. Every entry a_{nj} in A is between 0 and 1. In addition, we would like to bound the total sum of all a_{nj} values to be within a maximum level of outlierness we expect. Thus, we formulate the constraint as $\sum_{n=1}^{N} \sum_{j=1}^{K_2} a_{nj} \le \mu$. As is shown in the formula, μ is the total amount of outlierness. We will show later how to estimate μ in the description of Algorithm 3.

Analyzing the constraint of outlierness sum. If the total amount of outlierness is unbounded, one can simply mark all entries as outliers and then there will be no useful community matching. This corresponds to the trivial solution of setting all A elements to very high values. For the optimization (Equation (1)), we need to put in a constraint based on the total amount of outlierness. Note that normal entries have small a_{nj} values, while outlier entries have large a_{nj} values. Thus, we bound the total sum of all a_{nj} values to be within a maximum level of outlierness we expect in the software evolution. We replace the formulation by an equality constraint to simplify computation. In fact, the semantic meanings of outlierness scores will not change by this action because we only care about the relative ranking of the scores. Essentially the equality claims that there is a certain level of outlierness in the entire version. The objective function can be minimized (local minimum) by alternately optimizing one of S and A while fixing the other. Next, we will derive iterative update rules for the correspondence entry (s_{ij}) and the outlierness scores (a_{nj}). Using the method of Lagrangian multipliers, we can rewrite the problem as follows. Here, β_i and γ are Lagrangian variables.

$$\min_{S,A} f = \sum_{n=1}^{N} \sum_{j=1}^{K_2} \ln\left(\frac{1}{a_{nj}}\right) \left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2 + \sum_{i=1}^{K_1} \beta_i \left[\sum_{j=1}^{K_2} s_{ij} = 1\right] + \gamma \left[\sum_{n=1}^{N} \sum_{j=1}^{K_2} a_{nj} - \mu\right] \tag{2}$$

Taking the partial derivative of Equation (2) with respect to a particular a_{nj} and setting it to 0, we obtain the following.

$$a_{nj} = \frac{\left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2}{\gamma} \text{ and } \sum_{n=1}^{N}\sum_{j=1}^{K_2} \frac{\left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2}{\mu} = \gamma \tag{3}$$

This gives us the update rule for a_{nj} as follows.

$$a_{nj} = \frac{\left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2 \mu}{\sum_{n'=1}^{N}\sum_{j'=1}^{K_2}\left(q_{n'j'} - \overrightarrow{p_{n'\cdot}} \cdot \overrightarrow{s_{\cdot j'}}\right)^2} \tag{4}$$

The numerator $\left(q_{nj} - \overrightarrow{p_{n\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)^2$ represents the squared error for the (n, j)th entry, which represents the error incurred by matching the community detection results between two versions on the nth function with respect to the jth community in Q. The denominator in Equation (4) represents the overall error across all the entries in matrix Q, given a particular S, which serves as a normalization factor. Intuitively, we assign a higher outlierness score to functions and communities that incur higher community matching error, while functions that are matched well with respect to certain communities across two versions are considered normal and thus receive lower outlierness score.

Now, we will obtain the update rule for s_{ij}. Taking partial derivative of f with respect to s_{ij}, we obtain the following.

$$\sum_{n'=1}^{N}\left[2\ln\left(\frac{1}{a_{n'j}}\right)\left(q_{n'j} - \overrightarrow{p_{n'\cdot}} \cdot \overrightarrow{s_{\cdot j}}\right)(-p_{n'i})\right] + \beta_i = 0 \tag{5}$$

After some algebraic simplifications, we obtain the following update rule for s_{ij}.

$$s_{ij} = \frac{\sum_{n'=1}^{N} 2\ln\left(\frac{1}{a_{n'j}}\right)p_{n'i}\left[q_{n'j} - \sum_{k=1,k\neq i}^{K_1} p_{n'k}s_{kj}\right] - \beta_i}{\sum_{n'=1}^{N} 2\ln\left(\frac{1}{a_{n'j}}\right){p_{n'i}}^2} \tag{6}$$

