SOFTWARE ARCHITECTURE RECOVERY THROUGH SIMILARITY-BASED GRAPH CLUSTERING

JIANLIN ZHU*
*Corresponding author: Room 459, Building 9, Institute of Particle Physics, Central China Normal University, Wuhan, 430079, China.

Key Laboratory of Quark and Lepton Physics (Ministry of Education)
Central China Normal University, Wuhan, 430079, P. R. China
zhujianlin1120@gmail.com

JIN HUANG
Research Institute 709
China Shipbuilding Industry Corporation, Wuhan, 430070, P. R. China
derick0320@gmail.com

DAICUI ZHOU† and ZHONGBAO YIN‡
†dczhou.iopp@gmail.com
‡zbyin@mail.ccnu.edu.cn

GUOPING ZHANG
College of Physical Science and Technology
Central China Normal University, Wuhan, 430079, P. R. China
gpzhang@phy.ccnu.edu.cn

QIANG HE
Faculty of Information and Communication Technologies
Swinburne University of Technology, Melbourne, 3122, Australia
qhe@swin.edu.au

Received 6 August 2012
Revised 2 November 2012
Accepted 14 February 2013

Software architecture recovery is to gain the architectural level understanding of a software system while its architecture description does not exist. In recent years, researchers have adopted various software clustering techniques to detect hierarchical structure of software systems. Most graph clustering techniques focus on the connectivity between program elements, but unreasonably ignore the similarity which is also a key measure for finding elements of one

*Corresponding author: Room 459, Building 9, Institute of Particle Physics, Central China Normal University, Wuhan, 430079, China.
In this paper we propose a novel hierarchy graph clustering algorithm DGH C, which considers both similarity and connectivity between program elements. During the transformation of program dependence graph edges representing similarity between elements are added. Then similar elements are grouped by density-based approaches. The alternative strategy is adopted to find groups of closely connected and similar elements. Meanwhile we adjust the contribution of connectivity and similarity by a flexible clustering algorithm based on short random walk model, which can obtain more structure information of software to find its multiple layers. Furthermore a new method called Multi-layer Propagation Gap is proposed to suggest stable layers of hierarchy clustering result as multiple layers of software system. Extensive experimental results illustrate the effectiveness and efficiency of DGH C in detecting hierarchy structure of software through comparison with various software clustering methods.

Keywords: Software architecture recovery; hierarchy graph clustering; similarity-based clustering; multiple stable layers.

1. Introduction

Software clustering is an important technique for software architecture recovery [1]. The general goal of software clustering is to cluster similar program elements. This technique has been widely used to identify software modules [2], as well as multiple levels of software systems for architecture recovery [3, 4].

Graph, as a reliable and expressive representation of software, is widely adopted to model the relationships between the program elements of software system. Documents are used to understand legacy software systems. However, these documents may be not well written by programmers. Along with the constant modification of software systems, these documents are seldom up-to-date. The names of program elements are also chosen to represent programs. However, different programmers may assign different names to a same element. If these names share no common parts, they will not be considered as similar names. Compared with these representations, graph is more stable and reliable. Furthermore, the features of programs should be expressed to produce measures for software clustering. Neither documents nor names can properly indicate the dependencies between program elements. In a graph, the dependencies between program elements are expressed intuitively with directed links, and similar elements usually share common in-links or out-links.

To recover the architecture of a software system, both connectivity and similarity measures should be considered. For example, if a function is only called by another function, then they are of high connectivity and should be placed in a same cluster. In addition, if two functions use common global variables, they are of high similarity and they also should be placed in the same cluster. However, most graph clustering approaches consider only connectivity. As a result, similar program elements are seldom grouped into one cluster and software modules cannot be recovered successfully. An ideal software clustering should generate modules with these heterogeneous measures, which should be coordinated reasonably.

The primary challenge we meet in this research is to cluster program elements of a software system with two heterogeneous measures, i.e. connectivity and similarity.
The goal is to produce complete modules in which similar and closely connected program elements are grouped into one cluster. Density-based approaches are effective in finding dense modules, in which program elements are closely connected to each other. However, similar program elements cannot be discovered effectively by density-based approaches because they are not connected at all. In addition, few clustering approach is proposed to group elements with connectivity and similarity at the same time. In order to cluster the program elements with two measures, strategies such as balance [5] and alternative approaches can be adopted. However, connectivity and similarity are two independent measures. In recent research, it is not clear how to balance the weights of two measures, which depend on the peculiarities of software itself [4].

Another challenge is to identify multiple levels of a software system for architecture recovery. Hierarchy clustering is seemingly an effective approach since its goal is to find multiple levels in graphs. However, recent researches focus on choosing one partition from hierarchy clustering result as the “best result” [6]. Existing hierarchy approaches can hardly find multiple stable levels of software system. For example, random walk-based approaches [7] take advantage of authority scores to find typical dense communities in networks. Also, some metrics such as Q-function [6] are used to measure the quality of partitions and their extreme values indicate the best/worst partitions. However, none solution can be used to identify multiple levels of software systems.

Our solution is based on the observation that a complete software module consists of closely connected and similar elements. With directed graph as the representation of software programs, elements are grouped following certain rules:

1. **Connectivity** — If the connectivity between two elements is high, the program elements are likely to implement the same functionality and they should be in the same cluster.

2. **Transitive Connectivity** — Even if two program elements are linked indirectly, they might still have certain connectivity. For example, if most of A’s successors only reference B, they are likely to belong to one cluster.

3. **Similarity** — If the similarity between two elements is high, their functions are similar and they should be grouped into one cluster.

4. **Transitive similarity** — Even if two elements have no successors in common, they might still have certain similarity. For example, if A calls C and B calls D, and C and D are of high similarity, A and B are also similar and thus they should be in the same cluster.

Both closely connected and similar program elements should be grouped into one cluster. A flexible clustering algorithm is required to coordinate different types of clustering. In the clustering algorithm, two issues should be achieved to detect multiple stable levels: (1) obtaining detailed structure information in graph, (2) adopting effective approach to detecting stable levels. Software architecture recovery
has attracted a lot of interest. A variety of approaches have been reported. In summary, the contributions of this paper are as follows:

1. **Graph Transformation** is adopted to generate heterogeneous graphs, which include two types of edges that represent connectivity and similarity between program elements respectively. After the transformation, density-based clustering approaches can be used to group similar program elements. Furthermore, an *alternating strategy* is proposed to group both closely connected program elements and similar program elements into one cluster.

2. We improve authority-shift clustering [7] based on short random walk model. The *expansion factor*, which is used to set the steps of random walk, enables the coordinate of density-based and similarity-based clusterings. In addition, more local information of graph can be obtained to identify multiple levels of software systems.

