Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements

Brian S. Mitchell, Spiros Mancoridis
Department of Mathematics & Computer Science
Drexel University, Philadelphia, PA, USA
{bmitchel, smancori}@mcs.drexel.edu

Abstract

Decomposing source code components and relations into subsystem clusters is an active area of research. Numerous clustering approaches have been proposed in the reverse engineering literature, each one using a different algorithm to identify subsystems. Since different clustering techniques may not produce identical results when applied to the same system, mechanisms that can measure the extent of these differences are needed. Some work to measure the similarity between decompositions has been done, but this work considers the assignment of source code components to clusters as the only criterion for similarity. We argue that better similarity measurements can be designed if the relations between the components are considered.

In this paper we propose two similarity measurements that overcome certain problems in existing measurements. We also provide some suggestions on how to identify and deal with source code components that tend to contribute to poor similarity results. We conclude by presenting experimental results, and by highlighting some of the benefits of our similarity measurements.

1. Introduction

Most interesting software systems are large and complex and, as a consequence, understanding their structure is difficult. Such software systems are composed of many resources (*e.g.,* classes, modules, variables) that depend on each other in intricate ways (*e.g.,* procedure calls, inheritance relationships, variable references). The structure of these systems can be represented as a graph where the nodes are the resources and edges are the relations between the resources.

Without automated assistance the software structure graph provides little value when being used to understand the design of practical systems because of its large number of nodes and edges. System maintainers would find it easier to work with design documentation that provides a simplified description of the software structure. Unfortunately, we often find that accurate design documentation does not exist, and that the original developers of the system are unavailable for consultation. With nowhere else to turn, developers will generally modify source code without a thorough understanding of how their modifications affect the overall system structure.

To address the problem mentioned above researchers in the reverse engineering community have been developing clustering tools. These tools decompose source-level resources into subsystems by partitioning the software structure graph. Subsystems generally consist of a collection of source code resources that collaborate with each other to implement a feature or provide a service to the rest of the system. Typical resources found in subsystems include modules, classes, and possibly, other subsystems. Subsystems facilitate program understanding by treating sets of source code resources as high-level entities.

The primary goal of clustering tools is to propose subsystems that expose abstractions of the software structure. However, the various clustering tools use different algorithms, and make different assumptions about how subsystems are formed. Most techniques use source code component similarity [8, 16, 6, 15], optimization [13, 7, 12], concept analysis [11, 20, 1], or implementation information such as module, directory, and/or package names [2, 3] to determine the clusters.

Now that many clustering techniques exist, some researchers have turned their attention to evaluating their relative effectiveness [10, 18, 19, 2, 3]. There are several reasons for this:

- Many of the papers on software clustering formulate conclusions based on case studies, or by soliciting opinions from the authors of the systems presented in the case studies.

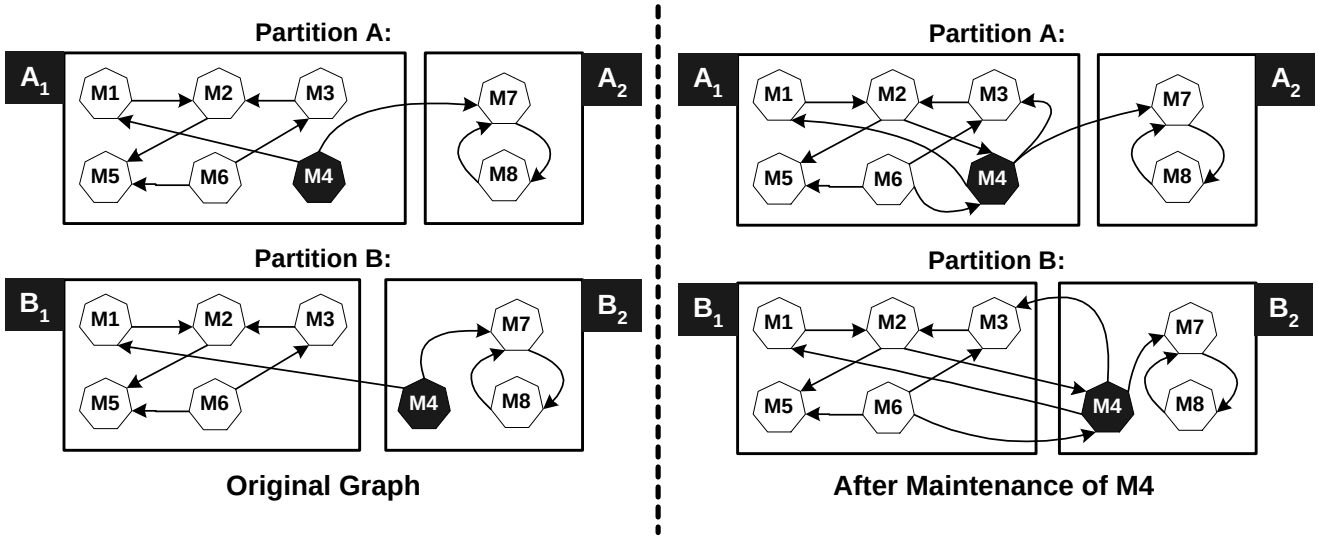

Figure 1. Example Dependency Graphs for a Software System

- Much of the research on measuring the similarity between decompositions only considers the assignment of the system's modules to subsystems. We argue that a more realistic indication of similarity should consider how much a module depends on other modules in its subsystem, as well how much it depends on the modules of other subsystems. We illustrate this point in Figure 1 where the graph nodes represent source-level modules and the edges represent inter-module relations. Notice that the two decompositions on the left side of the figure are identical to the two decompositions on the right side of the figure as far as module placement is concerned. However, if the dependencies are also considered, it is obvious that the two decompositions shown on the left side of the figure are more similar than the two decompositions on the right side of the figure.