The intuition behind Equation (6) is as follows. Our goal is to compute s_{ij} when other elements of the matrix S are fixed. s_{ij} involves matching of all functions with respect to the ith column of P and the jth column of Q. Contributions from each function n' are weighted by $\ln\left(\frac{1}{a_{n'j}}\right)$ such that highly outlying functions contribute little to matching. Fixing other elements of S, $\left[q_{n'j} - \sum_{k=1,k\neq i}^{K_1} p_{n'k}s_{kj}\right]$ represents the part of $q_{n'j}$ that functions to be explained by $p_{n'i}s_{ij}$. β_i and the denominator of Equation (6) are used to make sure that $\sum_j s_{ij} = 1$. $\beta_i's$ can now be computed easily using Equation (6) and the constraints $\sum_{j=1}^{K_2} s_{ij} = 1$ $(\forall i = 1, \cdots, K_1)$. This value of β_i can then be substituted back in Equation (6) to obtain the update rule for s_{ij}.

3.4. **Algorithm SECO-detection.** S captures the average evolution trend for the communities in the two software versions. It also captures the permutation effect when matching two communities. s_{ij} represents the degree to which the community i in version X_1 contributes to the community j in version X_2. Thus, s_{ij} averages the evolution/match across all the functions belonging to i in X_1 and j in X_2. If a community i splits into two parts j_1 and j_2, s_{ij_1} and s_{ij_2} will have non-zero values. Similar to this split case, s_{ij} can be used to represent community merges, community expansion, community shrinking and a mix of such scenarios. Now, if there are evolutionary outliers, they will possess values quite different from the average. However, an outlier will have most of its mass moving from community i in X_1 to communities other than j in X_2. Presence of such outliers can affect the computation of s_{ij} itself because outliers can lead to significantly different average values. In our formulation, a weight is given to a function n when computing the community evolution values for community j. For normal (function, community), a_{nj} should be low, while for outlying (function, community), a_{nj} should be high. A is designed to capture such evolutionary outlier.

Algorithm 3 SECO-Detection Algorithm

```
Input: function belongingness matrix $P$, $Q$
Output: Estimates of $S$ and $A$
1: Initialize $\mu$ to 1;
2: Initialize all $s_{ij} \Leftarrow \frac{1}{K_2}$ and $a_{nj} \Leftarrow \frac{1}{NK_2}$;
3: while NOT converged do
4:     Update $A$ using Equation (4) (Outlier Detection Step);
5:     Update $S$ using Equation (6) (Community Matching Step);
6: end while
7: $\mu \Leftarrow \frac{\sum_{n'=1}^{N}\sum_{j'}^{K_2}\left(q_{n'j'}-\overrightarrow{p_{n'\cdot}}\cdot\overrightarrow{s_{\cdot j'}}\right)^2}{\max_{n,j}\left(q_{nj}^2\right)}$;
8: Repeat Steps 2 to 6;
```

Algorithm SECO-Detection does initialization in line 1 and line 2. Perform line 3 to line 6 iteratively until the change in the value of the objective function is less than a threshold. In the first procedure, μ is initialized to 1. Then, μ is estimated as the ratio of overall error to the maximum entry value, as shown in Line 7 of Algorithm 3. The algorithm uses this estimated μ for the second procedure. Since μ increases compared to the first procedure, a_{nj} values are relatively large. This reduces the overfitting of S to the outlier entries. Hence, matching for non-outlier entries improves, and so outlier detection improves too. We initialize all a_{nj} values to $\frac{1}{NK_2}$, i.e., we begin by considering all functions to be equally anomalous. We initialize all s_{ij} to $\frac{1}{K_2}$, i.e., we assume that each community in X_2 evolves equally from each of the communities in X_1.

4. Experiment and Analyses.

In this section, we evaluate the results on real and synthetic datasets. Firstly, we evaluate the results on real datasets using case studies. We perform comprehensive analysis of functions to justify the top few outliers returned by the proposed algorithm. Real dataset contains two open source software datasets cflow and tar. Moreover, evolutionary outlier detection algorithms are quite difficult due to lack of ground truth. We perform experiments on multiple synthetic datasets, each of which simulates real scenarios. We will evaluate outlier detection accuracy of the proposed algorithm based on outliers injected in synthetic datasets. Furthermore, comparison of the proposed algorithm with other algorithms on computational complexity will be performed in Section 4.3. Experiment is conducted on 64 bit Windows 7 ultimate, Xeon CPU E5-2603 @1.80GHz, 8G Memory and Ubuntu14.04. We get the experiment data with the help of pvtrace, CodeVize, Gephi and Graphviz.