3. A novel hierarchy clustering approach is implemented to detect multiple levels of software systems. All stable layers are detected with *Multi-layer Propagation Gap*.

4. Different software systems, including some large-scale ones, are used to evaluate our approach. A series of experiments are designed and conducted to validate the effectiveness and efficiency of our approach.

This paper is organized as follows. We give the motivating example in Sec. 2. In Sec. 3 we propose our graph clustering approach. The setup of our experiment is given in Sec. 4. In Sec. 5 we evaluate our approach with several cases. We introduce related work in Sec. 6. Finally, we conclude the paper in Sec. 7.

2. Motivating Example

In this section, we present an example to motivate our research. The example is shown in Fig. 1(a), where a vertex represents a method or a variable and a directed edge represents the reference relationship between two elements. In this example, there are two groups of elements. One includes the functions for "logbook operation". Another includes the program elements for "buffer management". As presented, functions \(\{f_1, f_2, f_3\}\) operate on logbook and use variable \(v_1\), while functions \(\{f_4, f_5, f_6, f_7\}\) work on buffer management and use \(\{v_2, f_8\}\). Using different clustering measures, the graph can be partitioned into two clusters in three different ways:

1. **Connectivity-based Clustering.** Figure 1(b) shows the clustering result based on the connectivity dependency relationship between vertexes. The elements within clusters are closely connected. However, similar elements are not grouped, e.g. some "logbook" functions are not placed into the cluster — "logbook".

2. **Similarity-based Clustering.** Figure 1(c) shows the clustering result based on similarity, i.e. functionality. Functions within a cluster implement the same
functionality. However, because the connectivity between program elements is ignored, the closely connected program elements are lost, such as utility functions and used variables.

(3) **Similarity and Connectivity Clustering.** Figure 1(d) shows the clustering result based on both the connectivity and similarity between the program elements. This clustering result reflects the coordination of connectivity and similarity between program elements. The elements within one cluster are closely connected. Meanwhile, they implement different parts of one functionality. This is the ideal cluster we want to find in this work.

From this example, we could know that the relationship existing between program elements in a software is not only connectivity but also similarity. The connectivity and similarity measures are independent of each other, and these two heterogeneous measures should be coordinated reasonably while recovering the architecture of a software system. Based on these two heterogeneous measures, the suitable clustering approaches are suggested to detect the modules in the software systems. The clustering approaches could be density-based clustering approaches, or hierarchy clustering approaches. The clustering approaches designed in our research work is according to the features of the problems we want to solve. Additionally,
there are many extra approaches needed for the clustering approaches to deal with these two heterogeneous measures.

3. Software Clustering

In this section, firstly, we consider two different types of correlations as the distance measurement: Similarity and Connectivity. Secondly, we introduce a novel similarity-based Directed Graph Hierarchy Clustering approach (DGHC) detecting the modules of software and analyzing the clustering results for software architecture recovery.

3.1. Correlations on directed graph

Software measurement has been studied along with the development of software [8]. A proper definition of distance between program elements is the primary challenge for clustering a directed graph. In recent research on complex network, new concepts of distance and correlation have been proposed. In our previous research, two types of correlations are defined. One is SimCorr, which is the distance that describes the similarity between program elements based on their features and behaviors in the context. The other is ConnCorr, which is the distance that describes the correlation between program elements based on their connectivity features.

3.1.1. Similarity

Two program elements are considered to be similar if they reference many common or similar elements, or they are referenced by many common or similar elements. In a program dependence graph such as call graph, two methods are considered to be similar if they reference a lot of common methods or fields. Figure 2(a) presents an example. However, they are not considered to be similar if they are called by common methods. The reason is that methods from different modules can be called in the same method. Figure 2(b) presents such an example. Furthermore, it is common that similar methods are not called by the same method since they may implement different functions of a module. Inspired by SimRank [9], we define an iterative equation, i.e. Eq. (1), for computing the similarity scores between program elements.

Definition 1. [Similarity Correlation] \( O(a) \) is the element set of \( a \)'s successors, \( |O(a)| \) is the out-degree of \( a \), \( O_i(a) \) is \( a \)'s \( i \)th successor. \( |I(O_i(a))| \) is the in-degree of \( a \)'s \( i \)th

![Fig. 2. SimCorr correlation.](image)
successor. \( s_k(a, b) \) denotes the similarity score between vertexes \( a \) and \( b \) at \( k \)th iteration. \( c \) is a constant between 0 and 1. \( t \) is the \textit{Crosscutting Factor} for optimization purpose. The value for \( t \) is no less than 0. The initiation of computation is that \( s_0(a, b) = 0 \) if \( a \neq b \), otherwise \( s_0(a, b) = 1 \). The distance of similarity correlation \( s_{k+1}(a, b) \) from \( a \) to \( b \) is defined as:

\[
 s_{k+1}(a, b) = \frac{c}{|O(a)| \cdot |O(b)|} \sum_{i=1}^{\|O(a)\|} \sum_{j=1}^{\|O(b)\|} s_k(O_i(a), O_j(b)) \frac{(|I(O_i(a))| \cdot |I(O_j(b))|)^t}{C^2}. 
\]  

(1)

The program elements that call many common or similar elements will obtain higher similarity scores. However, some program elements in the graph have a large scatter degree. This scatter feature affects the accuracy of the similarity computation. So we need to optimize the computation of similarity to reduce the side effect caused by the scatter between program elements. Take Fig. 2(c) as an example, where ellipses like \( E \) and \( F \) stand for different modules in the software system, the node 5 means a frequency called program element. So node 5 has a high scatter degree. According to the theory of similarity computation, the references from modules \( E \) and \( F \) to crosscutting node 5 significantly increase similarity score of different modules unexpectedly. To eliminate the side effect of crosscutting program elements, an optimization is taken through dividing the similarity scores of their successors by their in-degrees, as presented in Eq. (1). Finally, it is quite easy to prove the convergency of the optimized equation which is similar to the proof for SimRank [9].

