M. Sc. Engineering Thesis

Software Restructuring using Hierarchical Clustering

by

Aftab Hussain

Submitted to

Department of Computer Science and Engineering

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE & ENGINEERING

Department of Computer Science and Engineering

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Dhaka-1000, Bangladesh

November, 2012

The thesis titled "**Software Restructuring using Hierarchical Clustering**", submitted by Aftab Hussain, Roll No. 1009052007F, Session October 2009, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on November 11, 2012.

Board of Examiners

1. _____

Dr. Md. Saidur Rahman — Chairman
Professor — (Supervisor)
Department of CSE, BUET, Dhaka 1000.

2. _____

Dr. Abu Sayed Md. Latiful Hoque — Member
Professor & Head — (Ex-officio)
Department of CSE, BUET, Dhaka 1000.

3. _____

Dr. Md. Mostofa Akbar — Member
Professor
Department of CSE, BUET, Dhaka 1000.

4. _____

Dr. Mohammed Eunus Ali — Member
Assistant Professor
Department of CSE, BUET, Dhaka 1000.

5. _____

Dr. Suraiya Pervin — Member
Professor — (External)
Department of Computer Science and Engineering
University of Dhaka, Dhaka 1000.

Candidate's Declaration

This is to certify that the work presented in this thesis entitled "**Software Restructuring using Hierarchical Clustering**" is the outcome of the investigation carried out by me under the supervision of Professor Dr. Md. Saidur Rahman in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka. It is also declared that neither this thesis nor any part thereof has been submitted or is being currently submitted anywhere else for the award of any degree or diploma.

Aftab Hussain
Candidate

Contents

List of Figures

List of Tables

Acknowledgments

The successful completion of my thesis has been a tremendous achievement for me, something I could never have accomplished without the support of the people around me. I humbly take this opportunity to thank them, although it would not be possible to mention them all. Nevertheless, my gratitude to each one of them remains beyond measure.

First and foremost my deepest gratitude goes to my supervisor, Prof. Dr. Md. Saidur Rahman, who was the only person behind motivating me to work in this engaging field of software restructuring. In addition to that, he inspired me to explore new ways of contributing to the field using graph algorithms. As a result, by virtue of his brilliant guidance, I learnt various exciting aspects of inter-disciplinary research. Apart from being grateful for all that I have gained from his vast scientific knowledge, I have a lot to thank him for his persistent encouragement throughout my endeavour. At tough times he has given full-hearted support, providing me with the most insightful of directions. Indeed, he played the most instrumental role in my thesis completion and I shall forever treasure this entire period that I have spent working with him.

Much of my appreciation goes to Dr. Md. Tanvir Parvez for his valuable inputs towards my research. I would also like to extend my sincere gratitude to Abdulaziz Alkhalid, Dr. Mohammad Alshayeb, and Prof. Dr. Sabri Mahmoud for sharing their implementation techniques of the A-KNN algorithm, which was studied in this thesis.

I earnestly thank my thesis committee members, Prof. Dr. Abu Sayed Md. Latiful Hoque, Prof. Dr. Md. Mostofa Akbar, Dr. Mohammed Eunus Ali and Prof. Dr. Suraiya Pervin, whose suggestions were very beneficial for this work.

I am heavily indebted to each and every member of the Graph Drawing and Information Visualization Laboratory, the place where most of this work was

done. Through their magnanimous support and friendship, they engendered a congenial and intellectually stimulating research environment in the lab. In particular, I would like to thank Md. Manzurul Hasan, Safique Ahmed Faruque and Md. Iqbal Hossain, who have always shown willingness to advise me on my work.

Finally, I would deeply like to thank my loving parents, my dear grandfather, and my ever-caring sister for consistently encouraging me, with great patience, at every moment of this wonderful and challenging journey of completing my thesis.

Abstract

Bad designs in software code have a significant impact on the total cost incurred in the development of software. This is because software code with bad designs has poor structure, which decreases its readability, understandability and maintainability. Software restructuring is thus a crucial activity in software development. *Cohesion* is an important measure in assessing the quality of software. The cohesion of a software module is the degree to which module components belong together. An ill-structured software code is characterized by low cohesion. Software restructuring techniques based on hierarchical agglomerative clustering (HAC) algorithms have been widely used to restructure large modules with low cohesion into smaller modules with high cohesion. These techniques generate clustering trees (or dendrograms) of the modules. The clustering trees are then sliced at different cut-points to obtain the desired restructurings. Choosing the appropriate cut-points is a difficult problem in clustering. This problem is exacerbated in previous HAC techniques as those techniques generate clustering trees which have a large number of cut-points. Moreover, many of those cut-points return clusters of which only a few lead to a meaningful restructuring.

In this thesis, we propose a new hierarchical clustering technique for restructuring software at the function-level that generates clustering trees where the number of cut-points is reduced, and the quality of the cut-points is improved. To establish this we compare the results of our technique with those of four previous hierarchical clustering algorithms. We also develop an easy-to-use software tool that allows the user to generate clustering trees of functions using five different clustering algorithms, including the algorithm proposed in this thesis. Finally, we give a characterization of clusters returned by cut-points, in the context of software restructuring.

Chapter 1

Introduction

Software code in the industry undergoes frequent change due to the varying needs of the user. Increased modifications of the code lead to the gradual degradation of its structure, and the consequent emergence of bad designs. Bad designs refer to structural flaws that are usually associated with bad programming practices. Although such structural flaws may not necessarily affect the intended functionality of the software code, their presence reduces the readability, reusability, and therefore the quality of the code. This adversely affects two of the costliest phases of software development: maintenance and evolution[1].

A key step involved in both software maintenance and software evolution is code comprehension, which constitutes 40% to 60% of the efforts given in these two phases [AJ04]. Since code comprehensibility becomes difficult with ill-structured software code, performing these two phases on such code becomes harder and hence costlier, having a ballooning effect on the overall cost of software development. These significant implications of ill-structured software code have led to the emergence of extensive research in the field of software restructuring.

Software restructuring is the process of reorganizing the internal structure of existing software systems in order to improve its quality, without modifying its external behaviour [KB99, FBB+99]. One of the most important measures of assessing the quality of software code is cohesion. The *cohesion* of a software

[1]It has been found that 60% to 70% of the total costs of developing a typical software product are incurred in the maintenance and evolution phases of software development [Mal08, LXZS06].

module is the degree of functional strength of the module [Mal08]. In other words, it is the degree to which a software module does a particular activity. Ill-structured software code is characterized by low cohesion [Pre04]. In practice, a software module could represent a function, class, package, library, etc.

Over the years, there has been numerous cohesion-based software restructuring techniques developed, many of which have been used to great effect. Among the most successful software restructuring techniques were those which used hierarchical clustering techniques. Compared to other types of techniques, these techniques were found to be more efficient, and also to give good restructuring suggestions.

Software restructuring techniques based on hierarchical clustering deploy hierarchical agglomerative clustering algorithms on raw data that represent the components of a software module that is to be restructured. The outputs consist of clustering trees (which are alternately referred to as dendrograms) that provide suggestions on how to restructure the software. The suggestions can be extracted by observing the clusters obtained from different cut-points of the clustering tree.

An important problem with past hierarchical clustering techniques is that they generate dendrograms that have a large number of cut-points, which return a large number of redundant clusters. *Redundant clusters* do not lead to a meaningful restructuring of the software. A *meaningful restructuring* of the software corresponds to a restructuring that is logically coherent and in-line with the original purpose of the software. It becomes extremely difficult for the programmer to choose the appropriate cut-point(s) and sieve out meaningful clusters from a clustering tree that gives a large number of cut-points, which return a large number of redundant clusters. Also, inspecting redundant clusters wastes precious time of the developer during analysis of the clustering tree. In this thesis, we concentrate on the problem of minimizing the number of cut-points and the number of redundant clusters. In order to do so we have developed an entirely new hierarchical clustering technique based on k-core decomposition [BZ03]. The new technique intuitively limits the number of cut-points and redundant clusters returned in its clustering tree output by using structural properties of the software components, without the use of any user-determined threshold values. We also compared the performance of our

technique with those of four previous hierarchical clustering techniques that have been found to be successful for software restructuring.

In the rest of this chapter we explore the various aspects of software restructuring, previous research in the field, and the significance of the problem that has been dealt with in this thesis. - In Section 1.1, we give an example of an ill-structured software and show how it can be improved in the cohesion perspective. In Section 1.2, we give an overview of the various software restructuring techniques used, emphasizing on hierarchical clustering techniques, which is the domain of our thesis. In particular, in the section we mention the problems faced by the hierarchical clustering techniques. In Section 1.3, the objectives of this thesis are stated. In Section 1.4, we briefly present the results that were obtained in this thesis work. Finally, in Section 1.5, the organization of this entire thesis book is outlined.

1.1 An Example of Ill-Structured Code

In this section, we explain the notion of cohesiveness in a sample function and show how its cohesiveness can be improved by suitably rewriting the function.

```
void sum_and_prod(int n, int[] arr) {
sum = 0;
prod = 1;
for ( int i = 1 ; i < n ; i ++ )
{
  sum = sum + arr [ i ] ;
  prod = prod * arr [ i ] ;
}
avg = sum / n ;
}
```

Figure 1.1: The sum_prod function [KB99].

Fig. 1.1 shows an example of a low-cohesive function. The function calculates the sum, product, and average of a set of integers. The three values are stored in the variables `sum`, `prod`, and `avg` respectively. The set of integers are input using the array variable, `arr`. Although the function correctly calculates the values, it is not optimally cohesive as it is carrying out a number of tasks in a single block

of code. A better and a more cohesive version of this code can be written in the manner shown in Fig. 1.2. Here the function in Fig. 1.2(a) only deals with the calculations relevant to `sum`, whereas the function in Fig. 1.2(b) only deals with the calculation of `prod`. Although both versions of the code fulfil the same functional objective, the restructured version is clearly easier to understand. The design flaw that has been corrected in this example is referred to as "long method", as termed in the authoritative book on refactoring (another term for restructuring) by Fowler *et al.* [FBB+99]. It has been widely recommended that long methods in codes should be split into shorter methods such that each method performs a unique task, as far as possible. This simplifies code structure, enhancing code readability, and is consequently in-line with object-oriented programming principles.

```
void sum(int n, int[] arr) {
sum = 0;
for ( int i = 1 ; i < n ; i ++ )
{
 sum = sum + arr [ i ] ;
}
avg = sum / n ;
}
```

(a)

```
void prod(int n, int[] arr) {
prod = 1;
for ( int i = 1 ; i < n ; i ++ )
{
 prod = prod * arr [ i ] ;
}
}
```

(b)

Figure 1.2: Restructured version of the function in Fig 1.1.

1.2 Software Restructuring Techniques

In this section, we discuss the objectives of software restructuring techniques at different levels of the software paradigm. In addition to that, we present the previous works that have been carried out in the field of software restructuring; in Subsection 1.2.1 we discuss partitional techniques, in Subsection 1.2.2 we discuss optimization techniques, and finally in Subsection 1.2.3 we discuss hierarchical clustering techniques.

As was mentioned earlier, software restructuring is the process of improving the quality of software code. Cohesion is a key aspect of the quality of soft-

ware code. Software restructuring techniques that focus on cohesion provide suggestions as to how to rewrite and thus *remodularize* a large, low-cohesive module into smaller modules of higher cohesion, as shown in Fig. 1.3. A *software module* can be a function, class, package, library, etc. Consequently, a restructuring technique can work at different levels. A restructuring technique that works at the function-level restructures functions, a technique that works at the class-level restructures classes, and so on.

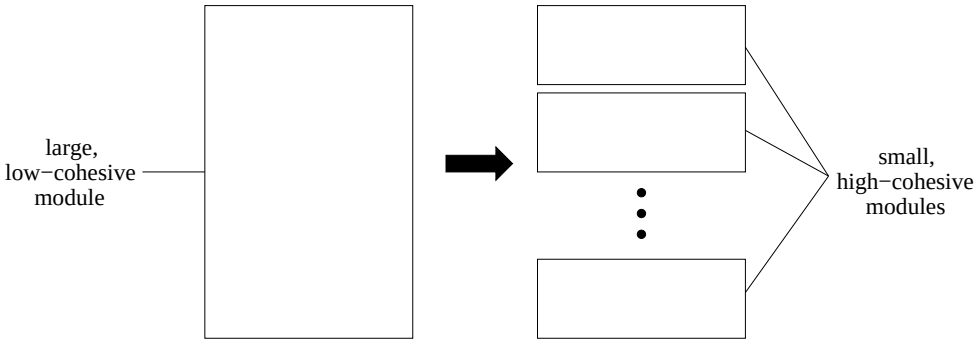


Figure 1.3: Cohesion-based software restructuring.

A software module consists of *components*, which in turn consist of *attributes*. In order to remodularize a software module to increase its cohesion, a restructuring technique categorizes the components of a module based on the similarity between its components. The similarity is evaluated based on the number of common attributes among the components.

As an example (see Fig. 1.4), say we have a package in which classes correspond to its components. The attributes are resembled by the usages of various functions in the classes. On applying a software restructuring technique on this package, suggestions on how the classes should be grouped are obtained. In particular, the technique recommends to group classes that are found to share a relatively large number of functions. The groupings serve as the basis of new packages which can be constructed from the original software module. Constructing new packages in this manner will increase the overall cohesion of the system.

However, it is important to note that when applying restructuring techniques different restructuring results may be obtained, of which some may not even lead to a meaningful restructuring of the software. The final decision lies with the software developer as to how to restructure the code. Thus, when using these

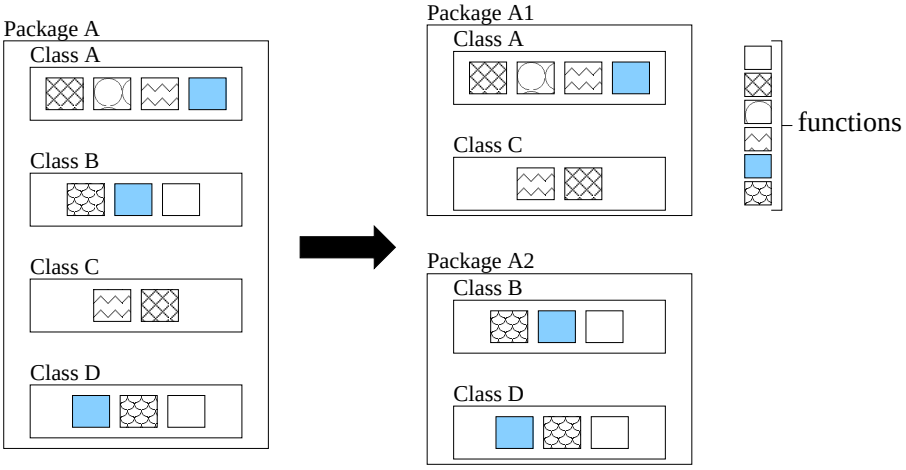


Figure 1.4: Cohesion-based restructuring at the package-level.

techniques, the developer will *only take help* from the suggestions, while mainly using his/her programming expertise and intuition in order to obtain the desired restructuring. Over the years, there have been many software restructuring techniques proposed. In this section we discuss some of the techniques, while mainly focussing on hierarchical clustering techniques.

1.2.1 Partitional Techniques

Partitional techniques address the software remodularization problem as a software clustering problem. These techniques try to construct a partitioning of a set of entities (i.e., the module components) into a set of non-overlapping groups (or clusters) such that a given criterion is optimized [And03]. For software clustering, the criterion is usually a metric that resembles the cohesion of the software system. Generally, an initial partitioning is chosen and then the membership of the entities is changed in order to obtain a better partitioning. This process is carried out iteratively until the given criterion is optimized.

The main problem faced by partitional algorithms is that even for a small number of entities, there exists a large number of possible partitions, of which all are assessed by the partitional algorithms. This makes the algorithms computationally very expensive, even more so for software clustering where the number of entities may be very large. Another disadvantage of partitional algorithms is the need to predetermine the ideal number of clusters. Since software systems are widely varying in structure, such a predetermination is difficult.

In [CS06], Czibula and Serban proposed and implemented a partitional technique for remodularizing software at the class-level. Although they provided a heuristic for predetermining the number of clusters, their technique was found to take high execution time.

Chatzigeorgiou *et al.* [CTS06] consider the software clustering problem as a graph partitioning problem. They suggested a partitioning technique based on spectral graph clustering, without any implementation. They mentioned that the calculation of eigen vectors, a step performed in spectral graph clustering, is an extremely expensive task (computationally), which might lead to prohibitive run times if implemented on large software systems.

1.2.2 Optimization Techniques

In order to find good partitions quickly, optimization techniques were proposed, which treat the graph partitioning problem as a search problem. The techniques try to significantly reduce the search space of partitioning problems and search for results such that a certain objective function/criterion is optimized.

An example is the work of Mancoridis *et al.* [MMR+98]. They deployed optimization algorithms, like hill-climbing algorithms and genetic algorithms, in order to find optimal partitions of software systems based on a cohesion criterion. However, the algorithms are nondeterministic in nature as they don't yield the same result for every execution. This is because in each execution, the algorithms begin with a random partition of the system and converge to a local maximum. Moreover, not every initial partition of the system leads to an optimal solution (in the perspective of the cohesion criterion). To overcome this, Mitchell and Mancoridis [MM06] used an initial population of random partitions on which they applied their technique. The random partitions led to different results, from which the best result was ultimately chosen. Although increasing the size of the population increased the probability of getting an optimal result, it aggravated the overhead of their technique.

Similar to what was done in the previous work, Kata *et al.* [PHY11] used a multi-objective optimization technique based on a genetic algorithm. In addition to the cohesion criterion, they introduced more objective functions for their algorithm to optimize. The new objective functions correspond to other quality parameters of software modules, such as function size, uniformity of

function size, etc. They showed that their technique gave better results than the previous techniques for many examples. However, because their technique considers more objective functions during each execution, their technique was found to take much longer to run.

1.2.3 Hierarchical Clustering Techniques

Hierarchical clustering techniques have been widely used for cohesion-based software restructuring. These techniques have been observed to work faster than optimization techniques [LZN06]. Most hierarchical clustering techniques use hierarchical agglomerative clustering algorithms (HACs) [AL99, AL03, LXZS06] or algorithms *based* on HACs [SC08, AAM10, AAM11]. These algorithms and the methodology by which they're deployed for software restructuring are discussed in depth in Chapter 2.

HAC techniques give results in the form of clustering trees (or dendrograms), for example like the one in Fig. 1.5. As shown in the figure, the dendrogram represents a hierarchy of the entities (or components) of a software module, where the vertical axis represents a scale of similarity, and the horizontal axis indicates the set of components of the module. In order to obtain a partitioning of the software module, the tree is sliced at different cut-points[1], indicated by the red, dashed horizontal lines in Fig. 1.5. Each cut-point returns a set of clusters which are used to obtain the desired restructuring. Now, choosing the appropriate cut-points is a difficult problem in clustering [AL99]. This problem is of even greater significance in software clustering because not all cut-points yield meaningful clusters.

The present hierarchical clustering techniques worsen the problem as they generate clustering trees with many cut-points, which yield a large number of redundant clusters. This clutters clustering trees, and makes it difficult to select appropriate cut-points that would lead to a desired restructuring of the software module.

Ways to solve the problem may be to pre-specify the cut-point height, or to pre-specify the number of clusters in the desired remodularized version of the module. However, software code can be widely varying in structure, which is

[1]The formal definitions of *dendrograms*, *similarity*, *cut-points*, and their significance in the context of software restructuring are given in Chapter 2.

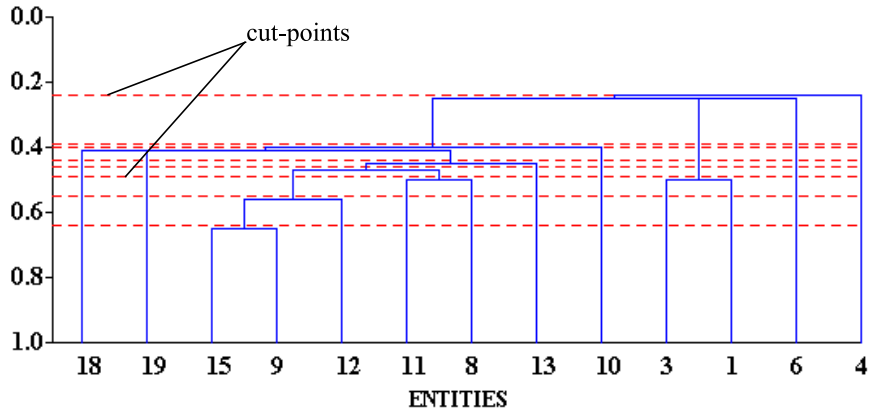


Figure 1.5: A clustering tree or dendrogram of the components of a software module.

why such pre-specifications are difficult to make. Serban and Czibula [SC08] gave a heuristic for predetermining the number of clusters in the desired restructuring. However, their heuristic depends on a user defined threshold value. In addition to that, their technique has been found to give unfavourable results for many examples.

A reasonable and intuitive way to address this problem would be to design a technique that generates clustering trees of software modules, where the number of cut-points in the trees is reduced and the quality of the cut-points is improved (i.e., the number of redundant clusters yielded by the cut-points is reduced). However, no such technique has been developed yet.

1.3 Thesis Objectives

Building upon the problems faced by hierarchical clustering techniques mentioned in the previous subsection, we state our thesis goals as under,

- Create a new hierarchical clustering technique, called (k, w)-Core Clustering ((k, w)-CC), that gives good suggestions for restructuring functions written in Java. The technique gives better clustering trees, than those generated by previous hierarchical clustering techniques, with comparable efficiency. In particular, the technique gives clustering trees that have cut-points of better quality, i.e., the total number of meaningless clusters

returned by the cut-points is reduced. In addition to that, the technique generates clustering trees with smaller number of cut-points.

- Give a characterization of clusters returned by cut-points in dendrograms, in the context of software restructuring at the function-level.
- Give an experimental comparison of the results of our algorithm with those of four previous hierarchical clustering algorithms (SLINK, CLINK, WPGMA, and A-KNN). The following parameters are compared: execution time, number of cut-points and bad clusters generated, cohesion improvement.
- Develop an easy-to-use software tool that allows the user to generate clustering trees of functions using the technique developed in this thesis, and the four previous techniques mentioned above.

1.4 Results Obtained

Based on our experiments, we have obtained the following results,

- Overall, (k, w)-CC generates 40.66%, 44.74%, 55.93%, 41.71% lesser number of cut-points than SLINK, CLINK, WPGMA, and A-KNN, respectively.
- Overall, (k, w)-CC generates 65.01%, 66.32%, 70.72%, 62.98% lesser number of bad clusters than SLINK, CLINK, WPGMA, and A-KNN, respectively.
- (k, w)-CC is faster by approximately 59.72% than SLINK, CLINK, and WPGMA. However, A-KNN is the fastest; it is 94.77% faster than SLINK, CLINK, and WPGMA.
- For most cases, the maximum cohesion improvement attained by all the techniques, including (k, w)-CC are the same.

1.5 Thesis Organization

The remaining part of this thesis is organized as follows: In Chapter 2, we describe hierarchical clustering algorithms and explain how they are used for restructuring software code. In the chapter, we also discuss the drawbacks of the previous hierarchical clustering techniques. In Chapter 3, we explain the new hierarchical clustering technique proposed in this thesis. In Chapter 4, we give a characterization of cut-points in clustering trees of software functions. In Chapter 5, we give an experimental comparison of the results of the new technique with those of the previous techniques. We conclude our thesis in Chapter 6, where we summarize the contributions of this work and give possible directions for further research in this field.

In Appendix A, we present a new software restructuring tool, called CohesionOptimizer, giving basic guidelines on how to use the tool. In Appendix B, the implementation code of the new clustering technique that was developed in this thesis is provided.

Chapter 2

Hierarchical Clustering Techniques

In this chapter, we explain hierarchical clustering and its application in software restructuring at the function-level. In Section 2.1, we give some preliminary definitions on clustering and hierarchical clustering. In Section 2.2, we explain the four previous hierarchical clustering algorithms [LXZS06, AAM10] that have been studied in this thesis. In Section 2.3, we describe the restructuring approach of [LXZS06, AAM10] in detail. Finally in Section 2.4, we discuss the problems faced by those techniques.

