МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2014
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2014
Proceedings of the Forty Third Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 2–6, 2014

A COMPOUND ALGORITHM FOR CLUSTERING SOFTWARE*

Todor Cholakov, Dimitar Birov

The support of a software product requires knowledge of the design decisions that have been taken and its software architecture. Usually this knowledge is missing, is too outdated, or it doesn't correspond to the source code of the software product. An important step towards the understanding of such products is to extract knowledge about the software architecture of the product, using different artifacts and mainly the available source code. The software clustering algorithms aim to automate this process to the best possible extent, giving the developers, designers and architects a good partitioning of the source code into modules. In this paper we discuss several algorithms for software clustering which implement different approaches to the problem and we propose a new one. Our purpose is to create a better partitioning algorithm in terms of both the quality of the resulting partitioning and speed.

1. Introduction. Software support is a part of the process of software evolution. Availability of respectable documentation is essential for developers to get acquainted with the design, architecture and product requirements. Usually in practice, the documentation is outdated or irrelevant to the source code. An essential step for good research in order to understand such software is to extract knowledge about the concerns, design concepts and software architecture out of various artifacts of the product and mainly the source code.

The location of the software elements amongst the file system does not give relevant information about the connections and interactions between the software units, modules and components, but it is an appropriate initial step for starting an automation analysis. It is difficult to make a conclusion about the ideas and design decisions that were applied in the development process. It is therefore necessary to automate the process of analyzing, understanding, recognition, and as a consequence, modification of the source code. This would be a great assistance towards the work of the software engineers, developers, designers, architects and so on, bringing better efficiency into the process of software support, evolution and development.

In the process of automated analysis and knowledge extraction from the program code an essential artifact is the software architecture. The usual practice (without automated

*2010 Mathematics Subject Classification: 68N19.
Key words: call graphs, reference graphs, reengineering, component recognition, clustering
The authors gratefully acknowledge financial support by the Bulgarian National Science Fund within project DO 02-102/23.04.2009.

instruments) is for some developer to explore and examine some fragments (snippets) of the code which is to be modified. It is possible for the significant areas of the code to be overlooked or misunderstood and as a result additional bugs can be introduced during the process of source code modification. In practice this process is accompanied by pressure of delivery deadlines. Without good tools, the process of modification substantially depends on the personal skills of the developers. And this is where the tools for automated analysis get applied – they prevent errors due to misunderstanding of the code and decrease the importance of the understanding skills of the supporting developer.

Automating the process of comprehension of the software code requires algorithms for automated component recognition to be created. These algorithms should use the dependencies and connections between the code fragments, as well as common coding and style patterns. It is suitable to represent the software code as a graph of interconnected elements. The nodes of the graph are pieces of code (depending on the required granularity these may be operators, methods, classes, components or bigger modules) and the edges represent the interconnections between them. A clustering algorithm is applied on the resulting graph, which groups its nodes into clusters. The nodes in each cluster are syntactically, semantically or logically connected and represent software entities.

In this paper we consider several algorithms for clustering of source code and we propose a new algorithm for clustering that aims to prevent some of their shortcomings and drawbacks. The proposed algorithm relies to some extent on knowledge about the structural principles of software code, and thus improves both the speed and the resulting partition, without neglecting the traditional metrics for determining the quality of the partition.

The paper is structured as follows: Section 2 presents a taxonomy of the types of algorithms used for clustering source code. Sections 3, 4 and 5 deal with three algorithms for clustering. In Section 6, we present our proposal for such a clustering algorithm.

2. Taxonomy of the algorithms for software clustering. The existing algorithms for clustering source code may be classified according to the following criteria:

1. The kind of graph that is used;
2. The approach used for the algorithm implementation;
3. The number of the executed iterations.