3.1.2. Connectivity

For clustering directed graphs, the other type of correlation named \textit{ConnCorr} is defined. We observe that two program elements have a high probability to be in the same module if one calls many elements that call the other and vice versa. \textit{ConnCorr} measures the connectivity between program elements which is calculated using Eq. (2). Inspired by the expected-f meeting distance [9], we choose the unidirectional connection. In Eq. (2), \( t : a \rightarrow b \) includes all the paths from \( a \) to \( b \), \( l(t) \) means the length of path \( t \), \( p \) is the probability of choosing the unidirectional connection \( C(a, b) \) from \( a \) to \( b \).

\[
 ConnCorr(a, b) = \sum_{t : a \rightarrow b} p(t) e^{l(t)}. 
\]  

(2)

The connectivity score \( ConnCorr(a, b) \) can be easily obtained through expected-f meeting distance, which can eliminate the side effect of hub nodes, i.e. the nodes that have weak connections to other nodes. Nodes 2 and 4 in Fig. 3(a) are the examples of hub node. In Fig. 3(a), because node 2 also connects to other nodes, node 2 has a weaker connection with node A than node 1. However, the connectivity score is influenced unexpectedly by scattering program elements. The unidirectional
connection to scattering program element is supposed to be weak, as presented in Fig. 3(b). Node B is a crosscutting node while C is not. The connection from nodes (such as node 7, 8, 9, 10, 11) to B must be lower than the connection from B to C. To achieve the optimization, the connectivity scores of scattering program elements should be reduced. With all the discussion mentioned above, we give the definition of connectivity correlation in Eq. (3), which is used to calculate the connectivity between program elements in the unidirectional connection.

**Definition 2.** [Connectivity Correlation] a is a vertex that stands for one program element. \(O(a)\) is the element set of a’s successors, \(|O(a)|\) is the out-degree of a, \(O_i(a)\) is the a’s ith successor, \(|I(O_i(a))|\) is the in-degree of a’s ith successor. \(ConnCorr_k(a, b)\) denotes the connectivity score between vertex a and vertex b at \(k\)th iteration. \(c\) is a constant between 0 and 1. To eliminate the side effect of scattering program elements, connectivity score \(ConnCorr(O_i(a), b)\) is divided by \(|I(O_i(a))|\). The initiation of computation is that \(ConnCorr_0(a, b) = 0\) if \(a \neq b\), otherwise \(ConnCorr_0(a, b) = 1\). The distance of connectivity correlation \(ConnCorr_{k+1}(a, b)\) from a to b is defined as:

\[
ConnCorr_{k+1}(a, b) = \frac{c}{|O(a)|} \sum_{i=1}^{|O(a)|} \frac{ConnCorr_k(O_i(a), b)}{|I(O_i(a))|}.
\]  

The proof of its convergence is omitted since it is similar to that of the expected-f meeting distance [9].

### 3.2. Graph transformation

Until now, clustering directed graphs is still a difficult problem. Firstly, directed graphs include rich content semantic meaning and multi-type meaning. It is hard to express and analyze these complex semantic meanings with the original directed graph. Secondly, the existing analytical approaches cannot conduct the analysis efficiently on directed graphs. Based on the above analysis, in order to cluster directed graphs efficiently clustering, a preprocess to the original directed graph is needed. In this section, we introduce Graph Transformation operation, which transfers a directed graph into an undirected graph, which we call Correlation
Graph. A Correlation Graph contains relation information between the nodes in the graph. The definition of Correlation Graph is given below.

**Definition 3.** [Correlation Graph] Given a directed graph $G(V, E)$, where $V = \{v_{ij}\}_{i=1,j=1}^{n,n}$ is the set of nodes and $E$ is the set of directed edges. A Correlation Graph is denoted as $G_c(V, E \cup E_s)$, where $E \cup E_s$ is the set of undirected edges. An edge $(v_i, v_j) \in E_s$ if there exists Similarity score between vertex $v_i$ and $v_j$ according to Eq. (1) which is SimCorr. The weight of edge $(v_i, v_j) \in E$ is Connectivity score, and the Connectivity score is calculated using Eq. (4) which is called ConSco and makes $ConSco(a, b) = ConSco(b, a)$.

$$ConSco(a, b) = ConnCorr(a, b) + ConnCorr(b, a).$$ (4)

In a Correlation Graph, SimCorr is used to measure the similarity between program elements, including those that are not connected. ConSco is used to measure the connectivity between program elements. The whole transformation from a directed graph $G$ to the correlation graph $G_c$ is called Graph Transformation operation which converts the similarity-based clustering problem to a density-based clustering problem. Since there are effective density-based clustering approaches to identify dense communities in undirected network, we draw lessons from the approaches on the undirected graph with the help of Graph Transformation, and get the clustering results of the original directed graph. In the Correlation Graph, we are able to cluster directed graphs indirectly.

Here we present an example to demonstrate how the Graph Transformation operation is performed on a directed graph. As presented in Fig. 4, in order to cluster the directed graph in Fig. 4(a), we first transfer it into a Correlation Graph, as presented in Fig. 4(b). This undirected correlation graph is also a heterogeneous

![Graph Transformation](image-url)
3.3. Basic graph clustering approach

Inspired by Authority-shift clustering using Personalized PageRank (PPR) [7], in this section, we introduce our basic density-based approach for graph clustering, namely Short Authority-shift. As presented in the last section, density-based graph clustering approaches can be applied to identify the communities in directed graphs after the process of Graph Transformation. Short Authority-shift is also a new authority seeking approach in graphs that computes the shift nodes using short random walk for authority shift clustering. Taking advantage of more structure information in the graph, it can identify clusters in a more proper way. Furthermore, the propagation of authority computation is discussed for hierarchy clustering.

3.3.1. Short random walk

For undirected graph clustering, a cluster consists of a dense node and the nodes nearby. Therefore, efficient identification of local dense nodes is crucial to authority shift clustering. We have designed a novel searching approach for authority shift clustering based on short random walk, which detects authority nodes of high authority scores as dense nodes. First, we define the short random walk model.

**Definition 4.** [Short Random Walk Model] Let $B$ be the transformation matrix in random walk which can be the matrix obtained from SimCorr or ConnSco. $t$ is the Expansion factor which denotes the length of random walk. $t$ is no less than 1. $LPR$ (Local PageRank) is the Expansion matrix which is used to search local shift nodes in both undirected and directed graph.

\[ LPR = B^t. \]  

The length of random walk should not be too long to obtain enough information from local part of graph. The most important thing is that the short random walk model searches for the highest authority score in a fixed scale, instead of directly
using the global authority node. As a result, the approach can use more structure information of the undirected and directed graph and produce more reliable clustering result in a more efficient way.

3.3.2. Authority propagation

Like authority shift clustering, the clustering approaches based on short random walk model can be applied to different applications for the computation of the authority value of nodes in the specified step length. Based on our short random walk model, we can identify dense nodes on different scales, as well as the multiple levels of structure in graph. We use Eq. (6) to iteratively update the $LPR$. As the propagation of authority scores iterates, we obtain the multi-scales authority scores for identifying authority nodes.

