Software Components Capture using Graph Clustering
Yves Chiricota, Fabien Jourdan, Guy Melançon

To cite this version:

HAL Id: lirmm-00191947
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00191947
Submitted on 26 Nov 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Software component capture using graph clustering

Yves Chiricota
Département d’informatique et mathématique
Université du Québec à Chicoutimi
555, boul. de l’Université
Chicoutimi (Qc), Canada, G7H 2B1
Yves.Chiricota@uqac.ca

Fabien Jourdan, Guy Melançon
LIRMM
161, rue Ada,
34392 Montpellier Cedex 5 France
fjourdan@lirmm.fr, Guy.Melancon@lirmm.fr

Abstract

We describe a simple, fast computing and easy to implement method for finding relatively good clusterings of software systems. Our method relies on the ability to compute the strength of an edge in a graph by applying a straightforward metric defined in terms of the neighborhoods of its end vertices. The metric is used to identify the weak edges of the graph, which are momentarily deleted to break it into several components. We study the quality metric $MQ$ introduced in [1] and exhibit mathematical properties that make it a good measure for clustering quality. Letting the threshold weakness of edges vary defines a path, i.e. a sequence of clusterings in the solution space (of all possible clustering of the graph). This path is described in terms of a curve linking $MQ$ to the weakness of the edges in the graph.

1 Introduction

The reverse engineering community has devoted much effort recently in designing techniques to help capture the structure of existing software systems or API [1, 2, 3, 4] (see also [5] for a list of references). The basic assumption motivating this research is that well-designed software systems are organized into cohesive subsystems that are loosely interconnected (the italicized words are borrowed from [6]). Most efforts aim at finding the natural cluster structure of software systems. That is, they offer techniques able to divide any given system into sub-components that relate to each other either from a logical or physical design point of view.

A common and popular approach is to define a metric measuring the cohesiveness of the different components of a clustering, with the implicit assumption that the metric is able to find the “best” of any two clusterings. The original problem can then be turned into an optimization problem, relying on various heuristics such as Hill Climbing or Genetic Algorithms to help find a satisfiable solution. Several different metrics have already been suggested by different authors, each attempting to capture what is meant by a good clustering (see [5] for references). Koschke and Eisenbarth [5] moreover defined several orderings on software components to help compare two distinct clustering of a same system. Mancoridis et al. defined the metric $MQ$ on a clustering capable of measuring its quality in absolute terms [1].

The problem of capturing the structure of a software system can be formulated in graph theoretical terms. Incidentally, the problem of finding a “best” cluster structure for a graph (with respect to a given criterion) is covered by a wide spectrum of the mathematical literature (see [7] for an exhaustive survey). The criterion is often turned into a target function of which one has to find a minimum. One popular instance of this problem is the so-called min-cut problem consisting in finding a clustering made of several distinct subsets or blocks $C_1, \ldots, C_p$ (covering the original set of
vertices) such that the number of edges connecting nodes of distinct blocks is kept to a minimum. A cut is thus given by the set of edges cutting through distinct blocks of the clustering. The problem then is to find a cut with minimal weight. Depending on the application domain, one may further require that the cut has a specified number of blocks. One difficulty with this type of approaches is that their theoretical complexity exclude any algorithmic and deterministic solution. Moreover, the heuristics used to “optimize” the target function usually have rather high computational cost. Indeed, genetic algorithms can sometimes take minutes to output a clustering of a small scale graph (~1000 vertices). A general strategy aiming at the improvement of such heuristics is to try to understand the structure of the solution space they have to explore, as well as the mathematical properties of the target function they optimize. This knowledge can then be used in several ways. Indeed, it can be used either to better evaluate the behavior of the algorithm implementing the heuristic, or to improve the behavior of the algorithm by suggesting “paths” to follow in the solution space.