- When decompositions are compared, all source code resources tend to be treated equally. We argue in Section 4 that special consideration should be given to certain modules.

- When decompositions are compared, conclusions are often formulated based on the value of a similarity or distance measurement. In Section 2.3 of this paper, we investigate whether poor similarity results can be useful for gaining any intuition about a system's structure.

The above points provide the motivation for this paper. In Section 2 we examine two similarity measurements that have been used to compare decompositions. In Section 3 we propose two new measurements to address some of the shortcomings of the similarity measurements described in Section 2. In Section 4 we argue that certain modules and/or classes warrant special consideration during the clustering process. In Section 5 we review the results of our comparative study of similarity measurements. We conclude, in Section 6, with a summary of the research contributions of this work.

2. Measuring Similarity

In an earlier paper [13] we show that the number of unique ways to decompose a software system into non-overlapping clusters (subsystems) grows exponentially with respect to the number of modules/classes in the source code. Thus, heuristic algorithms are necessary to automate this process. Over the past few years several clustering techniques have been presented in the literature, each making some underlying assumptions about what constitutes a "good" or a "poor" system decomposition. Ideally, the effectiveness of these clustering techniques could be measured against some standard (or at least agreed upon) decomposition of a system. However, we often find that standard benchmarks do not exist.

Many times clustering results (decompositions) are analyzed and evaluated by soliciting feedback from the developers and designers of the system. This evaluation mechanism is subjective, but is effective for determining if a result is reasonable. However, when clustering techniques produce different results for the same system, we would like know why they agree on certain aspects of the software structure, yet disagree on others.

As an initial step toward addressing some of these issues, researchers have begun formulating ways to measure the differences between system decompositions. For example, Anquetil et al. developed a similarity measurement based on Precision and Recall [2, 3]. Tzerpos and Holt authored a paper on a similarity distance measurement called MoJo [18]. Both of these measurements treat each difference between two decompositions of the same system equally. In Section 3 we present both a similarity and a distance measurement that ranks the individual differences between the decompositions, and applies an appropriate weighted penalty to each disagreement.

Before presenting our similarity measurements we review the MoJo and Precision/Recall techniques, highlighting some issues that we encountered when using them to evaluate software clustering results.

2.1. The MoJo Distance Similarity Measurement

MoJo measures the distance between two decompositions of a software system by calculating the number of operations needed to transform one decomposition into the other. The transformation process is accomplished by executing a series of *Move* and *Join* operations. In MoJo, a *Move* operation involves relocating a single resource from one cluster to another, and a *Join* operation takes two clusters and merges them into a single cluster.

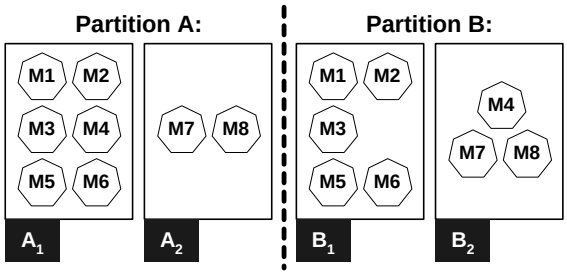

Figure 2. Two Slightly Different Decompositions of a Software System

We present an example, shown in Figure 2, to illustrate how to calculate MoJo. The MoJo value for this example is 1 because decomposition A can be transformed into decomposition B by a single move operation (move $M4$ from cluster A_1 to cluster A_2). Figure 1 shows two other decompositions that have the same structure as Figure 2 with respect to the placement of the modules. The only difference between the decompositions shown in Figure 1 and Figure 2 is that the edges are shown in Figure 1. Since MoJo does not consider edges, the MoJo value is also 1 for both of the decompositions shown in Figure 1. This seems appropriate for decompositions A and B shown on the left side of Figure 1 because $M4$ is a good candidate for either cluster(graph isomorphism). Things are not as clear for the decompositions presented on the right side of Figure 1. Clearly, module $M4$ is very connected to cluster A_1. Therefore, a MoJo value of 1 for transforming partition† A into partition B is misleading because $M4$ clearly should belong to the cluster it is most connected to. Thus, when relations are taken into account, partition A is not very similar to B, or at least not as similar to the respective partitions on the left side of Figure 1.

Tzerpos and Holt also introduce a quality measurement based on MoJo. The MoJo quality measurement normalizes MoJo with respect to the number of resources in the system. Given two decompositions, A and B, of a system with N resources, the MoJo quality measurement is defined as:

$$MoJoQuality(A,B) = \left[1 - \frac{MoJo(A,B)}{N}\right] \times 100\%$$

Since the other measurements discussed in this paper evaluate similarity as a percentage, we will refer to the MoJo quality measurement simply as MoJo from this point forward.