**Definition 5.** [Authority Propagation] Let $LPR$ be the Expansion matrix which denotes all the Local PageRank (LPR) scores between all nodes. The $i$th column vector of $LPR$ is denoted by $LPR(i)$, which denotes all the Local PageRank (LPR) scores from all nodes to node $i$. $LPR(i, j)$ from expansion matrix $LPR$ can be viewed as the importance of node $j$ with respect to node $i$. The order $n$ means that the random walker starts $n$ times again from the steady state of $LPR(i)$. Based on this concept, we define the $(n + 1)$th order $LPR$ by $LPR$ propagation as presented in Eq. (6), where the $LPR$ vector is recursively used for high-order personalization.

\[
LPR_{n+1}(i) = LPR \cdot LPR_n(i). \tag{6}
\]

$LPR_n(i)$ stands for a probabilistic landscape of the authority score around node $i$ and gradually propagates the authority score beyond node $i$ with increasing order $n$. Based on the $n$th order $LPR$, the authority node is assigned to node $i$ for each order $n$ by Eq. (7) which has the highest authority score for node $i$.

\[
Auth_n(i) = \arg \max_{s \in V} LPR_n(i, s). \tag{7}
\]

The identification of authority nodes for a fixed node is an aggregation process, which is shown in Fig. 5. The graph example is presented in Fig. 5(a). We compute the authority scores for the blue node. The color map is drawn with the different values of authority scores. The color of dark red represents high authority scores, while the color of dark blue represents low authority scores. At the beginning of authority propagation as shown in Fig. 5(b), the authority node is the one of high authority score nearby. As the computation proceeds, its authority node will gradually change to the one in larger scale. Eventually, its authority node will be the one in global scale, as shown in Fig. 5(f). Even though the example is an undirected graph, our approach is also valid in directed graphs. Because this algorithm only improves the authority propagation model and the whole algorithm is still the same as the original one, time complexity is the same as the original one and takes $O(N^3)$ time, where $N$ represents the number of nodes in the graph.
3.4. Hierarchy clustering

Based on the simple clustering approach presented in Sec. 3.3, in this section we introduce a hierarchy approach for directed graph clustering. It is an uniform approach of clustering directed graph of heterogeneous correlations. We first give the updating rules for computing correlations between clusters. Then we present the clustering algorithm. Finally we discuss the convergency and the number of clusters for the hierarchy clustering approach.

3.4.1. Correlation update

In the hierarchy clustering approach, the nodes in each cluster are aggregated into a super node recursively and the super node is the authority node of each cluster which we choose as the representation of each cluster. A higher-layer graph with fewer super nodes is constructed for the next run of hierarchy authority shift.

In order to measure the similarity score between two clusters, the definition of the similarity update rule is given to update the similarity scores in super graph, of which the vertex set is super nodes. In this update rule, the LPR relations between the clusters are updated by linear combinations of the $n$ order LPRs on the $l$ layer as presented in Eq. (8).

**Definition 6.** [Similarity Update Rule] Given $LPR_{Sim}$ be the similarity LPR matrix which denotes the similarity scores between all program elements calculated...
in Eq. (1). \( l \) presented the hierarchical layer and \( n \) is the order of LPR. \( C^l_a \) represents a cluster in the \( l \) layer, \( a \) is the authority node which belongs to this cluster in \( l \) layer. \( \sigma_{l+1}(a) \) is the number of nodes in cluster \( C^l_a \) and it will be used in the \( l+1 \) layer. There is \( \sigma_{l+1}(a) = \sum_{i \in C^l_a} \sigma_l(i) \). The similarity update rule for the clusters in \( l+1 \) layer is described in Eq. (8).

\[
LPR\_Sim^{l+1}(a, b) = \frac{1}{\sigma_{l+1}(a)} \sum_{i \in C^l_a, j \in C^l_b} \sigma_l(i) LPR\_Sim^l(i, j).
\] (8)

The definition of the connectivity update rule is given below about how to update the connectivity score between two clusters when aggregation proceeds to the next layer.

**Definition 7.** [Connectivity Update Rule] Let \( LPR\_Con \) be the connectivity LPR matrix which represents the connectivity scores between all elements calculated in Eq. (4). The connectivity update rule for the clusters in \( l+1 \) layer is described in Eq. (9).

\[
LPR\_Con^{l+1}(a, b) = \frac{1}{\sigma_{l+1}(a)} \sum_{i \in C^l_a, j \in C^l_b} \sigma_l(i) LPR\_Con^l(i, j).
\] (9)

### 3.4.2. Hierarchy clustering algorithm

For finding communities on directed graphs, we propose a hierarchy clustering approach on *Correlation Graph* that have two different measures — *Similarity* and *Connectivity*. Such a graph is referred to as a Directed Graph Hierarchy Clustering (DGHC) in this research. In this hierarchy clustering approach, the nodes in each layer are aggregated iteratively with *Similarity-shift* and *Connectivity-shift*. After each shift, the super graph of the *Correlation Graph* is updated by computing *Similarity* and *Connectivity* between clusters using Eqs. (8) and (9). The super graph itself is also a *Correlation Graph*. Our agglomerative update rule is derived from Authority-shift clustering [7]. Compared to Authority-shift, our updating rule is more stable since we adopt the short random walk model to detect the authority node in both *Similarity-shift* and *Connectivity-shift* processes. Furthermore, it improves hierarchy clustering quality since short random walk can obtain more structure information from local graph. This approach may slow down the convergence speed of clustering (the experiment shows that this approach improves the convergence speed of clustering a lot). The algorithm for clustering directed graph is summarized in Algorithm (1). As the final result, it generates a whole hierarchical representation of the directed graph.

### 3.4.3. Algorithm analysis

In the hierarchy clustering approach, each shift operation (including *Similarity-shift* and *Connectivity-shift*) corresponds to one layer in the authority propagation. In
Algorithm 1: Directed graph hierarchy clustering.

Input: Weight matrices \( W_S \in \mathbb{R}^{n \times n}, W_C \in \mathbb{R}^{n \times n} \).
Normalize \( W_S, W_C \) to stochastic matrices \( P_S, P_C \).
Compute the LPR matrices \( LPR_{Sim}, LPR_{Con} \).

\[ l = 0, \]

\begin{algorithmic}
\Repeat
\State \( l = l + 1, n = 0, \)
\Repeat
\State \( n = n + 1, \)
\State Identify authority nodes and clusters,
\State Propagate \( LPR_n \) with matrix \( LPR_{Sim} \) through Eq. (6).
\Until \text{one authority node is shifted;}
\State Update super graph by Eqs. (8) and (9),
\State \( l = l + 1, n = 0, \)
\Repeat
\State \( n = n + 1, \)
\State Identify authority nodes and clusters,
\State Propagate \( LPR_n \) with matrix \( LPR_{Con} \) through Eq. (6).
\Until \text{one authority node is shifted;}
\State Update super graph by Eqs. (8) and (9),
\Until \text{all nodes are in a single cluster;}
\end{algorithmic}

Output: a hierarchy of clustering.