We compare the proposed algorithm with three baseline methods: OneStage (1S), TwoStage (2S) and NearestNeighbor (NN). As discussed, our method SECO-Detection is named as $EC\mu$ because it integrates outlier detection and community matching, and μ is estimated using a two-pass procedure.

4.1. Results on real datasets.

We perform experiments using two real datasets: cflow and tar. We run algorithm SECO-Detection on two versions of cflow/tar to mine the software evolutionary community outliers. The results of the cflow and tar are showed below.

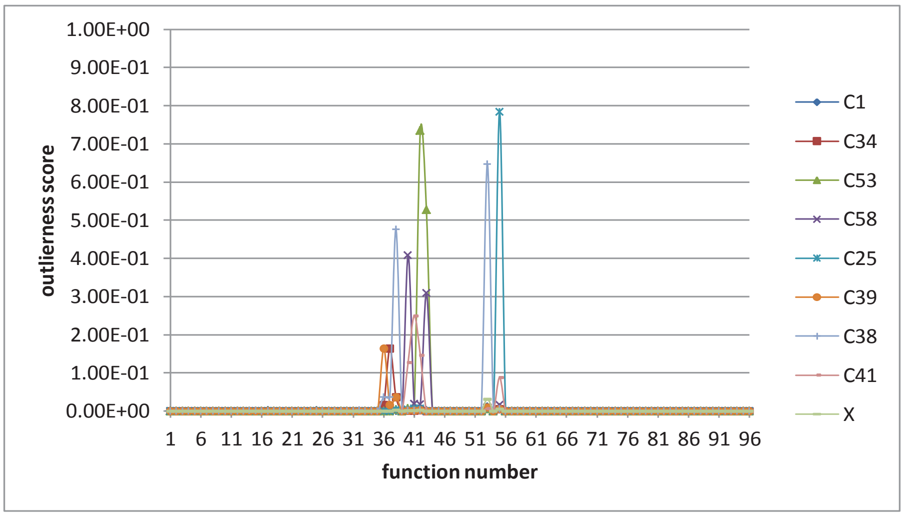

FIGURE 2. Outlierness score of all functions on cflow1.2 and cflow1.3

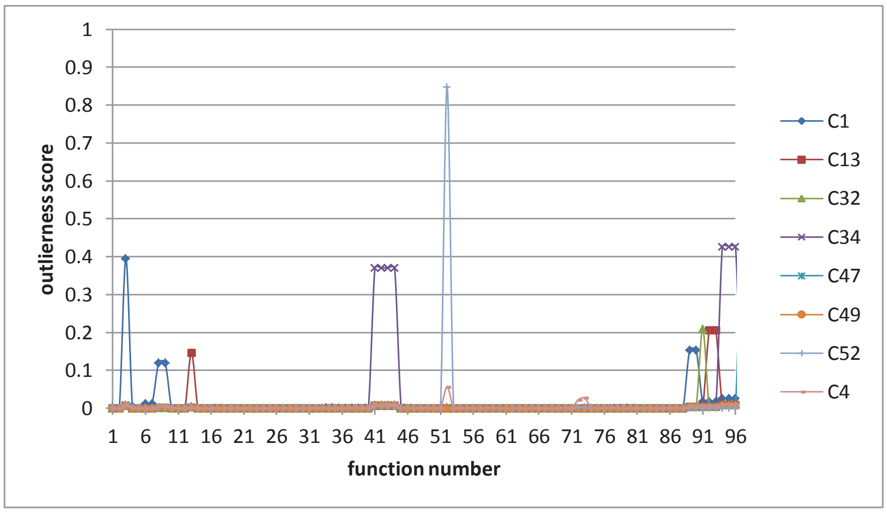

FIGURE 3. Outlierness score of all functions on tar1.25 and tar1.27

Different color represents different community's change between two software versions. High outlierness score represents obvious change of that function.

Here, we discuss case studies obtained from these datasets.