In this paper, we exhibit a simple, fast computing and easy to implement method for finding relatively good clusterings of software systems. Our method relies on the ability to compute the strength of an edge in a graph by applying a straightforward metric defined in terms of the neighborhoods of its end vertices. The metric is used to identify the weak edges of the graph, which are momentarily deleted to break it into several components. The method is fully described in section 2. Section 3 reports the result of our approach when applied to some known software systems. Next, in section 4, we concentrate on the target function $M_Q$ introduced in [1] and see it as an indicator of the quality of a clustering. The metric $M_Q$ can be seen as a reformulation of the min-cut problem, with the particular property however that a high value corresponds to a cut with low weight. The metric $M_Q$ actually possesses several mathematical properties that make it a good measure for clustering quality.

Also, letting the threshold weakness of edges vary defines a path or sequence of clusterings in the solution space (of all possible clustering of the graph). This path is described in terms of a curve linking $M_Q$ to the weakness of the edges in the graph (section 4.1). Perspectives and future work are discussed at the end.

## 2 Clustering metrics

In this paper, we describe a clustering technique based on the calculation of metrics on the edges of a graph $G = (V, E)$, that is a map $\phi : E \to \mathbb{R}$ assigning a real number $\phi(e) \geq 0$ to each edge $e \in E$. Assume the metric $\phi$ takes its value in the interval $[a, b]$ and fix a threshold value $t \in [a, b]$. We then define a graph $G'$ obtained from $G$ by removing any edge $e$ whose value $\phi(e)$ is below the threshold $t$. The subsets $C_1, \ldots, C_p$ corresponding to the connected components of $G'$ define clusters of $G$. Hence, any threshold value $t \in [a, b]$ defines a clustering of $G$. We refer to this method as a metric based clustering.

Figure 1 gives a clear illustration of our method. In this example, the graph is made out of four groups of nodes that can be visually identified (the graph in part (a) is pseudo-random and was actually built to bear this structure). The edges between distinct blocks are weak edges and can be identified as such by computing the metric since their value falls below a given threshold. Once those edges have been deleted (as shown on part (b) of the Figure), the connected components of the induced graph correspond exactly to the cluster structure we seek for.

As one may expect, the size and number of clusters calculated in this way depend on the value of the threshold $t$. We will take a closer look at the situation in section 4. We first present the metric into more details and motivate its use to cluster graphs.

### 2.1 Cluster measure of a vertex.

First observe that metrics can also be defined for vertices of a graph. The metric we now introduce is inspired from a clustering measure used to characterize the so-called “small-world graphs” [8, 9]. It is defined for each vertex $v$ in a graph $G$ as follow. Let $N_v$ denote the set of neighbours of $v$ and suppose it has size $k$. Let $e(N_v)$ denote the number of edges between vertices of $N_v$. The cluster measure for $v$,
we denote $c(v)$, is defined as

$$c(v) = \frac{e(N_v)}{(k_v)}$$

where $(k_v)$ denote the binomial coefficient $(k_v) = \frac{k(k-1)}{2}$. The value $(k_v)$ corresponds to the maximum number of edges that can connect vertices in $N(v)$ (the number of edges in the complete graph on the set of vertices $N_v$). So the metric measure the edge density in the neighborhood of the vertex $v$. Figure 2 illustrates the calculation of this metric. The value $c(v)$ represents the ratio of the actual number of edges between vertices of $N_v$ in relation to the maximal number of edges between these vertices. Since there are 4 edges connecting neighbors of $v$ (darker edges), we have $c(v) = \frac{4}{10}$. The clustering measure of a graph $G$ is obtained by averaging the clustering measure of all vertices,

$$\frac{1}{|G|} \sum_{v \in G} c(v).$$

Small-world graphs are graphs with high cluster measure and small average path length, in comparison to the same statistics computed on random graphs (see [8] and [9]). That is, these graphs correspond to networks where any two nodes are only a few steps away from each other, and where nodes are globally organized as closely linked subgroups. Table 2.1 reports those measures for several software systems, as well as for random graphs, providing arguments to the fact that software systems define small-world graphs.