Each layer the authority propagation is implemented with short random walk, which guarantees that it converges to a steady state toward the global state. Furthermore, our hierarchy clustering approach is bottom-up aggregation clustering method, which ends by the predefined iteration times. Therefore, our hierarchy clustering will eventually converge to one cluster or a few clusters as a steady state.

The time complexity of the hierarchy clustering approaches is mainly determined by the matrix multiplication. In the worst case, the non-hierarchy clustering algorithm takes \( O(N^3) \) time where \( N \) is the number of nodes. However, for the hierarchy clustering approaches, the value of \( N \) decreases a lot with the increase of layer. So in the worst case, the time complexity of hierarchy clustering approaches is less than \( O(N^3) \) and the actual execution time will be far less than the non-hierarchy clustering approaches.

3.4.4. Selecting clustering layers

In the hierarchy structure of clusters obtained through graph clustering, stable clustering layers are required to analyze clustering results in multiple scales. In this section, we propose \textit{Multi-layer Propagation Gap} which guides the selection of stable clustering layers for our hierarchy clustering and improves the original Propagation
Fig. 6. Selecting stable layers.

(a) Clusters in different layers

(b) Number of clusters in each layer

(c) Number of orders in each layer
Gap method [7]. Even though the layer selection in Authority-shift clustering helps us identify communities, it has no effect on analyzing hierarchy structure of software. To discover the stable layers in hierarchy clustering, we adopt Multi-layer Propagation Gap which considers the increment rate of the size of primary cluster in each layer. These primary clusters include the primary authority element, which is the authority node of the largest cluster in the final layer. The aggregation of primary cluster indicates correlation between clusters in different scales, which can be applied to detect hierarchy structure of software system.

Different increment rates of primary cluster’s size imply different phases during hierarchy clustering. We discover two types of phase — Restoration and Expansion. In the phase of Restoration, most subclusters are obtained separately so that the size of primary cluster increases slowly, while in the phase of Expansion many subclusters are propagated to the primary cluster so that its size increases rapidly. On the other hand, we can say that it is a good hierarchy clustering if these two types of phases are obviously detected. Finally we select the stable layer at the end of Restoration or the beginning of Expansion phase.

An example is shown in Fig. 6 which shows how to select the stable layers. As illustrated in Fig. 6(a), the nodes in the same color belong to the same cluster and the nodes with wide line are the authority nodes of the clusters. The goal is to choose the stable clustering layers \( L_S \) where \( S = \{ s_1 < s_2 < \cdots < s_n \} \). If layer \( L_{s_i} \) is stable and the clustering result is ideal, the number of clusters remains stable in the next layer. Take Fig. 6(b) for example, the layer set \( \{ 1, 3, 4 \} \) are chosen. Meanwhile, after getting to the stable layer, a large number of orders are required for authority propagation to reach the next level of hierarchy clustering. With the number of orders provided in Fig. 6(c), the layer set \( \{ 1, 3 \} \) are suggested. Finally, we consider the intersection of the previous two layer sets \( \{ 1, 3 \} \) as stable layers, in which layer 1 is the starting layer and layer 3 is the ending layer of the hierarchy structure. The analysis of hierarchy structure is further discussed in the experiment section with real graph data.

4. Experimental Setup

4.1. Datasets

We have performed experiments using four classic datasets which are listed in Table 1. There are two values in the column of “Number of Elements” for each case.

<table>
<thead>
<tr>
<th>Software system</th>
<th>Version</th>
<th>Language</th>
<th>Num. of elements</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>5.4</td>
<td>Java</td>
<td>3588/3800</td>
<td>11653</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>1.8.0.2</td>
<td>Java</td>
<td>3734/4012</td>
<td>70014</td>
</tr>
<tr>
<td>Prevayler</td>
<td>2.0.005</td>
<td>Java</td>
<td>1043/1225</td>
<td>3322</td>
</tr>
<tr>
<td>DAQ</td>
<td>7.6.2</td>
<td>C</td>
<td>4262/4717</td>
<td>153687</td>
</tr>
</tbody>
</table>
in Table 1. The bigger value represents all elements in the software programs and
the smaller value means the program elements without isolated elements. The
isolated elements are the elements both the in-degree and out-degree of which is
zero. The cases JHotDraw, HSQLDB and Prevayler are implemented in Java
language, while the case DAQ is implemented in C language. DAQ [10] is a Data
AcQuisition system in ALICE (A Large Ion Collider Experiment) which is one of
the four LHC (Large Hadron Collider) experiments performed at CERN (European
Organization for Nuclear Research). The whole DAQ system consists of hardware
system and software system. The software system of DAQ implements the dis-
tributed processes and the control mechanisms for a high-performance and reliable
data taking. It has been developed for 15 years and includes over 150,000 lines of
C source code.

We analyze four software systems using our program analysis tools based on JDT
[11] and CDT [12]. The clustering algorithms are implemented with Matlab 2008b.
The hardware configuration of the testing is Intel Core 2 Duo CPU 2.26 GHz and
8 GB memory, and the software platform is Windows 7 64bit edition and Scientific
Linux CERN Version 5 with JDK version 1.6.

4.2. Comparison methods and setup

To evaluate the quality of clusters generated by our approaches, we have designed
and conducted a series of experiments. In these experiments, we have evaluated the
quality and efficiency of DGHC from different aspects, selecting stable layers,
grouping similar elements, clustering software modules with different correlations
and detecting multiple levels of software system.

The setting of the Crosscutting Factor and Expansion Factor is as follows:

- **Crosscutting Factor** — The scattering elements may cause side effects on the
  similarity score. We use the in-degree of the common successors of two program
  elements to optimize the similarity between them. The Crosscutting Factor \( t \) in
  Eq. (1) is used to adjust the contribution of scattering to the similarity. If \( t \) is too
  small, it cannot eliminate the side effects. Meanwhile, this factor cannot be too big
  either. According to the experiences and experiments, we chose \( t = 1 \) when com-
  puting the similarity between program elements.

- **Expansion Factor** — The search scope of random walk has significant impact
  on finding dense communities in graph. We chose a proper random walk length
  to detect dense communities in a fixed scale. If the length is too large, it can
  hardly obtain enough information from the graph. The length is adjusted with
  the Expansion Factor \( t \) presented in Eq. (5) for both density-based clustering
  and similarity-based clustering. It also enables the coordination of these two
  clusterings. According to our experiences and experiments, the Expansion
  Factor for Similarity is \( t_{\text{sim}} = 2 \) and the Expansion Factor for Connectivity is
  \( t_{\text{conn}} = 1 \).
4.3. Evaluation measures

Precision and Recall are traditional measures which are used to evaluate the clustering quality for each module. In the experiments, we quantify the quality of software clustering with the average precision and average recall. The average precision is the expected value which is the weighted average of the precisions of all modules. The weight of each precision is determined by the size of clusters corresponding to its module. The average recall is the expected value which is the weighted average of all modules’ recalls.

