

Experimental Study and Comparison of Clustering Algorithms for Function Restructuring

Chung-Horng Lung Xia Xu
Department of Systems and Computer Eng.
Carleton University, Ottawa, Canada
{chlung, xiayu}@sce.carleton.ca

Marzia Zaman
Cistel Technology
Ottawa, Canada
marzia@cistel.com

Anand Srinivasan
EION Inc.
Ottawa, Canada
anand@eion.com

Abstract

The structure of a function, procedure, or method may not be well designed or may deteriorate over time due to evolution. As a result, the function may contain multiple activities or the cohesion may become very low, thus it becomes difficult to understand and maintain those functions. This paper deals with restructuring of program statements inside of a function. The objective is to support the restructuring of a large, low-cohesion functions into multiple higher cohesive functions. This paper adopts the hierarchical agglomerative clustering (HAC) method. We apply the method to program restructuring of industrial network software to support evolution. In addition, the paper reports a comparative study on the application of three different HAC algorithms, SLINK (single linkage), CLINK (complete linkage), and UPGMA (unweighted pair-group method using arithmetic averages). The results reveal that the approach provides useful information to facilitate the designers to reorganize the structure of complicated or large functions. In general, UPGMA generates better clusters, especially when the software size (LOCs) or cyclomatic complexity is high. SLINK is more inconsistent. CLINK, on the other hand, usually does not work well.

1. Introduction

Software evolves over time primarily due to changes in requirements and technologies. As a result, huge amount of effort is spent in maintenance and evolution. Software evolution usually accounts for more than 60% of total software costs [29]. In today's highly competitive era, software development is often driven by tight schedules. Hence, software designers often emphasize the functional quality attributes of a system. Even if a software product is well designed, the code is modified over time in response to the changing needs of customers. As a consequence, its original structure gradually drifts and quality degrades. The program becomes difficult to understand and change. As a result, it is often costly to maintain.

Müller, et al., indicate that 50-90% of software evolution work focuses on program comprehension or

understanding [25]. Program understanding could be at various levels, including architecture, design, and code. A large or poorly structured function usually involves multiple activities or has low functional cohesion, which makes the program difficult to understand and modify. Program restructuring or refactoring [8] can transform these functions to functions that are better organized and easier to understand, without changing their behavior. The new functions will usually be higher quality and less costly for further evolution. More importantly, a desirable restructuring should achieve high cohesion and low coupling.

Different approaches have been proposed to solve the problem of large or poorly structured functions. Among them, clustering [7,27] is a relatively new method, although many researchers in software restructuring have addressed the method in general. This paper uses the numerical taxonomy, or hierarchical agglomerative clustering (HAC), approaches. For HAC, two important factors are considered: (i) measurement of similarity and (ii) selection of a clustering algorithm, which is the focus of this paper. The first factor, measurement of similarity, is examined briefly.

There are four basic types of HAC algorithms – SLINK (single linkage), CLINK (complete linkage), weighted-average linkage, and unweighted-average linkage or UPGMA (unweighted pair-group method using arithmetic averages) [16]. Unfortunately, there is no clear separation on how to select a method in a particular application and research varies. This paper conducts a comparative study of SLINK, CLINK, and UPGMA in function restructuring at the code level (Weighted-average method is not included since it is not commonly used [27]). The objective is to provide practical experience and empirical results based on our extensive study of functions in research papers [4,5,13,14]; students assignments in operating systems courses; and industry software of a C code parser program and a network protocol software system. Designers who developed the functions validated the results for the industrial software.

The paper is organized as follows: Section 2 reviews related work on clustering. Section 3 presents an overview of the clustering technique and the method

adopted for this research. Section 4 provides the results of the experiment. Section 5 presents some empirical lessons and section 6 concludes the paper.

2. Related Work

2.1. Restructuring at Function Level

There has been extensive related research on program restructuring at the function level. Kim and Kwon [13] present a method of restructuring a poorly structured module. The approach applies program slicing to extract processing blocks and identify multi-function modules. The method uses module strength as a criterion to decide how to restructure the program. The processing blocks refer to tightly coupled sub-modules, similar to the data slices in [6], in which a slice is a group of data tokens that contribute to a particular output variable in terms of data and control dependence. Based on code implementation, module strength is defined in terms of the level of sharing between processing blocks.

Kang and Bieman [11,12] have introduced a method to restructure modules during the design or maintenance phase. The authors define the input/output dependence graph (IODG) of a module, similar to the variable dependence graph (VDG) in [17], to model the data dependence and control dependence relationship between input and output components of a module. They also define association-based design-level cohesion (DLC) measure, slice-based DLC and functional cohesion (FC) measures. Cohesion measures presented in these papers only consider dependence information between input and output components and do not reflect code fragments that are not related to the output components.

Lakhotia and Deprez [14,15] use tuck transformation to restructure programs by breaking large functions into small functions. Tuck includes three transformations: wedge, split, and fold. A wedge is a subset of statements in a slice, which contains related computations. After a wedge is formed, it is split from the rest of the code and folded into a new function. The paper uses a rule-based approach proposed in [17] to compute pair-wise cohesion between variables in the function as a criterion of restructuring. The empirical study in the paper [14] shows that the approach has some limitations for industrial applications.

