An Experimental Study of Two Graph Analysis Based Component Capture Methods for Object-Oriented Systems

Jing Luo, Renkuan Jiang, Lu Zhang, Hong Mei, Jiasu Sun
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, P. R. China, 100871
{luoj, jiangrk, zhanglu, meih, sjs}@sei.pku.edu.cn

Abstract

The problem of how to partition a software system and thus capture its overall architecture and its constituent components has become a research focus in the community of software engineering. In the literature, many methods have been proposed for solving this problem. For example, both top-down and bottom-up methods based on analyzing the graph representation of software systems have been proposed. In this paper, we report an experimental study of a top-down method and a bottom-up method. In our study, we focus on the capability of component capture, the capability of architecture recovery and the time complexity for the two methods. According to our results on two real world systems, the studied bottom-up method is superior to the studied top-down method in both aspects, although the time complexity of the bottom-up method remains a big concern for large systems.

1. Introduction

With the increase of existing software systems, the maintenance and evolution of these systems have become a research focus in the software engineering community. An important and helpful step is to capture the architectures of these systems and/or candidate components in them, since components may be reused in system evolution and the architecture of a system can be used as a guidance to understand and maintain the system.

During the past a few years, quite a few methods (see e.g. [3], [10], [4], [12], [17], [8], [9], [6], [2], [11], [5] and [7]) have been brought forward for fulfilling this task. In general, we can classify those methods into two categories: knowledge matching methods (such as [4], [12] and [11]) and structure analysis methods (such as [3], [17], [8], [9], [6], [5] and [7]). The basis of the knowledge matching methods is some knowledge of the target components. All the elements in the source code that match with the knowledge are regarded as candidate components. A disadvantage of these methods is that it is usually quite difficult in practice to acquire and represent the knowledge and do the matching. In fact, all the existing methods somehow rely on maintainers' manual work for the matching. On the other hand, structure analysis methods abstract software systems as mathematical notations, such as a tree structure or a graph structure. Via analysis of these notations, a software system can be partitioned into sub-systems, and each sub-system is taken as a candidate component. Furthermore, the organization of these sub-systems for forming the entire system can be viewed as the architecture of the software system. As the graph structure can reserve much more information in the source code than the tree structure, most structure analysis methods use the graph analysis based approach.

In graph analysis based methods, there is a common assumption that well-designed software systems are organized with cohesive sub-systems that are loosely interconnected [9]. Based on this assumption, several top-down methods (e.g. software clustering [8] [9] [6] [5]) are proposed to partition the entire system into cohesive clusters. In these methods, the component capture problem is viewed as the problem of searching for the optimal partition. As this problem is NP-hard in general, some heuristics (such as genetic algorithms or hill climbing algorithms) are used to achieve a sub-optimal result. An implicit assumption is that there is no omnipresent common sub-system in the system, because such a sub-system can link most pieces of source code into one big sub-system [10]. In contrast, we have proposed a bottom-up method named the iterative analysis method in [7]. In this approach, components with small sizes are firstly identified and then larger components composed of small components are identified. A distinct feature of this approach is its capability of dealing with omnipresent common components.

In this paper, we report an experimental study of two graph analysis based methods (i.e. [5] and [7]) for capturing components in object-oriented systems featuring three basic criteria. The organization of the remainder of

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

this paper is as follows. In section 2, we briefly introduce the two methods studied in this paper. Section 3 presents the settings of the experimental study. In section 4, we present and discuss the results of experimental study. Finally, we conclude this paper and discuss some future work in section 5.

2. The Studied Methods

2.1. The Graph Clustering Method

In [5], a top-down method named graph clustering analysis is proposed to capture components. The process of this method is as follows. Firstly, the software system is abstracted as an undirected graph. Secondly, the weight of each edge can be calculated through an edge strength metric, which is based on the small world graph theory [15] [16]. Thirdly, the graph is divided into several sub-graphs after deleting the edges whose weight is smaller than a threshold. Finally, different values of the threshold are tested to acquire the best partition. Obviously, the central part of this method is the edge strength metric. According to [5], the definition of this metric is as follows.

Let $G(V, E)$ be the graph, for each $e(u, v) \in E$ be an edge and u, v be its endpoints, the neighbors of u and v can be partitioned into three sets: M_u denotes the vertices neighboring to u but not to v; M_v denotes the vertices neighboring to v but not to u; and W_{uv} denotes the vertices neighboring to both u and v. This situation is depicted in Fig. 1.

