Comparison of Clustering Algorithms in the Context of Software Evolution

Jingwei Wu, Ahmed E. Hassan, Richard C. Holt
School of Computer Science
University of Waterloo
Waterloo ON, Canada
{j25wu,aeehassa,holt}@uwaterloo.ca

Abstract

To aid software analysis and maintenance tasks, a number of software clustering algorithms have been proposed to automatically partition a software system into meaningful subsystems or clusters. However, it is unknown whether these algorithms produce similar meaningful clusterings for similar versions of a real-life software system under continual change and growth.

This paper describes a comparative study of six software clustering algorithms. We applied each of the algorithms to subsequent versions from five large open source systems. We conducted comparisons based on three criteria respectively: stability (Does the clustering change only modestly as the system undergoes modest updating?), authoritativeness (Does the clustering reasonably approximate the structure an authority provides?) and extremity of cluster distribution (Does the clustering avoid huge clusters and many very small clusters?).

Experimental results indicate that the studied algorithms exhibit distinct characteristics. For example, the clusterings from the most stable algorithm bear little similarity to the implemented system structure, while the clusterings from the least stable algorithm has the best cluster distribution. Based on obtained results, we claim that current automatic clustering algorithms need significant improvement to provide continual support for large software projects.

1 Introduction

A well documented architecture can improve the quality and maintainability of a software system. However, many existing systems often do not have their architecture documented. Moreover, the documented architecture becomes outdated and the system structure decays as rapid changes are made to the system to meet market pressure [7, 18]. A high rate of turnover among developers makes the situation even worse. Maintenance of architectural documentation is one of many problems that confront today's large software projects. Software clustering holds out the promise of helping in this task [13, 24].

Software clustering refers to the decomposition of a software system into meaningful subsystems. It plays an important role in understanding legacy software systems [15], assisting in their architectural documentation [3, 13], and supporting their re-modularization[22, 27]. For example, a misplaced procedure or file can be automatically discovered and relocated to the proper subsystem to reduce unexpected dependencies and prevent the decay of the architecture [12].

Ideally, a software clustering algorithm should be automated to provide continual support throughout the lifetime of a large software system. More importantly, the algorithm must produce meaningful clusterings in a stable manner. To be meaningful, the algorithm must produce clusterings that can help developers understand the system. A clustering algorithm which places all source files from a system into two large clusters will not be helpful to developers analyzing the code. To be stable, the algorithm must produce clusterings that do not vary widely from one version of the system to the next. An algorithm which produces clusterings that vary widely even though no major code restructuring occurred between versions is likely not to be used by developers.

In this paper, we will compare six clustering algorithms by applying them to subsequent versions of five large open source systems. Our objective is to clarify to what extent software clustering algorithms can be used to support remodularization and architectural documentation during the life cycle of a large software system. We use three criteria to evaluate the usefulness of these algorithms: *C1*. When a system changes modestly, the clustering produced by an algorithm should also change modestly; *C2*. An automatically produced clustering should approximate the clustering produced by an authority (e.g., the architect); and *C3*. Automatically produced clusters should generally not be either huge (i.e., containing hundreds of source files) or tiny (i.e., containing very few source files).

The rest of this paper is organized as follows: Section 2

Systems	Prog. Lang.	# Versions	# Source Files	Size (KLOC)	Graph Data	Raw Data	Extraction Time
Ruby	C	73	90 – 261	74 – 187	6.5MB	1.0 GB	21 mins
KSDK	C/C++	70	21 – 1156	3 – 263	82.5MB	2.5 GB	52 mins
OpenSSL	C	73	593 – 845	164 – 278	74.4MB	4.2 GB	57 mins
PGSQL	C	73	771 – 947	182 – 519	99.1MB	4.5 GB	59 mins
KOffice	C/C++	70	1358 – 3266	272 – 962	1235.5MB	42.0 GB	325 mins

Table 1: Properties of five target systems from 1999 to 2004. The Raw Data refers to CTSX output and Graph Data refers to lifted graphs. Due to a CVS update problem, the last three monthly versions of KSDK and KOffice in 2004 are not available.

provides an overview of five systems chosen for the experimentations. Section 3 describes the experimental setup in the context of software evolution. Section 4 introduces a simple ordinal measure for comparing a number of data series. Section 5 describes the experimental results obtained using ordinal evaluation techniques. The results show that the studied clustering algorithms exhibit distinct characteristics in terms of stability, authoritativeness, and extremity. Section 6 discusses several interesting observations. Section 7 considers related work and Section 8 concludes this paper.

2 Target Systems

To evaluate software clustering algorithms, we chose to apply them to a number of real-life systems, all of which are open source software and hence available for study. Table 1 gives a summary of key properties of five target systems. They represent distinct application domains and have gone through a number of years of development. We now briefly describe these systems.

1. **KSDK** is a software development kit for the K Desktop Environment (KDE) [8]. It offers powerful framework support for various kinds of KDE applications.
2. **KOffice** is a free, integrated office suite for KDE. This suite contains 12 major applications: KWord, KChart, KSpread, KPresenter, Kivio, Karbon14, Krita, Kugar, KPlato, Kexi, KFormular, and Filters [9].
3. **OpenSSL** is a cryptography toolkit implementing the Secure Socket Layer (SSL) and Transport Layer Security (TLS) network protocols and related cryptography standards [17].
4. **PostgreSQL** is a free, large, SQL compliant object Relational Database Management System (DBMS) [19] that originated at the University of California at Berkeley in 1996. The rest of this paper will refer to PostgreSQL as PGSQL.
5. **Ruby** is an interpreted scripting language designed for quick and easy object oriented programming [21]. It has many convenient features for text file processing and system management.