Regarding the kind of graph used we classify the algorithms according to which code constructs create links between the graph nodes. Depending on the chosen graph, we may get different partitions of the code into components. The algorithms for clustering source code use mainly the following graphs for code representation:

1. *Call graph.* In this graph only direct calls between the methods of the classes create links between the nodes. On the one hand this allows better granularity of the graph going to method level. On the other hand the links that are created in this way are only a small part of the possible links.
2. *Reference graph.* The reference graph takes into account all possible kinds of connections between the classes. These include method calls, inheritance, referencing variables of the other class, variable declarations, casting and using static fields of the other class.

Regarding the approach used, the algorithms are classified depending on the starting partition of the code parts into clusters and depending on the number of the iterations that are executed until the final result is achieved [6].

The following algorithms are known according to initial partitioning:

1. *Bottom up.* These algorithms start from an initial partition where each node is in its own cluster and according to a pre-specified criterion they merge into larger clusters.
2. *Top down.* These algorithms start from a partition where all nodes belong to a single large cluster. At each step a single cluster is divided into two smaller clusters or two clusters are merged together. The goal is to optimize the partition according to specific connectivity criteria.
3. *Random partition.* These algorithms start from a random partition and at each step a cluster is divided into two clusters or two clusters unite. Usually these algorithms aim to optimize some connectivity criteria.

The following algorithms are known depending on the number of the executed iterations:

1. *Iterative.* These algorithms alter the partition until a pre-specified condition is met. In most cases this is an optimization criterion or limited condition, which indicates that the resulting partition is of good quality.
2. *Straight forward.* These algorithms partition the graph by executing only one pass through its nodes.

3. Bunch's hill-climbing algorithm. Mancoridis et al. [1] represent a hill-climbing algorithm based on the idea of searching a suitable partition amongst the space of possible partitions of the graph. This is an iterative algorithm starting from a random partition, which may take various graphs as input. The algorithm uses a module dependency graph as an input. Usually this is a reference graph, but a call graph may be used as well.

The algorithm starts from a group of randomly selected partitions of the graph and at each step one node is moved between the clusters (this includes the node forming a new cluster on its own) trying to improve the quality criteria. In order to avoid reaching a local extreme value of the quality criteria, which is far from the optimal, the algorithm starts from several random partitions. Each of them is improved in several ways and only the best of them are chosen. The process continues until the maximum of the population is reached (the number is a parameter of the algorithm). After reaching the maximum, the population gradually converges by choosing only the best partitions until a single best partition is chosen. The described algorithm does not guarantee that the optimal partition is achieved, but usually the result is close to the optimal one.

The criterion that this algorithm uses to estimate the quality of a partition is:

$$MQ = \sum_{i=1}^{k} CF_i \quad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \text{otherwise} \end{cases} \tag{1}$$

Where μ_i are the edges which are inside the ith component, while $\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ are the arcs starting from the ith to the jth component and from the jth to the ith component, respectively. This criterion is used to evaluate the potential effect of moving a node between the clusters of the partition and is calculated at each step for all possible movements of nodes.

This algorithm shows good results and although the starting partitions may vary, the results possess similar quality. Based on multiple executions of the algorithm with the

same data, the authors have developed a method for discovering the optimal partition, but overlapping the results achieved by the different runs [2].

4. A multi-objective approach for software module clustering. This algorithm is very similar to the previous one. Again the algorithm is iterative, starts from a random partition, and is able to process various graphs as input. The main difference between them is the way in which the partitions are estimated. While Mancoridis et al. estimate the partitions according the MQ function (1), the authors of the multi-objective approach [3] propose to search for an optimal partition according several functions simultaneously. Using this method it is impossible to tell by how much a given partition is better than another one, and the algorithm just states if one of them is better than the other. For this purpose, for each of the created partitions all of the estimation functions are calculated.

In order to compare two partitions the values of the corresponding functions are compared. The second partition is considered better than the first if the value of each of the estimation functions for the second is greater or equal to the corresponding value for the first partition and at least one value is strictly greater.