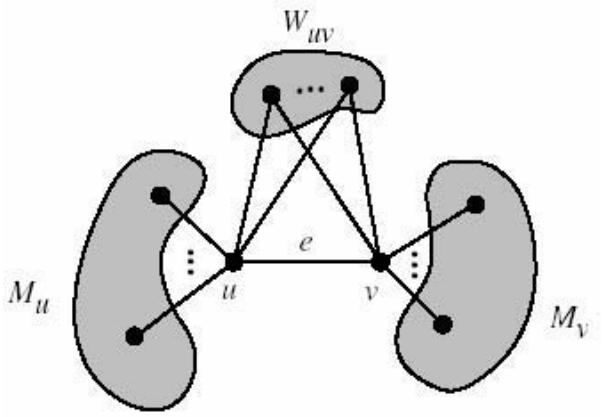


Fig. 1 Partition used to calculate the strength metric of an edge.

Based on the partition, two edge density metrics can be defined. The 3-cycle edge density and the 4-cycle edge density for edge $e(u, v) \in E$ are defined in (1) and (2) respectively.

$$\gamma 3(e) = |Wuv| / (|Wuv| + |Mv| + |Mv|) \quad (1)$$

$$\gamma 4(e) = s(Mu, Wuv) + s(Mv, Wuv) + s(Mu, Mv) + s(Wuv) \quad (2)$$

The strength metric for e is defined as the sum of the two densities:

$$\text{Strength}(e) = \gamma 3(e) + \gamma 4(e) \quad (3)$$

The functions $s(U, V)$ and $s(U)$ used in (2) are defined as follows. Let U and V be two subsets of vertices. The function $s(U, V)$ is defined as the ratio of the actual number of edges between the sets U and V with respect to the maximum number of possible edges between those two sets. Formally, the definition of $s(U, V)$ is represented in (4).

$$s(U,V) = \frac{e(U,V)}{|U||V|} \quad (4)$$

In (4), $e(U, V)$ denotes the number of edges connecting a vertex of U to a vertex in V. Furthermore, the function $s(U)$ is defined as $s(U, U)$.

2.2. The Graph Iterative Analysis Method

In [7], we have proposed a bottom-up method named graph iterative analysis. In this method, the process of capturing components is as follows. Firstly, the software system is abstracted as a directed weighted graph. Secondly, sub-graphs of the graph are iteratively examined and the most independent sub-graphs are selected as candidates. In the following, we present the system abstraction, the system decomposition and the independency metrics for this method respectively. Please refer to [7] for more details.

2.2.1. System Abstraction

In the graph iterative analysis method, an object-oriented software system is abstracted as a directed weighted graph. In this graph, each vertex represents a class, each edge represents a relationship between two classes, and the direction of an edge denotes the direction of the corresponding relationship. Formally, the directed weighted graph for an object-oriented software system is a tuple $G=(V, E)$, where V is the set of vertices, $E=\{<v1, v2, t>|v1, v2 \in V, t \in T\}$ is the set of edges, and $T=\{inheritance, aggregation, association, dependency\}$ is the set of types for edges. The concepts of the four types of relationships come from UML. An inheritance relationship means a class is a sub-class of another. An aggregation relationship means a class consists of another class. An association relationship means a class is explicitly linked with another class. A dependency relationship means a class relies on another class for its implementation. A weight function is used to assign edges of one type a same weight. In this paper, we use W_{in} to denote the weight for inheritance relationships, W_{ag} for aggregation relationships, W_{as} for association relationships and W_{de} for dependency relationships.

2.2.2. System Decomposition

In the graph iterative analysis method, the algorithm for decomposing an object-oriented system is depicted in Fig. 2.

Graph Iterative Analysis
Input: directed weighted graph ***G*** representing an objected-oriented software system; ***k*** as the threshold size of sub-graphs under checking; ***d*** as the independency threshold to choose the highly independent sub-graphs; ***s*** as size threshold for the final graph
Output: the set of **c**andidate sub-graphs ***Candidates***
Algorithm :

```
Candidates=Ø
While (sizeOf(G)≥s)
Begin
    NewCandidates=Ø
    subGraph=nextSubGraph(G, k)
    While(subGraph≠NULL)
    Begin
        IMv=calculateIM(subGraph)
        If(IMv>d)
        NewCandidates.add(subGraph)
    End
    G=constructNewGraph(G, newCandidates)
    Candidates=Candidates∪NewCandidates
End
```

Fig. 2 Algorithm for graph iterative analysis

In Fig. 2, there are two nested loops in the algorithm. In the outer loop, it is always checked whether the size of the graph has been shrinked under the threshold value ***s***. While the size is greater than **s**, its sub-graphs whose scale is under the threshold ***k*** are enumerated, and thus checked to select those highly independent ones. These highly independent sub-graphs are chosen as candidate components and will be shrinked into a vertex in the next round iteration. As the size of ***G*** in each time of iteration will always decrease, this algorithm can terminate in the end. When checking the independency of a sub-graph, an independency metric is used. The definition of this metric will be presented in section 2.2.3.