3 Experimental Design

Figure 1 illustrates the design of our experiments. We begin with the repository (stored in CVS) of a target system (such as PGSQL) that we use to evaluate clustering algorithms. The repository contains successive versions of the target system. We retrieve monthly versions of the source code, called here V_1, V_2, and etc. In step 1 (program extraction), we extract a directed graph G_i from each version V_i. In this graph, each node represents a file in the target system. Each edge represents a static dependency (such as a reference to a variable or a data type, a call to a function, or a call to a macro) from one file to another. In step 2 (software clustering), we run a number of clustering algorithms, called A, B, etc., on each graph G_i, producing clusterings C_{iA}, C_{iB}, etc. Finally, in step 3 (comparative analysis), we use a set of criteria to evaluate the algorithms. We now discuss these steps in more detail.

3.1 Program Extraction

In the first step we extract graphs from 70 to 73 monthly versions (see Table 1) of the target systems, as illustrated in Figure 1. Extraction can be a difficult task due to the size of the systems, the number of versions, and irregularities present in the source code (e.g., syntactical errors, incomplete code, language dialects and software configurations). To circumvent these difficulties, we developed a lightweight C/C++ source code extractor called CTSX.

CTSX is built on CTags [5] and CScope [4]. Both tools are efficient and robust in handling software systems up to millions of lines of code. CTSX uses CTags to extract program entities (e.g., functions, variables, and data types) and CScope to retrieve references (e.g., function calls) to entities found by CTags. These references are then “lifted” to the level of source files to produce directed edges between nodes (files) in the extracted graph. This lifting to the level of files greatly decreases the size of the graph, which is an important consideration when dealing with many versions of a large target system.

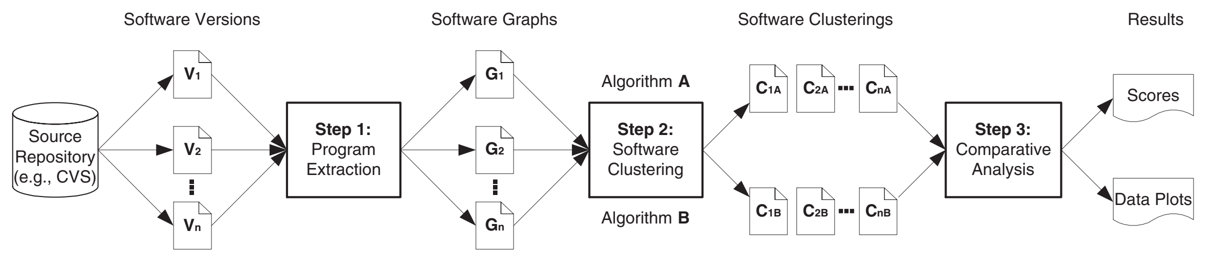


Figure 1: Comparative analysis of software clustering algorithms in the context of software evolution

3.2 Software Clustering

In the second step of the experiment (see Figure 1) a set of clustering algorithms is run over graphs extracted from each monthly version of the target system. A clustering is created for each version by each algorithm.

We chose a set of six clustering algorithms based on their availability and their discussion in the literature. We limited our choice to available implementations that run in batch mode since we needed to run each of these algorithms many times as they were applied to different target systems over a large number of versions. We selected software clustering tools produced by researchers Anquetil, Mancoridis, and Tzerpos. Anquetil designed a hierarchical clustering algorithm suite, which offers a selection of association and distance coefficients as well as update rules [1]. Four algorithms from this suite were chosen and given names of the form of CL** and SL** where ** encodes the parametrization, as described below. Mancoridis and Mitchell provided us with their Bunch suite [13]. Tzerpos provided us his algorithm for comprehension driven clustering (ACDC) [24]. We now give a brief description of the six algorithms.

1. **CL75** is an agglomerative clustering algorithm based on the Jaccard coefficient and the complete linkage update rule [1]. The cut-point height for the dendrogram is set to 0.75. The higher the cut-point, the smaller the number of clusters in the resulting dendrogram.
2. **CL90** has the same configuration as CL75 except that its cut-point height is set to 0.90 [1].
3. **SL75** is an agglomerative clustering algorithm based on the Jaccard coefficient and the single linkage update rule. The cut-point height is set to 0.75 [1].
4. **SL90** has the same configuration as SL75 except that its cut-point height is set to 0.90 [1].
5. **ACDC** is an algorithm based on program comprehension patterns and it attempts to recover subsystems that are commonly found in manually-created decompositions of large software systems [24].
6. **Bunch** provides a suite of algorithms that include Hill Climbing, Exhaustive, and Genetic Algorithms [6, 13]. We tried three different hill climbing configurations: NAHC (nearest ascend hill climbing), SAHC (shortest ascend hill climbing), and a customized configuration with the minimum search space greater than 55% and the randomized proportion of the search space equal to 20%. This paper will only discuss the third configuration since NAHC and SAHC is not much different from it based on the results we obtained. For brevity, it will be referred to as Bunch in the rest of this paper.

3.3 Comparative Analysis

In the third step of the experiment (see Fig 1), we compare the clustering algorithms based on three criteria, *C1*, *C2* and *C3*, which we have previously mentioned. We now detail the three criteria:

C1 Stability

Similar clusterings should be produced for similar versions of a software system. This criterion emphasizes the persistence of the clustering structure of successive versions of an evolving software system. Under conditions of small and incremental change between consecutive versions, an algorithm should be stable, i.e., it should produces similar clusterings to prior months.

C2 Authoritativeness

Clusterings produced by an algorithm should resemble clustering from some authority. An authoritative clustering may be produced by a (human) architect. It may also be derived from the directory structure of the target system. In the latter case, authoritativeness can be seen as adherence to the source folder structure.

C3 Extremity of Cluster Distribution

The cluster size distribution of a clustering should not exhibit extremity. In particular, a clustering algorithm should avoid the following two situations: (1) the majority of files are grouped into one or few huge clusters (sometimes called *black holes*), and (2) the majority of clusters are singletons (forming what are sometimes called *dust clouds*).