Lung and Zaman [19] apply the HAC concept to function restructuring and demonstrate how to restructure a low-cohesive function into high-cohesive functions using examples presented in the literature. They treat executable program statements as basic components, or entities, and variables as attributes. They also introduce artificial variables for iterative loops and logical control statements. Xia, et al. [33] extends the work to consider control and data

dependency. They further divide both entities and attributes into two types: control and data. All these authors present results based on systematic and extensive experiments to compare and determine the weight of attributes for similarity measure.

2.2. Restructuring at Design Level

There also has been extensive research on software clustering conducted at the design or architectural level. Tzerpos and Holt [30] survey clustering approaches and find that classic clustering techniques can be used in the software context and that there is research potential in the software clustering field. They point out that some structure is better than no structure. Wiggerts [32] provides a general overview of clustering techniques and their applications to system re-modularization, highlighting the benefit of the general theory of clustering analysis. Lakhotia [16] gives a survey on subsystem classification techniques and provides a unified framework for entity description and clustering methods to facilitate comparison between various subsystem classification techniques.

Previous software clustering approaches concentrate on software system modularization or re-modularization at the architectural or design level. The entities to be clustered could be functions (routines), global variables (for identifying abstract data types), or files. Their similarity measures are based on relationships between entities [9,18-20,24,26] or shared features [1,2,28], with or without giving weights to the relationships or features. Researchers use different information or formula to measure the similarity based on different perspectives.

2.3. Clustering Algorithms

The clustering algorithms used in previous works fall into three categories: hierarchical agglomerative clustering (HAC) [1,2,10,18-20,28], optimization algorithms [21,22,24-26], and graph theoretic algorithms [6,28]. The hierarchical algorithms are used the most.

Among the HAC algorithms, UPGMA is the most commonly used approach [27]. Lung [20] and Lung, et al [18] present reverse engineering and reengineering experiences for architecture or design recovery based on UPGMA. However, the survey in [16] suggests that most researchers prefer the SLINK algorithm in subsystem classification. Girard, et al, [9] tailor the SLINK algorithm because the approach generated very large groups that were not useful. Alternatively, [1] suggest the CLINK algorithm based on their experiments for software re-modularization using files. Maqbool and Babri [23] present a weighted combined linkage algorithm of software clustering to support architecture recovery and a comparative study with some clustering algorithms.

Different algorithms may be suitable for different applications. Recently, Wen and Tzerpos [30] presented a comparative study of software clustering based on *MoJo* distance for architecture decomposition, including HAC algorithms. The paper adopts the Jaccard algorithm to calculate the resemblance coefficient for those HAC algorithms. The approach also is based on the concept of un-supervised clustering, i.e. the clusters are automatically generated by the software. This paper, however, advocates a supervised approach. Many factors in software could affect relationships of entities or components, especially at the code level. In addition, testing needs to be conducted after function restructuring. Function restructuring based on unsupervised clustering could create a lot of burden in post-mortem analysis and testing.

Software clustering is a complicated research area. The user of the clustering approach needs to decide how to choose entities and attributes, how to measure and compute similarity, and which algorithm to use for a specific problem. This paper targets clustering at the program statement level inside of a function, which is a relatively new area and quite challenging.

3. Similarity Measure and Clustering

Measurement of similarity relates to entities and attributes. The HAC approaches have three common key steps:

- Obtain the input data set
- Compute the resemblance coefficients.
- Execute the clustering method.

An input data set is an entity-attribute data matrix. Entities are the components grouped based on similarities. For example, entities can be software subsystems, files/classes, functions/methods, or even program statements. The attributes of an entity are properties that constitute the entity. For example, attributes of a program statement can be variables, function calls and/or other code constructs.

In this paper, a resemblance coefficient for a given pair of entities indicates the degree of similarity between these two entities. The data may be represented by means of a binary value. In this case, a 1 value indicates that the component has the property. A resemblance coefficient could be qualitative or quantitative. The simplest form of qualitative value is binary representation; e.g., the value is either 0 or 1. Qualitative attributes can also be multi-state such as red/blue/green. A quantitative coefficient measures the literal distance between two components when they are viewed as points in a two-dimensional array formed by the input attributes.

3.1. Entities

In our approach for function restructuring, an entity is viewed as an executable program statement that is related to a functional activity and can be described by its attributes. Entities are further divided into control entities and non-control entities. A control entity refers to an entity that is either a predicate statement (such as an *if* or *switch* statement) or iteration statement (such as a *for* or *while* statement). Each entity is represented as number for simplicity.

3.2. Attributes

An attribute is a feature of an entity. We adopt multi-state values, 0, 1, and 2 for attributes. An entity usually has multiple attributes. In our paper, attributes are variable and function names used in a statement. Constants, operators, and key words are not attributes. Variables are further divided into data variables and control variables based on data dependence and control dependence relationship.

Data variable: A data variable refers to the variable that is directly used in a statement. Data variables include local variables, global variables, and parameters passed to the function. A composite variable, such as an array, a linked list, or a user-defined data structure (struct), is treated as one variable. In addition, a function name in a function call statement is also treated as a data variable.