The clustering precision for each module is calculated using Eq. (10), where $|s|$ is the size of clusters corresponding to one module, $SEN_s$ is the number of elements in these clusters belonging to module $s$, and $P_s$ expresses the clustering precision of module $s$. Then, by calculating the expectations of precisions for all modules in one dataset, the average precision is obtained for this dataset using Eq. (11), where $S$ is the collection of program elements in the dataset, $|S|$ denotes the number of program elements in the collection, and $APS$ is the average clustering precision of dataset $S$.

$$P_s = \frac{SEN_s}{|s|}$$  \hspace{1cm} (10)

$$APS = \sum_{s: s \in S} \frac{|s|}{|S|} \times P_s.$$  \hspace{1cm} (11)

The clustering recall for each module is computed using Eq. (12), where $R_s$ denotes the clustering recall of module $s$ and $C_s$ stands for the size of module $s$. The average recall is obtained using Eq. (13) where $ARS$ expresses the average clustering recall for dataset $S$.

$$R_s = \frac{SEN_s}{|C_s|}$$  \hspace{1cm} (12)

$$ARS = \sum_{s: s \in S} \frac{|s|}{|S|} \times R_s.$$  \hspace{1cm} (13)

5. Experiments

In this section, four experiments have been designed and conducted for software architecture recovery. We do not aim to reconstruct software architecture such as class or package, but group program elements of similar semantic with a hierarchy structure.

5.1. Selecting stable layers

How to select stable layers from the clustering result of the dataset JHotDraw is illustrated by Fig. 7 to discover its hierarchy characteristics through the identification of multiple stable layers. Figure 7(a) tells that the start layer of hierarchy clustering is 3, Fig. 7(b) shows that the end layer is 52. Figure 7(c) indicates the
number of elements in primary cluster from Layer 3rd to Layer 52nd, and the hierarchy structure of software could be detected through the analysis of the rate of primary cluster. For example, in Fig. 7(c), the phase in (38, 40) is Restoration and the phase in (40, 42) is Expansion. We manually check this Expansion phase and find that 16 clusters of about 1500 elements are grouped to the primary cluster in 2 orders. Using our approach, all stable layers of 4 software systems are listed in Table 2. These data also illustrate that our clustering approach is capable of producing clusters of good hierarchy structure.

Table 2. Stable layers.

<table>
<thead>
<tr>
<th>Software system</th>
<th>Set of stable layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>{3, 12, 16, 20, 32, 36, 40, 48, 52}</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>{4, 13, 17, 23, 29, 36, 43, 52, 57}</td>
</tr>
<tr>
<td>Prevayer</td>
<td>{2, 8, 15, 22, 33, 37, 40}</td>
</tr>
<tr>
<td>DAQ</td>
<td>{2, 6, 12, 19, 23, 31, 35, 38, 42}</td>
</tr>
</tbody>
</table>

Fig. 7. Jhotdraw hierarchy structure.
5.2. Grouping similar elements

Software modules contain similar program elements that implement common functionalities. Firstly, we use several cases to illustrate the effectiveness of our approach in grouping similar program elements. Then, we compare it with two clustering algorithms.

First of all, the clustering results of JHotDraw are analyzed in details. Since DGHC is a hierarchy approach and converges fast, we should choose the layer in the early stage of hierarchy clustering to avoid big clusters that contain different program modules. We chose Layer 12 as the initial stable layer of JHotDraw and analyze 10 largest clusters which are described in Table 3. As presented in Table 3, basic information of these modules are presented such as Name, Description, Authority Element, Size and Precision. Since our approach generates clusters of hierarchy structure, these clusters are not simple and they may be the combination of clusters or part of a large cluster. For example, modules {2, 3, 4, 7} are composite ones which include several sub-modules, while the 2th module Tool consists of four sub-modules

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
<th>Authority element</th>
<th>Size</th>
<th>Prec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Polygon operations</td>
<td>Operations on polygon figure, such as moving, deleting and chopping</td>
<td>PolygonFigure. ba-sicMoveBy()</td>
<td>57</td>
<td>94.7%</td>
</tr>
<tr>
<td>2</td>
<td>Tool</td>
<td>Various tools for figure and connection, such as creating figure and deleting connection</td>
<td>EventDispatcher. addToolListener()</td>
<td>122</td>
<td>98.4%</td>
</tr>
<tr>
<td>3</td>
<td>Command</td>
<td>Command related functions such as add listeners and execute command</td>
<td>EventDispatcher.removeCommandListener()</td>
<td>56</td>
<td>100%</td>
</tr>
<tr>
<td>4</td>
<td>Connection</td>
<td>Connection functions, such as get startPoint and endPoint of connection, draw connection</td>
<td>ChangeConnection StartHandle.target()</td>
<td>62</td>
<td>100%</td>
</tr>
<tr>
<td>5</td>
<td>Handle 1</td>
<td>Various handles such as rotation handle and radius handle, special handles for locating</td>
<td>AbstractHandle.owner()</td>
<td>75</td>
<td>94.7%</td>
</tr>
<tr>
<td>6</td>
<td>Handle 2</td>
<td>Handles similar with those in 5th cluster, special handles for line and connection</td>
<td>HandleEnumerator. nextHandle()</td>
<td>85</td>
<td>97.6%</td>
</tr>
<tr>
<td>7</td>
<td>Mouse actions</td>
<td>Functions for mouse actions, such as drag, double click and zoom</td>
<td>AreaTracker.mouseDrag()</td>
<td>101</td>
<td>100%</td>
</tr>
<tr>
<td>8</td>
<td>Scaling Graphic</td>
<td>Various functions for Scaling Graphic, such as create, setFont and drawImage</td>
<td>ScalingFontMetrics. ScalingFontMetrics()</td>
<td>59</td>
<td>100%</td>
</tr>
<tr>
<td>9</td>
<td>Figure</td>
<td>Operations on figures, such as create and display, read and write implement figure persistence</td>
<td>HTMLTextArea-Figure. clone()</td>
<td>66</td>
<td>93.9%</td>
</tr>
<tr>
<td>10</td>
<td>Locator</td>
<td>Handles for locating figures, such as RelativeLocator handles, Resize handles an North handles</td>
<td>BoxHandleKit. ad-dHandles()</td>
<td>63</td>
<td>96.8%</td>
</tr>
</tbody>
</table>
such as command and figure tools. Modules \{5, 6, 10\} are all parts of a larger module for implementing handle functions. We observe that the average precisions of these clusters are all over 90% (up to 100% in some cases).