The authors of the multi-objective algorithm propose two different approaches according the chosen set of estimation functions which they use:

1. The maximum cluster approach (MCA) includes the following functions:
    a. The sum of inter-edges (coupling, minimized).
    b. The sum of intra-edges (cohesion, maximized).
    c. The number of clusters (maximized).
    d. MQ (maximized).
    e. The number of isolated clusters (minimized).
2. The equal cluster size approach (ECA) is based on:
    a. The sum of inter-edges (minimized).
    b. The sum of intra-edges (maximized).
    c. The number of clusters (maximized).
    d. MQ (maximized).
    e. The difference between the number of nodes of the smallest and largest clusters (minimized).

The difference between the two approaches is that MCA tries to achieve a partition where as few nodes as possible are left without a cluster, while ECA aims to create equally sized clusters.

The proposed algorithm is genetic [4] – it starts from a population of random partitions and at each step they mutate and recombine. The best partitions according to the chosen set of estimation functions form the population for the next step. The algorithm finishes after a pre-specified number of steps are executed. Generally this algorithm performs better than the hill-climbing algorithm.

5. ACDC – An Algorithm for Comprehension Driven Clustering. The ACDC [5] algorithm is based on a set of measurable patterns, which are often used while writing source code. Using this set the authors create a frame for the partition and the nodes which are still not partitioned are orphaned by the most suitable cluster in the frame. This algorithm uses the bottom-up approach and is straightforward. The algorithm is designed to use a call graph.

An interesting feature of this algorithm is that the discovered modules are given suit-

able names, which leads to significantly better understanding of the resulting partition.

During the development of their algorithm the authors take into account the following patterns:

1. *The source files pattern*—the files are considered atomic entities and the code inside a given file belongs to the same subsystem (module).
2. *Folder structure pattern*—the files residing in the same folder are considered as belonging to the same subsystem.
3. *The body-header pattern*—in some programming languages such as C the implementation of a method is and its declaration are located in separate file and the two files have equal names, but different extensions.
4. *Leaf collection pattern*—this pattern is often observed in software systems, where a set of nodes are not connected to each other, but serve similar purposes. These nodes are usually leaves in the program's graph (they have an out-degree of zero).
5. *Support library pattern*—this pattern groups together those nodes of the graph that are too widely accessed. Usually these nodes represent some kind of library functions for the whole product and may be grouped together. These nodes usually have a bad influence on whatever clustering algorithm is used and may cause most of the nodes of the graph to be united in one or two very large clusters.
6. *The dispatcher pattern*—this is the opposite pattern of the support library. The dispatchers are nodes that refer to too many parts of the code. Their influence on the clustering algorithm also may cause the appearance of several very large modules and that is why they are excluded at the first step of the algorithm.
7. *Subgraph dominator pattern*—a node t is said to dominate a part of the graph if it divides the graph into two parts P and N, in such a way that for each route S from $p \in P$ to $n \in N$, $t \in S$ and for each $n \in N$ there exists a route from t to n. Essentially this means that each route to the elements of N must pass through t. In this case t and N form a module.

The algorithm itself consists of two stages:

1. The first stage of the algorithm creates a frame of the partition by searching for patterns in the following order:
   a. Each file is considered a separate subsystem named by the name of the file itself.
   b. Body-header – each couple of files according to this pattern form a subsystem named by the common name of the two files and an extension "ss".
   c. Leaf collection and support library – all leaves of the graph are collected in a separate list. Nodes are considered candidates for the support library if they have an in-degree of 20 or more. No subsystems are created at this step.
   d. Search for dominators. This is the main step of this algorithm. In the preparation stage for this step all nodes of the graph having an out-degree of 20 (empirically selected constant) or more are separated, thus eliminating the dispatchers. After that the nodes are processed searching for dominators, beginning with the nodes having the smallest out-degree. The purpose is to find clusters containing as few nodes as possible. Each of the formed clusters is named after the name of its dominator and an extension of "ss". The result is a hierarchical structure of subsystems. At the end of the step the nodes separated as possible dispatchers are considered and if they conform to the

dominator pattern, they are inserted in the skeleton.

e. All nodes separated in point c. as candidates for the support library are included in a module called "support.ss", unless they were added to another subsystem already.