Loop counter variable: A loop counter variable, used in *for* or *while* loops, is treated as another kind of data variable. Because the restructuring focuses on static functional structure, the number of times a loop is repeated is irrelevant. The loop body is considered. Therefore, the loop counter variable is not counted as an attribute of an entity.

Control variable: A control variable is an artificial attribute added by our tool to an entity in a control block. It is used to describe control dependence relationships between entities in a predicate statement. Entities with the same control variable indicate that they belong to the same control block, e.g., *if* or *while*, in source code.

Therefore, in our approach, data (except loop counter variables) and control variables are chosen as attributes to describe entities. Each attribute is measured on qualitative scale as binary representation. Each attribute is defined to have two states, either presence or absence in an entity (statement).

- 0 – absence state of a control or data attribute
- 1 – presence state of a control attribute
- 2 – presence state of a data attribute

In addition, the data attributes in control entities are treated as control attributes. Therefore, between any two

entities, there are six different types of matches described below.

- **1–1 match:** A control attribute is present in both entities.
- **2–2 match:** A data attribute is present in both entities in case neither of them is a control entity.
- **0–0 match:** An attribute is absent in both entities.
- **1–0 or 0–1 match:** Mismatch. A control attribute is present in one entity but absent in the other.
- **2–0 or 0–2 match:** Mismatch. A data attribute is present in one entity but absent in the other.
- **2–1 or 1–2 match:** A data attribute is present in both entities in case one of them is a control entity and the other is a non-control entity.

3.3. Similarity Measure

The similarity measure is used to evaluate cohesion and it is represented with a resemblance coefficient.

3.3.1. Attributes

Many factors need to be taken into account when considering cohesion for attributes. For example, data and control attributes contribute differently to cohesion. Two variables with a data dependence relationship have higher cohesion than variables with a control dependence relationship. You also need to consider that different data attributes are weighted differently and that a data attribute may appear in a non-control entity or a control entity.

Variable scope: A variable could be local to a function or shared by multiple functions in a program. But at the function level, from the functional activity point of view, there is no difference between a global variable and a local variable. In our approach, a global variable and a local variable in a function play the same important role in functional cohesion and are therefore, treated equally.

Function call: A function name in a function call statement is treated as a data variable. In this restructuring approach, a function call statement is treated as a non-control entity. Different functions usually perform different tasks or activities. A function name is used to distinguish different function calls that correspond to different functionality. Therefore, a function name in a function call is measured in the same way as a local variable.

Data attributes in control entities: When a variable or a data attribute appears in a control entity, it has no direct relatedness to an activity. However, when a data attribute is used in a non-control entity it is directly related to an activity. Therefore, data attributes in control statements or entities should be treated differently from those in non-control entities. In our

approach, data attributes used in control entities are simply treated as control attributes.

Therefore, all data attributes in data entities are considered to have equal importance for functional cohesion. That is, all the data attributes (variables) in the data entities (statement) have the same weight. As the data attributes in control entities are treated as control attributes, the problem of weighting attributes boils down to the problem of the weighting between control attributes and data attributes. We believe that data attributes should be weighted more than control attributes, since a data attribute affects a functional activity directly while a control attribute affects indirectly.

3.3.2. Matches

The following matches can be used when calculating the similarity coefficients.

0-0 match: A 0-0 match indicates that an attribute is not used in either of the two entities. In an entity-attribute matrix, there are usually many attributes that are used in a function. But for each individual entity, it consists of only few attributes. Hence, there are many 0-0 matches in the matrix. In program restructuring, the similarity of two entities is not affected by adding unrelated attributes to the function. Therefore, 0-0 matches are ignored.

1-2 / 2-1 match: This kind of match occurs between one control entity and one non-control entity when they share a common data attribute. When these two entities are in the same control block, they share a common control attribute and there is a 1-1 match that counts the control dependence. Thus there is no need to use 1-2 / 2-1 matches to describe control dependence again. When these two entities are not in the same control block, they do not have control dependence. So 1-2 / 2-1 matches are also ignored.

1-1 match and 2-2 match: 1-1 matches and 2-2 matches mean that two entities share common attributes, which have a positive contribution to the similarity measure. A 1-1 match indicates that two entities have a control dependence relationship, or two control entities share a common data variable. It reflects the control structure of a function. A 2-2 match shows that two entities have a data dependence. Because data dependence contributes more to cohesion than that of control dependence, a 2-2 match should have more weight than that of a 1-1 match.

1-0 / 0-1 match and 2-0 / 0-2 match: A 1-0 / 0-1 match is a mismatch on a control attribute and shows the dissimilarity on control dependence or control structure. A 2-0 / 0-2 match is a mismatch on a data attribute and describes the dissimilarity for data dependence. Both contribute to the dissimilarity between entities. If matches on common data attributes

(2-2 matches) play a more important role on similarities between entities than matches on common control attributes (1-1 matches), then mismatches on data attributes (2-0 / 0-2 matches), should also have more importance for dissimilarity than mismatches on control attributes (1-0 / 0-1 matches). Hence, 2-0 / 0-2 matches should be weighted more than 1-0 / 0-1 matches.