2.1 Preliminaries on Clustering

In this section, we present some of the basic concepts of clustering, which are used in this thesis.

Clustering is the process of organizing a set of *entities* into subsets or *clusters* such that entities in the same cluster are in "some aspect" more similar to each other than to those in different clusters [Eve74, Lak97]. A *hierarchical clustering algorithm* outputs a hierarchy of clustered entities, as shown in Fig. 2.1. Hierarchical clustering algorithms are of two types: divisive and agglomerative.

Divisive algorithms. Also known as top-down clustering algorithms, these algorithms begin the clustering process by considering all the entities to exist in one cluster, and then iteratively split the cluster into smaller clusters based

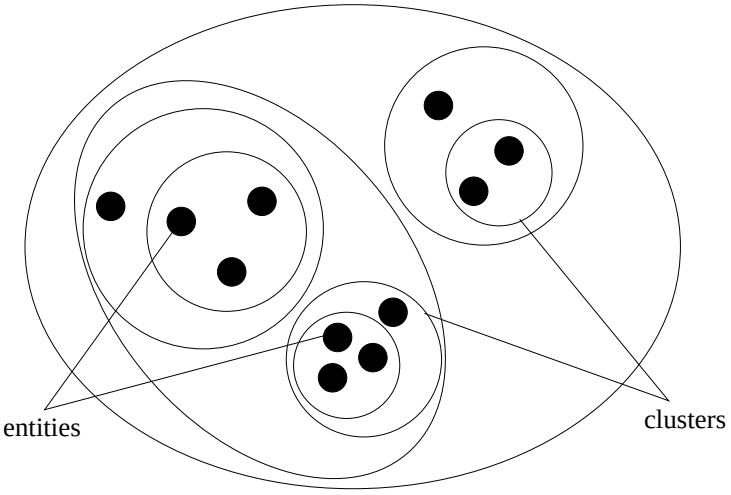


Figure 2.1: A hierarchy of clusters of a set of entities.

on some criteria. The process continues until all entities belong to separate individual clusters. Divisive algorithms suffer from excessive computational complexity, as there is an exponential number of ways to separate the clusters at every step [Tze01]. This is the main reason why these algorithms are not generally used for clustering.

Agglomerative algorithms. Also known as bottom-up clustering algorithms, these algorithms begin the clustering process by considering each entity to belong to a unique cluster. The process then continues by iteratively merging the closest cluster pairs until all entities belong to a single cluster. Hierarchical agglomerative clustering algorithms (HACs) have found much greater application than hierarchical divisive clustering algorithms [Tze01, MRS08, TSK09]. In Section 2.2, we explain HACs in greater detail and mention four previous HACs that have been used for software restructuring.

A common way to visualize the outputs of HACs is through clustering trees or dendrograms (see Fig. 2.2). A *dendrogram* is a two-dimensional diagram, in which a scale of similarity from 1 to 0 is represented in the vertical axis and the entities are indicated in the horizontal axis. The *similarity* quantitatively shows the extent to which the entities are similar. In the diagram, each horizontal line indicates a cluster, the height of which indicates the level of similarity of the cluster components (which can be entities or smaller clusters). Each cluster in the dendrogram, except the one which consists of all the entities, contributes

to a *cut-point*. A *cut-point* is the level of similarity at which a dendrogram can be cut in order to obtain a unique partition of entities. Clusters with the same level of similarity correspond to the same cut-point. The cut-points of the dendrogram in Fig. 2.2 are indicated by the red-dashed lines in Fig. 2.3.

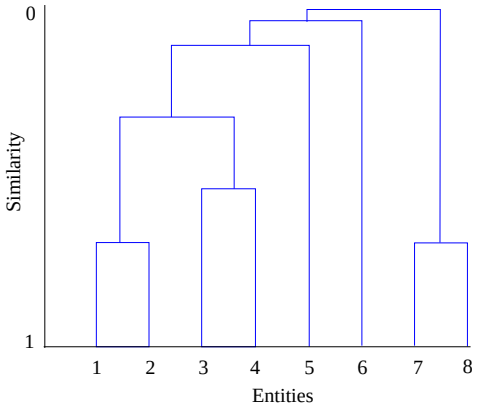


Figure 2.2: A dendrogram.

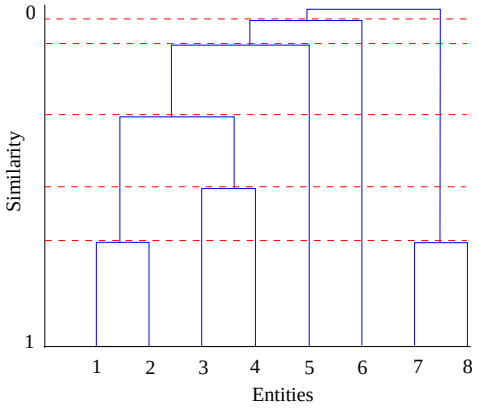


Figure 2.3: A dendrogram with cut-points.

Similarity values between entities are calculated using a metric which considers the various attributes that are common between the entities. The choice of metric depends on what the entities represent. For software restructuring at the function-level, the similarity metric is presented in Section 2.3. The manner in which the similarity values *between the resulting clusters* are calculated is what differentiates one HAC from another. Consequently, each HAC generates a different clustering and thus, a different dendrogram for the same set of entities. We now explore four HACs (SLINK, CLINK, WPGMA, and A-KNN) that have been previously used for software restructuring.

2.2 HACs

In this section, we first explain the basic mechanism of hierarchical agglomerative clustering algorithms. Then we explain four previous hierarchical clustering algorithms that were previously used for software restructuring; SLINK, CLINK, and WPGMA are explained in Subsection 2.2.1 and A-KNN is explained in Subsection 2.2.2.

Algorithm 1 shows the basic steps of any hierarchical agglomerative clustering algorithm (HAC). The algorithm finds a hierarchical clustering of a set of entities. As mentioned earlier, at the start each entity is assigned a separate cluster, which is termed as a singleton cluster. The algorithm proceeds as follows. In Step 1, a similarity matrix is calculated. The *similarity matrix* consists of similarity values (or resemblance values) for each pair of entities. The values are calculated using a formula known as the *resemblance coefficient*. The resemblance coefficient is application-specific and thus depends on what the entities signify in real-life. (One such coefficient is used for software restructuring, which is discussed later in Section 2.3.) The algorithm then enters a loop in Step 2, and repeatedly executes Steps 3 and 4. In Step 3, it merges a pair of clusters that have the highest similarity among all similarity values in the similarity matrix. Thus, a new cluster is formed by the merge. In Step 4, the algorithm recomputes the similarity values between the newly formed cluster and the other clusters. Steps 3 and 4 are repeated until all entities are in a single cluster, as specified in Step 5.

Algorithm 1 *Basic hierarchical agglomerative clustering (HAC) algorithm.*

1: Compute the similarity matrix
2: **repeat**
3: Merge the closest two clusters
4: Update the similarity matrix to show the similarity between the new cluster and the original clusters.
5: **until** Only one cluster remains.

The way in which the similarity between two clusters is calculated differentiates different HACs.

2.2.1 SLINK, CLINK, WPGMA

Single Linkage Algorithm (SLINK), Complete Linkage Algorithm (CLINK), and Weighted Pair Group Method of Arithmetic Averages (WPGMA) directly follow the steps of Algorithm 1. Their distinguishing feature is the way in which they calculate the similarity of two clusters, which is carried out in Step 4 of Algorithm 1. The differences can be illustrated as shown in Fig. 2.4. Each diagram of the figure consists of a pair of clusters consisting of entities. The entities are drawn in a Euclidean plane where the Euclidean distance between the entities is proportional to the dissimilarity between the entities. Thus, the greater the similarity between the entities, the smaller is the distance between them in the diagram.

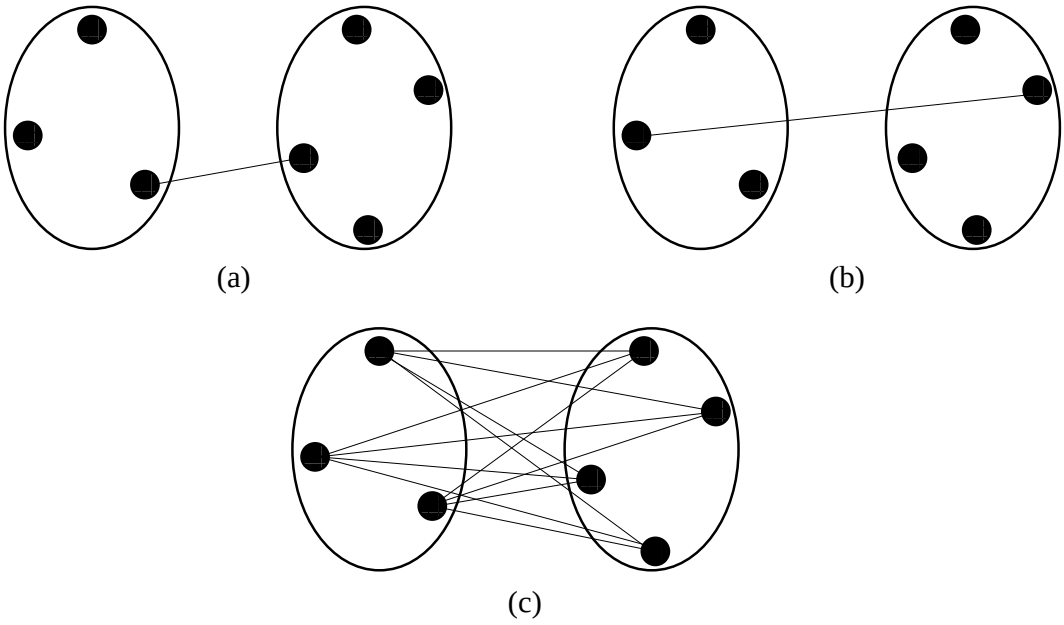


Figure 2.4: Cluster similarity in (a) SLINK, (b) CLINK, (c) WPGMA.

SLINK (Fig. 2.4(a)) defines cluster similarity as the similarity between the closest pair of entities, taking one from each cluster. CLINK (Fig. 2.4(b)) defines cluster similarity as the similarity between the farthest pair of entities, taking one from each cluster. WPGMA (Fig. 2.4(c)) defines cluster similarity as the average pairwise similarity between all pairs of entities from different clusters. Based on these notions of similarity the three algorithms merge clusters, repeatedly, until all entities become part of a single cluster. The output is a hierarchy of clusters.

2.2.2 A-KNN

The Adaptive K-Nearest Neighbour Algorithm (A-KNN) was introduced by Alkhalid *et al.* [AAM10]. Although A-KNN does not directly follow the conventional mechanism of an HAC, like HACs it also generates a hierarchical clustering of entities starting from singleton clusters. A-KNN is based on the idea of the K-Nearest Neighbour (KNN) algorithm introduced by Fix and Hodges [FH89]. The approach of KNN is to classify an entity based on the majority classifications of its nearest k-neighbours. Alkhalid *et al.* proposed a modified version of the KNN algorithm for k=3. Algorithm 2 shows the steps. The algorithm uses labels to indicate the cluster to which an entity belongs. So, for example, if, $L(A)$=$L(B)$, then entities A and B belong to the same cluster. We now explain the steps of the algorithm in detail.

In Step 1, each entity is assigned a unique cluster indicated by distinct labels. Step 2 computes the similarity matrix, just as was done in Step 2 of Algorithm 1. In Step 3, the algorithm enters a loop (Steps 3 to 14). In Step 4, it finds the closest three pairs of entities, $\{E_a, E_b\}, \{E_c, E_d\}$, and $\{E_e, E_f\}$. In the remaining steps of the loop (Steps 5 to 13), the algorithm checks a set of conditions in order to decide with which entity's cluster will the cluster of entity E_a be merged. A number of cases may arise. The first case corresponds to entities E_a, E_c, E_e having the same label, i.e., are in the same cluster (Step 5). There may be two situations in this case. First, E_a's farthest two neighbours among its three closest neighbours (i.e., E_d and E_f) have the same label. In this situation, the algorithm labels E_a with that of E_d (Step 7), merging the corresponding clusters of the respective entities. Second, the three neighbouring entities may all have different labels. In this situation, E_a's cluster is merged with that of its closest neighbour, E_b (Step 9). The last case corresponds to E_a, E_c, E_e all having different labels. In this case E_a's cluster is merged with that of E_b (Step 12). The algorithm repeatedly carries out Steps 3-14 until all entities have the same label and therefore are in the same cluster.

In their algorithm, each new cluster that is formed by merging a pair of clusters is assigned a similarity value equal to that between the closest pair of entities, taking one from each cluster. The output of the algorithm, like that of

the three previous HACs, is a hierarchy of clusters.

The key difference between A-KNN and the previous HACs (SLINK, CLINK, WPGMA) is that A-KNN performs the computation of the similarity matrix only once, whereas, the latter do it at every iteration. A-KNN is thus, more efficient than SLINK, CLINK, and WPGMA. Consequently, it was found to execute faster when it was implemented for restructuring software.

Algorithm 2 *A-KNN algorithm for K=3.*

1: Assign each entity i to a single cluster E_i and label each cluster uniquely. (Let $L(A)$ be the label of cluster A. Then initially $L(E_i) \neq L(E_j)$, $\forall$ $i, j \in \{1, .., n\}$ s.t $i \neq j$, where n is the no. of entities.)
2: Compute the similarity matrix
3: **repeat**
4: Find the most similar three pairs of entities, $\{E_a, E_b\}, \{E_c, E_d\}, \{E_e, E_f\}$, s.t $\text{Sim}(E_a, E_b) \geq \text{Sim}(E_c, E_d) \geq \text{Sim}(E_e, E_f)$, where Sim($A$,$B$) = similarity value for the pair of entities A and B)
5: **if** $L(E_a) = L(E_c) = L(E_e)$ **then**
6: **if** $L(E_d) = L(E_f)$ **then**
7: $L(E_a) \leftarrow L(E_d)$ //merge clusters of E_a and E_e
8: **else**
9: $L(E_a) \leftarrow L(E_b)$ //merge clusters of E_a and E_b
10: **end if**
11: **else**
12: $L(E_a) \leftarrow L(E_b)$ //merge clusters of E_a and E_b
13: **end if**
14: **until** All entities have the same label.

2.3 Restructuring Approach

In this section, we see how the algorithms discussed in Section 2.2 are utilized for restructuring software at the function-level, as was done in [LXZS06], [AAM10]. The three phases of the approach, "Entity-Attribute Matrix Generation", "Similarity Matrix Generation", "Applying Clustering Algorithms" are discussed in

Subsections 2.3.1, 2.3.2, and 2.3.3 respectively.

The overall framework of the restructuring approach is depicted in Fig. 2.5. The technique basically takes a function as input and returns a dendrogram showing a hierarchical representation of the function's entities. Three phases are adopted to carry out this process. In the first phase, "Entity-Attribute Matrix Generation" (Subsection 2.3.1), the presence state of attributes in the entities of the function is noted and stored in a matrix. In the next phase, "Similarity Matrix Generation" (Subsection 2.3.2), similarity values between each pair of entities of the function are calculated based on information provided by the matrix obtained in the previous phase. The output of this phase is a similarity matrix. In the final phase, "Applying Clustering Algorithms" (Subsection 2.3.3), different clustering algorithms are applied on the similarity matrix and cluster hierarchies of the entities are generated. The cluster hierarchies are viewed in the form of a dendrogram, which gives restructuring suggestions for the input function. We now explain each of these phases in detail.

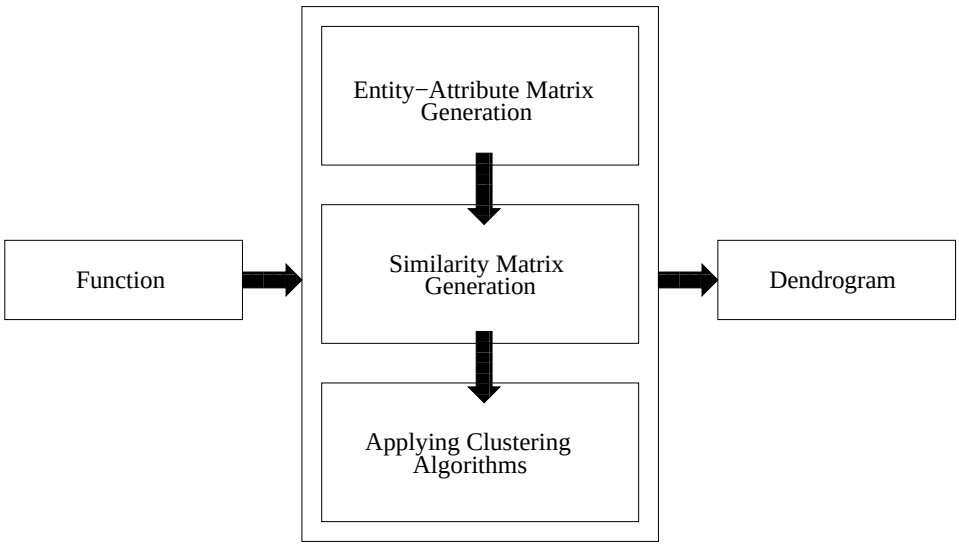


Figure 2.5: Overall approach for restructuring software at the function-level using hierarchical clustering techniques.

2.3.1 Entity-Attribute Matrix Generation

Before explaining this phase it is necessary to define specifically what entities and attributes of a function represent and elaborate on the different types of entities and attributes. All definitions are based on the terminology of Lung *et al.* [LXZS06].

Entities and Attributes

Entities. These resemble components, i.e., *statements*, of a function which are to be grouped by the restructuring process. There can be two types of statements: non-executable and executable. *Non-executable statements* include, declaration and comment statements, which, as mentioned in [LXZS06], "have no real effect on the functionality provided by the function", and hence are not selected as entities. *Executable statements* have a direct effect on the functionality of a function. They include assignment statements, condition statements (e.g., *if* and *else* LOCs), loop statements (e.g., *while* and *for* statements). Only executable statements are chosen as entities.

Entities are classified into two groups: control entities and non-control entities. A *control entity* is an entity which corresponds to a condition/loop statement (e.g., *if*, *else*, *elseif*, *for*, *while* statements or *try*, *catch* statements in Java). If an entity is not a control entity then it is a *non-control entity* (e.g., assignment statements, function call statements).

Attributes. An attribute is a feature or a property of an entity. Entities are grouped on the basis of the number of attributes they share. An entity may have many properties such as variables, constants, operators, keywords, brackets, function names (in function call statements). Since the objective of cohesion-based restructuring is to group entities of the function on the basis of the task they perform, only those properties which are related to a functional activity are chosen as attributes. In the context of function-level restructuring, only *variable names* and *function names* fulfil this criterion, and hence qualify as attributes. Constants, operators, and keywords do not fulfil this criterion, and thus are discarded. In this regard, loop variables are also not chosen as attributes. This is because restructuring is done on the basis of the static structure of the function and hence the number of times a loop body runs is

insignificant.

Attributes selected using the above criteria (variable names and function names) reveal data dependency relationships among entities and hence are classified as *data attributes*. In order to obtain control dependency relationships, new attributes are artificially added to the entities. These attributes are called *control attributes*. Entities that belong to the same control block in the source code (e.g., `if` block), are assigned the same control entity.

Before the start of the restructuring process, relevant entities (i.e., executable statements) are extracted from the function and an *entity-set* is created. Similarly, data attributes are extracted from the function and control attributes are added as necessary. All the attributes constitute the *attribute-set*. (The attribute selection criteria used in our restructuring process, explained in Chapter 3, differs from the attribute selection criteria of Lung *et al.* [LXZS06]. Therefore, in order to distinguish our attribute selection criteria from their criteria, we refer to our attribute extraction as "Selective", and Lung *et al.'s* attribute extraction as "Normal".) We now explain the construction of the entity-attribute matrix.

Entity-Attribute Matrix Construction

The entity-attribute matrix shows the presence states of attributes in each entity of the function. Fig. 2.6 shows the structure of the entity-attribute matrix. All attributes from the attribute-set are shown horizontally in the header of the matrix. Each row corresponds to an entity from the entity-set and indicates the presence states of attributes in the particular entity. Each cell, (i, j) (where i, the row number, refers to an entity, and j, the column number, refers to an attribute), of the matrix can have three values, 0, 1, or 2, which are described below,

0 - attribute j is absent in entity i,

1 - attribute j is present in entity i, where,

i. attribute j is a control attribute or,

ii. attribute j is a data attribute and entity i is a control entity,

2 - attribute j is present in entity i, where attribute j is a data attribute

and entity i is a non-control entity.

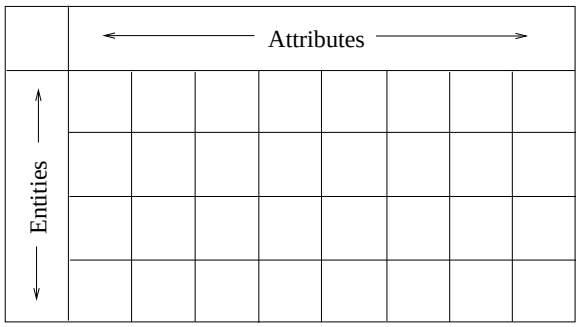


Figure 2.6: Entity-attribute matrix.

```
0  void sum_and_prod(int n, int[] arr) {
1  sum = 0;
2  prod = 1;
3  for ( int i = 1 ; i < n ; i ++ )
4  {
5  sum = sum + arr [ i ] ;
6  prod = prod * arr [ i ] ;
7  }
8  avg = sum / n ;
9  }
```

Figure 2.7: sum_prod function with each line enumerated.

Let us now see an example where we restructure the sum_prod function that was shown earlier in Fig. 1.1, Chapter 1. Fig. 2.7 shows the function with each statement (entity) of the function enumerated and Fig. 2.8 shows the entity-attribute matrix for the function. As can be seen, only the relevant entities and attributes are included in the entity-attribute matrix. The values of the matrix are based on the presence state of the attributes in the entities. For example, variable `sum` exists as a data attribute in entities 1, 5, 8. Therefore, the cells corresponding to these entities, under the `sum` column, are assigned the value '2'. The remaining cells of this column are assigned '0'. Variable `n` is present as a data attribute in entities 3 and 8. Since entity 3 is a control statement, the cell corresponding to entity 3, under n's column, is assigned '1'. The cell corresponding to entity 8 in this column is assigned '2' as usual. The *for* attribute is the extra control attribute that has been added to indicate the

control dependency relationship among the entities. The cells of all entities, which are inside the `for` block of the function, are assigned '1' under the `for` column.

In this manner, the restructuring technique generates the entity-attribute matrix for any input function. From the entity-attribute matrix, useful information about the similarities between the entities is obtained and processed for suitable quantitative representations. This is done in the next phase of the restructuring process.

Entity	Attributes					
No.	n	arr	sum	$prod$	avg	for
1	0	0	2	0	0	0
2	0	0	0	2	0	0
3	1	0	0	0	0	1
5	0	2	2	0	0	1
6	0	2	0	2	0	1
8	2	0	2	0	2	0

Figure 2.8: Entity-attribute matrix of sum_prod function.

2.3.2 Similarity Matrix Generation

This phase generates a similarity matrix from the information provided by the entity-attribute matrix generated in the previous phase. Fig. 2.9 shows the structure of a similarity matrix. The similarity matrix consists of similarity values for each pair of entities.

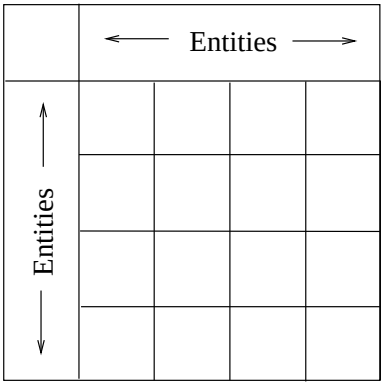


Figure 2.9: Similarity matrix.

The similarity values give a quantitative measure of the similarity that exists between each pair of entities. The values are calculated using a formula based on the Jaccard coefficient of similarity [AL99], which is given as under,

$$Sim(A, B) = \frac{a}{a + b + c},$$

where,
$Sim(A, B)$ is the similarity between entities A and B,
a is the number of common attributes among A and B,
b is the number of attributes present in A but not in B,
c is the number of attributes present in B but not in A.