The functions used in the algorithm are defined as follows.

- **nextSubGraph (G, k)** is used to get the next connected sub-graph whose size is not more than ***k*** which is set to be 3 in our experiments. Fig. 3 depicts a typical example of enumerating three-vertex sub-graphs containing *v*. In Fig. 3(a), vertex *v* is connected with three other vertices. Then two kinds of sub-graphs containing *v* can be enumerated. Fig. 3(b) and Fig. 3(c) depict the two situations respectively.

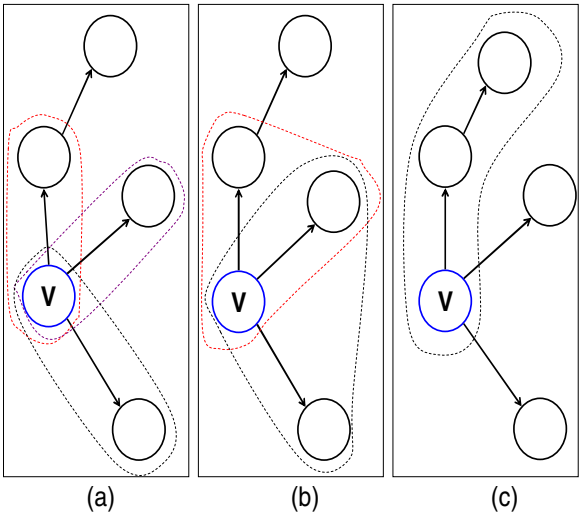


Fig. 3 An example of enumerating sub-graphs

- **calculateIM(subGraph)** is used to calculate independency value of subGraph according to the independency metrics defined in section 2.2.3.
- **constructNewGraph(G, newCandidates)** is used to shrink the candidate sub-graphs and form a new graph. The basic strategy is to shrink a candidate sub-graph into one vertex. If two candidate sub-graphs are over-lapping, the algorithm chooses the one of higher independency value. Fig. 4 shows an example of graph reconstruction.

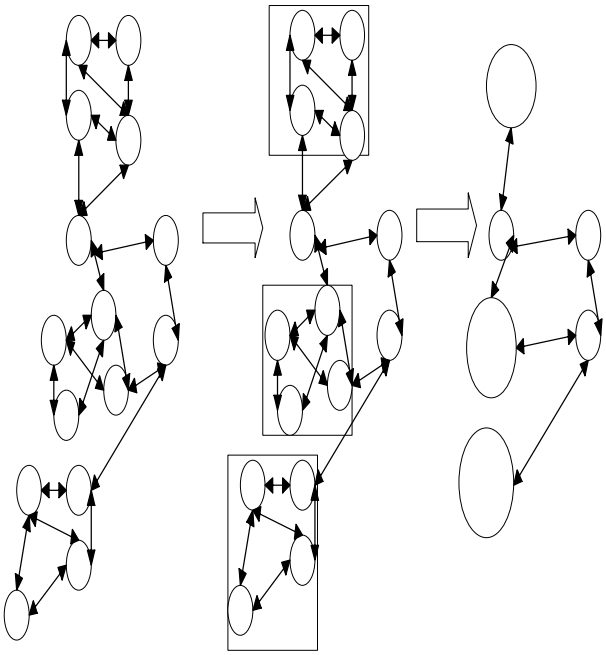


Fig. 4 An example of graph reconstruction

2.2.2. Independency Metrics

In the graph iterative analysis method, the independency metrics are derived from the metric for assessing the quality of system decomposition proposed in [9]. In this method, three factors are considered, which are depicted in Fig. 5. Among these factors, cohesion factor is positively related to the metric, while the other two factors are negatively related.

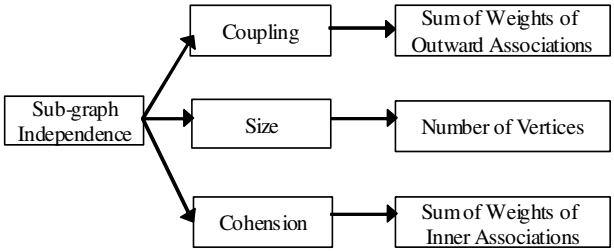


Fig. 5 Independency Metric Tree

For a given sub-graph, the sum of the weight of each edge is defined as the cohesion metric; the sum of the weight of each outgoing edge is defined as the coupling metric; and the number of vertices is defined as the size metric. Obviously, the direction is critical for calculating the coupling metric for directed graphs. Formally, let $G=(V, E)$ be the graph and $G'=(V', E')$ be a sub-graph, the set of outgoing edges is $O=\{<u, v, t> | <u, v, t> \in E, u \in V', v \in V-V'\}$. Thus the independency metric can be defined as (5). If the outgoing edge set O is empty, the sub-graph G' is viewed as having a very large independency value, denoted as *Max*.