5.3. Clustering with different correlations

In these series of experiments, we compare DGHC with other clustering algorithms for software clustering. More importantly, we discuss the effectiveness of the identification of software modules based on different correlations. The algorithms that we compare DGHC with are Short Authority-shift and SpectralKMeans. DGHC considers both connectivity and similarity between program elements, while Short Authority-shift and SpectralKMeans consider only similarity. The clustering results are shown in Fig. 8. For DAQ dataset, average precision of DGHC on DAQ dataset is 0.7625 and average recall is 0.7884 in Layer 6. There are 78 layers for Short Authority-shift algorithm when analyzing DAQ dataset, and the Layer 9 is chosen as the initial stable layer in which 213 clusters are found and the number of elements in the largest cluster is 2197. The average precision and recall of Short Authority-shift on DAQ dataset are 0.7245 and 0.2341. There are no layers for SpectralKMeans, the number of largest cluster is 2413 on DAQ dataset, average precision and recall of SpectralKMeans on DAQ dataset are 0.1229 and 0.4956. Generally, DGHC has the highest average precision and average recall, while SpectralKMeans has the lowest average precision and Short Authority-shift has the lowest average recall. More specifically, the average precisions of DGHC and Short Authority-shift are much higher than SpectralKMeans. Since partition clustering approaches like SpectralKMeans need the number of clusters in advance. However, it is difficult to obtain this number without domain experts. As a result, these approaches are unstable. If the number is too small, “big” clusters will appear so that the precision can be quite low and the average recall can be relatively higher.

Fig. 8. The comparison experiment for graph clustering approaches with different types of correlation.
As illustrated in Fig. 8, correlations chosen for software clustering have significant
effect on the identification of software modules. Since the average precisions of both
DGHC and Short Authority-shift are quite high, we conclude that the similarity
between program elements is quite important for finding modules. Compared with
Short Authority-shift, the average recall of DGHC is much higher because DGHC
detects modules with two types of correlations, which enables grouping both similar
and closely connected program elements. Short Authority-shift only considers Similarity
and thus the program elements closely connected to a module are lost. We
observe that the program elements, like global variables and the complex variables,
have high connectivity with specific module. However, they have no successors. Such
program elements should be grouped into modules according to connectivity.
According to this experiment, we can conclude that the clustering algorithms, which
use both connectivity and similarity, are more effective than those that consider only
a single correlation when detecting software modules.

5.4. Detecting multiple levels of software system

In these series of experiments, we use a case study to illustrate the ability of our
approach in detecting multiple levels of a software system. The clustering result of
DAQ at different layers is given in Table 4. In total, there are 42 layers in the

<table>
<thead>
<tr>
<th>Level</th>
<th>Stable layers</th>
<th>Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>2, 6</td>
<td>edmClient, edmAgent, ioHandler, ldcHandler, evbRuntimeHandler, monitorPartition, monitorBuffer, monitorGdcs, monitorClients, equipmentList_UDP, equipmentList_CTP, equipmentList_DDL, dbTrigger, dbAux, rorc_lib, rorc_send, rorc_receive, rorc_ddl, recordingLib, recordingValidator, DAQlogbook_tcl, infoLogger_tcl, recordingHandler, queuesSupport, changingConsumer, producer, consumer, shellInterface, fortranInterface, datenetperfIdc, datenetperfIdc, datenetperfcommom, validateQueues, dateRec, colePrint, coleParse, logbookDaemon, stream, mStreamRecorder_uuty, simpleFifo, simpleFifoValidate, tdsnPerformanceMeasurements, rorc_driver, dbInput, etc</td>
</tr>
<tr>
<td>Middle</td>
<td>19, 23, 31</td>
<td>mStreamRecorder, eventBuilder, rorc_driver, rorc, ValidateEvent, tdsnRequestWritePath, dumpDbs, DAQlogbook_tcl, dateStream, runControl, equipmentList, infoLoggerServer, edm, hltAgent, rcServer, tdsnPerformanceMeasurements, doroFlash_programmer, permanentFIFO, infoLogger, runControlH, rorc_driver, cole, dateBufferManager, prorc_ws, infoLogger_tcl, monitor, perf, logbookDaemon, validateQueue, datenetperf, fortranInterface, etc</td>
</tr>
<tr>
<td>High</td>
<td>35, 38, 42</td>
<td>DAQ, DAQlogbook_tcl, infoLogger_tcl, fortranInterface, checkLockedRuncontrol, checkLockedReservers, dateCleanup, tdsnPerformanceMeasurements, smi_timeout, runSubscribe, statsCollectorTrigger, fake_statsCollectorDaemon, dummy.bd, dummy_collector, backP_monitor, rorc_driver, etc</td>
</tr>
</tbody>
</table>
clustering results generated by DGHC. 8 layers are suggested as stable layers with Multi-layer Propagation Gap. In this table, the stable layers \{2, 6, 19, 23, 31, 35, 38, 42\} can easily be divided into three different levels: layers \{2, 6\} are the low level view of the software system which implements basic functionalities. Layers \{19, 23, 31\} are the middle level view which includes various modules. Layers \{35, 38, 42\} are the high level view which consists of a few main subsystems. The detailed information of different levels are presented in Table 4.

- **Low Level Modules.** Layers \{2, 6\} are the low level view of the software system which includes a lot of basic modules. For example, module \textit{ldcHandler} is used to handle the data from Local Data Concentrators (LDC); module \textit{edmHandler} is used by the event builder daemon to handle all the communication with the Event Destination Manager (EDM); module \textit{minitorGdcs} is used to monitor Global Data Collectors (GDC); module \textit{rorc.lib} is used to provide library to all the programs using Read-Out Receiver Card (RORC) which is a PCI master card that provides an interface between the Detector Data Link (DDL) and the PCI; module \textit{rorc.dl} is used to provide library to all the programs related with DDL.

- **Middle Level Modules.** Layers \{19, 23, 31\} are the middle level view which has various modules consisting of basic modules. For example, module \textit{eventBuilder} runs on a GDC, receives data from several LDCs, assembles the data into single events and records them to the output stream; module \textit{monitor} offers a uniform interface for the development of user-specific monitoring programs in C and C++ to monitor the process of taking experimental data; module \textit{rorc} contains library functions related to RORC and DDL, as well as some utility functions as interface to a RORC device.