The resemblance coefficient or similarity measure between two entities is defined as follows:

$$coeff = \frac{w_d a_d + w_c a_c}{w_d a_d + w_c a_c + w_d b_d + w_c b_c} \quad (1)$$

where: *coeff* - resemblance coefficient

a_d - # of 2-2 matches between two entities

a_c - # of 1-1 matches between two entities

b_d - # of 2-0 and 0-2 matches between two entities

b_c - # of 1-0 and 0-1 matches between two entities

w_d - weight of data attributes

w_c - weight of control attributes

$w_d > w_c > 0$

Here, the weight of an attribute represents its importance compared to other attributes. Attributes of the same type are weighted the same and the weight of data attributes is given more weight than control attributes. Previous work [33] suggested an empirical ratio of 8:3 of control and data dependence, for best results. Therefore the ratio of 8:3 is chosen to weigh the data attributes and the control attributes in the similarity measure throughout our experiments.

A sample program, as shown in Figure 1, is chosen to illustrate the scheme just described. The program calculates the sum and average, then selects the maximum element of an array. Table 1 provides the input data set. The following section will describe the results in details.

4. Experimental Comparison of UPGMA, SLINK, and CLINK

The UPGMA, SLINK, and CLINK clustering algorithms were applied to programs identified in

Section 2, related work, as well as assignments from students. The resulting functions revealed higher

```

1  procedure Sum_Max_Avg(n: integer; var arr:
   int_array; var sum, max: integer; var avg: float);
2  var i: integer;
3  begin
4    sum := 0;
5    max := arr[1];
6    for i:=1 to n do begin
7      sum := sum + arr[i];
8      if arr[i] > max
9        max := arr[i];
10   end;

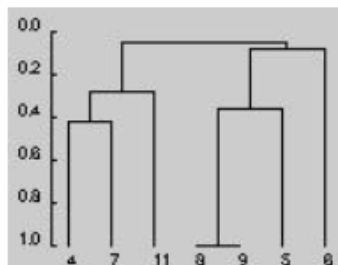
```

Figure 1. Sample program 1

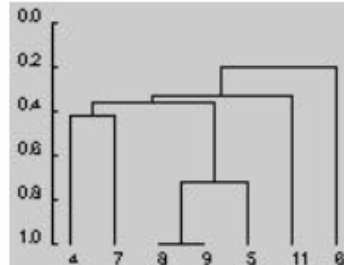
Table 1. Input data set for sample program 1

Entity	Attribute						
	N	arr	sum	max	avg	For	if
4	0	0	2	0	0	0	0
5	0	2	0	2	0	0	0
6	1	0	0	0	0	1	0
7	0	2	2	0	0	1	0
8	0	1	0	1	0	1	1
9	0	2	0	2	0	1	1
11	2	0	2	0	2	0	0

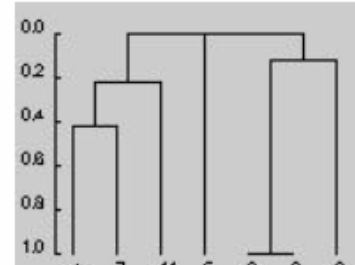
cohesion. Next, the approach was applied to industrial software, including functions of a C parser program and RSVP-TE network protocol [3] product. The RSVP-TE software has evolved over time from RSVP and will be further modified to support a new feature – fast reroute (FRR) in routing applications. Some functions in RSVP-TE share similar features with FRR. However, some RSVP-TE routines contained extra functional activities that will not be used in FRR and also the new feature will be implemented by a different designer. The approach and the tool are then used to support function restructuring with an aim to facilitate reuse at the function level. In addition, information about software



(a) UPGMA



(b) SLINK



(c) CLINK

Figure 2: Clustering results for sample program 1

complexity was captured for each function to better understand the relationship with clustering.

As mentioned earlier, the UPGMA, SLINK, and CLINK clustering algorithms were also applied to Sample Program 1 of Figure 1. UPGMA results in two clusters. Figure 2 presents the results. Figure 2(a) shows (4,7,11) and (8,9,5). Statement 6, for loop, does not belong to any cluster, rather it is shared by the two clusters. Figure 2(b) shows two clusters, (4,7) and (8,9,5). Statements 11 and 6 join two previous groups later, one by one. This is called a “chain” phenomenon. Figure 2(c) generates the worst result. Statement 5 does not show any relationship with other statements. Its resemblance coefficient is 0, which clearly does not reflect the real meaning of the function. One challenge of HAC is to determine the number of clusters or where the cut-point should be. Generally, the dendrogram, including the resemblance coefficients, provides heuristic information. The decision can be made automatically by setting a threshold value. However, this may generate clusters that are not useful in practice. Based on our experience [18,19,33], a supervised clustering, i.e. determining the cut off point for the clusters, and experienced evaluation of the clusters is highly desired. This is crucial, especially for software applications where many factors may affect the clustering result. Therefore, the results are evaluated manually by quickly reviewing the code and examining the dendrogram, though it may be subjective.

We first applied all three algorithms to twenty-one small functions (7 LOCs to 22 LOCs) presented in the related literature [4,5,14,15]. For all examples provided in the papers, the UPGMA had an effective rate of 100%. Alternatively, SLINK did not work well for 3 examples and CLINK did not work well for 5 examples used in [4,5].

Next, the three algorithms were applied to fourteen functions from students’ assignments that were designed for a variation of the classical smoker problem using semaphores and shared memory on the Linux operating systems. UPGMA worked well for all but one function. SLINK generated good clusters for seven functions and CLINK had good results for only for two examples. Table 2 demonstrates the results.