The idea we started from was to exploit the cluster measure of vertices to detect clusters in a graph. However, the cluster measure on vertices does not reveal itself as a good indicator. Indeed, Figure 3 shows a graph where several different vertices are assigned the same cluster measure. Thus, the metric indicates that all vertices with a metric value of 1/2 play similar or equivalent roles in the graph. However, one can argue that the left and right groups of vertices are its natural clusters and that the central edge weakly links them. Hence, we seek for an edge measure that would identify the central edge $e$ as the one to be removed.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Cluster measure</th>
<th>Av. path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resyn (access)</td>
<td>0.95</td>
<td>3.28</td>
</tr>
<tr>
<td>Leda (includes)</td>
<td>0.15</td>
<td>3.96</td>
</tr>
<tr>
<td>Leda (UML)</td>
<td>0.108</td>
<td>3.78</td>
</tr>
<tr>
<td>Linux (includes)</td>
<td>0.129</td>
<td>3.60</td>
</tr>
<tr>
<td>Mac OS9 (includes)</td>
<td>0.387</td>
<td>2.86</td>
</tr>
<tr>
<td>MFC (includes)</td>
<td>0.099</td>
<td>3.02</td>
</tr>
<tr>
<td>Random clustered</td>
<td>0.725</td>
<td>2.46</td>
</tr>
<tr>
<td>Random graph</td>
<td>0.016</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table 1. Cluster measure and average path length of several software systems.

2.2 Edge strength metric

We now extend the metric $c$ to a metric $\Sigma$ defined on edges of a graph. As we will see, this new metric has many interesting properties for metric based clustering of graphs. In particular, it resolves the situation we pointed at above since it is such that $\Sigma(e) = 0$ if $e$ is an isthmus. This metric corresponds to a measure of how much an edge is likely to separate a graph in two highly connected subgraphs. It measures the strength of edges in regard to this property. This metric is related to the density of edges in the neighborhoods of the end vertices of $e$, and thus appears as a generalization of the cluster measure to edges in a graph. As we will see, this density is calculated from the ratio of the number of paths of length three and four actually going through $e$ in relation to the maximal number of such paths.

We need to introduce notations before we can describe our extension of $c$ to edges. Let $e$ be an edge and $u, v$ be its endpoints. Denote by $N_u$ and $N_v$ the respective neighborhoods of $u$ and $v$ and define the set $M_u = N_u \setminus N_v$. The set $M_u$ contains neighbors of $u$ that are not neighbors of $v$. Similarly, define $M_v = N_v \setminus N_u$. Moreover, let $W_{uv}$ be the intersection of $N_u$ and $N_v$. That is, $W_{uv}$ gathers vertices that are neighbours of both $u$ and $v$. Observe that the sets $N_u$, $N_v$ and $W_{uv}$ form a partition of the set of vertices at distance 1 from $u$ or $v$. Figure 4 summarizes the situation.

This partition is useful to classify cycles of length four (4-cycles) going through the edge $e$. First observe that such a cycle contains four vertices. Two of them are $u$ and $v$, so
Figure 4. Partition used to calculate the strength metric of an edge.

Figure 5. A 4-cycle through e.

Figure 6. Extraction of clusters.