2.2. The Precision/Recall Similarity Measurement

Precision/Recall as defined by Anquetil et. al. [2] calculates similarity based on measuring *intra pairs*, which are pairs of software resources that are in the same cluster. Assuming that we are comparing two partitions, A and B, *Precision* is defined as the percentage of intra pairs in A that are also intra pairs in B. *Recall* is defined as the percentage of intra pairs in B that are also intra pairs in A.

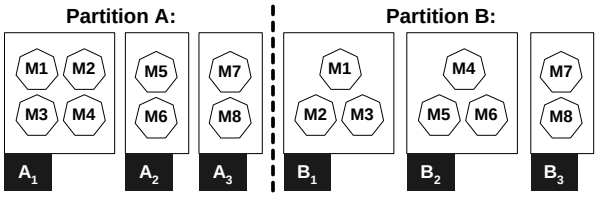

Figure 3. Another Example Decomposition

Like MoJo, the Precision/Recall measurement does not consider edges when calculating similarity. Thus, the partitions presented in Figure 1 and Figure 2 produce the same Precision(84.6%) and Recall(68.7%) values. Another undesirable aspect of Precision/Recall is that the value of this measurement is sensitive to the

†We will use the terms "partition" and "decomposition" interchangeably.

size and number of clusters. Thus, a few misplaced modules in a cluster with relatively few members will have a much larger impact on Precision/Recall than if the cluster has many members. Also, the number of clusters impacts Precision/Recall. We do not think that there should be a correlation between the size and number of clusters when measuring similarity. As an example, consider Figure 3. In this example, 8 modules are spread into 3 clusters. Only one module, $M4$, differs in its placement between A and B. Thus, the number of MoJo move and join operations remains 1 ($MoJoQuality = 87.5\%$). However, the Precision and Recall values shown in Figure 3 are 71.4% and 62.5% respectively (*i.e.*, lower than the Precision/Recall of Figure 2). This mistakenly leads us to believe that A and B in Figure 2 are more similar than the partitions shown in Figure 3.

2.3. Interpreting Similarity Results

Anquetil and Lethbridge [3] state that clustering techniques do not recover a software system's decomposition, but rather impose one. One might ask: What does it mean when the results produced by two different clustering techniques differ? Can anything meaningful be learned when two clustering algorithms produce significantly different partitions?

Obviously, when comparing a pair of partitions, a high similarity is desirable. Consider a simple experiment where a pair of partitions have a high similarity (*e.g.*, $> 95\%$). We then compare another pair (*i.e.*, a different system) of partitions and find a lower similarity value (*e.g.*, $< 60\%$). One can safely assume that the first pair of partitions is more similar to each other than the second pair.

Next, consider taking a pair of systems and clustering each one of them many times, using different clustering algorithms. Furthermore, assume that the average similarity found for the first system is high (>95%), but the average similarity for the second system is low (<60%). In cases where the average similarity is low, we often find that some modules in a system are assigned to particular subsystems with a high degree of certainty, while others tend to drift between a few subsystems, yet others do not seem to belong to any particular subsystem. Thus, a low similarity value may not be an indication of an inconsistent or unstable clustering algorithm, but rather of the tendency of some modules to be assigned to more than one subsystem across clustering results.

Similarity measurements can also be used to evaluate the stability [17, 18, 19] of a clustering algorithm. The idea behind stability is that small changes to a software system's structure should not result in large differences in how a clustering algorithm decomposes the system.

3. Our Similarity Measurements

In this section we present two similarity measurements that we have designed and implemented to overcome the shortcomings of Precision/Recall and MoJo that were discussed above. We begin by examining a similarity measurement that evaluates the differences between two partitions of a software system's structure. We then describe an algorithm that computes the distance between two partitions, which is based on the number of steps required to transform one of the partitions into the other. In Section 5, we compare our similarity measurements to Precision/Recall and MoJo.

3.1. Edge Similarity Measurement - *EdgeSim*

Given some of the problems mentioned in Section 2 we propose an approach to measuring similarity that considers both the vertices and edges of a partitioned graph. We next describe our approach formally along with an example.

Let a graph G=(V,E) represent the structure of a software system where V is the set of modules/classes and E is the set of the weighted dependencies between the modules/classes. The edge weights indicate the relative strength of the relationship between a pair of modules in the system[‡]. Graph G can be generated for most software systems using readily available source code analysis tools such as Acacia for C/C++ [5, 4] and Chava for Java [9].

Consider two partitions, A and B, of graph G. Let A_i, $1 \leq i \leq k$, be the clusters of A, and B_j, $1 \leq j \leq l$ be the clusters of B. Each A_i and B_j are subsets of the vertices in G. Figure 4 shows a sample graph and two similar partitions of it.

Consider the following sets:

$$\Phi = \{\langle u, v\rangle \in E \mid (u \in A_i \wedge v \in A_i) \wedge (u \in B_j \wedge v \in B_j), \\ where\ (1 \leq i \leq k)\ and\ (1 \leq j \leq l)\}$$

$$\Theta = \{\langle u, v\rangle \in E \mid (u \in A_i \wedge v \notin A_i) \wedge (u \in B_j \wedge v \notin B_j), \\ where\ (1 \leq i \leq k)\ and\ (1 \leq j \leq l)\}$$

$$\Upsilon = \Phi \cup \Theta$$

Given the above sets, Υ is the set of edges that are either intraedges (*i.e.*, edges that connect vertices within the same cluster) or interedges (*i.e.*, edges that connect

[‡]Müller et al. [15] provides several suggestions that can be used to determine edge weights. Similarity measurements should take edge weights into account.