Figure 2 shows that some function nodes present obvious anomaly against others in the cflow1.2 and cflow1.3 evolution process, and top 5 function outliers are shown in Table

TABLE 1. Top 5 function outliers in cflow

Function number	Function name	Outlierness score
55	*maybe_parm_list*	0.784
42	*dirdcl*	0.734
43	*skip_to*	0.527
53	*parse_function_declaration*	0.646
38	*parse_variable_declaration*	0.476

Table 2. Top 5 function outliers in tar

Function number	Function name	Outlierness score
52	*find_next_block*	0.847
94	*xattrs_acls_get*	0.427
97	*blocking_read*	0.424
3	*priv_set_remove_linkdir*	0.395
44	*timespec_cmp*	0.371

1. Here, we select two software evolutionary community outliers returned by SECO-Detection and discuss below. In cflow1.2, *dirdcl* belongs to community C41, but in cflow1.3, the weight of function *dirdcl* becomes higher than before, because of this, *dirdcl* evolves into a key node. From this evoluation, we can capture the knowledge that function *dirdcl* becomes more important. To the contrary, *parse_function_declaration* in cflow1.2 is a key node, but in cflow1.3, the weight of function *parse_function_declaration* becomes lower than before, because of this, *parse_function_declaration* evolves into an outlier node. We can capture functions which become not that important as first software version.

Figure 3 shows that some function nodes present obvious anomaly against others in the tar1.25 and tar1.27 evolution process, and top 5 function outliers are shown in Table 2. In tar1.25, *find_next_block* is a key node with high weight, and there are certain functions belonging to its community, but these functions evolve to other community in tar1.27. This phenomenon may reflect function *find_next_block* evolves to a discarded function. Through the analysis, *xattrs_acls_get* and *blocking_read* are new functions that emerged in tar1.27, and this suggests that these functions maybe relate to new application environment.

4.2. **Results on synthetic datasets.** In order to compare the performance of our algorithm with other similar algorithms, we generate a variety of synthetic datasets to capture different spatial cases of evolution. For each dataset, there are two snapshots. In each snapshot, we generate multiple clusters, each of which represents a community. Each cluster is modeled using a 2D Gaussian distribution, and evolution is modeled by changing the means and the variances of the Gaussians. For each of the N points, we obtain p_{nj} and q_{nj} as the probability with which the point can be generated from its cluster's Gaussian distribution. Using P and Q, we obtain S as the community matching matrix in absence of any outlierness. S captures evolution without the effect of outliers. Next, we inject outliers as follows. First we set an outlierness factor Ψ and choose a random set of objects, R with $N \times \Psi$ objects.

The experiment uses a variety of different settings. For each setting, we perform 100 experiments and report the mean values. Threshold ϵ for convergence fixes to 10^{-6}. We vary the number of objects as 1000, 5000 and 10000. The percentage of outliers injected into the dataset varies as 1%, 5% and 10%. Using these settings, we compare the actual outlier objects with the top outliers returned by various algorithms. The results of the three baselines and the proposed method are shown in Table 3 and Table 4.

Note that the proposed algorithm (SECO-Detection) outperforms the others in finding the top few outliers most precisely. The area under this curve (AUC) is a good measure of the effectiveness of the algorithm in identifying the outliers. We report the AUC values in Table 3. As the table shows, the proposed algorithm outperforms all the other algorithms for all the settings by a wide margin (sometimes as high as 30% better than the TwoStage method). Comparison with 1S will help us understand improvement in accuracy

TABLE 3. SynContractExpand & SynMerge dataset AUC (NN, 2S, 1S, $EC\mu$)

N	$\Psi(\%)$	SynContractExpand				SynMerge			
		NN	2S	1S	$EC\mu$	NN	2S	1S	$EC\mu$
1000	1	.656	.935	.961	.959	.746	.779	.829	.927
	5	.658	.864	.914	.957	.706	.664	.723	.855
	10	.647	.783	.837	.957	.667	.621	.660	.808
5000	1	.619	.946	.969	.959	.740	.779	.815	.925
	5	.641	.894	.926	.963	.736	.694	.742	.831
	10	.635	.776	.834	.964	.702	.645	.679	.795
10000	1	.606	.948	.972	.963	.745	.783	.816	.929
	5	.638	.911	.933	.965	.740	.713	.749	.822
	10	.630	.771	.820	.965	.719	.666	.700	.799

TABLE 4. SynSplit & SynMix dataset AUC (NN, 2S, 1S, $EC\mu$)