$$IM(G', G) = \frac{\sum_{e \in E'} W(e)}{|V'| \sum_{e \in O} W(e)} \quad (5)$$

To avoid the inconvenience of dealing with empty set of outgoing edges, an alternative metric is defined in (6).

$$IM(G', G) = \frac{\sum_{e \in E'} W(e) - \sum_{e \in O} W(e)}{|V'|} \quad (6)$$

According to our experience, these two formulae have not much difference in the experiments. It should be noted that the independency metrics can not only be used for identifying independent sub-graphs as candidate components, but also can be used as a reusability metric of candidate components. Components having higher independency values are more reusable.

3. The Experiment

In order to compare the two component capture methods, we performed an experiment on two real world systems based on three basic criteria (see Section 3.2).

3.1. Experimented Systems

The first experimented system is a mixed encrypt/decrypt tool [13] which is obtained from a code-base website. This system is about 12k LOC and includes 17 classes. The system mainly implements the mixed encrypt-decrypt algorithm together with a well-designed user interface. Fig. 6 shows the graph representation of its class diagram, in which each class is denoted as a vertex.

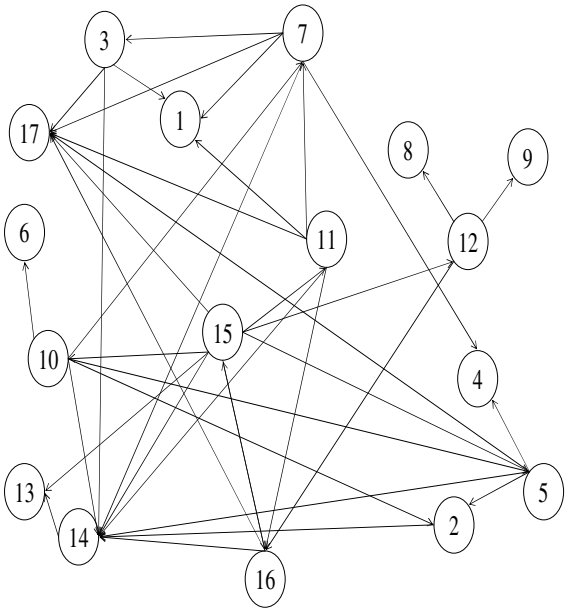


Fig. 6 Class diagram for the first experimented system

The second experimented system is an interface DLL [13] which is also obtained from a code-base website. This system is over 50k LOC and includes 85 classes. It provides facilities for implementing user interfaces like the MS Visual Studio. Interestingly, its class organization is a little disordered. We would like to see how the methods perform in this situation, which we think might be so for most real world systems. As this system is much larger than the first one, we will not show its class diagram.

3.3. Experimental Environment

The experimental environment is depicted in Fig. 7.

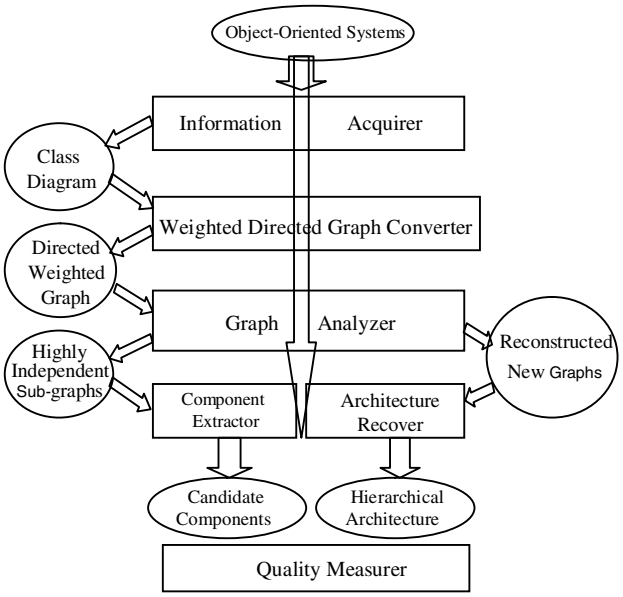


Fig. 7 Experimental environment

Fig. 7 depicts the main settings of the experiment, which include the following parts.