Let $U$ and $V$ be two subset of vertices. Define the ratio

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

where $e(U, V)$ denotes the number of edges connecting a vertex of $U$ to a vertex in $V$. Thus $s(U, V)$ computes 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. Also, we define

$$s(U) = \frac{e(U)}{\binom{|V|}{2}}.$$

Using our notations, we can express the edge density $\gamma_4(e)$ corresponding to 4-cycles going through an edge $e = u, v$

$$\gamma_4(e) = s(N_u, W_{uv}) + s(N_v, W_{uv}) + s(N_u, N_v) + s(W_{uv}).$$

Similarly, edge density related to 3-cycles going through $e$ can be computed as

$$\gamma_3(e) = \frac{|W_{uv}|}{|W_{uv}| + |N_u| + |N_v|}.$$ 

Finally, the strength metric of an edge $e$ is defined as the sum:

$$\Sigma(e) = \gamma_3(e) + \gamma_4(e).$$

This definition is related to edge density in the neighborhood of $e$ (where the neighborhood has been divided into three distinct parts as above). A low value for $\Sigma(e)$ indicates that the edge is more likely to act as an isthmus between clusters. Contrarily, a high value for $\Sigma(e)$ indicates that it is potentially at the center of a cluster. Consequently its endpoints and possibly its neighborhood should belong to a same cluster.

It is worth to note that the metric $\Sigma$ is related to the notion of shortcut used in the context in small-world graphs (see [8]). In fact, the value of $\Sigma(e)$ will be high if the edge $e$ is a shortcut for many 3-cycles and 4-cycles passing through $e$. The value will be low if there is not so many cycles of length 3 and 4 passing through $e$.
3 Applications

In this section, we present applications of our clustering technique to existing software systems, showing the relevance of our method to reverse software engineering. In all examples shown, graphs have been laid using a force-directed algorithm. Moreover, the thickness and saturation of an edge $e$ have been assigned according to its metric value $\Sigma(e)$.

3.1 Calculation on graphs related to logical design: Access graphs

Our first example shows the application of the metric $\Sigma$ to the edges of the access graph of the ResynAssistant software, written in Java at LIRMM. This software is dedicated to organic chemistry. Vertices of this access graph are Java classes. There is an edge between two classes if one of them has access to a method of the other. The access graph presented here have been generated from the work of by Ardourel et al. [10] The graph is shown in Figure 7.

![Figure 7. Application of the metric to ResynAssistant software.](image)

In this example, clusters extracted with our technique have been placed in boxes. The name of the respective packages is indicated in Figure 7. The designers have confirmed that the clusters we were able to identify correspond to the logical structure of the source code. Incidentally, the visualization of the access graph and a close study of the clustering led them to the identification of a design error. More precisely, the cluster labelled Others classes appear as a single cluster instead of two because of loosely designed accesses.

3.2 Calculation on graphs related to physical design

We have applied our method to the graph resulting from the include relations between source files of the MacOS9 operating system API. This API called Universal Headers is publicly available at the URL http://developer.apple.com/sdk. Includes relations are induced by the #include pre-compiler directive in C++. Figure 8 illustrate the clusters calculated with our technique. The threshold value used here has been chosen as described in Section 4.2. In Figure 8, every file in the cluster labelled CG belong to the Core Graphics component of MacOS. The cluster labelled CF contains files belonging to the Core Foundation Utility Routines component. Another cluster is labelled QT and corresponds to QuickTime. Many of the files in the cluster labelled ATS are related to the Apple Type Services component. Note however that there are files not directly in this component (for example, a file belonging to this cluster is related to threads). Remark that software components are less easily extracted from the include relations on source files, since this relation actually reflects the way the software is implemented.

The next example is about Microsoft Foundation Classes (MFC). We have applied our algorithm to the graph resulting from the include relations between source files. Figure 9 illustrate the result. The larger cluster, labelled AFX.ATL contains file from the Application Framework and Active Template Library components of MFC. Another cluster, labelled OCC, contains files from OLE Container Component. A small cluster concerns the strings mapping component. It is labelled MAP. Finally, we have found a cluster related to data base support, labelled DBSup, which contains
a few files. The two previous clusterings contain many iso-
lated vertices. This is related to the structure of the system
which can be seen as a collection of overlapping subtrees.
The threshold value used in the calculation of the previ-
ous clusters was obtained by the method described in sec-