In few assignments, all three clustering algorithms detected duplicated code segments. This shows replicated patterns in the dendrogram. The repeated portions can be replaced with an array, a function, or a loop depending on the nature of the code. Replacing repeated code segments or clones will improve the maintainability of the function.

We also applied the three algorithms to industrial software, C parser, and RSVP-TE network signaling protocol. First, the algorithms were applied to two functions from the C parser. UPGMA showed better results for both cases while SLINK generated functional

cluster for one function. Second, the algorithms were applied to twenty-five functions (ranging from 32 LOCs to 283 LOCs) from the RSVP-TE protocol software. The original designers evaluated the C parser and RSVP-TE restructuring. Table 3 summarizes the results of comparing the three algorithms for the C parser software and Table 4 summarizes the results for the rsvp signaling protocol software.

Table 2. Comparison between three algorithms for students’ programs

	UPGMA	SLINK	CLINK	LOCs	Cyclomatic Complexity	No. Main Activities
1	X			182	36	3
2	X			85	19	3
3	X			30	6	2
4	X	X		52	14	1
5	X	X		43	8	1
6	X	X	X	16	2	1
7	X	X	X	55	14	3
8	X			68	13	3
9	X			120	20	3
10	X	X		68	14	3
11		X		87	14	3
12	X	X		71	13	3
13	X			154	30	3
14	X			112	27	3

Note: X indicates the best algorithm

Table 3. Comparison between three algorithms for C parser software

	Function Name	
	process_body	Process_open_braces
UPGMA	X	X
SLINK	X	
CLINK		
LOCs	41	57
Cyclomatic Complexity	13	19
No. of Activities	1	3

The following observations are made:

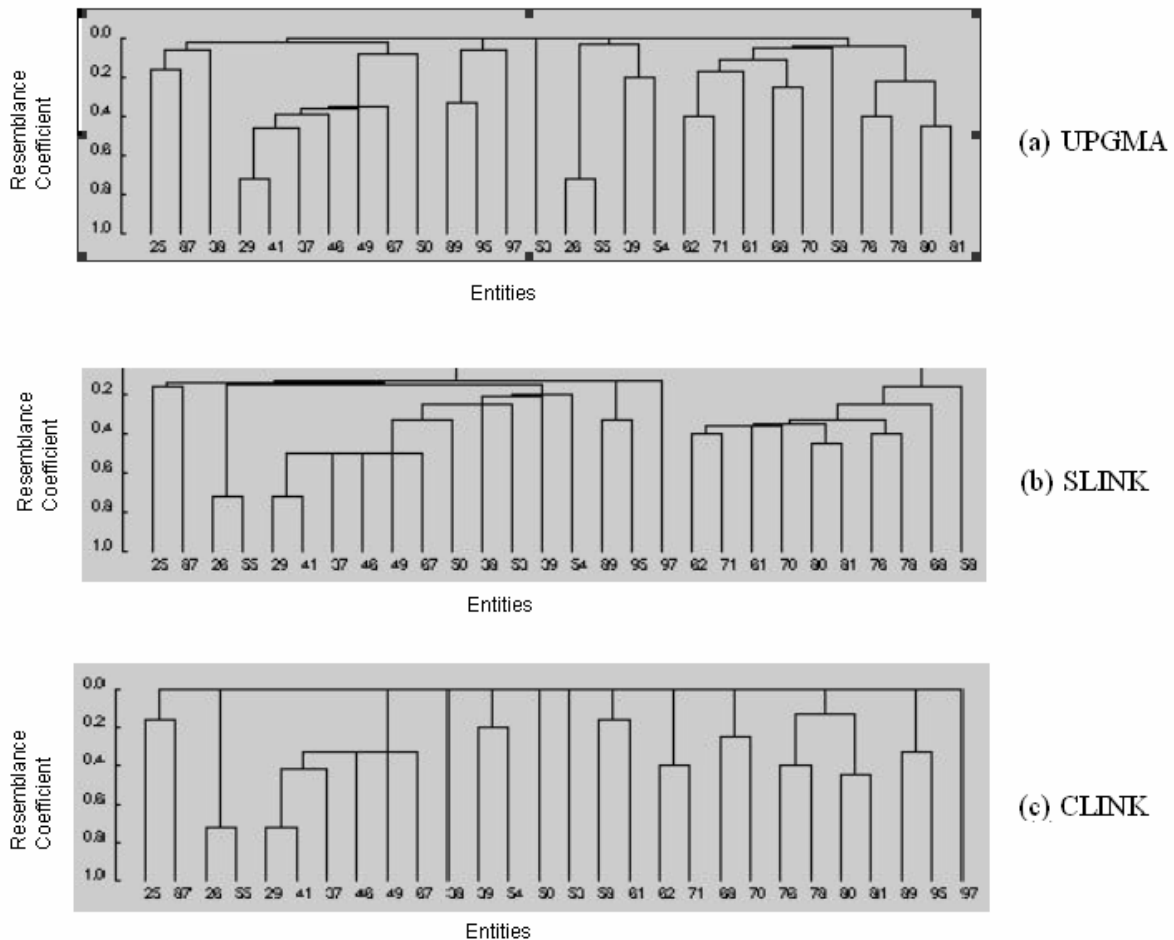
- UPGMA generated good functional clusters for 15 functions whereas SLINK generated 13 and CLINK generated 2.
- Only UPGMA generated good clusters for functions larger than 110 LOCs or had a cyclomatic complexity higher than 18.
- UPGMA generated functional clusters also when the size or complexity was small.
- UPGMA detects duplicated code segment.