- Information Acquirer analyzes the software system and creates the class diagram featuring the four kinds of relationships.
- Weighted Directed Graph Converter converts the class diagram into a directed weighted graph which will be transformed into an undirected unweighted graph for the graph clustering analysis method in Graph Analyzer.
- Graph Analyzer implements the iterative analysis method and the graph clustering analysis method.
- Component Extractor and Architecture Recover capture the candidate component and recover hierarchy architecture from the software system.
- Quality Measurer implements metrics used to evaluate the component capture capability and the architecture recovery capability.

3.2. Evaluation Criteria

As we mentioned above, this experimental study is mainly aiming at three aspects of the two methods. The evaluation criteria for the three aspects are as follows.

3.2.1. Evaluation of Component Capture Capability

In order to evaluate the component capture capability, we manually examine the first system and acquire the components in it. This is compared with the components captured by the two studied methods. As we find the manually captured components mainly have higher independency values defined by the metric in (6), we use this metric to evaluate the components captured by the two methods from the second system, since this system is too large for manual examination.

Another criterion we use to evaluate the capability of component capture is the distribution of the sizes of the captured components. Usually, candidates with very large sizes or very small sizes cannot be viewed as good possible components. Thus, the ideal distribution is a normal distribution where quite many candidates have reasonable sizes.

3.2.2. Evaluation of Architecture Recovery Capability

To evaluate the architecture recovery capability for the two methods, we use the partition quality metric MQ proposed in [9], as this metric has been used in several studies (such as [5]) since its publication.

Let $C(G_1, G_2, G_3, \ldots G_n)$ be a partition of $G(V, E)$, the MQ metric is defined as below:

$$MQ(C,G) = \frac{\sum_{i=1}^{n} s(G_i,G_i)}{n} - \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} s(G_i,G_j)}{n(n-1)/2} \qquad (7)$$

The MQ value denoting the quality of a given partition ranges from -1 to 1, and a higher value means better quality. We will apply it to assess architectures recovered from the same software system by different methods.

3.2.3. Evaluation of Time Complexity

As the settings for the two methods in the experiment are the same, we simply record the execution time for the time complexities of the two methods.

4. Results and Analysis

4.1. Capability of Component Capture

Component Number	Class Number	Corresponding Functionality	IM
1	2 13 14	File input and output	25
2	2 4 5	Des Decrypt and Encrypt	15
3	8 9 12	Dialog of About	12. 3
4	1 3	Algorithm for Big Prime Number Generation	10
5	3 7 11	Interface For Big Prime Number Generation	9. 3

6	7 11 1 3 17	Big Prime Number Generation	6.5
7	6 10 2 4 5	Algorithm for mixed decrypt and encrypt	−10

Table 1 Result of manual component capture

For the first experimented system, we performed the component capture manually before we applied the two methods. The result of the manual component capture is depicted in Table 1, in which there are in total seven components identified manually. The independency metric value for each component is also listed. This shows that the independency metric can be viewed as a metric for evaluating the capability of component capture.

Number	Directed Weighted Iterative analysis	Graph clustering analysis
1	13 14	1 3 11 14 17 7 15 16 2 5 10
2	2 13 14	1 3 7 11 14 17 4 10 15 16 2 5 13 12
3	13 14 1 3 17	1 3 11 14 17 7 15 16 2 5 10 12
4	1 3	1 3 7 11 14 17 10 15 16 2 5 13 12
5	15 16 8 9 12 13 14 1 3 17 6 10 2 4 5 7 11	1 3
6	13 14 1 3 17 6 10 2 4 5 7 11	1 3 11 17 14 15 16 5 10 2
7	13 14 1 3 17 7 11	10 15
8	7 11 1 3 17	16 17
9	8 9 12	11 17 16 14
10	13 14 1 3 17 16 8 9 12 7 11	

Table 2 Comparison of component capture

The results of the two component capture methods on this system are depicted in Table 2. For the iterative analysis method, we list the 10 candidates with highest independency values. Among the 10 candidates, four candidates are among the manually captured components. For the graph clustering analysis method, several values of the threshold for removing weak edges have been tested to acquire altogether 9 best candidates. The 9 candidates are also sorted by the independency values. Among them, only one candidate is in the manual results. Obviously, the iterative analysis method can capture more components than the graph clustering method.