4 Clustering quality measures

We now turn ourselves to the problem of evaluating the
quality of a clustering of a graph. This problem has been
previously addressed by a number of people in the software
engineering community [5, 2, 3]. From all the quality mea-
sures that have been defined, we focus on

\[ MQ \] introduced

by the authors of [3]. \( MQ \) actually computes a value for any
given partition \( C = (C_1, \ldots, C_k) \) of a graph \( G = (V,E) \). Edges in \( E \) contribute as a positive or negative weight ac-
cording to whether they are incident to vertices of a same
block \( C_i \) or to vertices of distinct blocks \( C_i, C_j \). It is de-
defined as follows, using the notations introduced in section 2.

\[
MQ(C; G) = \frac{\sum_{i=1}^{p} s(C_i, C_i)}{p} - \frac{\sum_{i=1}^{p-1} \sum_{j=i+1}^{p} s(C_i, C_j)}{p(p-1)/2}.
\] (1)

A straightforward consequence is that a higher \( MQ \) value

The proof of this fact is somewhat straightforward and is a
consequence of the central-limit theorem [11]. Indeed, each
of the terms \( s(C_i, C_i) \) in Equation 1 can be seen as a ran-
dom variable. These random variables are obviously inde-

Moreover, the mean value and standard deviation of this
gaussian approximation can be satisfactorily estimated as
\( \mu = -0.2 \) and \( \sigma = 0.2 \) as indicated in Figure 10. This
result can now be used to answer the questions we pointed at
above. Given a partition \( C \), we are now able to judge of its
goodness or badness with more confidence. For instance,
choosing a partition uniformly at random will most surely
give a clustering with a negative \( MQ \) value close to \(-0.2 \). Con-
versely, a partition with a positive but even small value,
is already a good clustering of a graph (compared to what an
average partition would give). Indeed, the probability that a
random partition has a positive value is approximately 15%. A bit more than 10% of all partitions have an $MQ$ value above 0.05 and only 2% of all partitions have an $MQ$ value above 0.2. A partition with an $MQ$ value above 0.31 can only be found in 0.5% of all partitions.

**Remark**
The above definition for $MQ$ actually differs from the one used in [6]. Contrarily to Bunch, we chose to define $MQ$ for undirected graphs. This choice does not affect the definition of the numerators but only changes the denominators (which differ by a factor $1/2$). This makes sense since some of the graphs we study correspond to the non-symmetric “includes” relations between physical files.

The graph for the ResynAssistant API is obtained differently. The vertices of the graph correspond to classes of the API. A vertex $u$ is adjacent to a vertex $v$ if it can access methods or attributes of $v$. Although the edges of the graph have natural orientations, we ignored them and considered the graph as non-directed.

### 4.1 Links between $MQ$ and the strength metric for edges

We now look at the quality of the clustering obtained by filtering the graph with the strength metric. The question we examine here is ‘Just how good is the clustering obtained by filtering out the edges?’ As a first evaluation, we have compared the clusterings we obtain with the ones produced by Bunch [6], which appears as one of the good clustering software used by the reverse engineering community. It should be well understood that our method does not compete with Bunch. As we understand it, Bunch implements standard optimization methods and tries to find a clustering with the highest possible $MQ$ value. The value of our approach, as we shall see, is that it produces clustering of good quality in short computing time, since it has low computational complexity. In our view, our technique could be used to significantly improve the performance of Bunch or similar tools or algorithms. Indeed, starting the search for a good clustering with an already good candidate usually improves the performance of such algorithms, both in time and quality. Moreover, our technique can be embedded in an interactive environment to let the user guide Bunch (or any other software based on heuristic algorithms) find a good clustering candidate in the minimum time.

The following table summarizes the comparisons we made with Bunch. The quality measures we report should be interpreted in view of the statistical distribution of the $MQ$ values we underlined in the preceding section. For each software system we examined, we report the $MQ$ value of a clustering found by Bunch and the clustering induced from the strength metric (using the best possible threshold value).