Based on these criteria, we conducted three comparisons to examine how the algorithms chosen for this study are different from one another. The detailed information on these comparisons will be presented in Section 5.

4 A Simple Ordinal Measure

To support our analysis of clustering algorithms over consecutive versions of a target software system, we created an ordinal measure for ranking a number of data series. A data series refers to a sequence of quantitative values, for example, <1,2,3,4>.

For series DS_i and DS_j, we define $Above(DS_i, DS_j)$ and $Below(DS_i, DS_j)$ as:

$$Above(DS_i, DS_j) = \frac{|\{n \mid DS_i[n] > DS_j[n], 1 \le n \le |DS_i|\}|}{|DS_i|}$$

$$Below(DS_i, DS_j) = \frac{|\{n \mid DS_i[n] < DS_j[n], 1 \le n \le |DS_i|\}|}{|DS_i|}$$

If we think of DS_i and DS_j as lines of points, the function $Above$ denotes the proportion of the line formed using DS_i above the other line formed using DS_j. A similar explanation can be given to $Below$. We say that DS_i is above DS_j if $Above(DS_i, DS_j) > Above(DS_j, DS_i)$.

Given K data series, DS_1, DS_2, ..., DS_K, we have the following equations for measuring the relative position of a particular DS_i with regard to all of the K data series.

$$Above(DS_i) = \sum_{j=1}^{K} Above(DS_i, DS_j) \tag{1}$$

$$Below(DS_i) = \sum_{j=1}^{K} Below(DS_i, DS_j) \tag{2}$$

In the next section, we will adapt these two equations to obtain different orderings of clustering algorithms in terms of stability, authoritativeness, and extremity.

5 Experiments and Results

This section describes the results we obtained.

5.1 Stability Comparison

Tzerpos defines a stability measure based on the ratio of the number of "good" clusterings to the total number of clusterings produced by a clustering algorithm [25]. He considers a clustering obtained from a perturbed version of the system to be good if the MoJo dissimilarity between that clustering and the one obtained from the original version of the system is not greater than 1% of the total number of resources (i.e., source files). We feel that it is difficult to determine a proper threshold in the context of real-life software evolution, and 1% seems too optimistic in reality.

We instead used our ordinal measure (defined in Section 4) to derive a stability ordering of an algorithm in comparison to other algorithms. In addition, we evaluated an algorithms' stability with three ordinal values: High, Medium, and Low. We referred to the latter method as the *HML ordinal evaluation*.

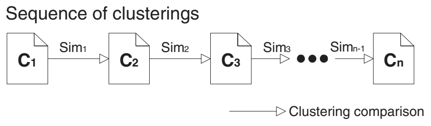


Figure 2: Intra-sequence clustering comparison

Before presenting our stability measure, we explain how to create similarity sequences using intra-sequence comparisons of consecutive clusterings as shown in Figure 2. Given a sequence of n clusterings, we have a sequence of $n-1$ similarity values denoted as $<Sim_1, Sim_2, ..., Sim_{n-1}>$. For a target system, a comparison of its two consecutive clusterings yields a similarity or dissimilarity value depending on the selection of similarity measures. For example, MoJo produces dissimilarity values [23], while EdgeSim produces similarity values [14]. Since some similarity measures like MoJo are not reflexive, we need to perform directional comparison in the same direction as the target system evolves. This is desirable if we think of a clustering as an entity moving along its own lifetime path from inception to death. We choose MoJo for its simplicity. Correspondingly, we have Sim_i defined to be $MoJo(C_i, C_{i+1})$ where $1 \le i \le (n-1)$.

A measure for relative stability

Given a target system S and K clustering algorithms, A_1, A_2, ..., and A_K, we use $MJS(A_i, S)$ to denote the sequence of MoJo values calculated based on intra-sequence comparison over the sequence of clusterings obtained by A_i from S. A relative score of A_i over A_j is defined as:

$$Score_{MJ}(A_i, A_j, S) = Below(MJS(A_i, S), MJS(A_j, S))$$

The relative score of A_i over all the K clustering algorithms is define as:

$$Score_{MJ}(A_i, S) = Below(MJS(A_i, S)) \tag{3}$$

where $Below$ is given in Equation 2 and MJ stands for MoJo. We say that A_i is more stable than A_j with regard to system S if $Score_{MJ}(A_i, A_j, S) > Score_{MJ}(A_j, A_i, S)$. When comparing K algorithms, we use Equation 3 to calculate a stability score for each algorithm. Using the K obtained scores, we can determine a relative stability ordering of K algorithms. The greater the score, the more relatively stable an algorithm.

A HML-based ordinal measure

Based on a data series of system growth SG, we create two new data series: $SG10$ and $SG30$. $SG10$ is proportional to SG and accounts for 10% of the number of source files with regard to each data point in SG, and correspondingly $SG30$ accounts for 30%. The three data series $SG10$, $SG30$, and SG divide the area below SG into three smaller regions. If the MoJo series $MJS(A_i, S)$ has at least 80% of its data points below $SG10$, we say the stability of A_i is H (High). If $MJS(A_i, S)$ has at least 80% below $SG30$, A_i has a score of M (Medium). Otherwise, A_i has a score of L (Low).

$$HML_{MJ}(A_i, S) = \begin{cases} H & \text{if } Score(MJS(A_i,S), SG10) \geq 0.8 \\ M & \text{elsif } Score(MJS(A_i,S), SG30) \geq 0.8 \\ L & \text{otherwise} \end{cases} \tag{4}$$

With this measure, we are actually measuring how stable an algorithm is with regard to a particular target system rather than simply stating which one is more stable than another. For example, if a clustering algorithm has a score of H, it means that the number of MoJo operations (i.e., moves and joins) is less than 10% of the total number of sources in the target system for 80 out of 100 transformations of pair-wise consecutive clusterings produced by that algorithm.