2. On the second stage of the algorithm, all nodes of the graph which still don't belong to some subsystem are orphaned by the most suitable existing subsystem.

This algorithm has the following advantages:

1. Each cluster has a name, corresponding to some extent to its functionality.
2. Each cluster separates some common functionality, because it is formed based on one or more control flows.
3. The algorithm doesn't try to optimize some metric, but rather to follow the structure and the flow of execution of the program, even at the price of a worse resulting ratio between cohesion and coupling.

6. Compound algorithm for software clustering. The algorithm that we present here combines the strengths of the aforementioned algorithms, by taking into account some often-used coding patterns and also some of the widely used metrics for estimating the quality of a software partition.

In the algorithm we use a weighted reference graph. This allows all of method calls, references to variables and class inheritance to imply links between the nodes in the graph. The algorithm uses the bottom-up approach and is straight-forward.

The algorithm accepts the minimum degree of connectivity ε that two nodes must have in order to be considered belonging to the same cluster (module) and a flag defining the behavior of the algorithm if at the end there are nodes that do not belong to any cluster.

The algorithm consists of the following steps:

1. All the nodes having too high an in-degree are removed from the graph. The criteria here is that at least 10% of the remaining nodes, but no fewer than 20 (the same constant as the one used by ACDC), refer to them. In this way all the nodes corresponding to potential system libraries are discarded and their influence on the clustering algorithm is eliminated. In the same time library classes used exclusively by a module or group of modules are left intact, which improves the recognition of such modules. For example the java.lang.System class will be recognized as a global library and removed at this step, but javax.swing.JFrame would be considered a local library used only by the UI module.
2. A weight of 1 is assigned to all arcs ending in nodes that have an in-degree larger than 20 and which are not removed at the previous step. In this way the influence of the library nodes on the forming of the clusters is reduced, but not entirely eliminated.
3. The nodes having an out-degree of more than 20 are removed from the graph.
4. The resulting graph is transformed in the following manner: for each two nodes a metrics for the degree of connectivity is calculated using the following formula:

$$f(x,y) = m(w(x,y), w(y,x)) + co1(x,y) + co2(x,y) + rf(x,y) \tag{2}$$

Where:

- $w(x,y)$ is the number of direct links from x to y (the weight of the arc from x to y)

- $m(w(x,y), w(y,x) = \begin{cases} \frac{w(x,y)}{2}, & \text{when } w(y,x) = 0 \\ \frac{w(y,x)}{2}, & \text{when } w(x,y) = 0 \\ w(x,y)w(y,x), & \text{otherwise} \end{cases}$
- $co1(x,y) = \frac{|out(x) \cap out(y)|^2}{|out(x)| + |out(y)|}$, where $out(x)$ is the set of all nodes referred by node x. This function adds connectivity between two nodes when the nodes referred by both of them are a relatively large part of all referred nodes.
- $co2(x,y) = \frac{|in(x) \cap in(y)|^2}{|in(x)| + |in(y)|}$, where $in(x)$ is the set of all nodes that refer the node x. This function adds connectivity when the set of nodes that refer both x and y forms a large part of the set of nodes that refer any of them.
- $rf(x,y) = \frac{\sum_{t \in (in(x) \cup x)} w(t,y)}{|in(x)+1|} + \frac{\sum_{t \in (in(y) \cup y)} w(t,x)}{|in(y)+1|}$
  This function adds connectivity between the two nodes when the average connectivity from one of the nodes and its neighbors to the other node is relatively high.