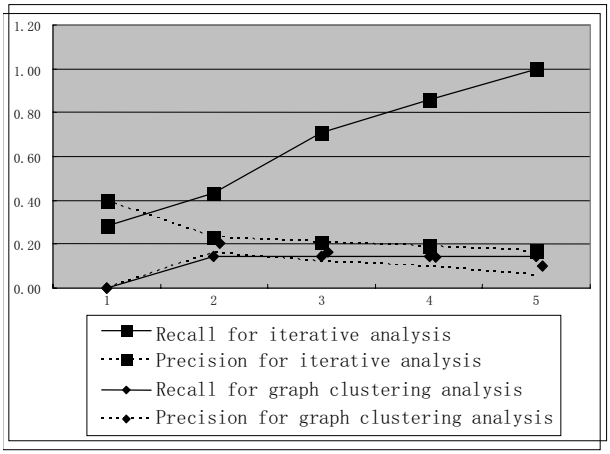


Fig. 8 Trends of precision and recall

Furthermore, we acquire the trends of the precision and recall for the two methods in Fig. 8. The variable for the graph clustering analysis method is the threshold for removing the weak edges. From this figure, we can see that both the precision and the recall for the graph clustering method are always much lower than those of the iterative analysis method.

As we cannot capture the components manually for the second system, we will use the independency metric and the component size distribution for evaluating the two methods. Before we analyze the results of the experiment on the second system, we apply the two criteria for the results on the first system. Fig. 9 depicts the comparison of the independency values for the candidates listed in Table 2. Obviously, the independency values of the candidates acquired by iterative analysis method are always higher than those of the graph clustering method.

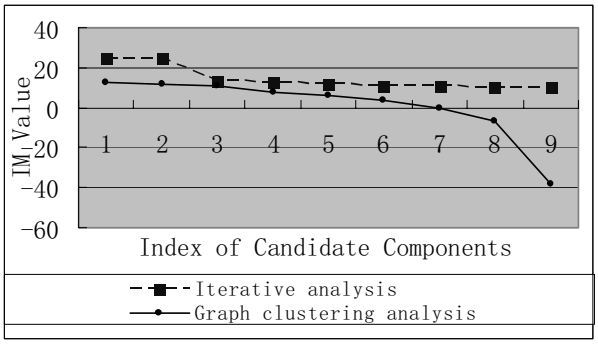


Fig. 9 Comparison of IM for the first system

The distribution of the sizes of the captured candidates for the two methods is depicted in Fig. 10. Although none of them can produce an ideal distribution, the distribution

for the iterative analysis is much nearer to be normal than the graph clustering analysis method. This also indicates than the iterative analysis method is superior to the graph clustering method with regard to the capability of component capture.

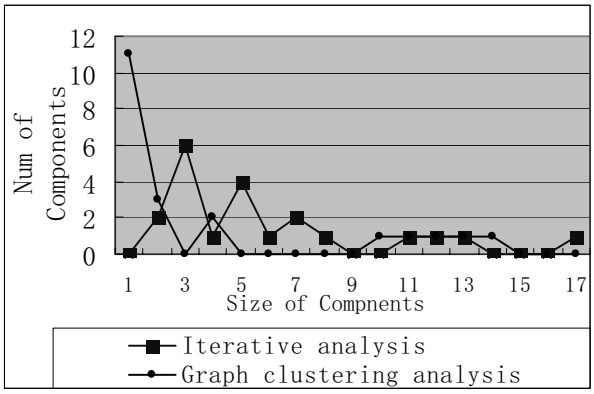


Fig. 10 Size distribution comparison for the first system

For the second experimented system, manual component capture result cannot be obtained because it is too large. The comparison of the independency values for the candidates acquired by the two methods is depicted in Fig. 11. The comparison of the size distribution is depicted in Fig. 12.

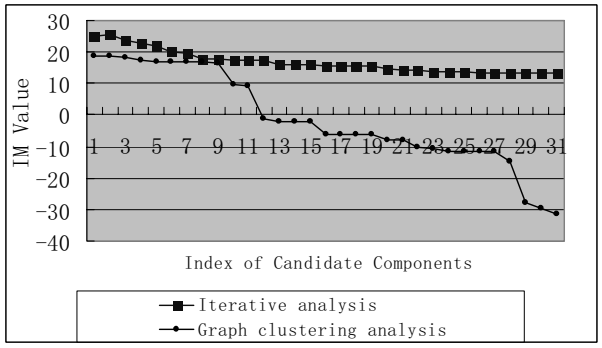


Fig. 11 Comparison of IM for the second system

In Fig. 11, once again the independency values of the candidates acquired by iterative analysis method are always higher than those of the graph clustering method. Similarly, in Fig. 12, the iterative analysis method outperforms the graph clustering analysis method on the size distribution.