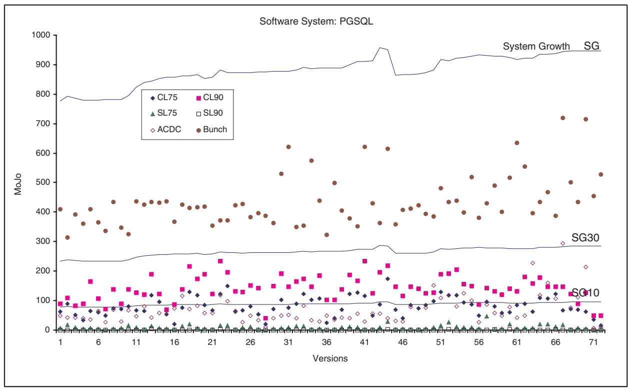


Figure 3: Stability comparison wrt PGSQL

Plots of MoJo series

Figure 3 shows a plot of data series of MoJo that were calculated for each of the six clustering algorithms with respect to PGSQL. From the figure, we can see that the MoJo values associated with Bunch are greater than the corresponding MoJo values associated with the other algorithms. Recalling MoJo is a dissimilarity measure, we can tell that Bunch is less stable than the other five algorithms. We examined the MoJo series for NAHC and SAHC. They exhibit similar behavior as the ones shown for Bunch in Figure 3. We omit them from this paper for the sake of brevity.

The data series associated with algorithms CL90, CL75, and ACDC reside in the middle of Figure 3. They appear intertwined at some locations. We can roughly see that ACDC appears more stable than CL75 and CL90. SL90 and SL75 are at the bottom of the figure. Clearly, they are the most stable algorithms.

	KSDK	KOffice	OpenSSL	PGSQL	Ruby	ALL
Bunch	0.07	0.00	0.00	0.00	0.01	0.02
CL90	1.00	1.06	1.40	1.08	1.35	1.18
CL75	2.01	1.97	2.46	2.29	1.93	2.14
ACDC	2.49	3.35	2.08	2.63	2.56	2.62
SL75	3.30	3.62	3.82	4.00	3.35	3.62
SL90	4.26	5.00	4.51	5.00	3.96	4.55

Table 2: Relative stability scores obtained using MoJo. The greater the score, the more relatively stable the algorithm.

	KSDK	KOffice	OpenSSL	PGSQL	Ruby	ALL
Bunch	L	L	L	L	L	L
CL90	M	M	M	M	M	M
CL75	M	M	M	M	M	M
ACDC	M	H	M	M	H	H
SL75	H	H	H	H	H	H
SL90	H	H	H	H	H	H

Table 3: HML-based stability scores obtained using MoJo

Ordinal evaluation

In Figure 3, when data series are highly intertwined it can be difficult to obtain an ordering of several clustering algorithms based on visual perception. A quantitative method is needed. Using Equation 3, we calculated relative stability scores for the six studied algorithms with regard to each target system as well as the concatenation of all five target systems. When comparing algorithms with regard to a concatenation of different systems, we actually concatenated the similarity series from all those systems for each clustering algorithm and then we applied Equation 3. The scores are given in Table 2. Those obtained using concatenation are listed in the `ALL` column and they are used to determine the overall stability ordering.

All the scores in Table 2 seem to tell the same story. In the direction of increasing stability, the chosen algorithms can be ordered as: Bunch, CL90, CL75, ACDC, SL75 and SL90. There is an exception with regard to OpenSSL where CL75 appears to be more stable than ACDC since the former scored 2.46 but the latter scored only 2.08 (see Table 2). The cells shaded in light gray indicates this disagreement.

Similarly, we calculated an HML ordering for the algorithms. The obtained scores are given in Table 3. From the table, we can see that SL90, SL75 and ACDC are highly stable algorithms. While Bunch is not.

To rule out the possibility that the obtained stability ordering is biased due to use of MoJo, we measured the ordering of these algorithms using other similarity measures, which include EdgeMoJo [26], EdgeSim and MeCl [14]. We found that EdgeSim strongly agrees with MoJo but the other two measures slightly disagree with MoJo on the ordering of CL75 and ACDC and rank CL75 as relatively more stable than ACDC. The experimental results obtained using other similarity measures are reported in [28].

Given that our results were obtained using evolution data from several hundred versions of multiple real-life systems and further verified by multiple similarity measures, we can safely state that the following stability order factually exists.

Low	Medium	High
Bunch	CL90 CL75	ACDC SL75 SL90

5.2 Authoritativeness Comparison

Unfortunately, a stable algorithm may not produce meaningful clusterings at all. For example, at one extreme, an algorithm that produces only singleton clusters is stable over time but it is not useful in practice. At the other extreme, an algorithm that groups all source file into one notoriously large clustering is obviously stable but not meaningful at all. A clustering algorithm of practical use should produce clusterings similar to authoritative decompositions of a software system by experienced software engineers [3]. However, an agreed upon architecture or authoritative decomposition of a complex software system often does not exist. This makes it difficult to do authoritativeness comparison.

In an attempt to perform authoritativeness analysis, we used a simple technique to create authoritative clusterings. Our technique comprises four steps: (1) create the subsystem hierarchy based on the directory structure; (2) relocate every header file (`.h`) to the subsystem that directly contains the related implementation file (`.c`); (3) merge a subsystem with its parent if it contains less than five files; (4) create a flat clustering with each subsystem in the remaining hierarchy as a cluster. For well structured software systems like KOffice and PGSQL, this technique is likely to produce a clustering which conforms to the mental model of the developers of the system. It can be easily automated to cluster a large number of versions.