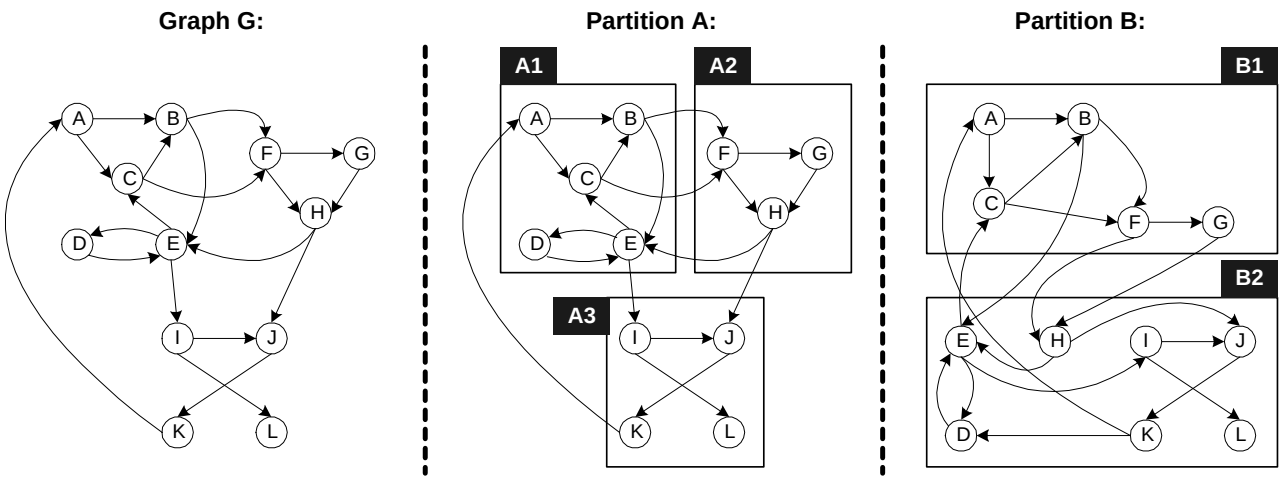

Figure 4. Example Clustering

vertices in distinct clusters) in both A and B. Figure 5 illustrates the Υ edges for partition A in Figure 4 using thick arrows. Since the Υ edges indicate agreement about the placement of a pair of vertices in both A and B, we aggregate the edge weights of Υ to calculate the $EdgeSim$ measurement.

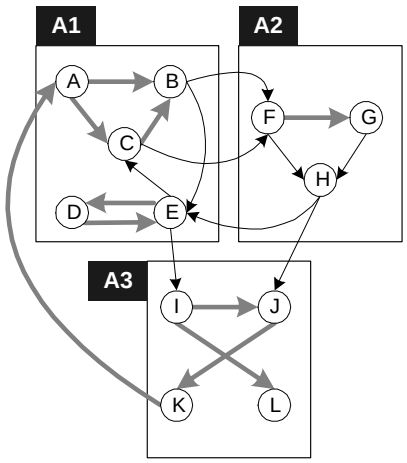

Figure 5. The Υ Set from Figure 4

The ratio of the weights in set Υ with respect to the weights of all edges in G normalizes the $EdgeSim$ measurement. Specifically, the similarity between A and B is:

$$EdgeSim(A,B) = \left\lceil \frac{weight(\Upsilon)}{weight(E)} \right\rceil \times 100$$

where,

$$weight(\Upsilon) = \sum_{e_i \in \Upsilon} weight(e_i)$$

If the graphs do not contain weighted edges, we assume that each edge has a weight of 1. The weight of the Υ set for the example presented in Figure 4 is 10 (*i.e.*, there are 10 bold edges in Figure 5). The $EdgeSim(A,B)$ measurement is therefore $[(10 \div 19)] \times 100$, or 52.6%. The *EdgeSim* measurement is reflexive ($EdgeSim(A,B) = EdgeSim(B,A)$) and can be calculated in $O(|E|)$ time.

3.2. Partition Distance Measurement - *MeCl*

In this section we present our measurement, called $MeCl$, to compute the distance between two partitions of a graph. $MeCl$ first splits the clusters in one partition, and then merges them back together to reproduce the other partition. Hence, the name ***Me***rge ***Cl***usters.