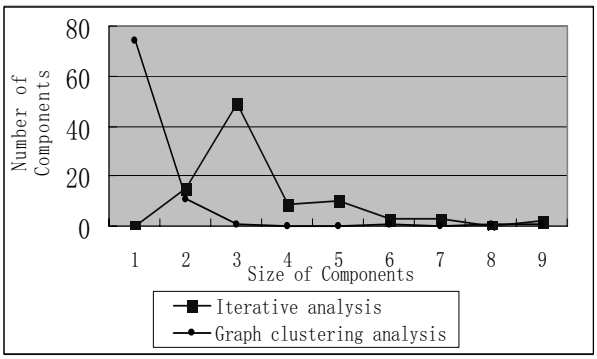


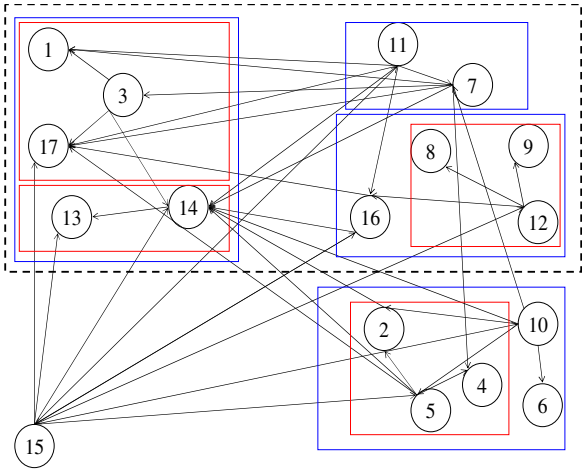
Fig. 12 Size distribution comparison for the second system

From the above results, we conclude that the iterative analysis method may have better capability of capturing components in object-oriented systems. We think the reason might be three-folds.

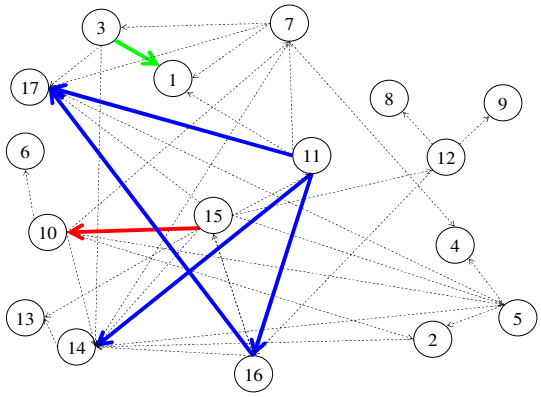
- In the iterative analysis method, more possible components are considered, while the graph clustering analysis method may ignore quite many good candidates.
- The directed weighted graph that the iterative analysis method uses as input better reflects the nature of object-oriented software systems, while the undirected graph used in clustering analysis may lose some important information.
- The iterative analysis method relies less on the assumptions discussed above. Thus it can effectively deal with more real world systems that do no exactly conform to the assumptions.

4.2. Capability of Architecture Recovery

Iterative analysis and graph clustering analysis can not only capture components but also give partition to the software system. We apply the MQ metric to measure the quality of each partition. As the first experimented system is rather small, we can present the architectures recovered by both methods. Fig. 13(a) shows the hierarchical architecture recovered by the iterative analysis method, while Fig. 13(b) shows the architecture corresponding to the partition achieved by the graph clustering analysis method. Usually the hierarchical architecture recovered by the iterative analysis method may be more helpful than the flat architecture recovered by the graph clustering analysis method. This difference is out of the scope of the MQ metric.



(a)



(b)

Fig. 13 Architectures recovered from the first system

For the first system, the MQ value for the architecture recovered by the iterative analysis method is 0.072, while that for the graph clustering analysis method is 0.034. The iterative analysis method can achieve better partition quality, but the difference is not much. For the second system, the MQ value for the architecture recovered by the iterative analysis method is 0.36, while that for the graph clustering analysis method is 0.031. The iterative analysis method can achieve much better partition quality than the graph clustering analysis method.

4.3. Time Complexity

Table 5 shows the running time for both component capture methods running on a 2.26GHz Pentium 4 PC with 256M memory. From this table, we can see the execution time of either method for either experimented systems is acceptable. However, the time complexity of the iterative analysis method remains a big concern, because both experimented systems are not very large. The execution time of this method for a system with more than a thousand classes may not also be acceptable.

Experimented System	Graph Clustering Analysis	Iterative Analysis
1	0.3	0.5
2	4.2	15

Table 5 Execution time (in seconds)

4.4. Discussion

In this section, we present a brief discussion of the main findings in this experimental study.

- **Component Capture**

For the component capture capability, the bottom-up approach is better than the top-down approach in terms of both precision and recall. However, the precision of neither approach is satisfactory. Furthermore, as the recall of the bottom-up approach is quite satisfactory, it is probable to be applied in real world systems together with human intervention.