We performed inter-sequence comparisons of clusterings from two parallel sequences. One sequence comprises clusterings produced by the algorithm in analysis, and the other contains authoritative clusterings obtained using the technique described above. Figure 4 shows how inter-sequence comparisons are done over time. The similarity Sim_i is calculated as $MoJo(C_i, C_{iA})$. Based on the obtained similarity series $<Sim_1, Sim_2, ..., Sim_n>$, we carried out authoritativeness comparison using the same scoring methods for stability analysis (see Equations 3 and 4).

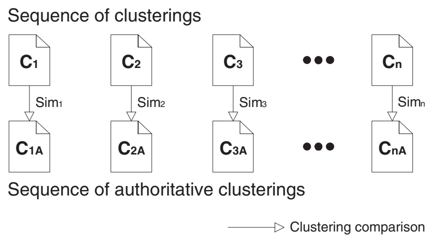


Figure 4: Inter-sequence clustering comparison

Plots of MoJo series

Figure 5 shows the MoJo series we obtained from PGSQL. At the top of the figure, there are two MoJo series associated with SL75 and SL90. Their data values are almost equal to the number of source files. This fact indicates that nearly every source file in PGSQL needs to be moved or joined in order to transform the clustering produced by SL75 and SL90 to the corresponding authoritative clustering. Given that PGSQL is well structured[1], we can tell that SL75 and SL90 produce clusterings bearing little similarity to the implemented structure of PGSQL.

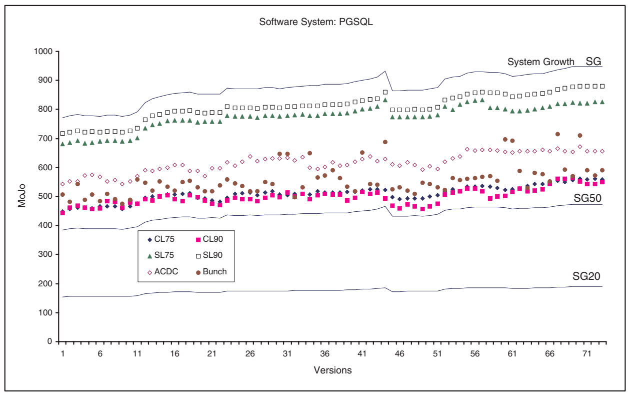


Figure 5: Authoritativeness comparison wrt PGSQL

The MoJo series for ACDC and Bunch reside in the middle of Figure 5, with the former staying mostly above the latter. It shows that ACDC is slightly less authoritative than Bunch. However, none of them produces clusterings resembling the implemented structure of PGSQL. Roughly speaking, 60-70% of all the source files in PGSQL need to be moved or joined in order to transform the clusterings produced by Bunch or ACDC to their authoritative counterparts. This may be due to the fact that clusterings produced by Bunch or ACDC are too coarse (containing few clusters) than their corresponding authoritative clusterings.

The most authoritative algorithms, CL75 and CL90, are at the bottom of Figure 5. Their data series are intertwined

[1]The PostgreSQL project has six Core Members (architects) who have strict control over the source code structure [16].

	KSDK	KOffice	OpenSSL	PGSQL	Ruby	ALL
SL90	0.07	0.00	0.00	0.00	1.85	0.39
SL75	1.06	1.00	1.00	1.00	1.29	1.07
ACDC	2.79	2.06	2.11	2.12	1.55	2.12
Bunch	3.86	4.03	3.66	2.92	3.26	3.54
CL75	2.61	4.03	4.62	4.16	2.19	3.56
CL90	4.43	3.84	3.56	4.77	4.27	4.18

Table 4: Authoritativeness scores calculated using MoJo. The greater the score, the more the clusterings produced by the algorithm resemble the implemented system structure.

and even overlapped. They are slightly better than Bunch.

We also manually examined the plots for other target systems. They roughly confirmed the above observations. We omit them from this paper for the sake of brevity.

Ordinal evaluation

We provide the obtained authoritativeness scores in Table 4. The greater the score, the more the clusterings produced by the algorithm resemble the implemented system structure, and consequently the more authoritative the algorithm turns out to be. The scores from the ALL column determine an overall ordering of the studied algorithms. The cells colored in light gray indicate the disagreement between the overall ordering and the one obtained with respect to a particular target system. To derive the HML-based ordering, we chose $SG20$ and $SG50$ instead of $SG10$ and $SG30$ to relax the requirements on how closer a clustering should resemble the implemented structure. We found that all the algorithms we studied were ranked Low. For this reason, we omit the obtained HML scores. The final ordering is shown below. None of these algorithms is satisfactory in producing clusterings that approximate the implemented structure of any of the five target systems.

Low
SL90 SL75 ACDC Bunch CL75 CL90

We verified this ordering using three similarity measures (EdgeMoJo, EdgeSim and MeCl), and found that they only disagree with MoJo on the ordering of CL75 and CL90. The experimental results are described in detail in [28].

5.3 Extremity Comparison

We studied cluster distributions over time in an attempt to examine whether a particular clustering algorithm avoids generating huge clusters (black holes) or many very small clusters (dust clouds).

We again base our discussions on PGSQL. Figure 6 displays six bubble charts. In each of these charts, the X axis represents software versions, the Y axis represents the size of the cluster, and the size of the bubble denotes the number of clusters of the same size. For example, the clustering we obtained using SL90 from version 72 of PGSQL has 14 clusters, in which 13 clusters are singletons and one huge cluster comprises 934 source files. This obtained clustering is represented using two bubbles in Fig. 6(b), located at the coordinates (72, 1) and (72, 934) respectively. The larger bubble is 13 times big as the smaller one though the latter (a black hole) represents a cluster of size 934. Though the axes X and Y may not be in consistent scales, but all the bubbles throughout seven figures share the same scale of measurement.