Table 4. Comparison between three algorithms for *rsvp* signaling protocol

Function Name	UPGMA	SLINK	CLINK	LOCs	Cyclomatic Complexity	No. Main Activities
rsvpTeRx	x			74	19	3
rsvpTeDecodeMsg	x			283	64	2
rsvpTeRxPath	x			104	22	2
rsvpTeProcessERO	x			118	23	2
rsvpTeReserve	x			100	32	3
rsvpTeUpdateRSB	x			58	12	2
rsvpTeBuildRSB	x	x		32	8	1
rsvpTeProcessFlowDescriptor	x	x	x	98	18	1
...						
rsvpTeRxResv		x		56	12	2
rsvpTeResvRefresh		x		57	12	2
rsvpTeRSBAddFilter		x		50	7	1
rsvpTeResvTearFD		x		70	17	2
rsvpTeRxPErr		x		52	8	2

Note: X indicates the best algorithm

Figure 3. Comparison of three algorithms for *rsvpTeRxResv*



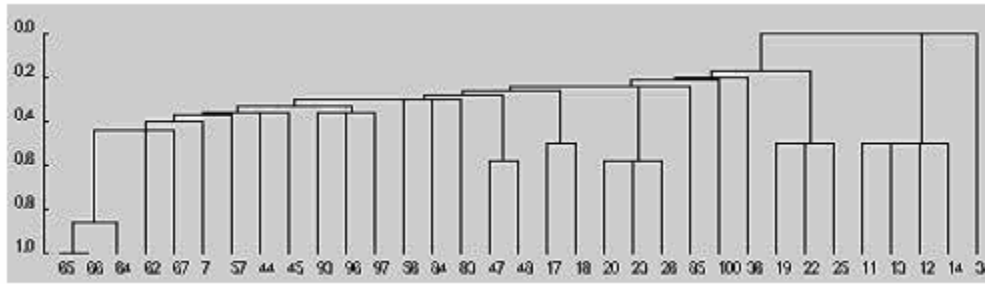


Figure 4. Chain phenomenon generated by SLINK

In general, SLINK and CLINK performed poorly compared to UPGMA. Figure 3 illustrates the clustering results for the three algorithms for the *rsvpTeRxResv* function. The clustering tree from SLINK shows that there are two big functional clusters: one functional cluster is related to the Resv message sanity check and the other is related to message processing. The clustering tree from CLINK shows thirteen small clusters and the clustering tree from UPGMA is between SLINK and CLINK. Different clustering algorithms generate different clusters. SLINK tends to build a small number of loose clusters or generate a “chain” phenomenon as illustrated in the first part of Figure 4. On the other hand, CLINK tends to form a large number of small but compact clusters. UPGMA provides a compromise between those two.

UPGMA was then selected to support RSVP-TE function restructuring. The designer used the clustering tool to analyze the code and restructure it if necessary. The original RSVP-TE software had twenty-four functions with about 2,200 LOCs. During the restructuring process, the designer restructured seventeen functions and decomposed them into sub-functions to increase program cohesion, reduce complexity, or remove duplicated or interleaved code. Table 5 demonstrates the results. McCabe’s cyclomatic complexity metric is calculated for each function before and after restructuring using the Krakatau metrics tool. The cohesion measure suggested by Anquetil and Lethbridge [1] is adopted to measure the functional cohesion.

Table 5. Size, complexity, and cohesion comparisons before and after restructuring

No.	Before Restructuring			After Restructuring				
	Function Name	LOCs	Complexity	Cohesion	Function Name	LOCs	Complexity	Cohesion
1	rsvpTeRx	74	19	0.048	rsvpTeRx	18	5	0.155
					rsvpTeMsgSanityCheck	51	12	0.081
2	rsvpTeDecodeMsg	283	64	0.182	rsvpTeProcessMsg	30	8	0.248
					rsvpTeDecodeMsg	223	54	0.178
					rsvpTeObjectListSanityCheck	38	8	0.536
					rsvpTeResetRsvpNodePointers	17	1	0.929
					rsvpTeProcessUnknownObj	21	4	0.098
3	rsvpTeRxPath	104	22	0.026	rsvpTeRxPath	64	12	0.038
					rsvpTeVerifyOpaqueObjLength	10	4	0.119
					rsvpTeVerifySndtmpObj	16	5	0.127
					rsvpTeDecodeEROandRROFromPMsg	40	6	0.103
4	rsvpTeProcessERO	118	23	0.090	rsvpTeProcessERO	49	10	0.056
					rsvpTeGetNextHopFromERO	57	12	0.079
					rsvpTeGetNextHopWithoutERO	25	5	0.119
...								
12	srvpTeRxPTear	70	13	0.078	srvpTeRxPTear	39	8	0.058
					rsvpTeForwardPTearMsg	36	5	0.048
13	rsvpTePSBTimeout	44	8	0.106	rsvpTePSBTimeout	16	2	0.257
					rsvpTePSBTimeoutUpdateRSBs	17	4	0.183
					rsvpTePSBTimeoutFreeSB	20	4	0.103