N	$\Psi(\%)$	SynSplit				SynMix			
		NN	2S	1S	$EC\mu$	NN	2S	1S	$EC\mu$
1000	1	.731	.928	.937	.913	.672	.872	.883	.919
	5	.683	.796	.890	.923	.641	.768	.816	.914
	10	.670	.736	.819	.917	.632	.729	.770	.915
5000	1	.697	.922	.946	.920	.668	.882	.896	.920
	5	.688	.811	.917	.926	.653	.830	.858	.922
	10	.676	.766	.840	.932	.637	.788	.831	.916
10000	1	.697	.943	.957	.922	.676	.887	.904	.921
	5	.688	.815	.929	.927	.651	.843	.864	.916
	10	.674	.768	.831	.932	.634	.801	.841	.921

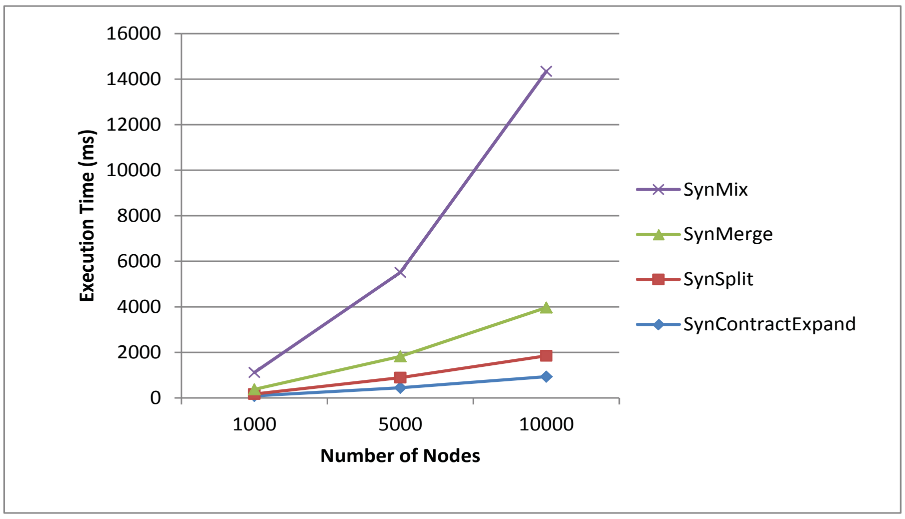

FIGURE 4. Running times (ms) for SECO-detection

by improved estimation of μ. Comparison with 2S will help us understand performing outlier detection and community matching (2S) in an integrated way (SECO-Detection) is better. NN performs commonly. This is because it does not consider evolution and assumes that the set of nearest neighbors does not change across the snapshots. To the contrast, the proposed algorithm detects outliers in the context of community evolution.

Figure 4 shows the execution time of different evolution types as the objects number increases. As Figure 4 shown, execution time rises steadily which indicates the algorithm we proposed have a good scalability with increasing dataset size.

4.3. **Comparison on computational complexity.** Recall that N is the number of objects (functions), K_1 and K_2 are the number of communities in the two software versions respectively. Sum of all error values in Equation (4) can be computed once for all a_{nj} in $O(NK_1K_2)$ time. Then, computation of each a_{nj} takes $O(NK_1)$ time. S consists of K_1K_2 entries. When computing S, at every iteration, for each s_{ij}, one needs to use Equation (6) which is $O(NK_1)$ itself. Thus, computational complexity of the SECO-Detection Algorithm is $O\left(NK_1{}^2K_2I\right)$ where I is the number of iterations. As can be seen, the running time is linear with respect to the number of objects. Usually the number of communities is small, and thus the proposed method scales well to large data sets.

1S is the one procedure version of SECO-Detection (line 1 to 6 of Algorithm 3). Computational complexity of the 1S Algorithm is $O\left(NK_1{}^2K_2\right)$ without I iterations. Although the complexity is reduced, the accuracy is much worse than SECO-Detection. Computational complexity of 2S is the same as SECO-Detection, 2S performs outlier detection after community matching. The accuracy decreases especially as the number of objects and outliers increases. For the algorithm NN, we find k-Nearest Neighbors set $NN_{X_1}(o)$ for every object o. An outlier entry (o, j) has a high score if belongingness of object o to community j in the second snapshot is quite different from the average belongingness of its X_1 nearest neighbors to the same community j in X_2. Computational complexity of NN is $O(NKK_1K_2I)$ where K is the object number of k-Nearest Neighbors set. It is not quite different from SECO-Detection. However, the accuracy of NN is not satisfactory when considering community evolution.