Based on the above, Lung *et al.* [LXZS06] used a new metric to calculate the similarity between two entities. The metric is based on attribute match count information of entity-pairs, which is obtained from the entity-attribute matrix. We first explain the different matches that can be obtained between two entities from the entity-attribute matrix.

1-1 matches. A 1-1 match either indicates that two entities have the same control attribute, i.e., have a control dependency relationship, or two control entities have the same data attribute.
2-2 matches. A 2-2 match indicates that two non-control entities have the same data attribute, i.e., have a data dependency relationship.
1-0/0-1 matches. A 1-0/0-1 match indicates either a mismatch between two non-control entities, where a control attribute is present in one entity and absent in the other, or a mismatch between two control entities, where a data attribute is present in one entity and absent in the other.
2-0/0-2 matches. A 2-0/0-2 match indicates a mismatch between two non-control entities, where a data attribute is present in one entity and absent in the other.
0-0 matches. A 0-0 match indicates that an attribute is present in neither of the two entities. The similarity of two entities is not affected by adding attributes to other entities of a function [LXZS06], because the absence of an attribute in a pair of entities does not convey any information regarding the similarity/dissimilarity between the entities. Moreover, since there can be many attributes used in a function, it is likely that a pair of entities may not have

many of those attributes and thus may have many 0-0 matches. It has been shown that considering such matches would distort clustering results [AL03]. 0-0 matches are thus ignored.

1-2/2-1 matches. A 1-2/2-1 match occurs between a control entity and a non-control entity where both have the same data attribute. There can be two scenarios for such type of matches. The first is that, *the two entities are in the same control block.* In this case, there already exists a 1-1 match between these two entities that considers their control dependency relationship, and thus there is no need to note it again with a 2-1 match. The other scenario is that *the two entities are not in the same control block,* in other words, the non-control entity is outside the control block. In this case, there is no control dependence between the entities. Therefore, 1-2/2-1 matches are ignored.

Lung *et al.* constructed an entity-pair similarity metric, known as the *resemblance coefficient*, which is based on the Jaccard coefficient described earlier. The resemblance coefficient uses match count information of 1-1, 2-2, 1-0/0-1, and 2-0/0-2 matches between any pair of entities. 0-0 and 1-2/2-1 matches are not considered. The formula is given as under,

$$coeff_{(A,B)} = \frac{w_d a_d + w_c a_c}{w_d a_d + w_c a_c + w_d b_d + w_c b_c},$$

where,
$coeff_{(A,B)}$ is the resemblance coefficient or similarity value for entity pair (A, B) that varies from 0 to 1,
a_d is the number of 2-2 matches between entities A and B,
a_c is the number of 1-1 matches between entities A and B,
b_d is the number of 2-0/0-2 matches between entities A and B,
b_c is the number of 1-0/0-1 matches between entities A and B,
w_d is the weight of data attributes,
w_c is the weight of control attributes,
$w_d > w_c > 0$.

As can be seen the resemblance coefficient, unlike the Jaccard coefficient, distinguishes between control attributes and data attributes in order to distinguish between data and control dependency relationships. This differentiation is made because two variables that have a data dependency relationship are

more cohesive than two variables that have only a control dependency relationship [Lak93]. As a result, more weight is assigned to data attributes than to control attributes. The weights are specified by weight parameters w_d and w_c. Different weight ratios (3:1, 5:2, 8:3) were used for restructuring functions in previous works. A weight ratio of 8:3 was found to give the most consistent restructuring results and was chosen by Lung *et al.* [LXZS06].

We now return to our example of restructuring the sum_prod function in Fig. 2.7. By using the resemblance coefficient and the match count information, obtained from the function's entity-attribute matrix (see Fig. 2.8), similarity values for each pair of entities of the function is computed. Below we show the calculation of the similarity value for entity pair (3, 5) using weights w_d=8, w_c=3,

From the entity-attribute matrix, we see that for entities 3 and 5, a_d=0, a_c=1, b_d=2, and $b_c = 1$. Therefore,

$$coeff_{(3,5)} = \frac{8*0+3*1}{8*0+3*1+8*2+3*1}$$
$$\simeq 0.14$$

In similar fashion, the similarity values for all the other entity-pairs are calculated. The results constitute the cells of the similarity matrix, as shown in Fig. 2.10. All calculations were made using the weight ratio of 8:3 for w_d:w_c.

	1	2	3	5	6	8
1	0	0	0	0.42	0	0.33
2	0	0	0	0	0.42	0
3	0	0	0	0.14	0.14	0
5	0.42	0	0.14	0	0.41	0.23
6	0	0.42	0.14	0.41	0	0
8	0.33	0	0	0.23	0	0

Figure 2.10: Similarity matrix of sum_prod function.

2.3.3 Applying Clustering Algorithms

Once the similarity matrix of a function is generated, HAC algorithms are directly applied on the matrix in the manner explained in Section 2.2. A hierarchy of entities is returned as the output. The hierarchy is visually represented as a dendrogram. As was mentioned earlier in Section 2.1, a dendrogram gives suggestions on how to partition the set of entities. The suggestions are given in the form of cut-points. Since each clustering algorithm returns a different clustering of the entities, the corresponding dendrograms returned by the algorithms also differ from one another, and hence give different cut-points. It is entirely for the developer to decide which cut-point(s) to choose for performing the restructuring. Factors that would influence the developer's decision include the restructuring objective (which in this case is to increase cohesion), the developer's experience and programming knowledge, etc. We now demonstrate how the restructuring process is carried out with the aid of a dendrogram.

Fig. 2.11 shows the sum_prod function and its corresponding dendrogram generated when the WPGMA algorithm is applied on the similarity matrix (Fig. 2.10) of the function. On analyzing the dendrogram, we see that it gives three cut-points, C_1, C_2, and C_3.

C_1 yields the partition {(6,2,3),(8,5,1)}, of which one cluster consists of all entities[1] of the function that deal with the `prod` variable, and the other cluster contains those that deal with the `sum` variable. The two clusters successfully identify the two tasks carried out by the function (resembled by `sum` and `prod` variables) and suggest each to be on separate functions. Since the goal of cohesion-based restructuring is to rewrite a function into separate functions where each function performs a unique task, this cut-point clearly yields suggestions that would lead to a desired restructuring (shown in Fig. 1.2, in Chapter 1). Note that entity 3 is a control entity (a `for` statement) that must exist with both entity 5 and entity 6. However, it was not included with entity 5 in cluster (8,5,1). Thus, in this cluster, entity 5 has an *implicit control dependence* with entity 3. Implicit control dependencies can be easily detected by the programmer and can accordingly be retained in the restructured version even without the dendrogram's assistance. Since cluster (8,5,1) correctly groups a

[1]Entities refer to the executable statements of the function (see Subsection 2.3.1).

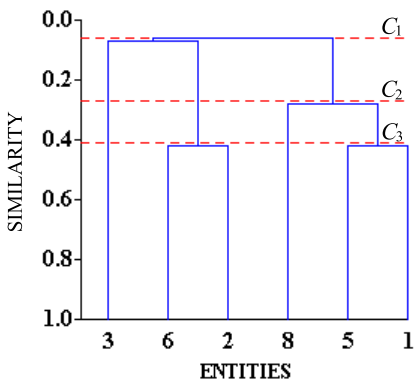
unique task of the function (resembled by the `sum` attribute usages), it is not a redundant cluster. (The properties of redundant clusters are elaborated upon in Chapter 4.)

C_2 yields the partition {(3),(6,2),(8,5,1)} of which the latter two clusters lead to the desired restructuring. Singleton cluster (3) is discarded. Singleton clusters do not give any grouping information and so do not give any advice on restructuring. Thus, *all singleton clusters are ignored during the analysis.*

C_3 yields the partition {(6,2),3,8,(5,1)}. This partition gives two groups of entities, (6,2) and (5,1). Cluster (6,2) groups all the entities that use `prod`, whereas cluster (5,1) does not include all the entities that use `sum`. In the context of this program, it is meaningless to keep entity 8 separate from entities 5 and 1 because doing so would imply either keeping entity 8 with entities 6 and 2, in case we were not to devote a separate function for entities 6 and 2, or keeping entity 8 alone in a separate function. Both these versions are impractical as one suggests keeping entity 8 with entities which have no relation to it, and the other suggests a function with a single entity. Therefore, cluster (5,1) is a redundant cluster.

```
0  void sum_and_prod(int n, int[] arr) {
1  sum = 0;
2  prod = 1;
3  for ( int i = 1 ; i < n ; i ++ )
4  {
5  sum = sum + arr [ i ] ;
6  prod = prod * arr [ i ] ;
7  }
8  avg = sum / n ;
9  }
```

(a)



(b)

Figure 2.11: (a) sum_prod function, and (b) its corresponding dendrogram (with cut-points) obtained by WPGMA algorithm.

2.4 Discussion on the Problems Faced

In this section, we elaborate on the problems faced by the previous hierarchical clustering techniques. In Subsection 2.4.2, we explain why the problems occur,

and in Subsection 2.4.3, we propose ways to address the problems.

2.4.1 The Problems

When implemented on functions, SLINK, CLINK, WPGMA, and A-KNN give dendrograms with the following properties,

Large number of cut-points. A greater number of cut-points clutters the dendrogram and makes it increasingly difficult for the developer to choose the appropriate cut-point(s) that lead to the desired restructuring.
Large number of redundant clusters. The cut-points return a large number of redundant clusters, i.e., clusters that do not lead to a meaningful restructuring of the function. Analyzing such clusters wastes precious time of the developer while he/she uses the clustering results to obtain a restructuring of the function.

Clearly, a dendrogram that gives a smaller number of cut-points that return a smaller number of redundant clusters is desirable.

2.4.2 Causes of the Problem

We identify the main reason behind the problems of the previous HACs, mentioned in Subsection 2.4.1, as the fact that they generate hierarchies with a large number of small clusters. Since each cluster in a hierarchy contributes to a cut-point in the corresponding dendrogram, the dendrogram would have a large number of cut-points. Consequently, when these cut-points are analyzed, many of them yield small clusters. In practice, it has been found that such clusters, because of their limited sizes, tend to leave out many related entities and as a result are likely to lead to a meaningless restructuring. For example, as elicited earlier, cut-point C_3 in Fig. 2.11 yields a cluster (5,1), which was found to be meaningless. Although the cluster successfully detects the *sum* attribute relationship between entities 5 and 1, it does not include entity 8 - an entity which also deals with the *sum* attribute. The larger cluster (1,5,8), produced by the other two cut-points, encompasses all the three related entities.

There are two factors in the mechanisms of the previous HAC techniques that contribute to the generation of small clusters. One factor is that their technique is based on merging clusters pairwise (see Algorithms 1, 2 in Section 2.2). This

means at every iteration only two clusters are merged to form a new cluster. The other factor that contributes to the generation of small clusters is that they only consider cluster/entity similarities when making the merge decisions. In doing so, they limit their merging decisions to only a single aspect of the entities' interrelationships, while discarding other aspects which may be present.

2.4.3 A Solution to the Problem

In order to reduce the number of cut-points in dendrograms and improve the quality of those cut-points (i.e., reduce the number of redundant clusters in cut-points), a hierarchical clustering technique that generates larger clusters could be used. Moreover, since software functions can be of varying design, it would be of utmost significance if the technique generates larger clusters intuitively. In other words, the technique should not require the user to predetermine any value on the minimum cluster size or the number of clusters. This objective may be fulfilled if the technique considers other structural properties of the entities in addition to their inter-similarities. In this thesis, we have developed such a technique, which is explained in the next chapter.

Chapter 3

Restructuring Using the Proposed Clustering Technique

In this chapter, we present a new hierarchical clustering technique, (k, w)-Core Clustering ((k, w)-CC), for restructuring software at the function-level. Since the new technique is based on (k, w)-core decomposition, a graph theoretic algorithm, we first give some basic graph definitions in Section 3.1. Then in Section 3.2, we explain (k, w)-CC. Finally, in Section 3.3, we propose a modified attribute selection strategy for (k, w)-CC instead of the one discussed in Subsection 2.3.1 of Section 2.3.

3.1 Graph Theory Definitions

In this section, we present some basic graph concepts which will be required to understand (k, w)-CC. In Subsection 3.1.1, we give definitions of standard graph-theoretical terms. In Subsection 3.1.2, we explain the concepts of k-cores and (k, w)-cores.

3.1.1 Graphs

A *graph* G is a tuple (V, E), which consists of a finite set V of *vertices* and a finite set E of *edges*; each edge is an unordered pair of vertices [NR04].

Fig. 3.1 depicts a graph $G = (V, E)$ where each vertex in $V = \{v_1, v_2, \ldots, v_6\}$ is denoted by a small circle and each edge in $E = \{e_1, e_2, \ldots, e_9\}$ is denoted by a line segment. An edge joining two vertices u and v of the graph $G = (V, E)$

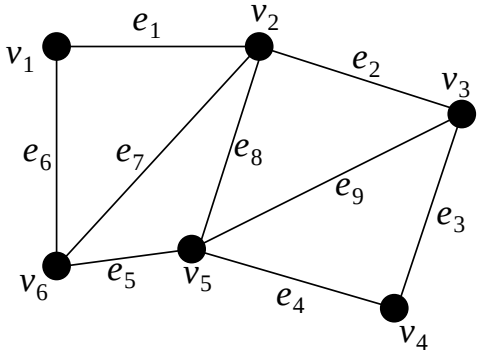


Figure 3.1: A graph with six vertices and nine edges.

can be denoted by (u, v) or simply by uv. If $uv \in E$, then the two vertices u and v of the graph G are said to be *adjacent*; the edge uv is then said to be *incident* to the vertices u and v; also the vertex u is said to be a neighbour of the vertex v (and vice versa). The *degree* of a vertex v in G, denoted by $d(v)$ or $deg_G(v)$, is the number of edges incident to v in G. In the graph shown in Fig. 3.1, vertices v_1 and v_2 are adjacent, and $d(v_6) = 3$, since three edges (e_5, e_6, e_7) are incident to v_6.

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. If G' contains all the edges of G that join vertices in V', then G' is called the subgraph induced by V'.

A *weighted graph* is a graph where a real value is associated with each edge of the graph.

A graph can be connected or disconnected. A graph G is *connected* if for any two distinct vertices u and v of G, there is a path between u and v. A graph which is not connected is called a *disconnected graph*.

A *(connected) component* of a graph is a maximal connected subgraph. The graph in Fig. 3.2(a) is a connected graph since there is a path between every pair of distinct vertices of the graph. On the other hand, the graph in Fig. 3.2(b) is a disconnected graph since there is no path between, say, v_1 and v_5. The graph in Fig. 3.2(b) has two connected components as indicated by the dotted lines. Note that every connected graph has only one component; the graph itself.

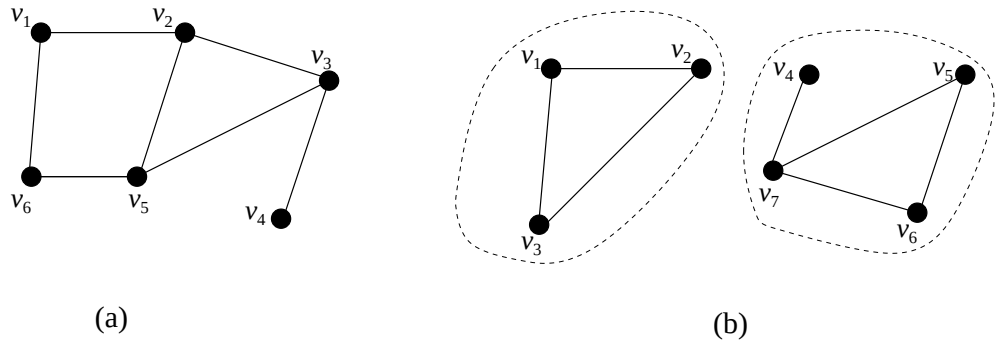


Figure 3.2: (a) A connected graph, (b) A disconnected graph with two connected components.

3.1.2 k-cores

We now give the definitions of two key elements of the (k, w)-CC technique: the k-core, first introduced by Seidman [Sei83], and the (k, w)-core introduced in this thesis.

Definition 3.1.1. ***k-core.*** *Let $G = (V, E)$, be a graph, where V is the set of vertices and E is the set of edges. A subgraph H_k of G induced by a vertex set $V' \subseteq V$ is a k-core of G if every vertex in V' has degree at least k in H_k, and H_k is the maximum subgraph with this property [BZ03].*

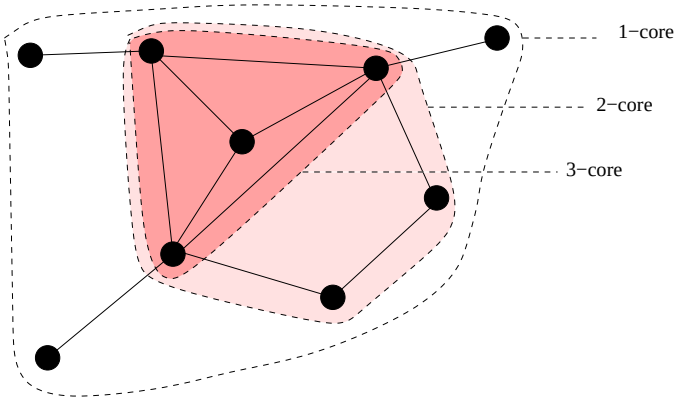


Figure 3.3: k-cores of a graph, G.

Fig. 3.3 indicates all the k-cores of G. We now present the following lemma,

Lemma 3.1.1 *If H_{k_1}, H_{k_2} are the k_1- and k_2-cores, respectively, of a graph G, where $k_2 > k_1$, then H_{k_2} is a subgraph of H_{k_1}.*

Proof. (Proof by Contradiction.) Let us assume that H_{k_1}, H_{k_2} are the k_1- and k_2-cores, respectively, of a graph G, such that $k_2 > k_1$. Suppose that H_{k_2} is not a subgraph of H_{k_1}.
By the definition of k-core, each vertex in H_{k_1} has a degree of at least k_1, and each vertex in H_{k_2} has a degree of at least k_2. As $k_2 > k_1$, then clearly each vertex in H_{k_2} has a degree $> k_1$. Thus, we have two subgraphs, H_{k_1}, H_{k_2} where each vertex of the subgraphs has degree $\geq k_1$. Since H_{k_2} is not a subgraph of H_{k_1}, then H_{k_1} is not the maximum subgraph in which all vertices have degree greater than or equal to k_1. Thus, H_{k_1} is not a k_1-core which is contradictory to what we had assumed. Therefore, H_{k_2} must be a subgraph of H_{k_1}.

$\mathcal{Q.E.D.}$

For our purpose we shall deal with weighted graphs. As was mentioned earlier, a real value is associated with each edge of a weighted graph. We introduce the notion of (k, w)-cores and present a lemma in this regard.

Definition 3.1.2. (k, w)***-core.*** *Let W be the set of different edge weights of graph G, where $w \in W$. Then a (k, w)-core of G is a subgraph of G where the degree of each vertex of the subgraph is at least k and the weight of each edge of the subgraph is at least w, and this subgraph is the maximum subgraph with this property.*

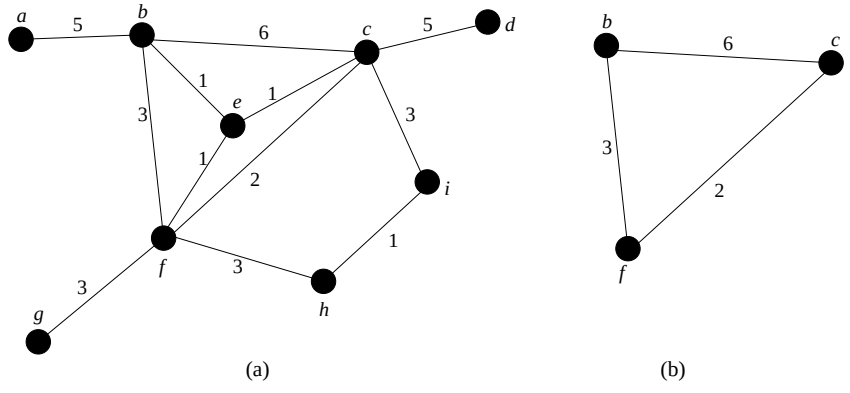


Figure 3.4: (a) A weighted graph G', (b) A $(2, 2)$-core of G.

Fig. 3.4 indicates the $(2, 2)$-core of G'. We now present the following lemma,

Lemma 3.1.2 *A (k, w)-core of a weighted graph, G, is a subgraph of a k-core of G.*

Proof. (Proof by Contradiction.) Let us assume that H_{k_1}, H_{k_1,w_1} are the k_1- and (k_1, w_1)-cores, respectively, of a graph G. Suppose that H_{k_1,w_1} is not a subgraph of H_{k_1}.

By the definitions of k-core and (k, w)-core, each vertex in H_{k_1} and H_{k_1,w_1} has degree $\geq k_1$. Since H_{k_1,w_1} is not a subgraph of H_{k_1}, then H_{k_1} is not the maximum subgraph in which all vertices have degree greater than or equal to k_1. Thus, H_{k_1} is not a k_1-core which is contradictory to what we had assumed. Therefore, H_{k_1,w_1} must be a subgraph of H_{k_1}.

$\mathcal{Q.E.D.}$

3.2 (k, w)-Core Clustering

In this section, we explain the new hierarchical clustering technique, (k, w)-Core Clustering ((k, w)-CC), that was introduced in this thesis. In order to implement our technique for software restructuring, we follow the restructuring approach explained in Section 2.3. In Subsections 3.2.1, 3.2.2, and 3.2.3, we describe the three main phases of our technique. In Subsection 3.2.4, we present a time complexity analysis of our entire technique.

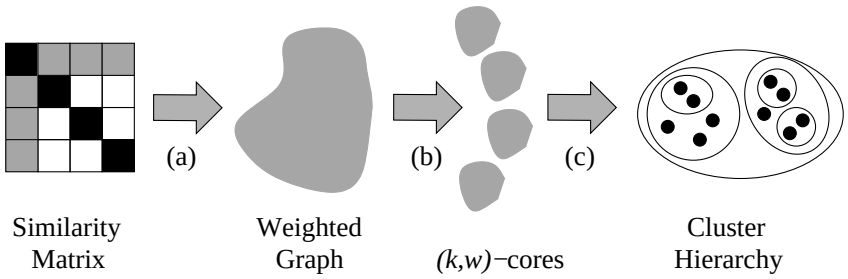


Figure 3.5: Overall approach of (k, w)-CC.

Fig. 3.5 illustrates the overall approach of (k, w)-Core Clustering. In the first step (Fig. 3.5(a)), a weighted graph is realised from the similarity matrix (which was obtained from the "Similarity Matrix Generation" phase). In the next step (Fig. 3.5(b)), all possible (k, w)-cores are generated from the weighted graph. In the final step of the approach (Fig. 3.5(c)), cores are systematically selected to form clusters, which together form a cluster hierarchy. Each of the steps are discussed in the following subsections.

3.2.1 Graph Generation

In this phase, a graph is obtained from the similarity matrix. Like the previous HACs, (k, w)-CC also takes the similarity matrix as an input. However, since it is a graph theoretic technique, it works on weighted graphs. A weighted graph is conveniently obtained from the similarity matrix, which serves as an adjacency matrix for the graph. In the graph, vertices represent entities, and edges incident to the vertices represent the presence of non-zero similarities between the corresponding entities. Each edge carries a weight equal to the similarity value between the corresponding entities.

3.2.2 Core Decomposition

This step comprises of generating all the (k, w)-cores of the graph obtained in the previous step. To carry this out, we developed new core decomposition algorithms. The basic idea of our approach is to first generate all the k-cores of the graph. Then from the k-cores we generate all the (k, w)-cores from the graph.

k-core Decomposition

Batagelj *et al.* [BZ03] gave an implementation for generating all the k-cores of a graph. The basic property of their algorithm is: if from a given graph G we recursively delete all vertices, and edges incident to them, of degree less than k, the remaining graph is the k-core of G. Based on this property, we developed algorithms for generating all k-cores and all (k, w)-cores of a graph.