We begin our explanation of $MeCl$ by presenting several definitions. Given a graph G=(V,E), we define a partition of G as $\Pi_G = \{G_1, G_2, \ldots, G_n\}$, where each G_i is a cluster in the partitioned graph. Specifically:

$$G_i = (V_i, E_i)$$
$$\bigcup_{i=1}^{n} V_i = V$$
$$\forall((1 \le i,j \le n) \wedge (i \ne j)),\ V_i \cap V_j = \emptyset$$
$$E_i = \{\langle u,v\rangle \in E \mid u \in V_i \wedge v \in V\}$$

We next define the vertices (V_i), edges (E_i), intraedges (Φ) and interedges (Θ) of G_i with respect to G:

$$V_G(G_i) = V_i$$
$$\Phi_G(G_i) = \{\langle u,v\rangle \in E_i \mid u \in V_i \wedge v \in V_i\}$$
$$\Theta_G(G_i) = \{\langle u,v\rangle \in E_i \mid u \in V_i \wedge v \notin V_i\}$$
$$E_G(G_i) = E_i = \Phi_G(G_i) \cup \Theta_G(G_i)$$

To illustrate the above definitions, consider cluster A_3 which is shown in Figure 4 and Figure 5. For this partition:

$$V_A(A_3) = \{I, J, K, L\}$$
$$\Phi_A(A_3) = \{\langle I, J\rangle, \langle J, K\rangle, \langle I, L\rangle\}$$
$$\Theta_A(A_3) = \{\langle K, A\rangle\}$$
$$E_A(A_3) = \{\langle I, J\rangle, \langle J, K\rangle, \langle I, L\rangle, \langle K, A\rangle\}$$

Now that some definitions have been provided, we proceed to measuring the distance between two partitions of a software system. Let A and B represent two partitions of graph $G = (V, E)$. We define the subgraphs (clusters) of A and B with respect to G as Π_A, Π_B such that:

$$\Pi_A = \{A_1, A_2, \ldots, A_k\}$$
$$\Pi_B = \{B_1, B_2, \ldots, B_l\}$$

The next step in determining the distance between A and B involves subpartitioning each A_i with respect to each of the subgraphs in Π_B. We refer to a *subpartition* of A_i with respect to subgraph B_j in Π_B as $C_{i,j}$. $C_{i,j}$ is formally defined for each $1 \leq i \leq k,\ 1 \leq j \leq l$ as follows:

$$C_{i,j} = (V_{i,j}, E_{i,j})$$
$$V_{i,j} = V_A(A_i) \cap V_B(B_j)$$
$$E_{i,j} = E_A(A_i) \cap E_B(B_j)$$

Figure 6 illustrates all of the non-empty subpartitions in A with respect to the example presented in Figure 4. In this figure, the subpartition labelled $A_{1,1}$ was formed by intersecting cluster A_1 with cluster B_1, subpartition $A_{2,1}$ was formed intersecting cluster A_2 with cluster B_1, and so on. Figure 4 does not show subpartition $A_{3,1}$ because there are no common vertices or edges between partitions A_3 and B_1.

Next, we define the set of intraedges in A that are interedges in B. These are defined for each $1 \leq i \leq k,\ 1 \leq j \leq l$ as follows:

$$\Upsilon_{i,j} = \Phi_A(A_i) \cap \Theta_B(B_j)$$

Once all of the subpartitions of each A_i (all $C_{i,j}$) are derived, we combine them to recreate the clusters in Π_B (each B_j) as shown in the bottom part of Figure 6. This merge process is performed for each $1 \leq j \leq l$ as follows:

$$B_j = (V_j, E_j)$$
$$V_j = \bigcup_{i=1}^{k} V_{i,j}$$
$$E_j = \bigcup_{i=1}^{k} E_{i,j}$$

During the merge process, we pay attention to the $\Upsilon_{i,j}$. For example, $\Upsilon_{1,1} = \{\langle B, E\rangle\}$ because edge $\langle B, E\rangle$ is an intraedge in cluster A_1 and an interedge in cluster B_1. All of the $\Upsilon_{i,j}$ for Figure 6 are shown below (for clarity we omit all $\Upsilon_{i,j} = \emptyset$):

$$\Upsilon_{1,1} = \{\langle B, E\rangle\}$$
$$\Upsilon_{1,2} = \{\langle E, C\rangle\}$$
$$\Upsilon_{2,1} = \{\langle G, H\rangle, \langle F, H\rangle\}$$

We next define the set of all interedges introduced during the merge process to form cluster j in partition B as:

$$\Upsilon_{B_j} = \bigcup_{i=1}^{k} \Upsilon_{i,j}$$

The bottom part of Figure 6 summarizes the merge process. Thus, to recreate partition B_1 we union subpartition $A_{1,1}$ with $A_{2,1}$. To recreate partition B_2 we union subpartitions $A_{1,2}$, $A_{2,2}$, and $A_{3,2}$. The intraedges introduced during the merge process are: $\Upsilon_{B_1} = \Upsilon_{1,1} \cup \Upsilon_{2,1} = \{\langle B, E\rangle, \langle G, H\rangle, \langle F, H\rangle\}$ and $\Upsilon_{B_2} = \Upsilon_{1,2} = \{\langle E, C\rangle\}$.