5. The nodes of the resulting graph are sorted in decreasing degree of connectivity which they have to any other node.
6. For each node in the list the following steps are performed:
   a. If the maximum degree of connectivity is more than ε and the node to which it is realized belongs to a cluster then the current node is attached to the same cluster.
   b. If the maximum degree of connectivity is more than ε, but the node for which it is realized doesn't belong to any cluster, a new cluster is created having the current node as its only element.
   c. If the maximum degree of connectivity is less than ε, the processing of nodes is stopped.
7. A new graph is created in the following manner:
   a. The nodes of the original graph that were removed at step 1. become nodes of the new graph.
   b. The nodes of the original graph that were removed at step 3. become nodes of the new graph.
   c. The elements of the list in step 6. which are not parts of a cluster become nodes of the new graph.
   d. All the created clusters become nodes in the new graph.
   e. There is an arc from node x to node y having weight $wt(x,y)$, if $wt(x,y)$¿0, where
   (3) $$wt(x,y) = \sum_{a_i \in x, b_j \in y} w(a_i, b_j)$$
8. The new graph is transformed by calculating the degree of connectivity metric in the same way as in step 4.
9. The nodes of the resulting graph that are not clusters are sorted in decreasing order by the maximum degree of connectivity to any other node.
10. For each element of the resulting list the following steps are performed:

a. If the maximum degree of connectivity is greater than ε, but the node to which it is realized does not belong to a cluster, a new cluster is created having the current node as its only element.
b. If the current node has been removed at step 3 and has a degree of connectivity greater than ε to two or more clusters, but the total number of elements in these clusters is less than 20, then these clusters are merged and the current node is added to the resulting cluster. If the total number of elements is 20 or more, the current node is added to a new cluster.
c. If the current node has been removed at step 3 and has a degree of connectivity greater than ε to exactly one cluster, the current node is added to that cluster.
d. If the maximum degree of connectivity is greater than ε and the node to which it is realized belongs to a cluster, but is not a cluster itself, the current node is added to that cluster.
e. If the maximum degree of connectivity is greater than ε but the node to which it is realized is a cluster, the following cases are considered:
    i. The current node has been removed at step 1 as a candidate for a library module. In this case we check if there are other clusters to which the node has a connectivity greater than ε. If there are no such clusters, the node is added to the referred cluster. Otherwise if the node has a degree of connectivity more than ε to any other node in the list, it is added to a new cluster.
    ii. The current node is one of the nodes that didn't belong to a cluster after step 6. In this case the node is added to the referred cluster.
f. If the maximum degree of connectivity is less than ε, the processing is stopped.

11. The nodes removed at step 1. which still don't belong to a cluster form a cluster named "Utilities".
12. The nodes that still don't belong to a cluster and have a maximum degree of connectivity 0 form a cluster named "Obsolete".
13. Depending on a parameter passed to the algorithm, the nodes that still don't belong to a cluster:
    a. are added to the most suitable cluster or
    b. are added to a new cluster named "Weak".
14. Each cluster that still has no name is assigned a name which is the name of the node that has the biggest number of inbound links from outside the cluster.

The complexity of the algorithm may be calculated depending of the number of nodes (n) or the number of edges (v) as follows:

Steps 1, 2, 3, 7, 10 and 14 have a complexity of $O(n^2)$ or $O(v)$, because they iterate twice on the nodes or once on the edges.

Step 4 has a complexity of $O(n^2)$ or $O(v)$. It is not $O(n^3)$, because at step 3 we ensured that there are no more than 20 nodes starting from each node.

Steps 6, 11 and 12 has a complexity of $O(n)$ and unknown complexity regarding v as they are not using the edges in any way. In real life we may assume that $n < v < n^2$, so the complexity would be $O(v)$, too.

Steps 5 and 9 have a complexity of $O(n \log(n))$ as we are using the quicksort algorithm. Taking the previous assumptions this is not more than $O(v \log(v))$.

Step 8 has a complexity of $O(n^3)$ or $O(v)$ in the worst case, although usually this step

should be much faster than 4 because most of the elements are already grouped.

Step 13 is either with complexity $O(n)$ or $O(n^2)$ depending on the parameter. The complexity is definitely less than $O(v)$.