Algorithm 3 shows the steps for generating all the k-cores of a weighted graph $G(V, E)$, where k belongs to the sorted set (in ascending order), D, of distinct degrees of the vertices of G. The algorithm takes input G in P (Step 1) and scans every vertex v of V_P to find the k-core for the smallest $k \epsilon D$. In each iteration, if the degree of a vertex v, $deg_P(v)$, is found to be below k, v is deleted (Steps 4-7) and the degrees of vertices adjacent to v in P are decremented using the update function in Algorithm 4 (Steps 2, 3 in Algorithm 4). In case the degrees of any of v's adjacent vertices, $N_P(v)$, fall below k as a result of the decrements, those vertices are deleted and the update function is recursively called again (Steps 4-7 in Algorithm 4). In this manner, the first k-core is

generated (Step 9, Algorithm 3). For generating the k-core for the next value of k in ordered set D, it is sufficient to analyze the last k-core that was generated instead of the entire graph G (see Lemma 3.1.1). Thus, at every stage, each k-core that is generated is again fed into Steps 3-8 of Algorithm 3 to obtain the next k-core. This process continues until the k-core for the largest value of $k \epsilon D$ has been generated.

Algorithm 3 *Generating all k-cores of weighted graph, $G(V, E)$*

1: $P(V_P, E_P) \leftarrow G(V, E)$ //Input
2: **for** each degree value k in D **do**
3: **for** each vertex v in V_P **do**
4: **if** $deg_P(v) < k$ **then**
5: $deg_P(v) \leftarrow 0$
6: update(v, k, P) //Completes deletion of v in P *(Algorithm 4)*
7: **end if**
8: **end for**
9: $H_k \leftarrow P(V_P, E_P)$
10: **return** H_k
11: **end for**

Algorithm 4 *Update connected vertices*

1: update(v, k, P)
 {
2: **for** each vertex u in $N_P(v)$ **do**
3: $deg_P(u) \leftarrow deg_P(u)$-1
4: **if** $deg_P(u) < k$ **then**
5: $deg_P(u) \leftarrow 0$
6: update(u, k, P)
7: **end if**
8: **end for**
 }

(k, w)-core Decomposition

By Lemma 3.1.2, for a particular value of k, say k_n, all (k_n, w)-cores of G can be obtained from the k_n-core of G. The steps required to achieve this are shown

in Algorithm 5. A k-core, H_k, is taken as input in P (Step 1). Let w belong to the sorted set (in ascending order), W, of distinct edge-weights of the original graph G. In Steps 2-11, for each weight w, all edges in P with weights (denoted by W) less than w are deleted and the remaining graph, the *intermediate graph*, is stored in I. In this way, intermediate graphs for all $w \epsilon W$ are obtained, and stored in I.

Now, the deletion of an edge (Step 6) decrements degrees of vertices on which the edge is incident. Henceforth, the degrees of some vertices of the intermediate graphs may have fallen below k, violating the k constraint of the (k, w)-core. Thus, in Steps 12-19, the degrees of the vertices of each intermediate graph of I is checked. In the process, for every vertex deletion (Step 15-16), the recursive update function (in Algorithm 4) is called, just as was done in Algorithm 3. The output is a set of all (k, w)-cores of G for the value of k, determined by the k-core. So for example, if a 3-core of G is input to Algorithm 5, all $(3, w)$-cores of G will be generated. Thus, in order to obtain all the (k, w)-cores of G, each k-core obtained from Algorithm 3 is input to Algorithm 5.

Algorithm 5 *Generating all (k, w)-cores of $G(V, E)$ for a certain value of k.*

$P(V_P, E_P) \leftarrow H_k(V_k, E_k)$
for each weight w in W **do**
  for each vertex v in V_P **do**
    for each vertex u in $N_P(v)$ **do**
      if $W(u, v) < w$ **then**
        ${E'}_k \leftarrow {E'}_k \cup (u, v)$ where ${E'}_k$ is the set of edges in H_k with $W < w$
      end if
    end for
  end for
  $I \leftarrow \{I \cup H'_k(V_k, E_k - {E'}_k)\}$
end for
for each intermediate graph $I'(V_{I'}, E_{I'})$ in I **do**
  for each vertex v in $V_{I'}$ **do**
    if $deg_{I'}(v) < k$ **then**
      $deg_{I'}(v) \leftarrow 0$
      update (v, k, I') //Completes deletion of v in I' *(Algorithm 4)*
    end if
  end for
  $H_{k,w} \leftarrow I'(V_{I'}, E_{I'})$
  return $H_{k,w}$
end for

3.2.3 Core Selection and Clustering Tree Generation

The final step of (k, w)-CC comprises of selecting cores and generating the clustering tree or dendrogram. Firstly, we calculate the relatedness of the entities in each core using a new metric, called core relatedness, which has been designed in this thesis. We then systematically select the cores for generating a clustering tree.

Core Relatedness Calculation

The *core relatedness* metric gives a quantitative measure of the level of similarity, in the range of 0 to 1, between the vertices (entities) of a core. We formulate

the relatedness, $R(H_{k,w})$, for a (k, w)-core, $H_{k,w}$, as,

$$R(H_{k,w}) = strength_k * share_k + strength_w * share_w,$$

where

$$strength_k = \frac{k}{degree_{max}},$$

$$strength_w = \frac{w}{weight_{max}}.$$

In the above formula, $strength_k$ resembles the *structural relatedness* of $H_{k,w}$, and $strength_w$ resembles the *weight relatedness* of $H_{k,w}$. $degree_{max}$ is the maximum degree in D and $weight_{max}$ is the maximum weight in W. $share_k$ and $share_w$ are the percentage contributions to the overall relatedness of the core by the structural and weight relatedness parameters respectively.

As can be seen, R considers both the structural relationship ($strength_k$) and the weight relationship ($strength_w$) of the vertices in a core in order to determine the overall relatedness of the core's vertices. $strength_k$ captures the state of interconnectivity of the vertices in a core as determined by the core's k value, which is an important aspect of the degree to which the core's vertices are interrelated. Considering this relationship to determine the overall relatedness of the core's vertices helps in fulfilling our objective of obtaining larger clusters, as cores with high $strength_k$ have high k values, and thus will have a larger number of vertices. Selecting such cores would yield larger clusters. However, it is important to note that evaluating cores solely on $strength_k$ would lead to the selection of cores that are impractically large. Moreover, while the interconnectivity state of the vertices is an important aspect of their relatedness, the extent to which one vertex is similar to the other as determined by the corresponding similarity value cannot be ignored. For this reason, R also considers the inter-entity similarity, $strength_w$, of the entities in a core. The parameters $share_k$ and $share_w$ denote the level of importance that should be given to these two relationships in determining the overall R value for a core. It has been experimentally found that better results are obtained if more importance is given to $strength_w$ in calculating the relatedness of a (k, w)-core. Best results were obtained with percentages of 30% and 70% for $share_k$ and $share_w$, respectively.

Using the above metric, R values for all the cores are computed. Then the

cores are sorted in descending order, based on their R values, and entered in set C. Since there may be disconnected cores in C, a connectivity check is made on the cores. If a core $H_{k,w}$ is found to be disconnected, consisting of x subcomponents, $H_{k,w}$ is removed from the ordered sequence in C and replaced by the subcomponents, $H^1_{k,w}, H^2_{k,w}, \ldots, H^x_{k,w}$, where each of the subcomponents are assigned the same R value as that of the removed core, $H_{k,w}$.

Generating Clusters from Cores

By Algorithm 6, cores are directly selected as clusters from the ordered set, C. At every iteration, a scanned core, $H_{k,w}$, is interpreted as a *candidate cluster*, $F_{k,w}$ (Step 3). If all entities of $F_{k,w}$ have already been clustered it is ignored (Steps 5-7). If $F_{k,w}$ consists of entities of which some have already been clustered, $F_{k,w}$ is merged with the immediately previous clusters in which the common entities are present (Steps 8-16). The relatedness of the new cluster formed being $R(F_{k,w})$. If none of the entities of $F_{k,w}$ have been clustered earlier, it is selected as a cluster (Step 16). The steps are repeated until all entities have been clustered. The output clusters are stored in set, C_{final}, which gives us the cluster hierarchy. However, if the last cluster in C_{final} is not found to contain all the entities in G, C_{final} would yield a disconnected hierarchy as the entities have not been clustered into a single cluster. In this situation, all the disjoint clusters in C_{final} are merged into a single cluster with an R value of 0 and added to C_{final}. Thus, C_{final} gives us a hierarchy of clusters, which can be conveniently viewed as a clustering tree or dendrogram.

Algorithm 6 *Generating clusters from cores, of $G(V, E)$, with $k > 1$*

1: Let Q be the set of entities representing each vertex in V and C_{final} be the final set of clusters
2: **for** each $H_{k,w}(V_{k,w}, E_{k,w})$ in the ordered set of cores, C, **do**
3: $F_{k,w} \leftarrow$ cluster consisting of all vertices in $V_{k,w}$
4: **if** $k \neq 1$ **then**
5: **if** $F_{k,w} \cap Q = \phi$ (the current cluster has no new entities) **then**
6: **continue**
7: **end if**
8: **for** each entity e in $F_{k,w}$ **do**
9: **for** each cluster F_i in C_{final} (starting from the last F_i in F) **do**
10: **if** $e \in F_i$ **then**
11: $F_{k,w} \leftarrow F_{k,w} \cup F_i$
12: **break**
13: **end if**
14: **end for**
15: **end for**
16: Enter $F_{k,w}$ in C_{final}, where $R(F_{k,w}) = R(H_{k,w})$
17: $Q \leftarrow Q - F_{k,w}$
18: $C \leftarrow C - H_{k,w}$
19: **if** $Q = \phi$ (all entities have been clustered) **then**
20: **break**
21: **end if**
22: **end if**
23: **end for**

A key component of this algorithm is specified by the condition statement in Step 4. It signifies that cores with $k = 1$ are ignored and hence given less preference in the cluster generation process. This is because $(1, w)$-cores, regardless of their R values, have the lowest structural relatedness among all the cores. As a result, $(1, w)$-cores are only selected when there are un-grouped entities remaining and all other cores in C have already been selected, in which case Algorithm 6 is repeated without the condition in Step 4.

3.2.4 Time Complexity Analysis

Time complexities of the algorithms were evaluated based on the most widely accepted complexity measure in computer science, the *running time*, which is the number of operations an algorithm performs before producing the final output [GJ79]. Before presenting the time complexity of our technique, we first briefly describe the basic ideas and notations related to time complexity, based on the definitions of [NR04].

Basic Ideas on Time Complexity

Since the number of operations required by an algorithm to produce the final output is not the same for all problem instances, all inputs of a given size are considered together, and the *complexity of the algorithm for that input size* is the worst case behaviour of the algorithm on any of these inputs. Consequently, the running time is a function of size n of the input.

The Notation $O(n)$. In analyzing the complexity of an algorithm, we are often interested only in the "asymptotic behaviour", that is, the behaviour of the algorithm when applied to very large inputs. To deal with such a property of functions we shall use the following notations for asymptotic running time. Let $f(n)$ and $g(n)$ be the functions from the positive integers to the positive reals. Then we write $f(n) = O(g(n))$ if there exists positive constants c_1 and c_2 such that $f(n) \leq c_1 g(n) + c_2$ for all n. Thus, the running time of an algorithm may be bounded from above by phrasing like "takes time $O(n^2)$".

Polynomial Algorithms. An algorithm is said to be *polynomially bounded* (or simply *polynomial*) if its complexity is bounded by a polynomial of the size of a problem instance. Examples of such complexities are $O(n)$, $O(nlogn)$, $O(n^{100})$, etc. The remaining algorithms are usually referred as *exponential* or *non-polynomial*. Examples of such complexities are $O(2^n)$, $O(n!)$, etc. When the running time of an algorithm is bounded by $O(n)$, the algorithm is called a *linear-time* algorithm or simply a *linear* algorithm.

Complexity of (k, w)-Core Clustering

Our technique comprises of three steps: k-core generation, (k, w)-core generation, and cluster hierarchy generation (refer to Fig. 3.5). The time complexities of each of these steps are given below.

Complexity of k-Core Generation

The algorithm "Generating all k-cores of weighted graph" (Algorithm 3) generates all the k-cores of a weighted graph. As was seen, it analyzes the degrees of each vertex of the graph with the aid of the recursive update function in Algorithm 4. For a particular value of k, if the degree of any vertex is found to be less than k, it is deleted and its neighbouring vertices' degrees are updated. Consequently, the degrees of the neighbouring vertices are also checked, and the same steps are carried out on them recursively. The worst case scenario for generating a k-core is signified by the situation when the first vertex that qualifies as a candidate for deletion is the last vertex that has been scanned in the graph, $G(V, E)$. In this situation the algorithm would have performed at most $n+m$ traversals on the graph, where $n = |V|$ and $m = |E|$. Since a vertex deletion entails updating its neighbours and checking their degrees recursively, in the worst case the algorithm would again scan the entire graph, performing another $n + m$ traversals. Thus, for each vertex deletion the algorithm would perform at most $2 \times (n + m)$ traversals. Now, for generating all the k-cores of a graph (i.e., k-cores for all values of k belonging to the set, D, of distinct degrees of the vertices in G), the algorithm would delete at most n vertices. Henceforth, the time complexity of the algorithm is $O(n \times 2 \times (n + m)) \Rightarrow O(n \times (n + m))$.

Complexity of (k, w)-Core Generation

The algorithm "Generating all (k, w)-cores of $G(V, E)$ for a certain value of k" (Algorithm 5) comprises of two parts. In the first part, it generates a set of intermediary graphs from a k-core by deleting edges in the k-core with weights less than w, for each w in the set, W, of distinct weights of the edges of G. To generate any intermediary graph from a k-core, the algorithm scans m edges, if the k-core has m edges. Thus, to generate all the intermediary graphs from a k-core the algorithm performs $|W| \times m$ steps. Since intermediary graphs are generated from all k-cores and that there can be a maximum of $|D|$ k-cores, on assuming that each k-core has m edges we obtain the complexity of generating

all the intermediary graphs from G as $O(|D| \times |W| \times m)$.

The next part of Algorithm 5 is to generate (k, w)-cores from the intermediary graphs. In order to do so, the algorithm scans each intermediary graph and deploys the same degree checking steps of the k-core generation algorithm, which was found to be $O(n \times (n + m))$. Since there can be a maximum of $|D| \times |W|$ intermediary graphs, we'll need to perform a maximum of $|D| \times |W| \times n \times (n+m)$ steps, where each intermediary graph is assumed to contain n vertices and m edges.

Thus, the overall complexity of (k, w)-core generation is,
$O(|D| \times |W| \times m) + O(|D| \times |W| \times n \times (n + m))$
$= O(|D| \times |W| \times n \times (n + m))$.

Complexity of Cluster Generation

In this phase of the clustering technique, first the relatedness values for the cores are computed. Given that we already have the $degree_{max}$ and the $weight_{max}$ values of a graph (refer to formula for core relatedness, Subsection 3.2.3), this step takes time $O(|D| \times |W|)$ (as there can be a maximum of $|D| \times |W|$ cores). Then the cores are sorted based on their relatedness values. This could be done using an efficient sorting algorithm, e.g., mergesort, in which case it would take $O(|D| \times |W| \times \log(|D| \times |W|))$ time [Knu98]. After sorting the cores, each core is checked for connectivity. If any are found to be disconnected they are split into connected components. The connectivity check for any graph $G(V, E)$ and the generation of its connected components could be done using depth-first search traversal in $O(n + m)$ time [THC09]. Therefore, performing the connectivity check for all the cores would take $O(|D| \times |W| \times (n + m))$ time.

The cores are then selected from the sequence using the algorithm "Generating clusters from cores of $G(V, E)$" (Algorithm 6). Here the vertices of each core are checked to see whether they have been chosen in the cluster hierarchy. In the worst case this would take $O(n)$ time, if a core contains n vertices. Thus, when deployed on all cores, the algorithm will take $O(|D| \times |W| \times n)$. (The same time will be taken when Algorithm 6 performs the steps without the condition in Step 4.)

On adding the complexities to obtain the complexity of the entire cluster generation process, we have,

$O(|D| \times |W|) + O(|D| \times |W| \times \log(|D| \times |W|))$

$+ O(|D| \times |W| \times (n+m)) + O(|D| \times |W| \times n)$
$= O(|D| \times |W| \times (n+m))$[1].

Overall Complexity
The overall complexity of (k, w)-CC can be obtained by adding the complexities of each of the phases of the technique. From the above discussion, we saw the k-core generation phase takes $O(n \times (n+m))$ time, the (k, w)-core generation phase takes $O(|D| \times |W| \times n \times (n+m))$ time, and the cluster generation phase takes $O(|D| \times |W| \times (n+m))$ time. On adding the complexities, we obtain $O(|D| \times |W| \times n \times (n+m))$ as the overall complexity of (k, w)-CC.

3.3 Modified Attribute Selection

In this section, we introduce a new attribute selection strategy for generating the entity-attribute matrix, which is a modified extension of the one discussed in Subsection 2.3.1. A class of attributes, called omnipresent attributes, chosen by the previous attribute selection strategy was found to distort results of (k, w)-CC. These attributes are described in Subsection 3.3.1. In Subsection 3.3.2, we explain a way by which we could extract these attributes by identifying certain properties of these attributes.

3.3.1 Omnipresent Attributes

We observed that (k, w)-CC returns distorted clustering results if it directly uses the attribute selection criteria of Lung *et al.*, which was discussed in Subsection 2.3.1. We found that attributes which tend to have a wide presence among the entities of a function adversely affect (k, w)-CC's restructuring results and consequently, we found that discarding such attributes improves restructuring results. These attributes are termed as *omnipresent attributes*. Because of their wide presence among the entities of a function, they give no information on suggesting how to separate, i.e., group the entities. Instead, in many situations, they may misleadingly convey a similarity between entities which are actually widely contrasting in functionality. Typical examples of omnipresent attributes

[1]The complexity $O(|D| \times |W| \times \log(|D| \times |W|))$ was ignored because both $|D|$ and $|W|$ have been found to be small compared to m for software graphs.

are system variables and system functions. It has been seen that in most industrial functions different segments of code use such attributes to access system resources. Despite using the same system resources, it has been seen that the segments are for widely contrasting purposes. Thus, considering system-related attributes would be of no help in separating these segments.

The effect of omnipresent attributes on (k, w)-CC's performance is particularly significant because (k, w)-CC considers the structural relationship in addition to the inter-similarity relationship between entities. As was mentioned earlier, the structural relationship of entities is directly dependent on their interconnectivity. With omnipresent attributes a large number of entities are shown to be highly interconnected. As a consequence, this leads to the generation of very large clusters having high relatedness values, which ultimately provides no restructuring advice.

Among the first works that considered the negative impact of omnipresent software modules on software clustering, and that stressed upon removing them is that of Muller *et al.* [MOTU93]. Following this work, different strategies for the detection of omnipresent modules have been proposed. Most strategies rely on setting a predetermined connectivity threshold and then assigning all modules that exceed the threshold as omnipresent (e.g., [MM06]). In [WT05], the strategy is slightly different, in which modules that are connected to a large number of subsystems, i.e., clusters, instead of entities, are assigned as omnipresent.

For our purpose, we give a new approach for detecting omnipresent attributes based on a novel categorization of attributes. We specify that, in a function an attribute can be *dependent* or *independent*. In simplistic terms, dependent attributes directly use the value of another attribute, whereas, independent attributes do not directly use the value of any attribute (formal definitions of these two types of attributes are given in Subsection 3.3.2). We observed that omnipresent attributes mostly tend to be independent. Thus, we discard independent attributes in (k, w)-CC. We refer to the attribute selection mode based on these criteria as *Selective Attribute Selection Mode*. In the following subsection, we elaborate on the notion of dependent and independent attributes and explain how attribute dependency is determined.

3.3.2 Identifying Dependent and Independent Attributes

Subject to certain conditions, an attribute is dependent if it directly uses the value of another attribute in an executable statement (entity) of a function. It trivially follows that an attribute can directly use the value of another attribute only in a non-control entity[2] through the assignment operator, `=`. Thus, when determining attribute dependency we only consider non-control entities that use `=` as an assignment operator. Control entities (condition statements, loop statements) are not considered in this regard. To illustrate the notion of attribute dependency in such an entity, consider the simple statement `a = b ;`. Here `a` directly uses `b` and hence is a dependent attribute. Note that only attributes denoting variables are considered for the purpose of attribute dependency detection. Attributes denoting function names are ignored, and are designated as independent attributes by default. Furthermore, in the example `a = b ;`, if `b` denotes a function name, then `a` does not directly use the value of `b`, and thus `a` cannot be designated as a dependent attribute solely on the basis of this statement.

In order to formally define attribute dependency of attributes in a non-control entity, we need to give a simplified form for the various types of non-control entities we may encounter. Since we are concerned with detecting dependency, only those non-control entities of the function which contain `=` as an assignment operator are considered. In order to present a suitable form of such entities, we consider a *refined version* of the entity in which all features (except the `=` assignment operator) that do not qualify as an attribute by Lung *et al.'s* criteria are excluded (e.g., class names, constants, keywords, operators, loop variables, etc.).

Let S be the set of all non-control entities of a function which use `=` as an assignment operator. Let A be the set of all attributes extracted from the function as per Lung *et al.'s* criteria. Then any entity $s \epsilon S$, in its refined form, can be represented as $L = R$, where $\{L, R\} \subseteq A$. The preliminary requirement for any attribute $a \epsilon A$ to be a dependent attribute in a function is that it must belong to L for at least one entity $s \epsilon S$. Before giving a complete definition of

[2]A non-control entity refers to assignment statements, function call statements (see Subsection 2.3.1}, Chapter 2)

a dependent attribute, it is necessary to understand the ramifications of the various of non-control entities we may encounter.

We have seen most of these entities have two basic characteristics. The characteristics, however, are not mutually exclusive as an entity can carry more than one of these traits. We now explore each of these two characteristics, and show how they affect the process of finding attribute dependencies in entities.

1. Variables assigned to variables. This trait resembles an entity where the value of a variable is assigned to another variable. An example of such an entity was discussed in the introductory paragraph of this subsection. Here's another example,

```
a[i] = b * (c + d);
```

Refining the above by removing all non-attributes, except the `=` operator, we have,

```
a i = b c d
```

On segregating the attributes, we have `a`ϵL and {`b`,`c`,`d`}ϵR. As can be seen, `a` uses the values of variables `b`, `c`, `d`, and thus depends on those variables. Note that variable `i` is not considered in the dependency assessment because it is an array index variable which only has referential use.

2. Functions assigned to variables. In these entities the value returned by a function is assigned to a variable. For example,

```
int a = fn1(b,c) ;
```

Refining the above by removing all non-attributes, except the `=` operator, we have,

```
a = fn1 b c
```

On segregating the attributes, we have `a`ϵL and {`fn1`,`b`,`c`}ϵR. As can be seen, `a`, via function `fn1`, uses the values of variables `b` and `c`, and thus depends on those two variables. The relationship of `a` with `fn1` is ignored for the purpose of determining attribute dependency because `fn1` is a function name and not a variable.

Here's an example from a real-life program that has both of these characteristics,

```
title = title + " - " + application.getName();
```

Here we see a variable, `title`, being assigned to a concatenation of itself, a

string, `" - "`, and the result obtained by calling a function `getName`. `getName` is accessed by a class object, `application`. By removing all non-attributes from the entity, except the `=` operator, we get,

```
title = title application getName
```

By segregating the attributes, we have `title`ϵL and {`title`, `application`, `getName`}ϵR. The relationship between the `title` attribute in L and the `title` attribute in R is ignored as both attributes refer to the same attribute and therefore their relationship does not give any meaningful information on `title`'s state of dependency. Also, `title`'s relationship with `getName` is ignored as the latter is a function name. `title` only has a dependency on `application`.

Based on the above discussion, we define the notion of attribute dependency as follows. Let x, y be attributes belonging to A, such that $x \neq y$. Then, x depends on y, or $x \rightarrow y$, iff x and y are found to belong to L and R, respectively, of any non-control entity $s \epsilon S$, provided that neither of x and y are function names, or array index variables.