Figure 6 shows cluster distributions for every sixth version in order to avoid the overlapping of bubbles in the X direction. Each algorithm exhibits a distinct cluster distribution pattern. From figures 6(a), 6(b), and 6(f), we see that SL90, SL75, and ACDC tend to produce extreme clusters, either huge or very small. By contrast, Bunch, CL75, and CL90 produce more distributed clusters. However, CL75 has a tendency to generate a relatively large number of singleton clusters as shown in Fig. 6(c).

A measure for non-extreme cluster distribution

In order to quantitatively evaluate the extremity of a cluster distribution, we defined a simple measure called NED (non-extreme distribution). In this experiment, we assumed that any clusters of size less than 5 or greater than 100 are extreme clusters. Such an assumption is reasonable with regard to the five target systems since very few clusters in the obtained authoritative clusterings contain less than five or more than a hundred source files.

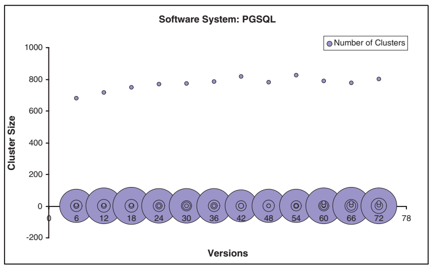
NED is defined as the ratio of the number of source files contained in non-extreme clusters to the total number of source files in the target system. Clearly, the larger the NED value, the better the distribution.

Figure 7 shows several NED data series we obtained for each studied algorithm with regard to PGSQL and KOffice. The NED data series obtained from KSDK, OpenSSL, and Ruby share very similar patterns with those in Figure 7(a). So they were omitted for brevity. From the figure, we can see that Bunch has the best cluster distribution. However, Bunch does not perform as well on KOffice as it does on PGSQL. The SL75 and SL90 algorithms always group the majority of source files into extreme clusters. The NED data series obtained for the other three algorithms mostly remain in the middle of the figure. In Fig. 7(b), the NED series for CL75 and CL90 show a downward trend as KOffice grows in size. An examination of cluster distribution reveals that both algorithms start to form black holes.

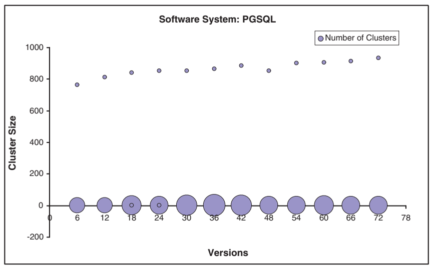
We extend NED to derive a relative ordering and HML ordering of non-extreme cluster distribution. We have two scoring methods defined as follows:

$$Score_{NED}(A_i, A_j, S) = Above(NS(A_i, S), NS(A_j, S))$$

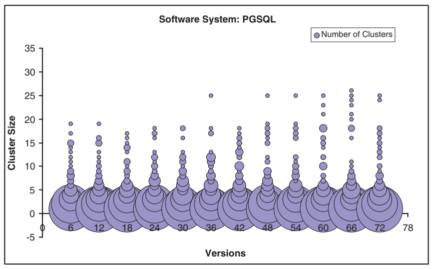
$$Score_{NED}(A_i, S) = Above(NS(A_i, S)) \qquad (5)$$



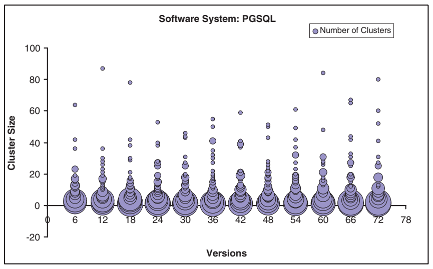
(a) Distributions obtained using SL75



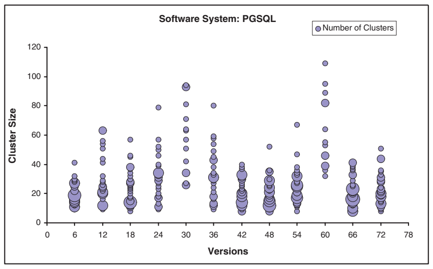
(b) Distributions obtained using SL90



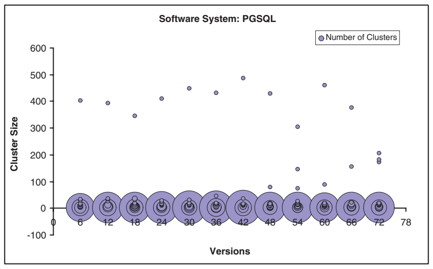
(c) Distributions obtained using CL75



(d) Distributions obtained using CL90

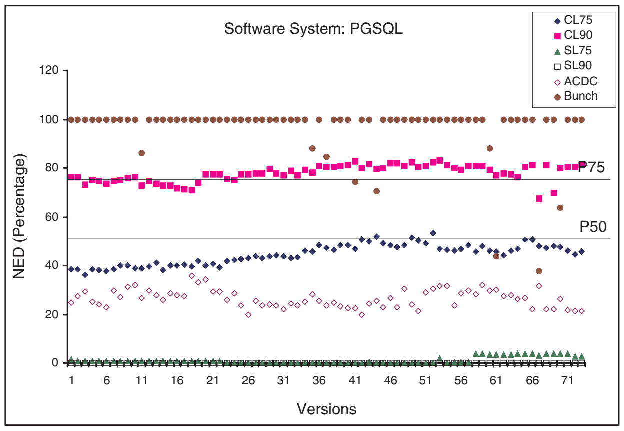


(e) Distributions obtained using Bunch

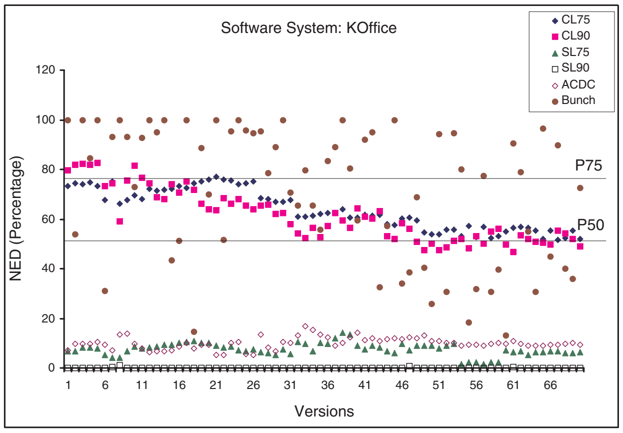


(f) Distributions obtained using ACDC

Figure 6: Bubble chart based distribution comparison of clustering algorithms wrt PGSQL