As a result the algorithm has a complexity of $O(n^3)$ or $O(v \log(v))$ in the worst scenarios. In the average case the complexity would fall to $O(n^2)$ and $O(v)$.

Bunch's hill-climbing algorithm and the multi-objective approach have a complexity of at least $O(n^2)$ or $O(v)$ (coming from the MQ formula), but they may need to iterate many times till reaching an optimal solution which may reach n or more, and finding the node to be moved between the clusters at each step, requires linear complexity, too. The ACDC algorithm is very similar to ours as we have used some of its elements and should have the same complexity $O(n^3)$.

The presented algorithm has the following advantages:

1. Compared to Bunch's hill-climbing algorithm and the multi-objective approach, in this algorithm the knowledge that the graph passed as input represents a software product is used to a greater extent.
2. This algorithm has a fixed number of steps before achieving the result. For large graphs this number is significantly less than the number of steps needed by the iterative algorithms.
3. As we don't rely on concrete patterns such as ACDC, we have the ability to discover clusters having more than one entry point.
4. The criterion for the candidates for the library module is a percent of the total number of nodes in the graph instead of a fixed number. In this way for large products, files that are widely used by a single module stay as part of that module.
5. The candidates for the library module that are referred by only one of the constructed modules become part of that module.

7. Conclusion. In this paper we present an algorithm for clustering a source code that combines the knowledge that the processed graph represents a software product and the similarity metrics, which makes it possible for nodes that do not conform to a specific code writing pattern, but have much in common, to belong to the same cluster.

REFERENCES

[1] S. Mancoridis, B. Mitchell, Y. Chen, E. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In: Proceedings of the International Conference on Software Maintenance (ICSM '99), IEEE Computer Society Press, Oxford, UK, August 1999.

[2] B. S. Mitchell, S. Mancoridis. Craft: a framework for evaluating software clustering results in the absence of benchmark decompositions. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001), 93–102, Suttgart, Germany, October 2001.

[3] K. Praditwong, M. Harman, Xin Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, **37**, *2* (March–April 2011) 264–282.

[4] D. Doval, S. Mancoridis, B. S. Mitchell. Automatic Clustering of Software Systems Using a Genetic Algorithm. Proc. Intl Conf. Software Tools and Eng. Practice, August–September 1999.
[5] V. Tzerpos, R. C. Holt. ACDC: an algorithm for comprehension driven clustering, in: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), 2000, 258–267.
[6] M. Shtern, V. Tzerpos. Clustering Methodologies for Software Engineering. *Advances in Software Engineering*, **2012** (2012), Article ID 792024, 18 pp.

Todor Plamenov Cholakov
e-mail: todortk@abv.bg
Dimitar Yordanov Birov
e-mail: birov@fmi.uni-sofia.bg
Faculty of Mathematics and Informatics
University of Sofia
5, James Bourchier Blvd
1164 Sofia, Bulgaria

КОМБИНИРАН АЛГОРИТЪМ ЗА КЛЪСТЕРИЗАЦИЯ НА ПРОГРАМЕН КОД

Тодор Чолаков, Димитър Биров

Поддръжката на един софтуерен продукт изисква знания за неговата архитектура. Често на практика те липсват или са твърде остарели, неточни и не съответстват на програмния код. Основна стъпка за доброто изследване с цел разбиране на такъв софтуер е извличане на знания за архитектурата от различни артефакти на продукта като основен източник е програмният код. Алгоритмите за клъстеризация на софтуер имат за цел да автоматизират този процес, като дадат на софтуерните специалисти – разработчици, дизайнери, архитекти и др. подходящо разделяне на програмният код на отделни компоненти. В настоящата статия правим сравнение между отделни алгоритми за клъстеризация, ползващи различни подходи и предлагаме нов подход и разработка на нов такъв алгоритъм. Нашата цел е да създадем алгоритъм, който е по-добър както по отношение на резултатното разпределение, така и в отношение на скоростта.