Definition 3.2.1. ***Dependent attribute.*** *In any function, an attribute x, where $x \epsilon A$, is a dependent attribute if the dependency $x \rightarrow y$ can be obtained from any non-control entity $s \epsilon S$, for any attribute $y \epsilon A$, such that $x \neq y$ and neither of x and y are function names, or array index variables.*

Definition 3.2.2. ***Independent attribute.*** *In any function, an attribute x, where $x \epsilon A$, is an independent attribute if the dependency $x \rightarrow y$ cannot be obtained from any non-control $s \epsilon S$, for any attribute $y \epsilon A$, such that $x \neq y$ and neither of x and y are function names, or array index variables.*

Based on the above definitions, we identify and discard independent attributes in the entity-attribute matrix generation phase when using the (k, w)-CC clustering technique. As elicited earlier, because independent attributes generally tend to be omnipresent, discarding them improves the results obtained by (k, w)-CC. Therefore, only dependent attributes are used.

Chapter 4

Characterization of Cut-Point Clusters

In this chapter, we elaborate on the various aspects of clusters obtained from cut-points in dendrograms of functions. An understanding of these aspects is necessary to interpret the implications of the experimental results presented in Chapter 5. In this chapter, we first revisit the significance of a cluster in software restructuring (Section 4.1). Then, in Section 4.2, we characterize clusters based on certain common patterns which we have observed.

4.1 Significance of Clusters in Restructuring

In this section, we emphasize on the basic idea and practical implications of clusters. In Subsection 4.1.1, we describe what clusters resemble and what role they play in function-level software restructuring. In Subsection 4.1.2, we discuss the practical issues that are associated with clusters during cluster analysis.

4.1.1 Basic Ideas on Software Clusters

As was mentioned earlier, any cut-point in a dendrogram of a function returns a partition of clusters. A cluster consists of a set of entities which correspond to the statements[1] of the function. Consequently, a cluster shows which statements

[1]In this discussion, we use the terms *statements* and *entities* interchangeably.

of the function should be grouped together, and thus extracted to form a new function.

Any cluster obtained from the cut-point of a dendrogram is of the form, $(e_1, e_2, \ldots, e_n)$, where e_i, $i\epsilon\{1, n\}$, represents an entity in the cluster corresponding to a statement in the function (n denoting the number of entities in the cluster).

4.1.2 Practical Implications

Clusters chosen for analysis

In the restructuring process, not all clusters are analyzed by the user of the clustering tree. The only clusters that call for inspection by the user are clusters for which $n > 1$. These clusters are called non-singleton clusters. A dendrogram may also produce singleton clusters, i.e., clusters with $n = 1$. Singleton clusters do not give any grouping information of the entities and thus are straight away discarded on detection. (This was discussed earlier in Subsection 2.3.3 of Chapter 2.)

Interpreting Clusters

As was discussed earlier, cohesion-based software clustering techniques give more importance to data dependency relationships than to control dependency relationships because only the former portray the most significant tasks of any function. This notion of differentiation is carried out even to the cluster analysis stage, where the clusters are interpreted to obtain meaningful restructurings. Any cluster, $(e_1, e_2, \ldots, e_n)$, may contain both control and non-control entities. Since only non-control entities indicate significant operations of a function by using data attributes, the main objective in cluster analysis is to find out how to separate the non-control entities of a function.

Consequently, any cluster that correctly groups a set of non-control entities of a function is a useful cluster, even if it excludes some directly related control entities. Any existing control dependency relationships of the non-control entities of such a cluster with control entities may be easily perceived by the user while viewing the function code. Therefore, when building a new function based on such a cluster, the user can easily identify and include any associated control

entities as necessary. We refer to such clusters as having *implicit control dependencies*, in which related control entities are not included. Such an example was already seen for the sum_prod function in Subsection 2.3.3 of Chapter 2. (A cluster that correctly groups a set of non-control entities is a redundant cluster only if it also consists of a control entity which is not related to any of the other entities in the cluster. This scenario is discussed in Section 4.2.)

Time Spent in Cluster Analysis

A significant aspect of cluster analysis is the time spent during the analysis. We note that inspecting a non-singleton cluster in order to determine whether or not a meaningful restructuring can be obtained from it consumes time. From this it follows that time spent on a non-singleton cluster from which no meaningful restructuring could be deduced is wasted. If the cut-points of a dendrogram generate a large number of meaningless or bad clusters, the likelihood of more time being wasted during the analysis phase increases. Although the actual time spent in analyzing such clusters varies from one person to another (due to differences in programming expertise), comparing the number of bad clusters returned by different clustering techniques would be a justifiable way to compare the efficiency with which proper restructuring results could be obtained from the dendrograms generated by those techniques. We performed such a comparison in this work, the results of which are presented in Chapter 5.

4.2 Types of Clusters

In this section, we discuss some common patterns of clusters. We have heuristically established that by following these patterns it is possible to identify meaningful and redundant clusters. We classify the patterns into two categories, "Definitive Patterns", which is discussed in Subsection 4.2.1, and "Generic Patterns", which is discussed in Subsection 4.2.2. In the rest of this discussion we refer to redundant clusters, or clusters that do not lead to a meaningful restructuring, as *bad clusters*.

4.2.1 Definitive Patterns

We define definitive patterns of clusters as those combinations of entities of a function which, when extracted to form a new function, never lead to a meaningful restructuring. We have identified three such patterns; they're given as under,

Clusters with only Control Entities

Clusters that consist of only control entities don't give any meaningful information on clustering the function. As was stated in Subsection 4.1.2, the main objective in cluster analysis is to identify data dependency relationships between non-control entities. Clusters with only control entities don't give any such information, and hence are regarded as bad clusters.

Clusters Splitting Conditional Constructs

Conditional constructs such as `if-else`, `if-elseif-...-elseif`, and `try-catch` (in Java), are widely used in software functions. Conditional constructs consist of multiple control blocks that are logically linked, and which are executed in a specific sequence. Let us see an example with a sample code segment [swe12],

```
0 if (homeName == null) {
1 homeDisplayedName = application.getUserPreferences().
                      getLocalizedString(HomeFramePane.class,"untitled");
2     if (newHomeNumber > 1) {
3     homeDisplayedName += " " + newHomeNumber;
4     }
5 }
6 else {
7 homeDisplayedName = this.contentManager.getPresentationName(homeName,
                      ContentManager.ContentType.SWEET_HOME_3D);
8 }
9 if (home.isRecovered()) {
10 homeDisplayedName += " " + application.getUserPreferences().
                 getLocalizedString(HomeFramePane.class,"recovered");
11 }
```

In the above code segment, there is an `if-else` construct from lines 0-8, defined by the control entities in statements 0 and 6. It can be clearly seen that

any restructured version of this code segment must ensure that the `else` block (lines 6-8) is executed *after* the `if` block (lines 0-5).

It has been observed that many clusters ignore the logical sequence of control blocks in such conditional constructs. Such clusters suggest results that incorrectly split the control block which defy the original flow of the program. These clusters, therefore, have conditional construct splitting patterns and are characterized as bad clusters.

For the sample segment above the cluster (1,2,3,9,10) (corresponding statements highlighted in code) is a bad cluster since using it to restructure the code would imply executing statements 9-10 immediately after the `if`-block (lines 0-5), defying the original logic of the program which was to perform the `else` condition check immediately after executing the particular `if`-block.

Clusters with Unrelated Control Entities

Clusters with these patterns group a set of unrelated control entities. Such clusters may be returned by clustering algorithms due to the 1-1 similarity matches (see Subsection 2.3.2 in Chapter 2) between the unrelated control entities that use the same data attributes. Here's an arbitrary example,

```
0 double[] input_arr = {1.4, 3.6, 0.9, 17.3, 8.2, 10.0};
1 int prod = 0;
2 int sum = 0;
3 for (double d : input_arr) {
4 prod *= d;
5 }
6 for (double d : input_arr) {
7 sum *= d;
8      if (sum>22.5) {
9      System.out.println(``Threshold Exceeded'') ; }
10      }
11}
```

For the example above the cluster (3,6,7,8,9) (highlighted in code) would be a bad cluster since it groups control entity 3 with entities with which it has no relation. Although control entities 3 and 6 use the same data attribute `input_arr` (that contributes to a 1-1 match between the two entities), the control entities have no control dependency relationship. It is therefore not meaningful to have

them in the same cluster.

4.2.2 Generic Patterns

Unlike definitive patterns, these patterns are more generic in nature as their interpretation depends on the developer's requirements and understanding of the relationship between the entities in a cluster with such patterns. The patterns are given as under,

Clusters without Related Entities

We have observed that there may be many clusters returned by dendrograms that omit entities closely related to the ones in the cluster. Although the restructurings obtained from such clusters may be programmatically valid, in many situations they may be deemed to be conceptually incoherent with respect to the requirements of the programmer, in which cases the clusters would be classified as bad clusters. Here's an example [KB99],

```
0 void fn1(int n, int[] arr) {
1 sum = 0;
2 prod = 0;
3 for ( int i = 1 ; i < n ; i ++ )
4 {
5 sum = sum + arr [ i ] ;
6 prod = prod * arr [ i ] ;
7 }
8 avg = sum / n ;
9 }
```

For the above code segment, the cluster (1,5) (highlighted in code) is a bad cluster. If this cluster is used to restructure the above code, we will have a new function with the entities 1 and 5, leaving out entity 8. It can be seen that all three entities carry out tasks related to the `sum` attribute. In the context of this program, such a new function, despite being programmatically correct, would be undesired as leaving out entity 8 from the new function would be conceptually inappropriate: doing so would imply keeping entity 8 with unrelated entities (entities 2 and 6) or dedicating a separate function for entity 8 only.

It is important to note that clusters which leave out related entities may not

always be perceived as bad. Consider the following example,

```
0 void fn2(int n, int[] arr) {
1 sum = 0;
2 prod = 0;
3 for ( int i = 1 ; i < n ; i ++ )
4 {
5 sum = sum + arr [ i ] ;
6 prod = prod * arr [ i ] ;
7 }
8 avg = sum / n ;
9 if (avg>50) {
10 System.out.println(‘‘Average exceeded 50 and is equal to "+ avg) ;
11 }
12 }
```

For the above code segment, the cluster (1,5) (highlighted in code) would not be a bad cluster. This is because despite the existing relationship between entities 1, 5, and 8, entity 8 may still be kept separate from the former two entities as it is also closely related to other entities in the function (entities 8-11). Therefore, using this cluster for restructuring would not necessarily lead to a conceptually incoherent clustering.

Therefore, clusters that omit related entities, if detected, may or may not lead to a meaningful restructuring for it entirely depends on the interpretation and requirements of the programmer.

Clusters that do not retain Execution Sequence

We have observed many instances of programs which contain certain entities that must be executed in a predefined sequence. For example, let us say that a function consists of four entities, e_1, e_2, e_3, e_4, that are supposed to execute in the order in which they're mentioned. For such a function, a cluster (e_1, e_3) would be a bad cluster since by using this cluster to restructure the function, it would not be possible to ensure that entity e_2 is executed before and after entities e_1 and e_3, respectively. Because this cluster omits e_2, using this cluster would violate the original execution sequence of the entities. Henceforth, any cluster that non-sequentially separates entities that are supposed to execute in a predefined order are regarded as bad clusters. This pattern in clusters is

similar to the "Splitting Conditional Constructs" pattern and could be viewed as a general form of the latter. Here is an example [swe12],

```
0 String homeName = home.getName();
1 if (homeName == null) {
2 JFrame homeFrame = getHomeFrame(home);
3 homeFrame.toFront();
4 homeName = contentManager.showSaveDialog((View)homeFrame.getRootPane(),
                    null, ContentManager.ContentType.SWEET_HOME_3D, null);
5 }
6 if (homeName != null) {
7      try {
8      getHomeRecorder().writeHome(home, homeName);
9      }
10      catch (RecorderException ex) {
11      ex.printStackTrace();
12      }
13 }
```

In the above code, line 0 declares and obtains the value of the `homeName` variable. In lines 1-5, `homeName` is modified. In lines 6-13, an operation is carried out based on the value of `homeName`. Since both line sets 1-5 and 6-13 use `homeName`, in any restructuring of the above code, line sets 1-5 and 6-13 must follow line 0, where `homeName` is declared. Also, line set 6-13 must follow line set 1-5 because the former carries out operations based on the value of `homeName` *after* it is modified in line set 1-5. Therefore, the logical flow of the program strictly follows a particular sequence. In that case, obtaining a function using the cluster that groups only the highlighted entities shown above would break this sequence for that would imply either executing the new function before line set 1-5, in which case line set 6-13 would precede line set 1-5, or executing the new function after line set 1-5, in which case line set 1-5 would precede line 0. Thus, the indicated cluster is a bad cluster.

Clusters with Extreme Sizes

Clusters with extreme sizes include those that contain a large number of entities and those which have very few entities. Although these clusters may be programmatically correct, they may still be rejected by the programmer and thus, be classified as bad clusters.

Clusters that are very large generally don't give any meaningful restructuring of the function. This is because in most cases such clusters do not lead to any significant separation of the various tasks of the function, and thereby do not fulfil the main objective of cohesion-based software restructuring. On the other hand, functions created from clusters that contain very few entities may be regarded as too small to meaningfully exist in a restructured version.

The characterization of clusters with extreme sizes is thus entirely dependent on the properties of the encompassing function and the requirements of the programmer.

Chapter 5

Experimental Results

In this chapter, we present and discuss the results that were obtained while performing our experiments in which we restructured Java functions using the HACs discussed in this thesis. In Section 5.1, we explain the design of our entire experiment. In Section 5.2, we present various experimental results in tabulated and graphical form. Finally, in Section 5.3, we provide a discussion on the basis of the results obtained.

5.1 Experimental Design

In this section, we describe the design of our experiment. In Subsection 5.1.1, we give an overview of the methodology of our experiment. In Subsection 5.1.2, we describe all the parameters that were measured while performing our experiment.

5.1.1 Overall Methodology

In our experiment, Java functions were restructured using the five different clustering techniques: SLINK, CLINK, WPGMA, A-KNN, and the new clustering technique developed in this work, (k, w)-CC. The clustering techniques were implemented to generate dendrograms of the functions. The dendrograms were then analyzed to obtain the restructured versions. In the process, parameters pertaining to the dendrograms of each clustering technique were measured on the basis of which we compared the techniques with each other. The time taken by each technique to generate the dendrograms was also measured. In addi-

tion to that, the qualities of the final restructured versions that were eventually obtained from each technique were compared.

As was mentioned earlier, for (k, w)-CC we have used the Selective Attribute Selection Mode for attribute selection due to reasons discussed in Section 3.3 of Chapter 3. For maintaining consistency there was a need to show that (k, w)-CC did not gain an advantage over the other techniques solely because of its modified attribute selection strategy. For this reason, we have implemented the remaining techniques using both their prescribed attribute selection mode, the Normal (N) Attribute Selection Mode, and the Selective (S) Attribute Selection Mode. Therefore, effectively, we have implemented nine different restructuring techniques: (k, w)-CC with S-Attribute Selection Mode only and SLINK, CLINK, WPGMA, A-KNN with both N- and S-Attribute Selection Mode.

Since data dependency relationships are given more emphasis than control dependency relationships during software restructuring, we used the weight ratio of 8:3 for the weights of data attributes to control attributes for the computation of the similarity matrix. (See "Similarity Matrix Generation" in Subsection 2.3.2 of Chapter 2.)

5.1.2 Parameters Measured

Execution Time

We measured the time taken, in milliseconds (ms), by each clustering technique to generate dendrograms for each of the functions that were analyzed in this thesis. The execution time was measured only for the Applying Clustering Algorithms phase (refer to Figure 2.5 in Chapter 2), i.e., from the point after obtaining the similarity matrix in the Similarity Matrix Generation Phase to the point when the final dendrogram is generated.

All executions were carried out in a system with a 2.4 GHz processor and a 4096 Mb RAM. Measuring the execution times of the techniques gave us a good estimate on the relative efficiency of the techniques.

Number of Cut-Points and Bad Clusters

The number of cut-points (N_{cp}) and the number of bad clusters (N_{bc}) are significant indicators of the quality of the dendrograms, particularly in reflecting

the ease with which proper restructuring results could be extracted from the dendrograms. The criteria followed for determining bad clusters were discussed in detail in Chapter 4.

Cohesion Improvement

We measured the maximum percentage increase in cohesion that was possible to achieve with each clustering technique while restructuring a function. For measuring cohesion we used the cohesion metric used by Lung *et al.*. Their metric follows Anquetil *et al.*'s [AL03] quantitative definition of cohesion, which ascribes the cohesion of a function to the average similarity between any two entities of the function. Cohesion ranges from 0 (worst) to 1 (best). The formula for cohesion, C, of a function, F, is given as under,

$$C_F = \frac{\sum_{i=1}^{m}\sum_{j=1}^{m} coeff_{(i,j)}}{m^2}, \text{ such that } i \neq j,$$

where m is the number of entities (executable statements) of the function and $coeff_{(i,j)}$ is the similarity value for entity-pair (i, j). (Mentioned in Subsection 2.3.2 in Chapter 2.)

Since restructuring each function resulted in the creation of more functions, the cohesion of the final restructured version was evaluated as the average cohesion of all the functions in the system. Therefore, the cohesion of a restructured version R that consists of n functions is given as,

$$C_R = \frac{\sum_{i=1}^{n} C_{F_i}}{n}.$$

5.1.3 Functions Analyzed

In our experiment we examined low-cohesive functions extracted from published papers and an industrial Java application, SweetHome3D [swe12]. Sweet Home 3D is a popular cross-platform interior design application for drawing two-dimensional plans of houses. The application also provides three-dimensional renderings of the interior designs of houses. We selected five large, low cohesive, functions from the application for analysis. A full list of the names of the

functions analyzed in this thesis along with their respective cohesion measures (C) and lines of code (LOC) is given in Table 5.1,

List of Functions Restructured			
Function Id.	Function Name	LOC	C
1	sum_max_avg [LXZS06]	11	0.191
2	sum_and_prod [LXZS06]	9	0.139
3	sale_pay_profit [AAM10]	19	0.1524
4	sum1_or_sum2 [BK98]	14	0.073
5	prod1_and_prod2 [BK98]	8	0.121
6	fiboAvg [BK98]	10	0.28
7	deleteLevel [swe12]	41	0.13
8	displayView [swe12]	22	0.053
9	exitAfter3DError [swe12]	37	0.1022
10	updateFrameTitle [swe12]	37	0.0587
11	updateSunLocation [swe12]	40	0.113

Table 5.1: Functions analyzed.

Overall, by implementing the nine different clustering techniques, we analyzed a total of 99 dendrograms obtained from the 11 functions. (The Id. numbers shown in Table 5.1 are used to refer to the functions in the tables and diagrams given in the rest of this chapter.)

5.2 Results and Analysis

In this section, we present the various results obtained in our experiment along with our respective analyses. In Subsection 5.2.1, we provide the execution times of the different techniques implemented. In Subsection 5.2.2, we tabulate the number of cut-points and number of bad clusters generated by each technique for each function that we analyzed. In Subsection 5.2.3, we provide results pertaining to the cohesion improvements that was attained by restructuring the functions using suggestions provided by each of the techniques. Finally, in Subsection 5.3, we provide the detailed restructuring result for one of the functions we analyzed.

5.2.1 Execution Time

We have seen that a graph can be easily obtained from the similarity matrix of a function. As such, the similarity matrix serves as an adjacency matrix

for the weighted graph. Since all clustering techniques work on the similarity matrix, we investigated the relationship of the execution time of the techniques with two different properties of the graph represented by the similarity matrix: the number of vertices (entities), the number of edges (non-zero inter-entity relationships).

Now, as we already mentioned, two attribute selection modes were used in the Similarity Matrix Generation phase. Since the S-Attribute Selection Mode discarded some of the attributes chosen by the N-Attribute Selection Mode, the two selection modes yielded different similarity matrices for the same function. In particular, while using the S-Attribute Selection Mode,

- Entities that only contained independent attributes were absent from the resulting similarity matrix. This led to a reduction in the number of vertices in the corresponding graph of the similarity matrix.
- Entity relationships that only existed because of independent attribute matches were absent in the resulting similarity matrix. This led to a reduction in the number of edges in the corresponding graph of the similarity matrix.

We observed that the reduction in the number of edges and the number of vertices in the graphs of the functions, while using the S-Attribute Selection Mode, was negligible. (See Table 5.2.) Consequently, we observed that changing the attribute selection mode did not make any noticeable difference in the execution time of any technique. Therefore, we only recorded the execution times of all the algorithms when they were implemented using their prescribed attribute selection modes, i.e., N-Attribute Selection Mode for SLINK, CLINK, WPGMA, A-KNN and S-Attribute Selection Mode for (k, w)-CC.

Function Id.	No. of Vertices (n)		No. of Edges (m)	
	N-Attr. Sel. Mode	S-Attr. Sel. Mode	N-Attr. Sel. Mode	S-Attr. Sel. Mode
1	7	7	11	10
2	8	6	11	7
3	13	13	51	48
4	8	8	8	8
5	5	5	5	5
6	7	7	10	10
7	25	25	280	279
8	17	15	33	32
9	24	22	112	99
10	25	25	78	73
11	34	34	561	561

Table 5.2: No. of vertices and no. of edges in the graphs of each function, obtained using the two attribute selection modes.

Table 5.3 provides the execution times for each of the techniques when they were implemented on the functions. As can be seen, SLINK, CLINK, and WPGMA took the same execution times. The reason for this is due to the fact that these three techniques deploy exactly the same algorithmic mechanism except for minor differences in the way they calculate cluster similarity.

Function Id.	Execution Times (ms)		
	SLINK(N) \ CLINK (N) \ WPGMA (N)	A-KNN(N)	(k, w)-CC(S)
1	0.41	0.11	0.328
2	0.175	0.021	0.077
3	3.45	0.092	1.5
4	0.55	0.039	0.11
5	0.08	0.015	0.048
6	0.2	0.02	0.1476
7	22.02	0.44	9.68
8	4.36	0.33	1.24
9	20.8	0.95	4.49
10	21.954	0.62	3
11	34.01	3.01	22.88

Table 5.3: Execution times of the clustering techniques for each function.

Figures 5.1, 5.2 show plotted-graphs for the techniques' execution times. In Fig. 5.1, the functions are horizontally presented in increasing order of number of vertices in their corresponding graphs. In Fig. 5.2, the functions are horizontally presented in increasing order of number of edges in their corresponding graphs.

From the figures, we can see that there exists a positive correlation between the execution times of SLINK, CLINK, WPGMA and the number of vertices in the graph. Similarly, we can see that (k, w)-CC's execution time has a positive correlation with the number of edges in a graph. No such relationship was possible to establish for A-KNN.

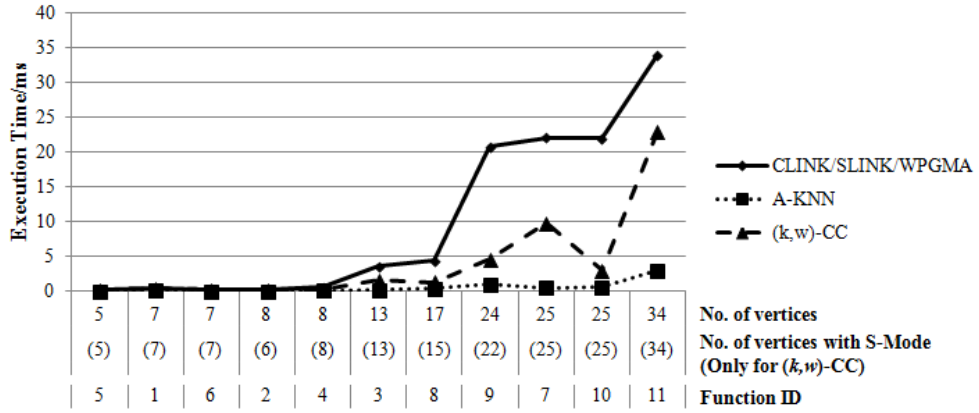


Figure 5.1: Execution times of the clustering techniques for each function, with functions arranged by no. of vertices in their corresponding graphs.