(a) NED-based distribution comparison wrt PGSQL



(b) NED-based distribution comparison wrt KOffice

Figure 7: NED-based distribution comparison of clustering algorithms wrt PGSQL and KOffice

$$HML_{NED}(A_i,S) = \begin{cases} H & \text{if} \quad Score_{NED}(NS(A_i,S),P75,S) \geq 0.8 \\ M & \text{elsif } Score_{NED}(NS(A_i,S),P50,S) \geq 0.8 \\ L & \text{otherwise} \end{cases} \tag{6}$$

where A_i $(1 \leq i \leq K)$ denotes K algorithms and $NS(A_i, S)$ denotes the NED data series obtained from the clusterings produced by A_i with regard to system S. The $Above$ is defined in Equation 1. We use $P75$/$P50$ to refer to a data series comprising only data value 0.75/0.50. The $Score_{NED}$ evaluates whether a NS is above another NS. If so, it means that the algorithm producing the former NS has a better cluster distribution than the algorithm producing the latter NS. The HML_{NED} evaluates whether an algorithm really has a good cluster distribution or not. If A_i has a score of H, it means that 80% of the clusterings produced by A_i have a NED value greater than 0.75.

Ordinal evaluation

We conducted a similar ordinal evaluation as we did to stability and authoritativeness. For brevity, we only give the final NED-based ordering as follows, omitting the relative scores and HML scores.

Low	Medium	High
SL90 SL75 ACDC CL75	CL90	Bunch

5.4 Summary

Table 5 provides a summary of ordinal evaluation we obtained for each of the studied algorithms. This table serves as the main discussion base in the next section.

Algorithms	Stability	Authoritativeness	Non-extremity
CL75	Medium	Low	Low
CL90	Medium	Low	Medium
SL75	High	Low	Low
SL90	High	Low	Low
ACDC	High	Low	Low
Bunch	Low	Low	High

Table 5: A summary of ordinal evaluation

6 Discussions on the Results

This section discusses some of the obtained experimental results. In particular, we will focus on the following observations.

SL75/SL90, the most stable but the least useful

From the figures 6(a) and 6(b), we can see that SL75 and SL90 tend to merge smaller clusters (including singletons) one by one into one super large cluster. As the height of the cut point increases from 0.75 to 0.90, the number of clusters in the produced clustering decreases, and the size of the super large cluster increases. We can think of the super large cluster as a black hole, which eventually attracts every small cluster surrounding it. The number of source files consumed by the black hole accounts for about 84.8-86.3% of the total number of source files in the case of SL75 and 79.8-98.7% in the case of SL90. This explains why SL75 and SL90 appear more stable than the other four algorithms. Once the majority of the source files are put into one super large cluster, the resulting clustering bears little similarity to the directory-based authoritative clustering. In terms of MoJo, the number of moves and joins is almost equal to the number of source files because nearly every file in the super large cluster needs to be moved to a directory-based authoritative cluster and every singleton cluster needs to be joined. This explains why SL75 and SL90 are the least useful.

A clustering of the majority of source files into one black hole cluster is equivalent to no clustering.

Algorithms in CL less stable than those in SL

Because of the complete linkage update rule, the algorithms from the CL class are known to produce more compact clusters than those from the SL class. Raghavan and Yu have shown that graph theoretic clustering methods that produce more compact clusters are less stable [20]. The stability ordering we obtained agrees with the theoretic results (see Section 5.1).

Bunch, the least stable with the best distribution

In our configuration of Bunch, every time a better neighbor is being sought after, 20% of the entire search space is randomized and at least 55% is examined in order to find a better MQ (Model Quality) [13]. Consequently, the hill climbing algorithm in Bunch does not perform in a deterministic way and it even produces different decompositions for the same input graph. By contrast, the other algorithms use less randomization.

Bunch (including NAHC and SAHC) produces the best cluster distribution. It favors neither a group of very small clusters nor an overwhelmingly large cluster. This can be seen from Figure 6(e).

No gain for ACDC from overusing program patterns

Since ACDC is a program pattern based algorithm [24], we had expected that it would produce clusterings more similar to the authoritative clusterings we obtained. However, it did not meet our expectations.

ACDC internally has an upper limit for the size of the obtained cluster. By default this limit is set to 20. However, Figure 6(f) shows that ACDC produces both very large clusters and very small ones. To investigate what goes wrong in ACDC, we manually examined the clusterings obtained from versions 12, 24, 36, 48, 60, and 72 of PGSQL. We found two main problems regarding the use of program patterns in ACDC.

1. The body-header pattern [24] results in the grouping of the interface file (`.h`) and its implementation file (`.c`).

However, clustering of the obtained body-header pairs is not done sufficiently. This results in a relatively large number of small clusters of size 2.

2. The subgraph dominator [24] is somewhat overused. For example, the file `utility.c` located in the directory `tcop` was recognized by ACDC as a dominator. After we removed `utility.c` from the input graph, ACDC cut the size of the largest cluster approximately by 50%. After we further removed 2 or 3 dominators from the input graph, no cluster in the obtained clustering contains more than 100 source files. We suspect that ACDC has internal defects in processing subgraph dominators.