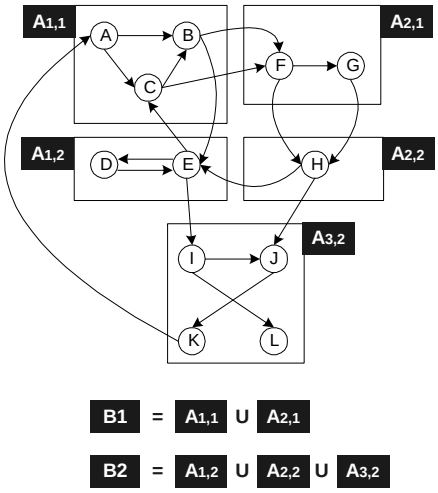

Figure 6. The SubPartitions of A and B from Figure 4

To compute $MeCl$ we evaluate the total cost of the merge operation (the Υ_{B_j} set) as the sum of the weights of the interedges introduced during the merge process:

$$\Upsilon_B = \bigcup_{j=1}^{l} \Upsilon_{B_j}$$
$$MeCl(A, B) = \left[1 - \frac{weight(\Upsilon_B)}{weight(E)}\right] \times 100$$

The $MeCl$ measurement rewards the clustering technique for forming cohesive sub-clusters in A with

respect to B. The cost for introducing new interedges is paid only when the subgraphs in A (*i.e.,* each $C_{i,j}$) are merged to reproduce the original clusters in B. No cost is ever assigned for the common inter- and intraedges in A and B, as these edges indicate agreement between the clustering results.

As with our $EdgeSim(A, B)$ measurement, if G does not contain weighted edges, the $MeCl(A, B)$ expression can be simplified by only computing the cardinality of Υ_B set. However, unlike the $EdgeSim(A, B)$ measurement, the *MeCl* measurement is not reflexive. Thus, in general, $MeCl(A, B) \neq MeCl(B, A)$. When the direction of transformation is not important, and the goal is to obtain a measure of relative "closeness", we suggest using the value $MeCl_{min}(A, B) = min[(MeCl(A, B), MeCl(B, A)]$.

Revisiting the example illustrated in Figure 4, we have already shown that the Υ_B set contains 4 edges $(\{\langle B, E\rangle, \langle E, C\rangle, \langle G, H\rangle, \langle F, H\rangle\})$. These are the intraedges that are converted to interedges during the transformation of partition A into partition B. However, if we consider transforming partition B into partition A we find that the Υ_A set contains 5 edges $(\{\langle E, I\rangle, \langle H, J\rangle, \langle B, F\rangle, \langle C, F\rangle, \langle H, E\rangle\})$. Thus, since $weight(\Upsilon_A) > weight(\Upsilon_B)$, and the software graph in this example contains a total of 19 edges, $MeCl(A, B)$ is $[(1 - (5/19)) \times 100]$, or 73.7%.

The presentation of the *MeCl* measurement in this section shows how the measurement is calculated. *MeCl* can be evaluated in $O(|E|)$ time.

4. Isolating Certain Modules to Improve Similarity

Being able to measure the similarity between two decompositions of a software system is important when evaluating the effectiveness of a clustering technique. In the cases where a benchmark decomposition of a system does not exist, being able to measure similarity alone is of little value. However, if a system is clustered several times, using different clustering algorithms, and similar decompositions are produced, we gain confidence that the proposed clusters are good [14].

Because various clustering algorithms use different criteria to form clusters, we need to identify if large differences in similarity are due to the clustering algorithms, or if the differences are a result of the way that certain "special" modules bias the results. Since our goal is to gain intuition about the structure of the system, one may find it beneficial to exclude the special modules (and their associated dependencies) from the similarity process, especially if this exclusion improves the results produced by the clustering algorithm.

Several questions must now be asked. First, what are the "special" modules? How do we identify them? What do we do with them? To answer the first question, we revisit the work of H. Müller et al. [15] where the authors introduce the notion of an *Omnipresent Module*. Omnipresent modules are defined as modules that are highly connected to many of the other modules in the same system. These modules tend to obscure the system structure, and as such, should be removed from consideration when forming subsystems. We also think that modules that appear to have a high in-degree and 0 out-degree should be removed from the clustering process because they do not seem to belong to any particular cluster (*i.e.*, they behave like libraries in the system).

Without automation, the identification of omnipresent and library modules is tedious. To address this, we added a feature to identify these special modules to our clustering tool called Bunch [12]. Bunch can propose candidate omnipresent and library modules automatically, while allowing the user to add or remove these special modules if necessary. Once these modules are identified, the software engineer may wonder what to do with them. In Bunch, these modules, and their associated dependencies, are removed from the clustering process. The clustering results are presented with the omnipresent and library modules collected into special clusters that are named accordingly. Also, when visualized, these modules are shown with a special shape so that the user knows that the module did not participate in the clustering process.

The final consideration when measuring the similarity between partitions is to account for isomorphic modules. Isomorphic modules are modules that are connected to two or more clusters with the same total edge weight. For example, in the decompositions shown on the left side of Figure 1, we could consider Partition A and Partition B to be identical - one can argue that Module $M4$ could be a member of either cluster. Thus, when performing similarity analysis we think that the isomorphic modules should be included as a virtual member of all subsystems to which they are connected.