No.	Before Restructuring			After Restructuring				
	Function Name	LOCs	Complexity	Cohesion	Function Name	LOCs	Complexity	Cohesion
14	rsvpTeRxRTear	191	38	0.062	rsvpTeRxRTear	43	9	0.035
					rsvpTeProcessRTearFlowDescriptor	70	15	0.149
					rsvpTeForwardRTearMsg	99	16	0.121
15	rsvpTeResvTearFD	70	17	0.047	rsvpTeResvTearFD	52	13	0.044
					rsvpTeDeleteFilters	33	6	0.127
16	rsvpTeRxPErr	52	8	0.071	rsvpTeRxPErr	28	5	0.043
					rsvpTeIngressProcessPErr	25	3	0.160
					rsvpTeForwardPErrMsg	12	2	0.176
17	rsvpTeRxRErr	140	28	0.033	rsvpTeRxRErr	50	11	0.030
					rsvpTeDecodeFlowDescriptorFromRErr	28	4	0.150
					rsvpTeEgressProcessLargeRROErr	23	4	0.157
					rsvpTeProcessAdmissionErr	33	5	0.146
					rsvpTeRSBGenRErr	38	10	0.108
Total		1611			Total	1902		
Average		94.76	19.94	0.080	Average	37.29	7.69	0.160

5. Empirical Lessons

In this case study, the clustering results demonstrate an overall structure or related entities of a function. This information is useful to designers for function restructuring.

There are four main phases to restructuring: identify candidate modules; apply clustering; perform the restructuring; and conduct testing. The first step involves performing the clustering analysis on all the modules (functions in this particular case) and analyzing the results to identify the modules that require restructuring. It is not necessary to apply clustering to all functions. Designers usually have a good idea which functions are complex. After this step, the cluster analysis results (for those modules that require clustering) are used to restructure the code. This process is manual; however, the results help considerably in achieving the restructuring. As well, the time required to restructure is significantly reduced. To ensure that no new bugs are introduced in the process of restructuring, sufficient testing is required, as was done in this case study. Other lessons or observations learned by the experimental case study are as follows:

Restructuring may not necessarily always reduce the size and/or complexity: In some cases, the size or complexity of a function may not be reduced much after restructuring. For example, *rsvpTeDecodeMsg* function in Table 5 has similar metric values before and after the exercise. This particular function has a switch statement that consists of a large number of cases, which is used to decode different possible RSVP objects in an RSVP message. It is logical to keep those cases in one place although they may not be related to each other.

Customization of algorithm: In this case study, we modified the resemblance coefficient or similarity measure specifically for our needs. Other factors may need to be considered for some programs. The formula may need to be tailored for specific cases.

Using the concept of program slicing: The approach, conceptually, shares some similarity with intra-function program slicing. For instance, variable or attribute sum, in Table 1, appears in statements 4, 7, and 11. In other words, to find a slice, we can examine a column or a set of columns in the input table. Further experiments or comparisons can be made in this area.

Identification of aspects: Clusters may represent some aspects. For instance, security aspect was later added to the RSVP-TE code, which is very different from the functional perspective. The clustering tool can help identify those code segments specifically for the security aspect. Whether the code needs to be restructured is determined by the designers based on quality attributes, e.g., performance, security, etc.

6. Conclusions

Clustering has been studied at various levels for software analysis. This paper focused on code level and restructured functions to improve code cohesion. Previous research on clustering did not have a consistent conclusion regarding the effectiveness of the three commonly used clustering algorithms: UPGMA, SLINK and CLINK. This paper presented an empirical study on several small and medium-size software programs as well as an industrial telecommunication software.

Based on the study of more than 60 functions in different areas, UPGMA generally works best among the three commonly used algorithms, especially when the software size or complexity is high. It improves the

quality of the code and supports evolution, resulting in software that is more understandable, maintainable, and reusable. SLINK works well in some cases, but the results are more difficult to predict. Alternatively, CLINK does not produce good functional clusters. To reveal more cohesive clusters, UPGMA and SLINK can be applied at the same time.