- **Architecture Recovery**

The bottom-up approach seems more promising for recovering system architectures, as it can provide hierarchical structures and a better partition based on the MQ metric. However, real world experience with these approaches is essential to determine whether they are actually useful for this purpose.

- **Time Complexity**

The time complexity of the bottom-up approach is much higher than the top-down approach. This may be a strong hindrance for its application.

5. Conclusions and Future Work

Research on component capture and/or system partition for existing systems has become a focus in the community of software engineering. Many methods have been proposed for this problem. Among these them, most are based on analysis of the graph representation of object-oriented systems. Both top-down and bottom-up analysis methods have been proposed.

In this paper, we have empirically compared and evaluated two graph analysis based methods (representing the top-down approach and the bottom-up approach respectively) against two real world systems. According to our results and analysis, the tested bottom-up method outperform the tested top-down method in terms of the capability of component capture and the capability of architecture recovery, while the top-down method has less time complexity than the bottom-up method.

We think that there are quite a few interesting issues that need further investigation. First of all, we think the study reported in this paper is still rather preliminary. We would like to perform more experiments on more and larger real world systems involving more component capture methods. Secondly, the results of this study inspire us that a method combining bottom-up analysis and top-down analysis may have a better performance. We will pursue this in the future. Finally, there are still several issues hindering the adoption of component capture methods in real world practice. For example, the naming of the captured components and the visualization of the recovered architecture may also need further investigation. This may be our next research topic.

Acknowledgements

This effort is sponsored by the National 973 Key Basic Research and Development Program No. 2002CB31200003, the State 863 High-Tech Program No. 2001AA113070, and the National Science Foundation of China No. 60125206 and 60233010.

References

[1] J. Bansiya, "A Hierarchical Model for Quality Assessment Of Object-Oriented Designs," [Ph.D.Dissertation]. Huntsville: University of Alabama in Huntsville, 1997.

[2] K.S. Barber and T.J. Graser, "Tool support for systematic class identification in object-oriented software architectures," Proceedings of 37th International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS-Pacific), 20-23 Nov. 2000, 82-93

[3] L.A. Belady and C.J. Evangelisti, "System Partitioning and its Measure," Journal of Systems and Software, 2(1), pp. 23-29, February 1982.

[4] T. Biggerstaff, B. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," Proceedings of International Conference on Software Engineering, May 1993, 482-498.

[5] Y. Chiricota, F. Jourdan, and G. Melancon, "Software Components Capture using Graph Clustering," Proceedings of 11th IEEE International Workshop on Program Comprehension, 10-11 May 2003, 217-226.

[6] D. Doval, S. Mancoridis, and B.S. Mitchell, "Automatic Clustering of Software Systems using a Genetic Algorithm," Proceedings of 9th International Workshop on Software Technology and Engineering Practice, 1999, 30 Aug.-2 Sept. 1999, 73-81.

[7] J. Luo, W. Zhao, T. Qin, R. Jiang, L. Zhang, and J. Sun, "A Decomposition Method for Object-Oriented Systems Based on Iterative Analysis of the Directed Weighted Graph," Journal of Software (in Chinese) (to appear).

[8] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," Proceedings of 6th International Workshop on Program Comprehension (IWPC '98), 24-26 June 1998, 45-52.

[9] S. Mancoridis, B. Mitchell, Y. Chen, and E. R.Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," Proceedings of International Conference on Software Maintenance (ICSM'99), August 1999, 50-62.

[10] H. Muller, M. Orgun, S. Tilley, and J. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification," Journal of Software Maintenance: Research and Practice, vol. 5, 1993, 181–204.

[11] M. Pinzger and H. Gall, "Pattern-Supported Architecture Recovery," Proceedings of 10th International Workshop on Program Comprehension, June 27-29, 2002, 53 –61.

[12] M. Siff and T.Reps, "Identifying Modules via Concept Analysis," Proceedings of the International Conference on Software Maintenance, October 1997, 170-179.

[13] VC Code Base, Mixed Encrypt-Decrypt System [http://www.vckbase.com/code/downcode.asp?id=1781] (in Chinese)

[14] VC Code Base, BCGControlBar Pro 6.21 [http://www.vckbase.com/tools/viewtools.asp?id=76] (in Chinese)

[15] D.J. Watts and S. H. Strogatz, "Collective Dynamics of Small-World Networks," Nature, vol. 393, 1998, 440–442.

[16] D.J. Watts, Small World. Princeton University Press, 1999.

[17] T.A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization," Proceedings of the Fourth Working Conference on Reverse Engineering, 6-8 Oct. 1997, 33-43.