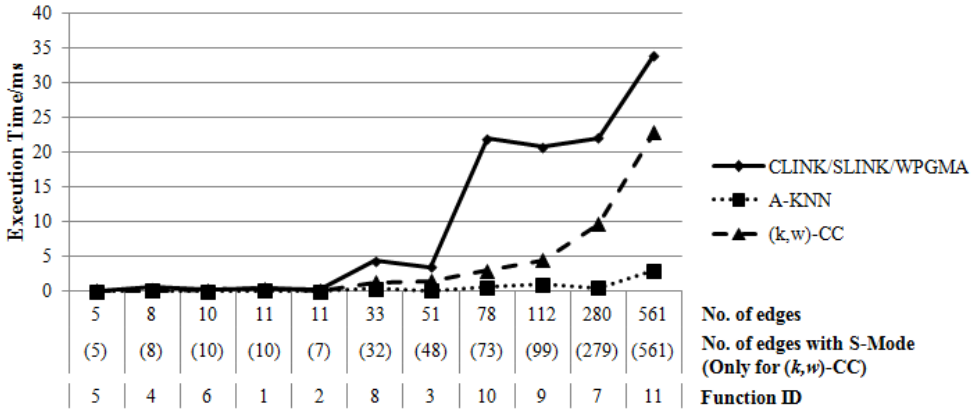


Figure 5.2: Execution times of the clustering techniques for each function, with functions arranged by no. of edges in their corresponding graphs.

Regarding the comparative speeds of the techniques, SLINK, CLINK, and WPGMA consumed the greatest execution times, while A-KNN performed the quickest. (k, w)-CC's performance was intermediary, performing slower than

A-KNN but significantly faster than the other three; on average, (k, w)-CC performed 59.72% percent faster than SLINK, CLINK, and WPGMA.

5.2.2 Number of Cut-points and Bad Clusters

The number of cut-points and bad clusters obtained by implementing SLINK, CLINK, WPGMA, A-KNN and (k, w)-CC on each function, with their prescribed attribute selection modes, are given in Table 5.4 and Table 5.5 respectively. Coloured stacked chart representations of the data in these tables are given in Figures 5.3 and 5.4.

Function Id.	No. of cut-points (N_{cp})				
	SLINK (N)	CLINK (N)	WPGMA (N)	A-KNN(N)	(k, w)-CC(S)
1	5	4	5	5	2
2	3	2	3	3	2
3	8	8	10	8	5
4	4	3	5	4	2
5	1	1	2	1	1
6	4	3	4	4	3
7	8	12	11	8	4
8	12	9	14	12	7
9	12	12	14	12	8
10	20	16	22	20	9
11	19	22	23	19	3
TOTAL	96	92	113	96	46

Table 5.4: Number of cut-points generated for (k, w)-CC with S-Attribute Selection Mode and SLINK, CLINK, WPGMA, A-KNN with N-Attribute Selection Mode.

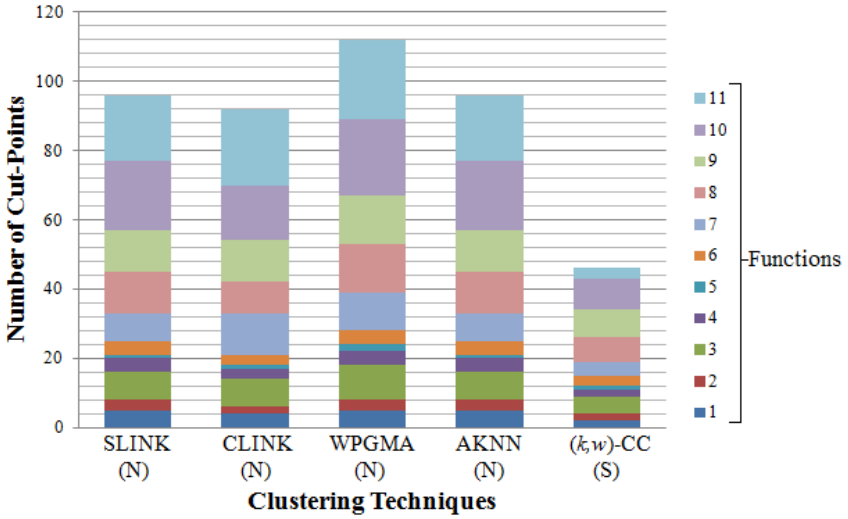


Figure 5.3: Number of cut-points generated for (k, w)-CC with S-Attribute Selection Mode and SLINK, CLINK, WPGMA, A-KNN with N-Attribute Selection Mode.

Function Id.	No. of bad clusters (N_{bc})				
	SLINK (N)	CLINK (N)	WPGMA (N)	A-KNN(N)	(k, w)-CC(S)
1	4	3	2	4	0
2	3	1	1	3	0
3	7	6	5	7	1
4	2	2	2	2	0
5	0	0	0	0	0
6	1	1	1	1	1
7	12	14	13	12	3
8	8	5	8	8	4
9	17	13	19	16	5
10	15	13	17	15	9
11	15	18	17	14	0
TOTAL	84	76	85	82	23

Table 5.5: Number of bad clusters generated for (k, w)-CC with S-Attribute Selection Mode and SLINK, CLINK, WPGMA, A-KNN with N-Attribute Selection Mode.

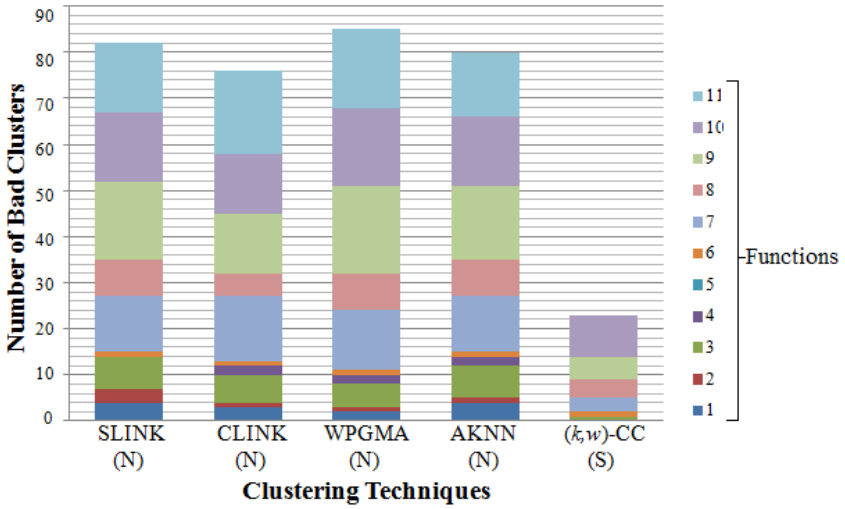


Figure 5.4: Number of bad clusters generated for (k, w)-CC with S-Attribute Selection Mode and SLINK, CLINK, WPGMA, A-KNN with N-Attribute Selection Mode.

As can be seen, overall, (k, w)-CC was found to give both a smaller number of cut-points and a smaller number of bad clusters than all the other techniques. For many of the functions, (k, w)-CC gave almost zero bad clusters. On average, (k, w)-CC gave 52.08%, 50.00%, 59.29%, 52.08% fewer number of cut-points than did SLINK,(N) CLINK(N), WPGMA(N), and A-KNN(N), respectively. In addition, (k, w)-CC gave 72.62%, 69.74%, 72.94%, 71.95% fewer number of bad clusters than did SLINK(N), CLINK(N), WPGMA(N), and A-KNN(N), respectively.

In order to ascertain that (k, w)-CC did not produce better results solely because of it's modified attribute selection strategy, we also implemented the previous HACs using the S-Attribute Selection Mode. Likewise, we have provided the results obtained in Tables 5.6, 5.7 and in Figures 5.5, 5.6.

Function Id.	No. of cut-points (N_{cp})				
	SLINK (S)	CLINK (S)	WPGMA (S)	A-KNN(S)	(k, w)-CC(S)
1	5	4	5	5	2
2	2	2	3	2	2
3	8	10	10	8	5
4	3	2	3	3	2
5	1	1	2	1	1
6	3	3	4	3	3
7	5	8	11	5	4
8	6	5	9	6	7
9	8	10	11	8	8
10	11	13	18	13	9
11	13	18	21	13	3
TOTAL	65	76	97	67	46

Table 5.6: Number of cut-points generated for all techniques with S-Attribute Selection Mode.

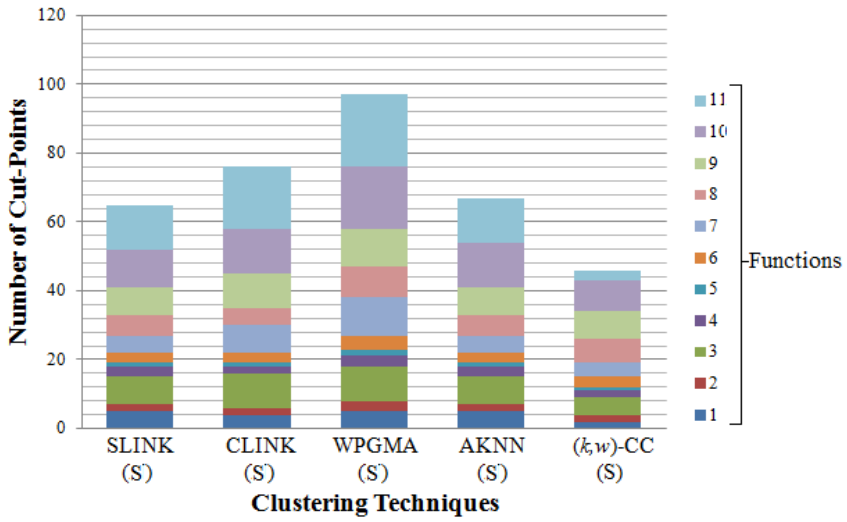


Figure 5.5: Number of cut-points generated for all techniques with S-Attribute Selection Mode.

Function Id.	No. of bad clusters (N_{bc})				
	SLINK (S)	CLINK (S)	WPGMA (S)	A-KNN(S)	(k, w)-CC(S)
1	2	3	2	2	0
2	1	1	1	1	0
3	4	5	5	4	1
4	2	2	2	2	0
5	0	0	0	0	0
6	1	1	1	1	1
7	5	7	11	4	3
8	4	3	6	4	4
9	9	12	12	9	5
10	13	14	17	13	9
11	13	14	16	10	0
TOTAL	54	62	73	50	23

Table 5.7: Number of bad clusters generated for all techniques with S-Attribute Selection Mode.

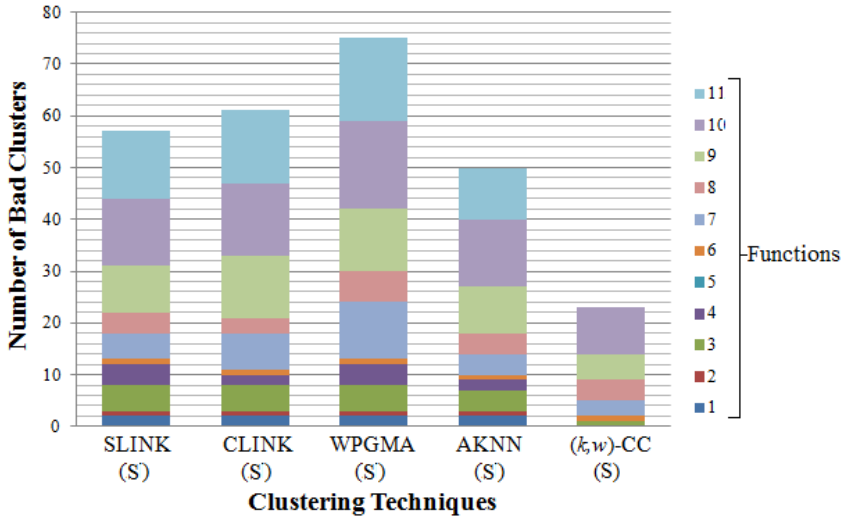


Figure 5.6: Number of bad clusters generated for all techniques with S-Attribute Selection Mode.

From the above findings, we see that although the modified attribute selection technique did improve the N_{cp} and N_{bc} measures of SLINK, CLINK, WPGMA, and A-KNN, overall their results were still considerably worse than (k, w)-CC's results. On average, (k, w)-CC gave 29.23%, 39.47%, 52.58%, 31.34% fewer number of cut-points than did SLINK(S), CLINK(S), WPGMA(S), and A-KNN(S), respectively. In addition, (k, w)-CC gave 57.41%, 62.90%, 68.49%,

54.00% fewer number of bad clusters than did SLINK(S), CLINK(S), WPGMA(S), and A-KNN(S), respectively.

5.2.3 Cohesion Improvement

After analyzing the dendrograms returned by the different clustering techniques for each of the functions, we restructured the functions based on the clusters that had been obtained from the dendrograms. Consequently, we measured the quality of the restructurings obtained by the different techniques, i.e., we measured the overall cohesion of each of the restructured versions we obtained. (For this purpose, we used the cohesion metric described in Subsection 5.1.2.) Since each clustering technique gave different restructuring versions in terms of cohesion, we recorded the maximum cohesion that was attainable through the suggestions of each technique, for every function we analyzed. Subsequently, we measured the percentage increases in cohesion.

For the purpose of this work our primary focus was on the relative quality of the results obtained through (k, w)-CC. Thus, we compared (k, w)-CC's results only with those techniques which gave the best results with respect to cohesion. Among the previous clustering techniques, we found CLINK and WPGMA to give the best results in this aspect, while using both the attribute selection modes. A comparison of (k, w)-CC with CLINK, WPGMA using N- and S-Attribute Selection Modes based on percentage improvements in cohesion for each function, are shown in Figures 5.7 and 5.8 respectively.

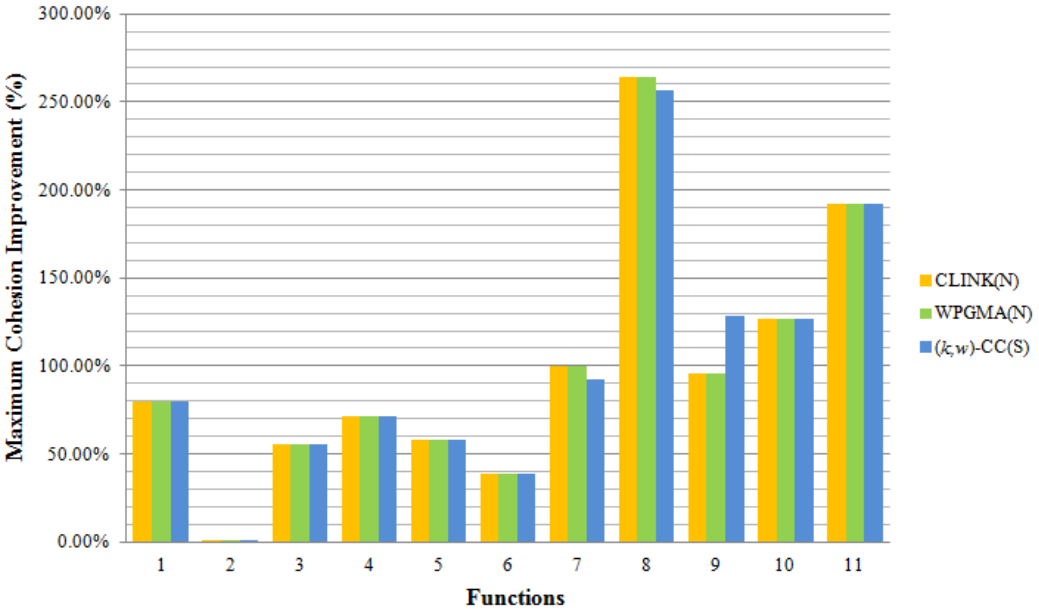


Figure 5.7: Maximum cohesion improvement through (k, w)-CC, CLINK(N), WPGMA(N).

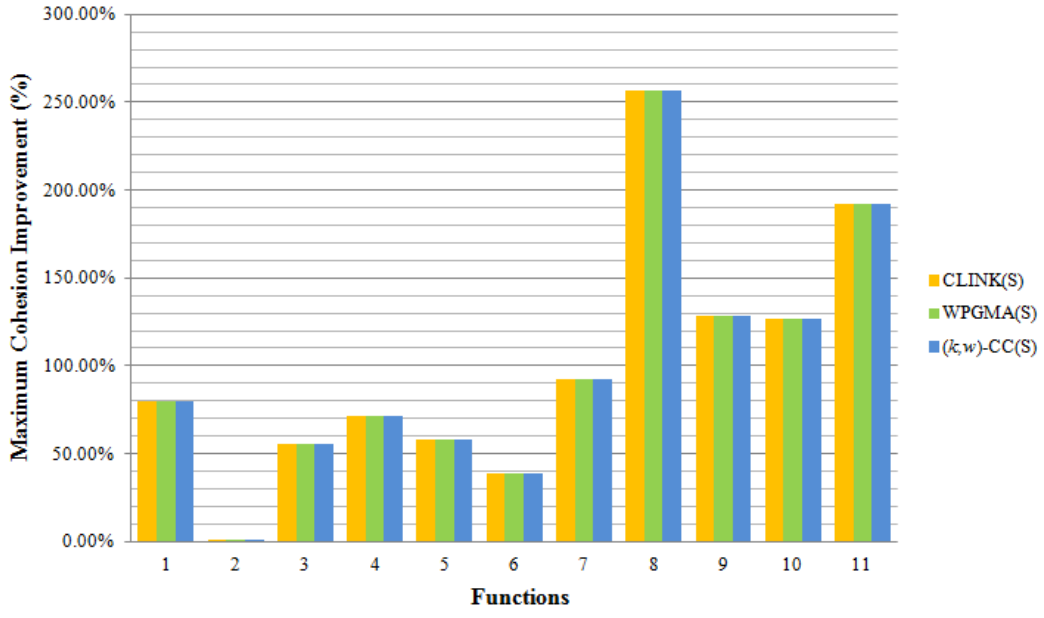


Figure 5.8: Maximum cohesion improvement through (k, w)-CC, CLINK(S), WPGMA(S).

The charts in the above figures indicate that (k, w)-CC provides competitive

results in terms of the quality of the restructurings obtained. In fact, with the majority of the functions, it was seen that the quality of the restructuring results obtained using (k, w)-CC were just as good as those obtained using the remaining techniques.

5.2.4 Sample Restructuring Result

In this subsection, we provide the full restructuring result of a large function, exitAfter3DError(), from the SweetHome3D application. The code of this function is given below,

```
0 private void exitAfter3DError() {
1 boolean modifiedHomes = false;
2 for (Home home :  getHomes()) {
3 if (home.isModified()) {
4 modifiedHomes = true;
5 break;
6 }
7 }
8 if (!modifiedHomes) {
9 show3DError();
10 }
11 else if (confirmSaveAfter3DError()) {
12 for (Home home :  getHomes()){
13 if (home.isModified()) {
14 String homeName = home.getName();
15 if (homeName == null) {
16 JFrame homeFrame = getHomeFrame(home);
17 homeFrame.toFront();
18 homeName = contentManager.showSaveDialog((View)homeFrame.
        getRootPane(),null,ContentManager.ContentType.SWEET_HOME_3D,null);
19 }
20 if (homeName != null) {
21 try {
22 getHomeRecorder().writeHome(home, homeName);
23 }
24 catch (RecorderException ex) {
25 ex.printStackTrace();
26 }
27 }
```

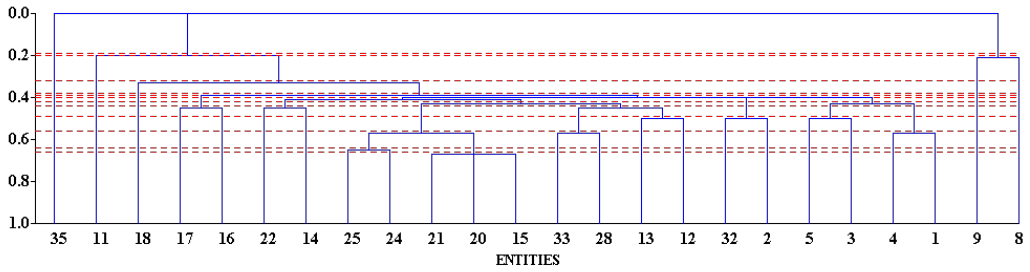
```
28 deleteHome(home);
29 }
30 }
31 }
32 for (Home home :  getHomes()) {
33 deleteHome(home);
34 }
35 System.exit(0);
36 }
```

The complete characterization of the clusters obtained for the above code along with the corresponding dendrograms for each of the techniques is presented in Tables 5.8-5.16. The number of cut-points and the number of bad clusters returned by each technique are also indicated in the tables. An explanation of the notation used in the tables is given as under,

1. $\{X, Y, Z\}$ represents a partition of clusters X, Y, Z, obtained from a cut-point, where each of X, Y, Z contain more than one entity. Thus, any singleton clusters returned by the partitions are not shown. (Such clusters are ignored in the cluster analysis stage - see Subsection 4.1.2, Chapter 4.)
2. (x, y) represents a cluster consisting of entities x and y.
3. $(x \leftrightarrow y)$ represents a cluster in the dendrogram consisting of all the entities listed on the horizontal axis of the corresponding dendrogram that are between x and y, including x and y.
4. Any cluster that is struck out, e.g., ~~(x, y)~~ or ~~$(x \leftrightarrow y)$~~, indicates that the cluster is a bad cluster.

Table 5.8: Cluster analysis of dendrogram obtained by SLINK(N) for exitAfter3DError().

SLINK (N)



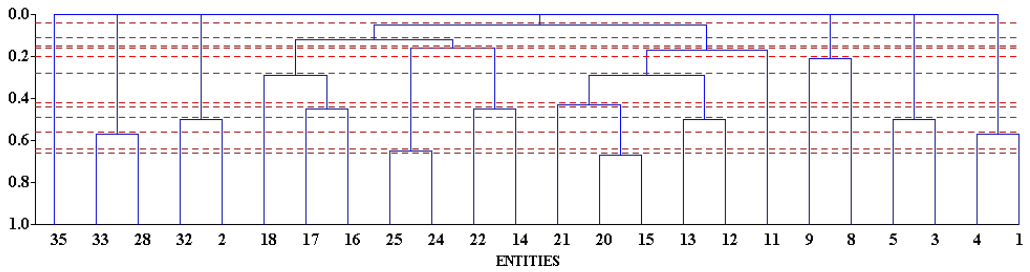
Cut-Point Partitions:

{(~~21 ↔ 15~~)}, {(~~25 ↔ 24~~)}, {(~~25 ↔ 15~~),(~~33,28~~),(4,1)}, {(~~13,12~~),(~~32,2~~),(~~5,3~~)}, {(~~17,16~~),(~~22,14~~),(~~33 ↔ 12~~)}, {(~~25 ↔ 12~~),(5 ↔ 1)}, {(~~22 ↔ 12~~)}, {(~~22 ↔ 1~~)}, {(~~17 ↔ 1~~)}, {(~~18 ↔ 1~~)}, {(~~9, 8~~)}, {(~~11 ↔ 1~~)}

$N_{cp} = 12$, $N_{bc} = 17$

Table 5.9: Cluster analysis of dendrogram obtained by CLINK(N) for exitAfter3DError().

CLINK (N)



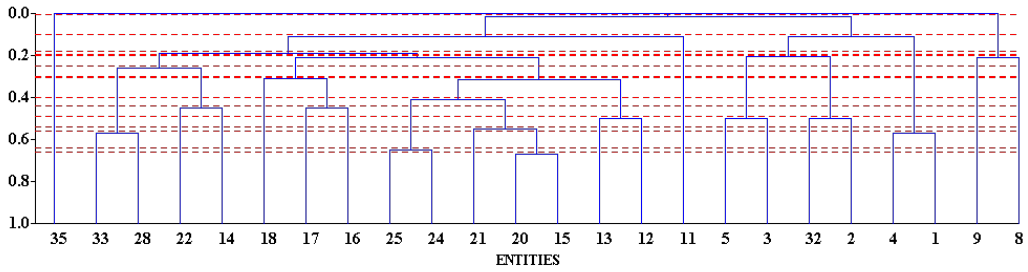
Cut-Point Partitions:

{(~~20,15~~)}, {(~~25,24~~)}, {(~~33,28~~),(4,1)}, {(~~32,2~~),(~~13,12~~),(~~5,3~~)}, {(~~17,16~~),(~~22,14~~)}, {(~~21↔15~~)}, {(~~18↔16~~),(~~21↔12~~)}, {(~~9,8~~)}, {(~~21↔11~~)}, {(~~25↔14~~)}, {(18↔14)}, {(18↔11)}

$N_{cp} = 12$, $N_{bc} = 13$

Table 5.10: Cluster analysis of dendrogram obtained by WPGMA(N) for exitAfter3DError().