5. Comparative Study

In this section we present a comparative study to illustrate the effectiveness of the similarity measurements presented in Section 3, and to validate our assumptions about removing special modules from the clustering process. We will use the Bunch [13, 7, 12] clustering tool because it has several unique features that lend themselves well to our study. In particular, Bunch uses randomization in its optimization approach to form clusters, therefore, because of the very large

System	Original Modules								Special Modules Removed							
	Regular				Isomorphic				Regular				Isomorphic			
	PR	MJ	ES	MC	PR	MJ	ES	MC	PR	MJ	ES	MC	PR	MJ	ES	MC
compiler	72	85	79	93	83	85	91	97	78	87	91	98	80	87	95	100
bison	53	73	66	88	58	73	75	88	66	92	69	90	71	92	75	90
incl	78	84	84	95	80	84	89	95	91	86	99	100	94	86	99	100
rcs	60	75	66	89	63	75	72	89	58	75	81	93	61	75	84	95
boxer	71	80	74	96	80	80	87	96	100	100	100	100	100	100	100	100
grappa	75	88	84	95	76	88	87	96	63	91	91	100	63	91	99	100
ispell	63	79	70	91	68	79	76	91	80	88	95	98	82	88	96	99
cia	56	74	68	67	61	76	74	70	88	93	92	98	89	93	93	98
mtunis	80	85	81	93	82	85	84	94	85	92	93	98	88	92	96	99
linux	39	58	83	92	42	58	87	93	75	74	96	98	76	74	97	99

Table 1. Similarity Comparison Results (Numbers Indicate Percentage Agreement)

search space, it is unlikely that repeated runs will produce the exact same decomposition of a software system. Before we continue, we present a brief overview of the clustering technique implemented by Bunch.

Bunch starts by generating a random partition of the software structure graph. We refer to this graph as the *Module Dependency Graph* (*MDG*). The *MDG* represents all of the modules/classes in the system as nodes, and all of the dependencies between the nodes as edges. The edges in the *MDG* may or may not be weighted to emphasize the relative strength of the dependency between nodes. Bunch determines candidate clusters by moving nodes between the clusters, or in some cases creating new clusters in order to maximize an objective function that we call *Modularization Quality* (*MQ*) [13]. Our *MQ* function has the property of rewarding cohesive clusters, while penalizing excessive inter-cluster coupling.

Bunch's non-determinism also provides the basis for further study about why some systems produce relatively consistent results, while other systems exhibit variability in their results when Bunch is used to cluster the system's *MDG* many times. In the latter case, Bunch always converges to a similar *MQ* value, however, the placement of certain modules in the system, along with the number of clusters in the result varies slightly from one run to another.

Our primary goal is not to focus on the clustering approach implemented in Bunch, but to investigate if variation in the clustering results can be used to infer useful information about a system's structure.

Our study was conducted using 10 systems of varying size. Most of these systems are open-source or used in the academic environment. All of the sample systems were clustered 100 times using Bunch. For each system, the 100 individual results were pairwise compared to each other using the four similarity measurements that were discussed in Section 2 and Section 3. We then recomputed each of the similarity pairs, taking into account the isomorphic modules. This was done by treating an isomorphic module as a member of all clusters to which it is attached.

In our final test, we repeated the experiment described above with one change. This time, for each of the 10 systems, we removed all of the omnipresent and library modules (along with their associated dependencies) from the input *MDG*. In Table 1 we show the results of the experiments. The data for each test represents the average percentage of similarity based on 4950 (*i.e.*, $n(n-1)/2$) samples.

The following are some observations we made from the data shown in Table 1[††]:

- All four of the similarity measurements examined in this study seem to behave consistently with respect to the 10 systems that were examined. This is a good indication that all of the measurements are suitable for measuring similarity. Although the values of the measurements differ across each individual system, we expect that when comparing the results of two systems that the individual measurements will be related in the same way. Thus, if we compare all four measurements to each other across two systems we expect to see all of the values for one of the systems to be larger than all of the corresponding similarity values for the other system. The data in Table 1 shows this relationship constantly for all of the systems examined. For example, since $PR($`bison`$) \leq PR($`ispell`$)$, we also expect to see that $MJ($`bison`$) \leq MJ($`ispell`$)$, $ES($`bison`$) \leq ES($`ispell`$)$, and $MC($`bison`$) \leq MC($`ispell`$)$.

[††]In this table the abbreviations shown in column headers are as follows: PR is Precision/Recall, MJ is the MoJo quality measurement, ES is EdgeSim, and MC is MeCl.

- We indicated in Section 4 that the removal of special (*i.e.*, library, omnipresent) modules should improve similarity. The results of our study validate this claim. The data in Table 1 indicates an average 12.4% improvement for Precision/Recall, 9.7% improvement for $MoJo$, 13.6% improvement for $EdgeSim$, and 7.3% improvement for $MeCl$ when the omnipresent and library modules are removed. The best overall improvement was for the `cia` system. The most likely cause for this improvement is because almost 40% of the nodes in `cia` are omnipresent or library modules (far more than any other system). The average number of special modules for the other 9 systems examined in this study was approximately 19%. The `grappa` and `rcs` systems are the only 2 cases where the value of the Precision/Recall measurement dropped after the removal of the special modules. These systems had very few clusters, and as such, small deviations between the partitions were probably amplified by the shortcomings of the Precision/Recall measurement that were discussed in Section 2.

- The data from the study also indicates that a slight improvement in similarity is attained by allowing isomorphic modules to become virtual members of all clusters to which they are connected. There were no counterexamples in the data showing that special treatment of isomorphic modules had a negative impact on similarity measurement results. The `compiler` and `boxer` systems appear to have improved the most when isomorphic modules were considered. This result was probably because the `compiler` and `boxer` systems were the smallest systems in the study, consisting of 13 and 18 modules respectively.