5. **Conclusions and Future Work.** In this paper, we focus on mining software evolutionary community outliers in function dependency network. We propose a novel approach based on community matching to conduct outlier detection. First we get functions and the relationships of function calling or dependency from source code with the help of some tools and map them to a weighted function dependency network. Next, key-nodes based algorithm FBM-Gen is put forward for constructing function belongingness matrix. Finally, algorithm SECO-Detection is designed to generate correspondence matrix S and outlierness matrix A. Case study on versions of two open source software datasets reveals some interesting and meaningful evolutionary outliers. Experiments on a series of synthetic data show capability of the proposed algorithm is remarkable in detecting various types of community evolution outliers. Although the proposed algorithm focuses on two versions, it can detect trends and outliers, as versions can consist of short or long intervals. Moreover, our future work includes extending the approach to software network at class granularity and multiple versions of open source software.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61572420, No. 61472341 and the Natural Science Foundation of Hebei Province China under Grant No. F2013203324, No. F2014203152 and No. F2015203326.

REFERENCES

[1] H. P. Breivold, I. Crnkovic and M. Larsson, A systematic review of software architecture evolution research, *Information and Software Technology*, vol.54, no.1, pp.16-40, 2012.

[2] I. Neamtiu, G. Xie and J. Chen, Towards a better understanding of software evolution: An empirical study on open source software, *JSME*, 2011.

[3] C. R. Myers, Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs, *Physical Review E*, vol.68, no.2, 2003.

[4] Z. Liu and Q. Zhang, Analysis on evolution features of software network, *IIICEC*, pp.749-752, 2015.

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

[6] L. Subelj and M. Bajec, Community structure of complex software systems: Analysis and applications, *Physica A Statistical Mechanics & Its Applications*, vol.390, no.16, pp.2968-2975, 2011.

[7] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton and E. Tempero, Understanding the shape of java software, *Proc. of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.397-412, 2006.

[8] D. Li, Y. Han and J. Hu, Complex network thinking in software engineering, *Proc. of the International Conference on Computer Science and Software Engineering*, pp.264-268, 2008.

[9] L. Zemanová, C. Zhou and J. Kurths, Structural and functional clusters of complex brain networks, *Physica D Nonlinear Phenomena*, vol.224, no.1, pp.202-212, 2006.

[10] G. Zamora-López, C. Zhou and J. Kurths, Cortical hubs form a module for multisensory integration on top of the hierarchy of cortical networks, *Frontiers in Neuroinformatics*, vol.4, no.1, 2010.

[11] D. Chen, L. Lü, M. S. Shang et al., Identifying influential nodes in complex networks, *Physica A Statistical Mechanics & Its Applications*, vol.391, no.4, pp.1777-1787, 2012.

[12] V. Chandola, A. Banerjee and V. Kumar, Anomaly detection: A survey, *ACM Surveys*, vol.41, no.3, 2009.

[13] V. J. Hodge and J. Austin, A survey of outlier detection methodologies, *AI Review*, vol.22, no.2, pp.85-126, 2004.

[14] J. Gao, F. Liang, W. Fan et al., On community outliers and their efficient detection in information networks, *KDD'10*, pp.813-822, 2010.

[15] M. Gupta, J. Gao, X. Yan et al., On detecting association-based clique outliers in heterogeneous information networks, *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp.108-115, 2013.

[16] L. Liu, J. Le, S. Qiao et al., Trajectory outliers detection based on local outlying degree, *Chinese Journal of Computers*, vol.34, no.10, pp.1966-1975, 2011.

[17] D. Pokrajac, A. Lazarevic and L. J. Latecki, Incremental local outlier detection for data streams, *IEEE Symposium on Computational Intelligence and Data Mining*, pp.504-515, 2007.

[18] C. C. Aggarwal, Y. Zhao and P. S. Yu, Outlier detection in graph streams, *Proc. of the 27th Intl. Conf. on Data Engineering*, pp.399-409, 2011.

[19] M. Gupta, J. Gao, Y. Sun et al., Integrating community matching and outlier detection for mining evolutionary community outliers, *KDD'12*, pp.859-867, 2012.