WPGMA (N)



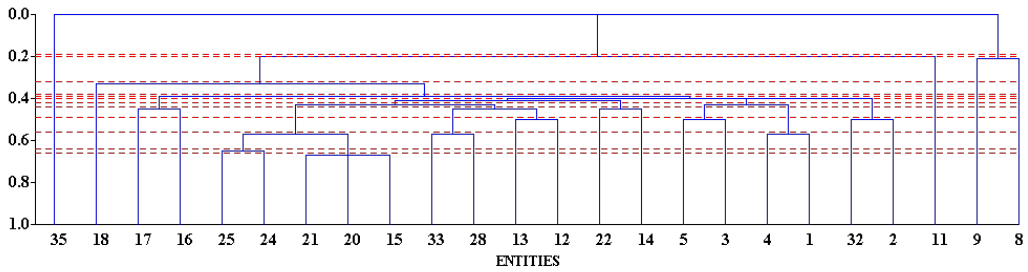
Cut-Point Partitions:

{(~~20,15~~)}, {(~~25,24~~)}, {(~~33,28~~),(4,1)}, {(~~21↔15~~)}, {(~~13,12~~),(~~5,3~~),(~~32,2~~)}, {(~~22,14~~),(~~17,16~~)}, {(~~25↔15~~)}, {(~~25↔12~~),(18↔16)}, {(~~33↔14~~)}, {(~~18↔12~~),(~~9,8~~)}, {(~~5↔2~~)}, {(~~33↔12~~)}, {(~~33↔11~~),(~~5↔1~~)}, {(~~33↔1~~)}

$N_{cp} = 14$, $N_{bc} = 19$

Table 5.11: Cluster analysis of dendrogram obtained by A-KNN(N) for exitAfter3DError().

A-KNN (N)



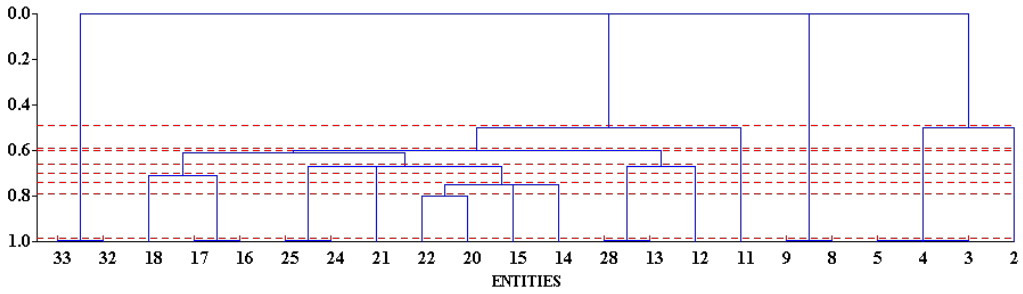
Cut-Point Partitions:

{(~~21↔15~~)}, {(~~24,24~~)}, {(~~25↔15~~),(~~32,28~~),(4,1)}, {(~~5,3~~),(~~32,2~~),(~~13,12~~)}, {(~~33↔12~~),(~~17,16~~)}, {(~~25↔12~~),(5↔1)}, {(~~25↔14~~)}, {(~~25↔2~~)}, {(~~17↔2~~)}, {(~~18↔2~~)}, {(~~9,8~~)}, {(~~18↔11~~)}

$N_{cp} = 12$, $N_{bc} = 16$

Table 5.12: Cluster analysis of dendrogram obtained by SLINK(S) for exitAfter3DError().

SLINK (S)

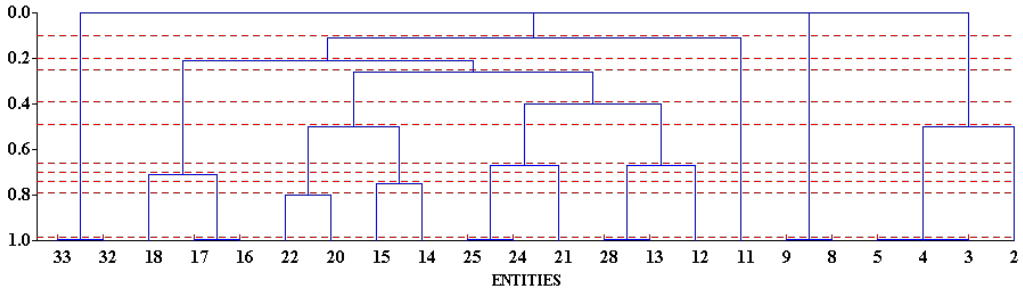


Cut-Point Partitions:
{(~~33,32~~),(~~17,16~~),(~~25,24~~),(~~28,13~~),(~~9,8~~),(5↔3)}, {(~~22,20~~)}, {(~~22↔14~~)}, {(18↔16)}, {(~~25↔14~~),(~~28↔12~~)}, {(18↔14)}, {(18↔12)}, {(18↔11),(5↔2)}

$N_{cp} = 8$, $N_{bc} = 9$

Table 5.13: Cluster analysis of dendrogram obtained by CLINK(S) for exitAfter3DError().

CLINK(S)



Cut-Point Partitions:
{(~~33,32~~),(~~17,16~~),(~~25,24~~),(~~28,13~~),(~~9,8~~),(5↔3)}, {(~~22,20~~)}, {(~~15,14~~)}, {(18↔16)}, {(~~25↔21~~),(~~28↔12~~)}, {(~~22↔14~~),(5↔2)}, {(~~25↔12~~)}, {(~~22↔12~~)}, {(18↔12)}, {(18↔11)}

$N_{cp} = 10$, $N_{bc} = 12$

Table 5.14: Cluster analysis of dendrogram obtained by WPGMA(S) for exitAfter3DError().

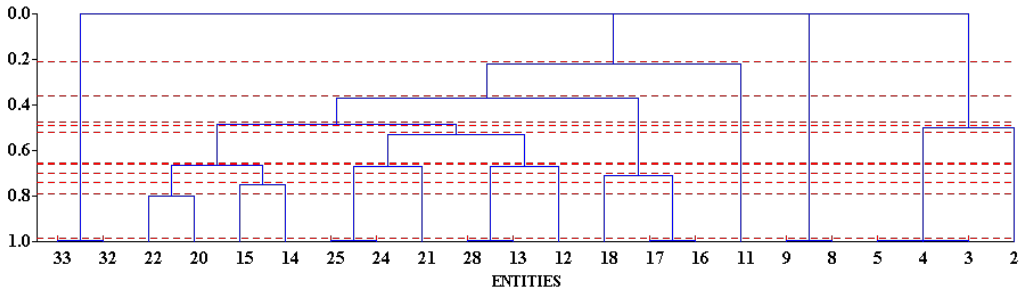
WPGMA(S)

Cut-Point Partitions: {(~~33,32~~),(~~17,16~~),(~~25,24~~),(~~28,13~~),(~~9,8~~),(5↔3)}, {(~~22,20~~)}, {(15,14)}, {(18↔16)}, {(~~25↔21~~),(~~28↔12~~)}, {(~~22↔14~~)}, {(~~25↔12~~)}, {(5↔2)}, {(~~22↔12~~)}, {(22↔16)}, {(22↔11)} $N_{cp} = 11$, $N_{bc} = 12$

Table 5.15: Cluster analysis of dendrogram obtained by A-KNN(S) for exitAfter3DError().

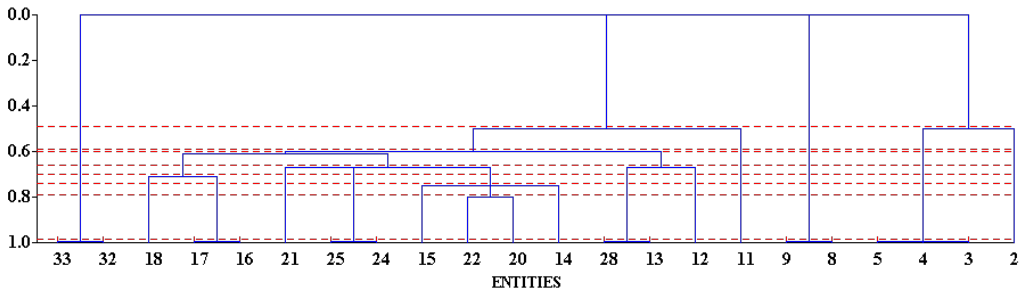
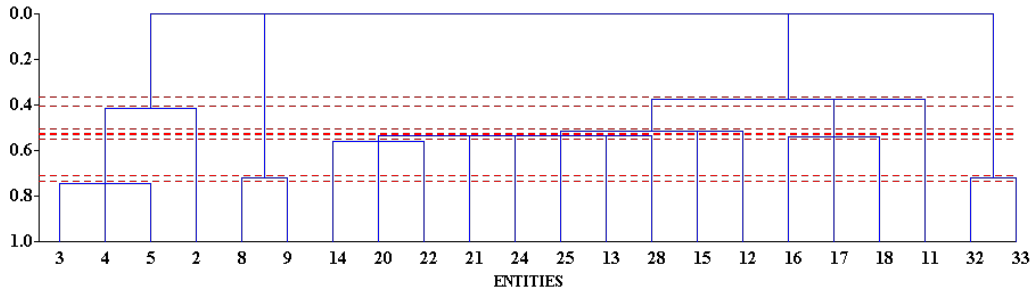
A-KNN(S)

Cut-Point Partitions: {(~~33,32~~),(~~17,16~~),(~~25,24~~),(~~28,13~~),(~~9,8~~),(5↔3)}, {(~~22,20~~)}, {(~~15↔14~~)}, {(18↔16)}, {(~~21↔14~~),(~~28↔12~~)}, {(18↔14)}, {(18↔12)}, {(18↔1),(5↔2)} $N_{cp} = 8$, $N_{bc} = 9$

Table 5.16: Cluster analysis of dendrogram obtained by (k, w)-CC(S) for exitAfter3DError().

(k, w)-CC(S)



Cut-Point Partitions:

{(3↔5)}, {~~(8,9),(32,33)~~}, {~~(14↔22)~~}, {(16↔18)}, {~~(14↔28)~~}, {~~(14↔12)~~}, {(3↔2)}, {(14↔11)}

$N_{cp} = 8$, $N_{bc} = 5$

Cluster Patterns Observed

During cluster analysis, there were several patterns observed in the clusters obtained from the dendrograms. Most of the bad clusters we found conformed with at least one of the patterns mentioned in Section 4.2 of Chapter 4.

Firstly, many of the clusters were found to be of extreme sizes. For example, referring to the code of the exitAfter3DError() function in pages 74-75, clusters (25,24), (33,32), (22,20), (28,13), which were returned by most of the algorithms, were too small to be considered as the constituents of individual functions. On the other hand, clusters like (18 ↔ 11) in the dendrogram of A-KNN(N) were too big to form any meaningful restructuring.

Then, there were several clusters which did not include all related entities, e.g., cluster (17,16), which omits the related entity 18. Although entities 17 and 16 are more similar to each other than they are with entity 18, all the three entities use the `homeFrame` variable and are in the same control block. Thus, it would be conceptually incoherent to leave out entity 18 from a function that contains entities 17, 16.

There were also clusters which only contained control entities. These clusters were designated as bad clusters for reasons mentioned in Section 4.2 of Chapter 4. The cluster (13,12) is one such example. Among such clusters,

there were clusters which contained unrelated control entities, e.g., (32,2).

Finally, there were clusters which had patterns that split conditional constructs. An example is cluster ($25 \leftrightarrow 12$) which is returned by SLINK(N). It suggests grouping the highlighted entities shown below,

```
11 else if (confirmSaveAfter3DError()) {
12 for (Home home : getHomes()){
13 if (home.isModified()) {
14 String homeName = home.getName();
15 if (homeName == null) {
16 JFrame homeFrame = getHomeFrame(home);
17 homeFrame.toFront();
18 homeName = contentManager.showSaveDialog((View) homeFrame.
        getRootPane(),null,ContentManager.ContentType.SWEET_HOME_3D,null);
19 }
20 if (homeName != null) {
21 try {
22 getHomeRecorder().writeHome(home, homeName);
23 }
24 catch (RecorderException ex) {
25 ex.printStackTrace();
26 }
27 }
28 deleteHome(home);
29 }
30 }
31 }
32 for (Home home : getHomes()) {
33 deleteHome(home);
34 }
```

As can be seen, the cluster ($25 \leftrightarrow 12$) suggests in splitting the `try-catch` construct by omitting entity 22 in the `try` block. Consequently, deploying this cluster would violate the original execution sequence of the code.

Restructured Versions

Based on the meaningful clusters that were returned by the dendrograms, two restructured versions of the function were obtained. The first version was suggested only by SLINK(N), CLINK(N), WPGMA(N), and A-KNN(N).

The second, more cohesive version was suggested by SLINK(S), CLINK(S), WPGMA(S), A-KNN(S), and (k, w)-CC. The versions suggest specific sets of lines that should be extracted to form a new function. They are shown below, (lines that are highlighted with the same colour are extracted into the same function),

Restructured Version 1

```
0 private void exitAfter3DError() {
1 boolean modifiedHomes = false;
2 for (Home home : getHomes()) {
3 if (home.isModified()) {
4 modifiedHomes = true;
5 break;
6 }
7 }
8 if (!modifiedHomes) {
9 show3DError();
10 }
11 else if (confirmSaveAfter3DError()) {
12 for (Home home :  getHomes()){
13 if (home.isModified()) {
14 String homeName = home.getName();
15 if (homeName == null) {
16 JFrame homeFrame = getHomeFrame(home);
17 homeFrame.toFront();
18 homeName = contentManager.showSaveDialog((View) homeFrame.
        getRootPane(),null,ContentManager.ContentType.SWEET_HOME_3D,null);
19 }
20 if (homeName != null) {
21 try {
22 getHomeRecorder().writeHome(home, homeName);
23 }
24 catch (RecorderException ex) {
25 ex.printStackTrace();
26 }
27 }
28 deleteHome(home);
29 }
30 }
31 }
```

```
32 for (Home home :  getHomes()) {
33 deleteHome(home);
34 }
35 System.exit(0);
36 }
```

Restructured Version 2

```
0 private void exitAfter3DError() {
1 boolean modifiedHomes = false;
2 for (Home home : getHomes()) {
3 if (home.isModified()) {
4 modifiedHomes = true;
5 break;
6 }
7 }
8 if (!modifiedHomes) {
9 show3DError();
10 }
11 else if (confirmSaveAfter3DError()) {
12 for (Home home : getHomes()){
13 if (home.isModified()) {
14 String homeName = home.getName();
15 if (homeName == null) {
16 JFrame homeFrame = getHomeFrame(home);
17 homeFrame.toFront();
18 homeName = contentManager.showSaveDialog((View) homeFrame.
        getRootPane(),null,ContentManager.ContentType.SWEET_HOME_3D,null);
19 }
20 if (homeName != null) {
21 try {
22 getHomeRecorder().writeHome(home, homeName);
23 }
24 catch (RecorderException ex) {
25 ex.printStackTrace();
26 }
27 }
28 deleteHome(home);
29 }
30 }
31 }
32 for (Home home :  getHomes()) {
```

```
33 deleteHome(home);
34 }
35 System.exit(0);
36 }
```

Both the restructured versions were found to have almost the same cohesion, each giving nearly a two-fold improvement with respect to the original cohesion of the function. (The cohesions of restructured versions 1 and 2 are 0.2 and 0.233, respectively. The original function's cohesion is 0.1022, shown in Table 5.1.)

Benefits of (k, w)-CC

As can be seen from Table 5.16, (k, w)-CC generated eight cut-points which was lower than that generated by all the other techniques using the N-Attribute Selection Mode. (k, w)-CC's N_{cp} measure was also lower than that of the other techniques, except SLINK and A-KNN, when the other techniques used the S-Attribute Selection Mode. SLINK(S) and A-KNN(S) produced the same number of cut-points as did (k, w)-CC. Nevertheless, (k, w)-CC's N_{bc} measure was lower than that of all the other techniques when they used both the attribute selection modes. From the restructuring perspective, the maximum cohesion that was attainable using all these techniques was 0.233. (k, w)-CC was among the techniques which suggested this improvement. This showed that (k, w)-CC retained the best restructuring suggestion for this function despite reducing the number of cut-points.

Therefore, for the exitAfter3DError function, (k, w)-CC has generated a clearer dendrogram by generating a lower number of cut-points and a lower number of bad clusters. In the process, it has also retained the best restructuring result that was attainable from all the techniques.

5.3 Discussion

From the results obtained in our experiments it can be established that, overall, (k, w)-CC produces both a smaller number of cut-points and a smaller number of bad clusters in its dendrogram outputs. Consequently, the dendrograms are easier to analyze and more readily usable for the purpose of restructuring

code than are those of previously good software clustering techniques (SLINK, CLINK, WPGMA, and A-KNN). It was also established that the quality of (k, w)-CC's outputs was not noticeably compromised as a result of reducing the two parameters. A key finding in this regard was that the previous algorithms gave many meaningful results, of varying quality in terms of overall cohesion. In contrast, (k, w)-CC, for most functions, gave only a single restructuring result, which turned out to be as good as the best restructuring results returned by the other techniques. Thus, (k, w)-CC was found not only to discard redundant results, but also to discard meaningful results which were of inferior quality.

As was mentioned earlier, (k, w)-CC uses the Selective Attribute Selection strategy. For maintaining consistency, we used this strategy with SLINK, CLINK, WPGMA, and A-KNN as well. The new strategy was found to improve the results of the other techniques in terms of the numbers of cut-points and bad clusters returned. This was expected since the new strategy discarded a number of attributes, leaving lesser attributes on the basis of which entity similarities were determined. As a result, the effect of omnipresent attributes on the similarity values was cancelled out. This increased the likelihood of obtaining a fewer number of distinct similarity values in the similarity matrix and thus increased the chances of obtaining more clusters with identical similarity values while using the previous clustering techniques. Since clusters with the same similarity correspond to the same cut-point in the dendrogram, the chances of obtaining lesser number of cut-points was also increased. In addition to that, any similarity that existed between widely contrasting entities (in terms of functionality) due to matching omnipresent attribute usage was discarded. This helped the algorithms to group entities meaningfully and generate a lesser number of bad clusters. Nevertheless, overall both of (k, w)-CC's N_{cp} and N_{bc} measures were still better than those of the other techniques when all used the new attribute selection strategy.

Regarding (k, w)-CC's limitations, (k, w)-CC was found to suffer when implemented on functions which contained omnipresent attributes that qualified as dependent attributes by the criteria of the Selective Attribute Selection Strategy. Although such cases were rare, this observation showed that the Selective Attribute Selection Strategy cannot always eliminate all omnipresent attributes of a function. In view of the overhead of the techniques, despite being signifi-

cantly faster than SLINK, CLINK, and WPGMA, (k, w)-CC was much slower than A-KNN. Finally, the benefits of (k, w)-CC over the other techniques were established on the basis of the results obtained from restructuring Java functions only.

We summarize our results as follows,

- Overall, (k, w)-CC gave smaller numbers of cut-points and bad clusters in its dendrograms than did all the other clustering techniques. The percentage improvements by (k, w)-CC over each of the other techniques in these two aspects is shown in Table 5.17 below,

Parameter	Percentage Improvement by (k, w)-CC							
Name	SLINK (N)	CLINK (N)	WPGMA (N)	A-KNN(N)	SLINK (S)	CLINK (S)	WPGMA (S)	A-KNN(S)
N_{cp}	52.08%	50.00%	59.29%	52.08%	29.23%	39.47%	52.58%	31.34%
N_{bc}	72.62%	69.74%	72.94%	71.95%	57.41%	62.90%	68.49%	54.00%

Table 5.17: Percentage improvement by (k, w)-CC in N_{cp} and N_{bc} over other techniques

- (k, w)-CC was faster by approximately 59.72% than SLINK, CLINK, and WPGMA. However, A-KNN was the fastest; it was found to be 94.77% faster than SLINK, CLINK, and WPGMA.

- For most of the functions, the maximum cohesion improvement attained by all the techniques, including (k, w)-CC were equal.

Chapter 6

Conclusion

Software restructuring has become an important research area particularly because of the costly impact of ill-structured code in the software industry. In this work, we have developed a new hierarchical clustering technique based on (k, w)-core decomposition, (k, w)-Core Clustering ((k, w)-CC), for restructuring software functions in order to improve cohesion, one of the most crucial aspects of software quality. Hierarchical clustering techniques generate dendrograms or clustering trees which give suggestions on how to restructure software code. Unlike previous hierarchical clustering algorithms, (k, w)-CC has been found to generate better dendrograms that are easier to analyze and hence from which restructuring solutions are more readily retrievable. In particular, (k, w)-CC has generated dendrograms that contain a smaller number of cut-points and a smaller number of bad clusters. To establish this, we have compared the dendrograms of (k, w)-CC with those of four previous hierarchical clustering algorithms (SLINK, CLINK, WPGMA, and A-KNN), that were known to give good restructuring solutions. The techniques have been implemented on functions extracted from published papers and real-life software. Overall, we have found our technique to produce the same quality restructuring solutions as those of the other techniques, through dendrograms that were more readable. In terms of performance, although (k, w)-CC was found to be slower than A-KNN, it was found to execute considerably faster than SLINK, CLINK, and WPGMA. In this thesis work, we have also developed a software tool that can automatedly generate dendrograms of Java functions, using (k, w)-CC and the four previous hierarchical clustering algorithms. We have also given a heuristic

characterization of the various clusters that may appear while analyzing the dendrograms.

Software restructuring using hierarchical clustering techniques have been widely popular particularly due to the efficiency and effectiveness with which they could be implemented. Among the early implementations were those of Anquetil *et al.* [AL99], which deployed hierarchical clustering algorithms to restructure functions. More recently, Lung *et al.* [LXZS06] gave an improved implementation of the algorithms by providing a more refined similarity metric, a key component of such algorithms. Alkhalid *et al.* [AAM10, AAM11] extended their work and designed a new and efficient hierarchical clustering algorithm for restructuring software at the class and package levels. Since all these techniques generate dendrograms, choosing the appropriate cut-points from these dendrograms, from which meaningful restructuring suggestions can be obtained, has always been a difficult problem. The previous algorithms complicate this problem by returning a large number of cut-points which yield a large number of meaningless clusters. Our approach intuitively addresses this problem by considering the structural relationship (interconnectivity) of entities, in addition to their inter-similarities. As a consequence, (k, w)-CC produces larger and more meaningful clusters.

Based on the findings obtained from this research work, there is scope for further research in several directions. Below we give ideas on some of the areas where progress could be made.

- The (k, w)-CC algorithm presented in this thesis reduces the cut-point and bad cluster counts in dendrograms, two important measures for assessing the readability of dendrograms. There is scope for building efficient clustering techniques that can achieve even further reductions in these two measures. In this regard, time may be well spent in investigating other properties of the relationships of software entities that could form as the basis of the new clustering techniques, just the way entity interconnectivity serves (k, w)-CC.

- A theoretical analysis on the ideal ratio of $share_k$ and $share_w$, two parameters which denote the level of importance given to the interconnectivity and inter-similarity properties in calculating a core's relatedness, could be carried out. In this thesis, we have determined this ratio experimentally.

- In our experiments it was seen that, overall, using the new attribute selection strategy reduced the N_{cp} and N_{bc} measures of SLINK, CLINK, WPGMA, and A-KNN. A rigorous analysis, both theoretically and experimentally, could be performed to establish defining relationships between the strategy and the characteristics of those algorithms in order to give in-depth explanations of this phenomenon.

- (k, w)-CC was found to suffer from the presence of omnipresent attributes in functions, in which it generated clusters that were too large to induce a meaningful restructuring. To deal with this a new strategy was used for choosing attributes. The strategy helped in discarding many of the omnipresent attributes, which were classified as independent. Although this improved (k, w)-CC's results for almost all cases, there were some cases where omnipresent attributes existed as dependent attributes and thus were not discarded by the selection strategy. In this aspect, there is scope for advancement in the development of clustering algorithms that are less sensitive to the presence of omnipresent attributes, while at the same time achieving the goals of producing clearer dendrograms.