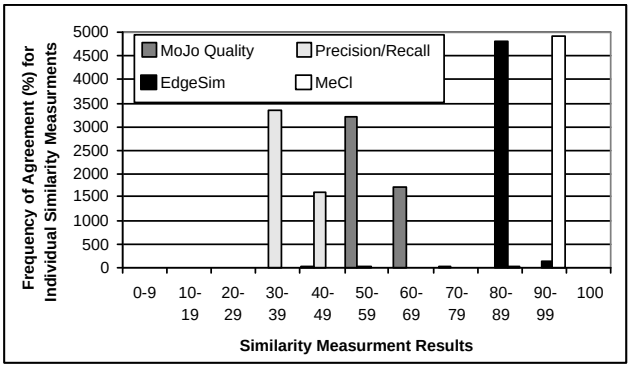

Figure 7. Frequency Distribution for `linux`

- Similarity measurements such as $EdgeSim$ and $MeCl$, which consider the relations between the source code components, seem to produce results that are less variable than similarity measurements such as Precision/Recall and $MoJo$, which only consider the differences in module assignments. Since our study was conducted by pairwise comparisons of a 100 runs of the exact same system, we would expect to see very little variability in the individual similarity measurement results. In Figure 7 we show a frequency distribution for the results of clustering the `linux` system. This graph illustrates that the variance in the Precision/Recall and $MoJo$ measurements is much larger than the variance in the $EdgeSim$ and $MeCl$ measurements. Due to space restrictions, we only show the frequency distribution for the `linux` system in Figure 7, although the other systems examined in our case study share this behavior. For the `linux` system, Table 2 shows the min-max spread as well as the variance for each similarity measurement.

Similarity Measurement	**max-min**	**Standard Deviation**
`Precision/Recall`	19%	2.93
`MoJo`	23%	3.69
`EdgeSim`	9%	1.81
`MeCl`	5%	0.69

Table 2. Similarity Measurement Variance for `linux`

- The data from the study indicates that the average values for the $MeCl$ and $EdgeSim$ measurements tend to be larger than the Precision/Recall and $MoJo$ measurements. Also, the $MeCl$ measurement is very high (almost always over 90%) for all of the systems examined. This result could be related to using the Bunch clustering technique, as the optimization approach used by Bunch tries to minimize coupling between subsystems. Thus, we expect minimal changes in the interedges between successive clustering runs. As future work we suggest that this study be repeated using multiple clustering approaches to investigate if the $MeCl$ and $EdgeSim$ measurements remain high.

It should be noted that MoJo numbers did not change when considering isomorphic modules. We used an implementation of MoJo that was provided by Tzerpos, but because we did not have access to the MoJo source code, we were unable to treat the isomorphic modules differently.

6. Summary

The primary goal of this work was to investigate the importance evaluating software clustering results. We examined measurements based on similarity and distance, which have been used to evaluate software clustering results, and raised some concerns about these measurements. To address these concerns, we argued that measuring software clustering results should take into account not only the placement of modules into clusters, but the connectivity of the system modules as well. We next presented two new measurements, *EdgeSim* and *MeCl*, that address these concerns, and conducted a study to investigate the effectiveness of these measurements. We also addressed the importance of recognizing special and isomorphic modules when comparing clustering results, and proposed some ways to deal with these types of modules.

The *MeCl* and *EdgeSim* measurements have been integrated into the Bunch API and Bunch GUI. Bunch can be downloaded from the Drexel Unviersity Software Engineering Research Group (SERG) web page at http://serg.mcs.drexel.edu/bunch.

As followup to this work we would like to investigate the effectiveness of our similarity measurements when comparing decompositions produced by clustering algorithms other then Bunch. We have already taken some of the initial steps towards this goal by creating a framework [14] for software evaluation.

7. Acknowledgements

This research is sponsored by grants CCR-9733569 and CISE-9986105 from the National Science Foundation (NSF). Additional support was provided by the research laboratories of AT&T, and Sun Mircosystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, U.S. Government, AT&T, or Sun Microsystems.

We would like to thank Vassilios Tzerpos for the MoJo calculator.

References

[1] N. Anquetil. A comparison of graphis of concept for reverse engineering. In *Proc. Intl. Workshop on Program Comprehension*, 2000.

[2] N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with hierarchical clustering algorithms as software remodularization methods. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[3] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[4] Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[5] Y. Chen, E. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proc. 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1997.

[6] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.

[7] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, 1999.

[8] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, Aug. 1985.

[9] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, 1999.

[10] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proc. Intl. Workshop on Program Comprehension*, 2000.

[11] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, 1997.

[12] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, Aug. 1999.

[13] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, 1998.

[14] B. S. Mitchell and S. Mancoridis. Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proc. Working Conference on Reverse Engineering*, Oct. 2001.

[15] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[16] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[17] V. Tzerpos and R. Holt. The orphan adoption problem in architecture maintenance. In *Proc. Working Conf. on Reverse Engineering*, 1997.

[18] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clustering. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[19] V. Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proc. Intl. Workshop on Program Comprehension*, 2000.

[20] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. International Conference on Software Engineering*, Aug. 1999.