<table>
<thead>
<tr>
<th>Graph</th>
<th>MQ/Bunch</th>
<th>MQ/Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResynAssistant (access)</td>
<td>0.435 (10)</td>
<td>0.368 (10)</td>
</tr>
<tr>
<td>Mac OS9 (includes)</td>
<td>0.137 (4)</td>
<td>0.015 (251)</td>
</tr>
<tr>
<td>MFC (includes)</td>
<td>0.044 (4)</td>
<td>0.011 (373)</td>
</tr>
<tr>
<td>Random clustered</td>
<td>0.322 (6)</td>
<td>0.346 (6)</td>
</tr>
</tbody>
</table>

Table 2. Comparison of $MQ$ values between Bunch and Strength metric

Table 4.1 shows that, in most cases, the clustering obtained from the Strength metric compares well with the one obtained from Bunch, since both clusterings have very close $MQ$ values. For instance, the results obtained for the ResynAssistant API compare very well, since the subset of partitions having an $MQ$ value lying between 0.368 and 0.435 represents only a bit more that one tenth of a percent of all partitions. It should also be remarked that in both cases the partitions found consist of the same number of blocks.

The random compared for the Mac OS9 and MFC softwares admits a simple explanation. The structure of these graphs roughly compares with a large and highly coupled component having several and smaller hierarchies attached to its periphery. To be able to get at the 4 components identified by Bunch, our method needs to filter the edges with a high threshold value, thus leaving a rather large number of nodes isolated. Each of these nodes corresponds to a cluster, and edges stemming from them count as inter-cluster, which explains the bad $MQ$ score we get. Collecting the isolated nodes and grouping them to one of the large component (through a DFS for instance) would undoubtedly lead to a better $MQ$ value and to a much lower number of clusters.

The random clustered graph example (see Figure 1) was run for sake of completeness. This graph, although not random from a strictly theoretical point of view, is built by selecting a number of clusters, prescribing upper and lower bounds for their number of inter-cluster edges and extra-cluster edges. It is not at all surprising that both methods find a clustering with the exact number of blocks. The surprise is that with this example, our method was able to obtain a better score than Bunch. In another random clustered example, not reported here, both methods found exactly the same clustering.

### 4.2 Automating the process

We have mentioned that our method could well be embedded in an interactive environment, giving a user the freedom to browse through the clusterings we are able to produce. However, when considering our method as a possible input for Bunch, we need to be able to automatically identify the
clustering to use as a starting point. This problem translates into the question of finding the threshold value corresponding to the clustering with the highest possible MQ score we are able to find. Hence we were naturally led to study the possible correlation between the interval of thresholds and the MQ values we reach.

Figure 11 shows the variation of MQ as the threshold goes through the interval of all possible values for the strength metric. The metric values on the x-axis have been normalized to the [0, 1] interval. The curve gives the MQ score associated with the clustering obtained from the corresponding threshold value on this interval. The variation along the curve also relates to a path in the set of all possible partitions. Indeed, suppose a threshold value \( t \in [0, 1] \) has been chosen and denote by \( C_t \) the clustering induced from this threshold \( t \). Then, the clustering \( C_{t'} \) induced from a slightly larger threshold \( t' > t \) can be obtained from \( C_t \) by dividing some of its blocks into two or more parts. Hence the ordered list of all clusterings obtained by letting \( t \) vary over the whole interval \([0, 1]\) gives an ascending path in the lattice of all partitions. That is, the path corresponds to a curve extracted from a high-dimensional space (the space of all possible partitions and their corresponding MQ value).

As Figure 11 shows, once the curve has been computed, it is straightforward to find its maximum value. It could actually be useful to find all local maxima which can be indicators for good clustering candidates (a local maxima could just be a point sitting close to a peak reached by the curve). Indeed, the MQ/Strength curve can admit many local maxima, as shows Figure 12. Also, it should be noted that the cost for computing the curve is proportional to the cost of computing MQ for a given partition of a graph (since the \([0, 1]\) interval is interpolated at values \( t \) a constant number of times).