- **High Level Modules.** Layers \{35, 38, 42\} are the high level view of the program system that has a super module called \textit{DAQ} including 3463 elements. Module \textit{DAQ} is the primary cluster that provides all the necessary functionalities to perform data acquisition activities in the distributed environment. Other modules on this level may be interfaces for supporting other systems or scripts languages, such as \textit{DAQlogbook_tcl} and \textit{fortranInterface}, or commands for checking states of system, such as \textit{statsCollectorTrigger} and \textit{checkLockedReservers}.

As illustrated in Table 4, we can see many aggregation of software modules related to functionalities — monitor, database, event handler, rorc, stream recorder and log. Some basic modules on the low level are aggregated into larger modules on a higher level. For example, basic modules \{equipmentList\_UDP, equipmentList\_CTP, equipmentList\_DDL\} are aggregated into cluster \{equipmentList\}. Many modules on the middle level are aggregated into a super modules called \textit{DAQ} module on the high level. These modules are \{mStreamRecorder, infoLogger, logbook, rorc, runControl, simpleFifo, readout, readList, cole, recordingLib, db, eventBuilder, edm, physmem, bufferManager\}. In addition, some modules are independent from others and their sizes increase slowly on different levels, such as \{tdsmPerformanceMeasurements, rorc\_driver, DAQlogbook\_tcl, infoLogger\_tcl\}. 
Finally, we show the hierarchical structure of clustering result generated by DGHC through an example presented in Fig. 9. The basic modules are \{ldcHandler, edmHandler, memHandler, rorc\_lib, rorc\_ddl\}. The module eventBuilder on the middle level includes the basic modules \{ldcHandler, edmHandler, memHandler\}, while module rorc includes the basic modules \{rorc\_lib, rorc\_ddl\}. The eventBuilder and rorc modules are aggregated into module DAQ on the high level. In fact, module DAQ is composed of several parallel readout streams. Each of these read-out streams carries the data produced by the electronics of detector. These electronics are controlled and readout by LDC, which builds events for recording data. The functions in module rorc are used to interact with electronics, while the functions from module eventBuilder are responsible for receiving, assembling and recording data. As demonstrated by Fig. 9, this hierarchical structure produced by our approach shows that its high effectiveness in automatically finding the modules of software, as well as in detecting multiple levels of software systems for software architecture recovery.

6. Related Work

6.1. Clustering approaches

Clustering techniques aim at detecting groups of elements that are close to each other. Traditional clustering techniques [13, 14] such as KMeans [14] have been widely adopted in various domains. Due to high time complexity, they are too time-consuming, especially for large-scale dataset. In addition, they suffer from several disadvantages, such as selection of cluster number and initiation of cluster centers. For graph data, researchers have proposed new graph clustering techniques [15, 6, 16, 7]. Some of these techniques [15, 6, 16] are only applicable to undirected graph while others [17–19, 7, 20] are designed for directed graphs. A few hierarchical methods are used to discover multiple layers in graphs. Authority shift clustering [7] is a hierarchical method for both undirected and directed graphs. It overcomes the limitations of traditional clustering techniques and is quite efficient for detecting typical clusters.

As the amount of data in various domain grows fast, various query techniques are required to meet the need of retrieval service. Most classic techniques are to detect dense communities in networks [21, 6]. Recently, a most popular retrieval service is to
find a group of object similar to each other, such as web search [22] and community
detection for social networks [20]. Two main types of graph clustering techniques are
listed as follows:

(1) **Density-Based Clustering** — To detect dense communities in complex networks.
density-based methods [15, 6, 17, 18, 16, 7] have been widely studied. In recent
years, spectral methods [15, 17] and Random walk methods [18, 16] have been
adopted to find dense communities. However, those techniques cannot detect
the number of clusters. Newman [21, 6] proposes *Q-function* to obtain the
optimal partition of undirected networks. Shift-clustering [7] works with mean-
shift strategy.

(2) **Similarity-Based Clustering** — Several methods [19, 22, 5, 23, 20] have been
proposed based on *Similarity*. Some of those methods [19, 20] utilise symmetric
matrix for processing directed graphs, considering only similarity measure. In
some research [22, 5] tag (attribute) information is considered in undirected
graph clustering. The authors of [23] try to consider connectivity and similarity
at the same time for queries. However, no clustering method considering both
measures is introduced for directed graph.

### 6.2. Software clustering

Software clustering techniques have been adopted for software comprehension and
reengineering tasks [1], such as topics identification [24], software architecture re-
covery [3, 25–27] and subsystem structure identification [28, 29, 2]. The quality of
clustering depends not only on the similarity measure but also on the clustering
algorithm. Different features of program elements are used to define the similarity/
distance between them. Anquetil [28] used names of files to find software modules.
Program references are considered in the research presented in [29]. Using these
features, various measures of cohesion and coupling are defined as basic distances for
software clustering. Some measures, such as Jaccard-like ones [25, 30], indicate the
similarity between program elements, while others [29] indicate the dependencies
strength between them. Unfortunately, it costs too much time to calculate the Eu-
clidean distance. Finally, different types of clustering approaches are adopted, such as
hierarchical clustering [31, 32, 4, 33] and partition methods [34, 35].

Recent researches [36, 35, 37] tend to apply graph-theoretical approaches to
software architecture recovery. A variety of dependencies in programs are used to
generate graph representations, such as *Containment Relationship* [35] and *Use
Relationship* [38]. Several density-based approaches have been adopted to identify
software modules, such as *edge betweenness clustering* [35] and *Spectral graph
partition* [38]. However, none of those similarity-based approach are used for soft-
ware clustering. Since graph-based similarity measure is not considered in finding
software modules, similar elements that are not connected are hardly grouped
together. Along with the development of analyzing complex techniques [39, 7, 20],
new directed graph clustering and hierarchy approaches are applied to software architecture recovery.

7. Conclusion

In this paper, we have presented software architecture recovery with graph clustering, which focuses on two different measures: Connectivity and Similarity. In the proposed approach, directed graphs are adopted to represent software programs and calculate SimCorr and ConSco as the measures between program elements. Based on the measures, Graph Transformation is used to generate the Correlation Graph, with which similar program elements can be grouped through density-based clustering. Then DGHC has been proposed, which coordinates density-based clustering and similarity-based clustering with alternative strategy, and implement hierarchy clustering of software systems. Furthermore, multiple levels of software are detected with stable layers of hierarchy clustering, which is suggested by Multi-layer Propagation Gap. Finally experimental results on four real software systems demonstrate the effectiveness and efficiency of our approach in software architecture recovery.

Acknowledgments

This work is supported by the NSFC Key Grant 11020101060, IRG11221504 and 1005044, and the “973” Grant of MOST of China 2013CB837803. We thank the ALICE Collaboration and ALICE DAQ team for supporting this work. We would also like to thank Sylvain Chapeland for providing us with DAQ system’s source codes, documents and technical support.

References