7 Related Work

Current research is mainly focused on designing algorithms that partition large software systems into meaningful subsystems. For example, early work by Belady *et al.* identified automatic clustering as a means to produce high-level views of large software systems [2]. Bunch has evolved to become a suite of algorithms to fit into various contexts in reverse engineering [6, 13]. In addition, a variety of evaluation frameworks have been proposed to evaluate the quality of clustering techniques [10, 11]. However, clustering techniques have not been sufficiently tested and evaluated against lifetime versions of systems from diverse domains. In particular, the stability of clustering algorithms has not attracted much attention.

Tzerpos and Holt examined the stability of a number of clustering algorithms by means of generating randomly perturbed versions of an example system and measuring differences between the obtained clusters and the one obtained from the original version of the system [25]. In this paper, we argue that random perturbation of a fixed size system is insufficient in simulating how changes occur in a real world software system, since changes to software systems rarely occur in a random fashion and most software systems are continuously growing in size. To be faithful to the reality, we conducted stability comparison on evolution data extracted from several target systems. In addition, our work shows that stability comparison of clustering algorithms should be augmented with the analysis of meaningfulness (adherence to authority and non-extreme distribution). Otherwise, the results of stability comparison could be misleading (i.e. seeing only one side of a coin).

8 Conclusions

In this paper we have asked the question of how useful clustering algorithms might be in large software systems undergoing evolutionary change.

In an attempt to gain insight into this question, we investigated the effectiveness of six clustering algorithms which represent a range of clustering techniques and which are supported by available batch implementations. We selected five open source software systems, each having roughly 70 monthly versions. These systems represent a range of applications, so they can be expected to be a reasonable testing base for clustering algorithms.

We proposed three criteria to evaluate the usefulness of software clustering algorithms. We ran experiments to measure how well each clustering algorithm satisfies these criteria. Table 5 provides a brief view of the overall results we obtained. It indicates that:

- On the stability criteria (Are successive versions of a target system given similar clusterings?), three algorithms (SL75, SL90 and ACDC) are ranked as having high quality, two (CL75 and CL90) as medium and the remaining algorithm (Bunch) as low.
- On the authoritativeness criteria (Do the clusterings reasonably approximate the implemented structure of the target system?), all six algorithms are ranked as having of low quality.
- On the extremity criteria (Are non-extreme cluster distributions normally produced?), Bunch is ranked as high, CL90 as medium, then CL75, ACDC, CL75 and SL90 as low.

Although SL75 and SL90 ranked high on stability, it appears that this is largely due to the fact that they repeatedly produced black holes (overly large clusters) or dust clouds (many too small clusters), as indicated by their low NED ranking. This suggests that in practice these two algorithms may not be that helpful.

The fact that all six algorithms are ranked low on authoritativeness suggests that they may not be mature enough for use in production on large systems undergoing evolutionary change. However, it is also possible that our technique for generating authoritative clusterings is biased toward the as-implemented structure of the target system.

These results are discouraging, suggesting that, for large systems, such as we used as a basis for our testing, and for the criteria we chosen, more work needs to be done before these clustering algorithms are ready to be widely adopted. However, it may be that such algorithms are useful in less stringent environments. For example, these algorithms may be useful in a reverse engineering exercise, by producing a basic partitioning of a particular version of a target system, thus eliminating a significant amount of manual effort.

Our hope is that our results spur on further efforts both to create/improve automated clustering algorithms and to subject these algorithms to empirical evaluations such as we have reported in this paper.

References

[1] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 235–255, Atlanta, Georgia, USA, October 1999.

[2] L. A. Belay and C. J. Evangelisti. System partitioning and its measure. *The Journal of Systems and Software*, 2:23–29, 1981.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, pages 555–563, Los Angeles, California, May 1999.

[4] Cscope. Website, 2004. http://cscope.sourceforge.net.

[5] Ctags. Website, 2004. http://ctags.sourceforge.net.

[6] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the International Conference on Software Technology and Engineering Practice*, pages 73–91, Pittsburgh, PA, September 1998.

[7] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, January 2001.

[8] KDE. K Desktop Environment. Website, 2004. http://www.kde.org.

[9] KOffice. Website, 2004. http://www.koffice.org.

[10] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 201–210, Limerick, Ireland, June 2000.

[11] A. Lakhotia and J. M. Gravley. A unified framework for expressing software subsystem classification techniques. *The Journal of Systems and Software*, 36(3):211–231, March 1997.

[12] R. Lange and R. W. Schwanke. Software architecture analysis: A case study. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 19–28, Trondheim, Norway, June 1991.

[13] 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 the 15th International Conference on Software Maintenance*, pages 50–59, Oxford, England, September 1999.

[14] B. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 744–753, Florence, Italy, November 2001.

[15] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to system structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[16] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85, Orlando, Florida, May 19-20 2002.

[17] OpenSSL. Website, 2004. http://www.openssl.org.

[18] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Sorrento, Italy, May 16-21 1994.

[19] PostgreSQL. A free open source database system. Website, 2003. http://www.postgresql.org.

[20] V. V. Raghavan and C. Yu. A comparison of the stability characteristics of some graph theoretic clustering methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3.

[21] Ruby. Website, 2004. http://www.ruby-lang.org.

[22] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, Austin, Texas, United States, May 1991.

[23] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 187–193, Atlanta, Georgia, USA, October 1999.

[24] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension driven clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 258–267, Brisbane, Australia, November 2000.

[25] V. Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 211–218, Limerick, Ireland, June 2000.

[26] Z. Wen and V. Tzerpos. Evaluating similarity measures for software decompositions. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 368–377, Chicago, IL, USA, September 2004.

[27] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 33–43, Amsterdam, The Netherlands, October 1997.

[28] J. Wu and R. C. Holt. Do clustering similarity measures agree with each other? 2005. To Submit.