It should be remarked at this point that the MQ values reported in the comparison table in section 4 have been computed by Bunch. However, the curves in Figure 11 and Figure 12 report values computed with our version of MQ, (which basically explains why the maximum MQ values reached by the curves is relatively higher).

Figure 13 shown as last example the graph of includes for the MFC API (see Figure 9). Again, the local maxima can be easily computed in order to get at a almost optimal clustering of the graph. Note however that the intrinsic structure of the graph makes it difficult to reach high MQ scores. We also ran this example with Bunch who produced clusterings having an MQ value just above zero, even by letting the
heuristics run extensively. In situations such as this, our method seems to offer a tangible advantage, be it simply that it can find similar candidates almost instantly.

5 Perspectives and future work

We have presented a simple, fast computing and easy to implement method aiming at the capture of software components from a logical and physical point of view. Our method exploits a metric based clustering of graphs. The metric we have introduced is a new metric that measures edge density in graphs and is inspired from the cluster measure defined by Watts [9] for the so-called small-world graphs. The motivation behind this is that software systems show resemblance with graphs of this class.

The relatively low complexity of the underlying calculations of our method allows it to be embedded in an interactive environment. Clusters on graphs of thousands of vertices can be done in a second.

From the experimentations we were able to conduct, our method appears to get better results for graphs corresponding to logical design that those resulting from physical design. Indeed, the graphs resulting from include relations do not always separate into well defined and large clusters but tend to explode into several small clusters. This is a consequence of the fact that the graph under study often depends on the quality of the underlying software. It should be mentioned that although our method does not appear as being well adapted to repair deficient design, it is useful to detect design flaws.

We have mentioned that our method appears as a useful preprocess step to optimization procedures such as the ones used by Bunch. More work is needed to demonstrate the use of our technique as a guiding strategy or as good initial solutions for optimization heuristics. This study should most probably go deeper into the comprehension of the structure of the space of all clusterings, with respect to $MQ$ seen as a similarity measure. Also, the actual quality of the clusterings we are able to produce suffer from the large number of isolated vertices. The quality can certainly be improved by agglomerating them to larger clusters using a DFS for instance.

References


Appendix: Complexity analysis

6.1 Strength measure

Let $G = (V, E)$ be a graph and write $|V| = n$ and $|E| = m$. Also, assume that the average degree of vertices is bounded by above, that is $\frac{1}{n} \sum_{v \in V} d(v) \leq c$, where $c > 0$ is a positive real number. Note that this assumption follows from empirical observations applies to all software systems that we studied.

Let $e \in E$ be an edge in the graph. The computation of the metric boils down to the computation of the seven sets $M_u, M_v, W_{uv}$ (described in section 2.2), $E(M_u, M_v)$, $E(M_u, W_{uv})$, $E(M_v, W_{uv})$ and $E(W_{uv})$. The computation of the first three sets is made in constant time, by virtue of the assumption on the average degree of vertices. The four last sets are built by looking at the neighborhood of all vertices belonging to $M_u$, $M_v$ and $W_{uv}$. Note that, the average size of each of these three sets is at most $c$. Hence, the computation of each of the set $E(U, V)$ (where $U, V$ stand for the appropriate neighborhoods) is done in time at most $c^2$ on average. To sum up, building the seven sets is made in constant time $O(c^2 + c)$. Consequently, the metric is computed on the whole graph in time $O(m)$ (observe that our assumption on the average vertex degree implies that $m$ is proportional to $n$).

6.2 Cluster retrieval

We now look at the actual cost for computing the clusters induced from a fixed threshold value $t \in [a, b]$. Note that this can be done through a Depth First Search algorithm. The complexity of this algorithm is $O(N + M)$. 