References

- [1] Anquetil, N. and Lethbridge, T. C. (2003). "Comparative study of Clustering Algorithms and Abstract Representations for Software Remodularisation", *IEE Proc. on Software*, 150(3), pp.185-201.
- [2] Anquetil, N., Fourier, C and Lethbridge, T. (1999). "Experiments with Hierarchical Clustering Algorithms as Software Remodularization Methods", *Proc. of Working Conf. on Reverse Eng.*
- [3] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V. and Swallow, G. (2001). *RSVP-TE: Extensions to RSVP for LSP Tunnels*, RFC 3209.
- [4] Bieman, J. M. and Kang, B.-K. (1998). "Measuring Design-Level Cohesion", *IEEE Trans. on Software Eng.*, 24(2), pp.111-124.
- [5] Bieman, J. M. (1994). "Measuring Functional Cohesion", *IEEE Trans. on Software Eng.*, 20(8), pp.644-657.
- [6] Choi, A. C. and Scacchi, W. (1990). "Extracting and Restructuring the Design of Large Software Systems", *IEEE Software*, 7(1), pp.66-71.
- [7] Everitt, B. (1974). *Cluster Analysis*. Heineman Educational Books, London.
- [8] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.
- [9] Girard, J.-F. Koschke, R. and Schied, R. (1999), "A Metric-based Approach to Detect Abstract Data Types and State Encapsulations", *Automated Software Eng.*, vol. 6, Issue 4, pp. 357 – 386.
- [10] Hutchens, D. and Basili, V. R. (1985). "System Structure Analysis: Clustering with Data Bindings", *IEEE Trans. on Software Eng.*, 11(8), pp.749-757.
- [11] Kang, B.-K. and Beiman, J. M. (1999). "A Quantitative Framework for Software Restructuring", *J. of Software Maintenance: Research and Practice* 11, pp.245-284.
- [12] Kang, B.-K. and Bieman, J. M. (1998). "Using Design Abstractions to Visualize, Quantify, and Restructure Software", *The J. of Sys. and Software*, 42, pp.175-187.
- [13] Kim, H. S. and Kwon, Y. R. (1994). "Restructuring Programs through Program Slicing", *Int'l J. of Software Engineering and Knowledge Eng.*, 4(3), pp.349-368.
- [14] Lakhotia, A. and Deprez, J. C. (1999). "Restructuring Functions with Low Cohesion", *Proc. of Working Conf. on Reverse Eng.*, pp.36-46.
- [15] Lakhotia, A. and Deprez, J. C. (1998). "Restructuring Programs by Tucking Statements into Functions", *J. of Info. and Software Technology*, 40(11-12), pp.677-689.
- [16] Lakhotia, A. (1997). "A Unified Framework for Expressing Software Subsystem Classification Techniques", *J. of Sys. and Software*, 36, pp.211-231.
- [17] Lakhotia, A. (1993). "Rule-based Approach to Computing Module Cohesion", *Proceedings of the 15th Int'l Conf. on Software Eng.*, pp.35-44.
- [18] Lung, C.-H., Zaman, M. and Nandi, A. (2004) "Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring", *J. of Sys. and Software*, 73(2) pp 227-244.
- [19] Lung, C.-H. and M. Zaman (2004), "Using Clustering Techniques to Restructure Programs", *Proc. of Software Engineering Research and Practice Conf*, pp. 853-858.
- [20] Lung, C.-H. (1998). "Software Architecture Recovery and Restructuring through Clustering Techniques", *Proc. of the 3rd Int'l Workshop on Software Architecture*, pp.101-104.
- [21] Mancoridis, S., Mitchell, B., Chen, Y. and Gansner, E. (1999). "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Organizations of Source Code", *Proc. of Int'l Workshop on Program Comprehension*.
- [22] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y. and Gansner, E. R. (1998). "Using Automatic Clustering to Produce High-Level System Organizations of Source Code", *Proc. of the 6th Int'l Workshop on Program Comprehension*, pp.45-52.
- [23] Maqbool O. and H.A. Babri, H.A, (2004). "The Weighted Combined Algorithm: A Linkage Algorithm for Software Clustering", *Proc of 8th Euromicro Working Conf on Soft. Maintenance and Reengineering*, pp. 15-24.
- [24] Mitchell, B. S. and Mancoridis, S. (2001). "Comparing the Decompositions Produced by Software Clustering Algorithm Using Similarity Measurements", *Proc. of Int'l Conf. of Software Maintenance*.
- [25] Müller, H. A., Wong, K. and Tilley, S. R. (1995). "Understanding Software Systems Using Reverse Engineering Technology", *Object-Oriented Technology for Database and Soft. Sys., World Scientific*, pp.240-252.
- [26] Müller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S. (1993). "A Reverse Engineering Approach to Subsystem Structure Identification", *J. of Software Maintenance: Research & Practice*, 5(4), pp.181-204.
- [27] Romesburg, H. C. (1990). *Cluster Analysis for Researchers*, Krieger Publishing Company.
- [28] Schwanke, R. W. (1991). "An Intelligent Tool for Re-engineering Software Modularity", *Proc. of the 13th Int'l Conf. on Software Eng.*, pp.83-92.
- [29] Sommerville, I. (1996). *Software Engineering*, 5th Edition, Addison-Wesley, England.
- [30] Tzerpos, V. and Holt, R. C. (1998). "Software Botryology Automatic Clustering of Software Systems", *Proc. of the 20th Int'l Conf. of the IEEE*, 3, pp.811-818.
- [31] Wen, Z. and Tzerpos, (2004) V., "An Effectiveness Measure for Software Clustering Algorithms", *Proc. of the 12th Int'l Workshop on Program Comprehension*, pp. 194-203.
- [32] Wiggerts, T. A. (1997). "Using Clustering Algorithms in Legacy Systems Modularization", *Proc. of the 4th Working conf. on Reverse Eng.*, pp.33-43.
- [33] X. Xu, C.-H. Lung, M. Zaman, and A. Srinivasan (2004), "Program Restructuring Using Clustering Technique", to appear in *IEEE Workshop on SCAM (Source Code Analysis and Manipulation)*, Sept.