- Given (k, w)-CC's benefits over the previous techniques at the function-level, there is good prospect in investigating the results of (k, w)-CC when applied to higher levels of software, e.g., class, package, or even architecture levels.

- Finally, there is considerable scope for research in the aspect of the usability of the dendrograms. In this thesis we have taken a significant step in this direction, in which we characterized clusters, obtained from dendrograms, based on some specific patterns. The characterization will aid users in assessing clusters as meaningful or redundant. Although the patterns we identified were found to be prevalent in clusters of software functions, there is room for development in identifying and classifying other complex patterns that may appear.

References

[AAM10] A. Alkhalid, M. Alshayeb, and S. Mahmoud. Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. *Journal of Advances in Engineering Software*, 41(10-11):1160–1178, 2010.

[AAM11] A. Alkhalid, M. Alshayeb, and S. Mahmoud. Software refactoring at the package level using clustering techniques. *IET Software*, 5(3):276–284, 2011.

[AJ04] A. Abran and W. M. James. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Science Society, 2004.

[AL99] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of 6th Working Conference on Reverse Engineering*, pages 235–255, 1999.

[AL03] N. Anquetil and T. C. Lethbridge. Comparative study of clustering algorithms and abstract representations for software remodularisation. In *Proceedings of IEE Software*, volume 150, pages 185–201, 2003.

[And03] K-H. Anders. A hierarchical graph-clustering approach to find groups of objects. In *Proceedings of 5th Workshop on Progress in Automated Map Generalization*, pages 1–8. Citeseer, 2003.

[BK98] James M. Bieman and Byung-Kyoo Kang. Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):111–124, 1998.

[BZ03] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. In *CoRR (Computing Research Repository), cs.DS/0310049*, 2003.

[CS06] I. G. Czibula and G. Serban. Improving systems design using a clustering approach. *International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[CTS06] A. Chatzigeorgiou, N. Tsantalis, and G. Stephanides. Application of graph theory to oo software engineering. In *Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*, pages 29–36. ACM Press, 2006.

[Eve74] B. Everitt. *Cluster Analysis*. Heinemann, London, England, 1974.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1st edition, 1999.

[FH89] E. Fix and J. Hodges. Discriminatory analysis: nonparametric discrimination: consistency properties (1951), report no. 4, project no. 21-49-004. *USAF School of Aviation Medicine*, Randolph Field, Texas, (Reprinted in *International Statistical Review*, 57(3):238–247, 1989.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, USA, 1979.

[KB99] B. K Kang and J. M. Bieman. A quantitative framework for software restructuring. *Journal of Software Maintenance: Research and Practice*, 11(4):245–284, 1999.

[Knu98] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley Professional, Redwood City, CA, USA, 1998.

[Lak93] A. Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings of the 15th International Conference on Software Engineering*, pages 35–44, 1993.

[Lak97] Arun Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, 1997.

[LXZS06] C. H. Lung, X. Xu, M. Zaman, and A. Srinivasan. Program restructuring using clustering techniques. *Journal of Systems and Software*, 79(9):1261–1279, 2006.

[LZN06] C. H. Lung, M. Zaman, and A. Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software*, 73(2):227–244, 2006.

[Mal08] R. Mall. *Fundamentals of Software Engineering*. Prentice-Hall, New Delhi, 2nd edition, 2008.

[MM06] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MMR+98] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 45–53. IEEE Computer Science Society, 1998.

[MOTU93] H. Muller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[MRS08] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 1st edition, 2008.

[NR04] T. Nishizeki and M. S. Rahman. *Planar Graph Drawing*. World Scientific, Singapore, 2004.

[PHY11] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[Pre04] R.S Pressman. *Software Engineering: A Practitioners Approach.* McGraw-Hill, 6th edition, 2004.

[SC08] G. Serban and I. G. Czibula. Object-oriented software systems restructuring through clustering. In *Proceedings of the Artificial Intelligence and Soft Computing ICAISC 2008*, pages 693–704. Springer-Verlag, 2008.

[Sei83] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[swe12] Sweethome3d, http://sourceforge.net/projects/sweethome3d/?source=directory, July 2012.

[THC09] C. E. Leiserson R. L. Rivest T. H. Cormen, C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[TSK09] P-. N Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Pearson Education, 3rd edition, 2009.

[Tze01] V. Tzerpos. *Comprehension-Driven Software Clustering.* PhD thesis, Department of Computer Science, University of Toronto, 2001.

[WT05] Z. Wen and V. Tzerpos. Software clustering based on omnipresent object detection. In *Proc. of 13th IEEE Intl Workshop on Program Comprehension*, pages 269–278, 2005.

Appendix A

The CohesionOptimizer Tool

CohesionOptimizer is a GUI-enabled software tool that allows the user to automatedly obtain clustering tree visualizations of Java functions in the form of dendrograms. The tool has been entirely coded in Java. This section describes the various features of the tool. Subsection A.1 elaborates on how to use the tool. Subsection A.2 gives the minimum system requirements of the tool.

A.1 Using the Tool

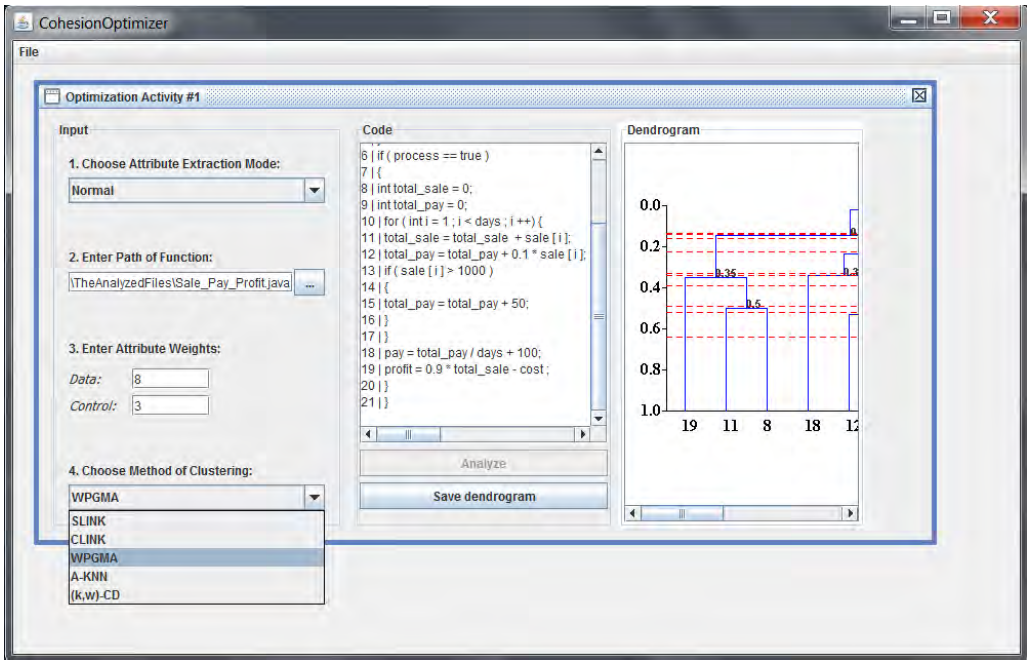


Figure A.1: The CohesionOptimizer software tool.

In order to execute the application the user has to open the .jar application file of the software. Then in order to perform a clustering of a function, the user will have to open a new "Optimization Activity" from the File menu. The window shown in Fig. A.1 would then appear.

A.1.1 Input

The user will have to input the following details in the "Optimization Activity" window,

- the attribute extraction mode,
- the weight ratio between the data and control attributes,
- the pathname of a .java file which contains the function to be restructured,
- the method of clustering that is to be used.

Format of Input File. In order to interpret[1] the input function, the tool expects the code of the function to follow traditional formats: Each statement of the code must exist in a single line. The first line of the code must be the name of the function with an opening brace and must be tagged with the text, "#HEADER#". Declaration statements must be tagged "#DEC#". Each control statement must start from a new-line. (E.g., the code "...} `else` {" must be separated into two lines, i.e., "...}" in one line and "`else` {" in the other.). Single-line comments, blank lines may be present.

Attribute Extraction Mode. Two attribute extraction modes are available: The Normal Attribute Extraction Mode directly follows the attribute selection criteria of Lung *et al.* [LXZS06] and Selective Attribute Extraction Mode, which follows the new attribute selection strategy presented in this thesis. For obtaining best results while using (k, w)-CC, we recommend the Selective Attribute Extraction Mode.

Weight Ratio. For best results, weight ratios of 8:3, 5:2, 3:1 are recommended for data to control attributes respectively.

Clustering method. Five clustering algorithms can be used to perform clustering: Single Linkage Algorithm (SLINK), Complete Linkage Algorithm

[1] A custom-built parser based on the Java StringTokenizer class was used for this purpose.

(CLINK), Weighted Pair-Group Method of Arithmetic Averages (WPGMA), Adaptive k-Nearest Neighbour Algorithm (A-KNN), and (k, w)-Core Clustering ((k, w)-CC).

After inputting the above information, the "Code" box shall display an enumerated version of the function, where each statement/LOC of the function is prefixed with a unique number. Unique numbers are given only to non-comment lines of code.

A.1.2 Output

The user can then generate a dendrogram for the input java function by clicking on the "Analyze" button. The dendrogram is displayed inside the "Dendrogram" box. The dendrogram displays the entities of the function in the horizontal axis, on a vertical scale of similarity ranging from 0 to 1. The blue lines indicate the clusters. The red-dashed lines indicate the possible lines of cut.

The "Save Dendrogram" button saves the output dendrogram as a .png image file in the same directory where the software tool application file (i.e., the .jar file) is located.

A.2 System Requirements

CohesionOptimizer is a cross-platform application that can be used in both Linux and Windows based machines with Java enabled. For best user experience we recommend the application to be used in a system configured with at least the following settings,

- Operation System: Windows XP (for Windows-based systems), Ubuntu 9.04 (for Linux-based systems)
- Processor: 1.4 GHz
- RAM: 1 Gb

Appendix B

(k, w)-CC Implementation Code

In this section we provide the implementation code for the (k, w)-Core Clustering algorithm that was developed in this work.

```
package javaapplication1;
import java.util.*;
import java.text.DecimalFormat;

class Core {
public ArrayList al = new ArrayList();
int prox[][];
int proxFull[][];
int kCMatrix[][];
int kwCMatrix[][];
int kCMax = 0, kwCMax = 0;
double cohArr[][];
double relClusters[][];
//the final set of cores to be used for clustering
int compArr[];
int compNo;
int disconnComp;
int entityCheckListMain[];
int lineNumbers[];
int total_line_count = 0;
int dC_row_cnt, dC_col_cnt;
double coeffArr[];

public Core(int[][] input, ArrayList al, int Wd, int Wc, int total_line_count, int attr_count) {
dC_row_cnt = al.size();
dC_col_cnt = al.size() + 2;
int dendseqCreator[][] = new int[dC_row_cnt][dC_col_cnt];
double coeffArr[] = new double[al.size()];
int entityCheckListMain[] = new int[al.size()];

for (int i = 0; i < al.size(); i++) {
entityCheckListMain[i] = 0;
```

```
}
this.entityCheckListMain = entityCheckListMain;

this.al = al;
Object ia[] = al.toArray();

int lineNumbers[] = new int[al.size()];
int[][] prox = new int[al.size() + 1][al.size() + 1];
//the proximity matrix
this.total_line_count = total_line_count;
int[][] proxFull = new int[total_line_count + 1][total_line_count + 1];
//the proximity matrix with all lines

for (int i = 0; i < al.size(); i++) {
lineNumbers[i] = ((Integer) ia[i]).intValue();
}

//initializing the clustering coefficients array
for (int i = 0; i < al.size(); i++) {
coeffArr[i] = 0;
}
this.coeffArr = coeffArr;

for (int i = 0; i < al.size(); i++) {
prox[0][i + 1] = lineNumbers[i];
prox[i + 1][0] = lineNumbers[i];
}

    coreDecomp();
}

//THE CORE DECOMPOSITION ALGORITHM
private void coreDecomp() {
int ProxSet[] = new int[al.size() * (al.size() - 1) / 2];
//Set of proximities
int DegSet[] = new int[al.size()];
int orderedProxSet[] = new int[al.size() * (al.size() - 1) / 2];
int orderedDegSet[] = new int[al.size()];
int maxDeg;

//generating ProxSet
for (int i = 1; i < ProxSet.length; i++) {
ProxSet[i] = 0;
}
int idx = 0;
boolean exists;
for (int i = 0; i < al.size(); i++) {
for (int j = i + 1; j < al.size(); j++) {
exists = false;
if (prox[i + 1][j + 1] != 0) {
for (int k = 0; k < ProxSet.length; k++) {
if (ProxSet[k] == prox[i + 1][j + 1]) {
```

```
exists = true;
break;
}
}
if (exists == false) {
ProxSet[idx] = prox[i + 1][j + 1];
idx++;
}
}
}
}

//ordering the ProxSet and storing it in orderedProxSet
int zeroLoc = 0;
for (int i = 0; i < orderedProxSet.length; i++) {
if (ProxSet[i] == 0) {
zeroLoc = i;
break;
}
}
System.arraycopy(ProxSet, 0, orderedProxSet, 0, ProxSet.length);
sort(orderedProxSet, 0, zeroLoc - 1);

kwCMax = orderedProxSet[zeroLoc - 1];

//generating DegSet
for (int i = 1; i < DegSet.length; i++) {
DegSet[i] = 0;
}
for (int i = 0; i < al.size(); i++) {
for (int j = 0; j < al.size(); j++) {
if (prox[i + 1][j + 1] != 0) {
DegSet[i]++;
}
}
}

//ordering the DegSet and storing it in orderedDegSet
System.arraycopy(DegSet, 0, orderedDegSet, 0, DegSet.length);
sort(orderedDegSet, 0, orderedDegSet.length - 1);

//kC MATRIX GENERATION
maxDeg = orderedDegSet[orderedDegSet.length - 1];
kCMax=maxDeg;
this.kCMatrix = new int[maxDeg][al.size() + 1];
/*an extra column is used to keep a count of the number of vertices in each column which will be
required for detecting the disconnected components in each of the cores*/

//kCMatrix initialization
for (int i = 0; i < maxDeg; i++) {
for (int j = 0; j < al.size() + 1; j++) {
this.kCMatrix[i][j] = 0;
```

```
}
}

//creating first row of the kC matrix
for (int j = 0; j < al.size(); j++) {
this.kCMatrix[0][j] = DegSet[j];
}
this.kCMatrix[0][al.size()] = al.size();

for (int i = 0; i < maxDeg - 1; i++) {
for (int j = 0; j < al.size() + 1; j++) {
this.kCMatrix[i + 1][j] = this.kCMatrix[i][j];
}

/* Constructs each row of the kC matrix, removing any vertices not satisfying the vertex function
kC, and consequently updating the degrees of the vertices
connected to the removed vertex using updateConnectedVerts */
for (int j = 0; j < al.size(); j++) {
if (this.kCMatrix[i + 1][j] < i + 2 && this.kCMatrix[i + 1][j] != 0) {
this.kCMatrix[i + 1][j] = 0;
this.kCMatrix[i + 1][al.size()]--;
updateConnectedVertskC(i, j, kCMatrix);
}
}
}

//kwC MATRIX GENERATION
this.kwCMatrix = new int[maxDeg * zeroLoc][al.size() + 1];
/*an extra column is used to keep a count of the number of vertices in each column which will be
required for detecting the disconnected components in each of the cores*/

//kwCMatrix initialization
for (int i = 0; i < maxDeg * zeroLoc; i++) {
for (int j = 0; j < al.size() + 1; j++) {
this.kwCMatrix[i][j] = 0;
}
}

//make direct copies
//filling rows of (1,w1), (2,w1), (3,w1), (kmax,w1)- cores
for (int i = 0; i < maxDeg; i++) {
for (int j = 0; j < al.size() + 1; j++) {
kwCMatrix[i * zeroLoc][j] = kCMatrix[i][j];
//safe to do direct copy since kCMatrix will not be required after this
}
}

for (int i = 0; i < maxDeg; i++) {
// for each degree
for (int k = 1; k < zeroLoc; k++) {
// for each proximity
for (int j = 0; j < al.size() + 1; j++) {
```

```
kwCMatrix[i * zeroLoc + k][j] = kwCMatrix[i * zeroLoc + k - 1][j];
}

/** * Constructs each of the remaining rows of the kwC matrix, removing any vertices not satisfying
w AND kC, and consequently updating the degrees of the vertices connected to the removed vertex using
updateConnectedVerts */
for (int j = 0; j < al.size(); j++) {
for (int m = 0; m < al.size(); m++) {
/*The following will delete any edge which does not satisfy the kwC constraint.  That is any edge
with proximity value less than the kwC value of the current core will be removed.  */
if (kwCMatrix[i * zeroLoc + k][j] != 0 &&
prox[j + 1][m + 1] != 0 &&
kwCMatrix[i * zeroLoc + k - 1][m] != 0 &&
prox[j + 1][m + 1] == (orderedProxSet[k - 1])) {
kwCMatrix[i * zeroLoc + k][j]--;
if (kwCMatrix[i * zeroLoc + k][j] == 0) {
kwCMatrix[i * zeroLoc + k][al.size()]--;
}
/* In the above steps, the edges are deleted based on the proximity value of the edge.  To reflect
this change in the data structure the corresponding entries of the concerned entities in the kwCMatrix
are decremented */
}
}
}
}

/* Now after decreasing the degrees it may so happen that the degree falls under the kC matrix constraint
So we'll need to check that as well.  */
for (int k = 1; k < zeroLoc; k++) {
for (int j = 0; j < al.size(); j++) {
{
//if the degree of an entity in the kwCMatrix is less than the core number of its parent core, we
//remove the entity (set it's degree to 0)
if (kwCMatrix[i * zeroLoc + k][j] < i + 1 && kwCMatrix[i * zeroLoc + k][j] != 0) {
kwCMatrix[i * zeroLoc + k][j] = 0;
kwCMatrix[i * zeroLoc + k][al.size()]--;
updateConnectedVertsP2(i * zeroLoc + k - 1, j, kwCMatrix, zeroLoc, orderedProxSet[k]);
}
}
}
}
}

//sort the elements of cohArr
sort(cohArr, 0, maxDeg * zeroLoc - 1);

/*CREATING RELCLUSTERS - Obtaining the relevant set of cores from kwCMatrix, ignoring cores with
0 elements.  */
double relClusters[][] = new double[al.size()][al.size() + 3];
int entityChecklist[] = new int[al.size()];
for (int i = 0; i < al.size(); i++) {
entityChecklist[i] = 0;
```

```
}
int k = 0;
double cof = 0, coreidx;
int i = maxDeg * zeroLoc - 1;

while (cohArr[i][0] > 0.0 && k < al.size()) {
boolean newEntityPresent = false;
cof = cohArr[i][0];
//starting from the highest cohArr element, which is the highest val.
coreidx = cohArr[i][1];
for (int j = 0; j < al.size(); j++) {
//checks whether the current core consists of a new entity
if (kwCMatrix[(int) coreidx][j] != 0 && entityChecklist[j] == 0) {
newEntityPresent = true;
break;
}
}
if (newEntityPresent == false) {
i--;
if (i == -1) {
break;
}
else {
continue;
}
}
else {
if (((int) coreidx / zeroLoc + 1) != 1) {
//ignoring cores with kCvalue=1 INITIALLY, since structural is the lowest in these cores
for (int j = 0; j < al.size() + 1; j++) {
relClusters[k][j] = kwCMatrix[(int) coreidx][j];
if (relClusters[k][j] != 0 && j != al.size()) {
if (entityChecklist[j] == 0) {
entityChecklist[j] = 1;
}
}
}
relClusters[k][al.size() + 1] = cof;
relClusters[k][al.size() + 2] = coreidx;
k++;
i--;
}
else {
i--;
}
}
if (i == -1) {
break;
}
}
//NOTE THAT IN THE ABOVE PROCESS, COHARR WAS USED, WHICH IS THE ORDERED SET OF RELATEDNESS
//HENCE THE RELCLUSTERS WE OBTAINED IS A SET OF CORES, ORDERED BY THEIR PROXIMITY VALUES
```

```
/*finally adding clusters from cores with kC values=1 in case some unclustered entities are still
remaining */
cof = 0;
coreidx = 0;
i = maxDeg * zeroLoc - 1;

while (cohArr[i][0] > 0.0 && k < al.size()) {
boolean newEntityPresent = false;
//whether or not the new cluster contains a new entity
cof = cohArr[i][0];
/*AGAIN HERE WE START FROM THE HIGHEST VALUED RELATEDNESS. HENCE ONLY THE MOST COHESIVE CORES WITH
kCVAL=1 ARE TAKEN*/
coreidx = cohArr[i][1];
if (((int) coreidx / zeroLoc + 1) > 1) {
//if the kCvalue of the core is greater than 1, then stop since those cores have already
//been taken care of
i--;
continue;
}
for (int j = 0; j < al.size(); j++) {
if (kwCMatrix[(int) coreidx][j] != 0 && entityChecklist[j] == 0) {
newEntityPresent = true;
break;
}
}
if (newEntityPresent == false) {
i--;
if (i == -1) {
break;
}
else {
continue;
}
}
else {
for (int j = 0; j < al.size() + 1; j++) {
relClusters[k][j] = kwCMatrix[(int) coreidx][j];
if (relClusters[k][j] != 0 && j != al.size()) {
if (entityChecklist[j] == 0) {
entityChecklist[j] = 1;
}
}
}
relClusters[k][al.size() + 1] = cof;
relClusters[k][al.size() + 2] = coreidx;
k++;
i--;
}
if (i == -1) {
break;
}
```

```
}
this.relClusters = relClusters;

//checking RelClusters for disconnected cores
int kCvalue, kwCvalue;
for (int m = 0; m < al.size(); m++) {
coreidx = relClusters[m][al.size() + 2];
kCvalue = (int) coreidx / zeroLoc + 1;
kwCvalue = orderedProxSet[(int) coreidx if (relClusters[m][al.size()] >= 2 * (kCvalue + 1)) {
int compNo = 0;
int compArr[] = new int[al.size()];
for (int j = 0; j < al.size(); j++) {
compArr[j] = 0;
}
this.compArr = compArr;

for (int j = 0; j < al.size(); j++) {
if (relClusters[m][j] != 0 && compArr[j] == 0) {
compNo++;
compArr[j] = compNo;
componentCreator(m, j, kwCvalue, compNo);
}
}

//space creation
disconnComp = 0;
spaceCreator(m);

//Splitting & storing cores with disconnected components
if (disconnComp > 1) {
splitCluster(m);
}
}
}

//deleting cores that contain entities that have already been clustered in previous cores
for (int c = 0; c < al.size(); c++) {
entityChecklist[c] = 0;
}
i = 0;
while (relClusters[i][al.size() + 1] > 0) {
boolean newEntityPresent = false;
for (int j = 0; j < al.size(); j++) {
if (relClusters[i][j] != 0 && entityChecklist[j] == 0) {
entityChecklist[j] = 1;
newEntityPresent = true;
}
}
if (newEntityPresent == false) {
for (int j = 0; j < al.size(); j++) {
if (relClusters[i][j] != 0) {
relClusters[i][j] = 1;
```

```
}
}

//delete the row
for (int j = 0; j < al.size() + 3; j++) {
relClusters[i][j] = 0;
}

//shift up the rows below
for (int x = i; x < al.size() - 1; x++) {
for (int y = 0; y < al.size() + 3; y++) {
relClusters[x][y] = relClusters[x + 1][y];
relClusters[x + 1][y] = 0;
}
}
continue;
}
i++;
}

relCluster_srt(relClusters, 0, al.size() - 1);

//simplifying relClusterValues to 1
for (int m = 0; m < al.size(); m++) {
for (int n = 0; n < al.size(); n++) {
if (relClusters[m][n] != 0) {
relClusters[m][n] = 1;
}
}
}

//Clearing kC columns of empty rows in RelClusters
for (int j = 0; j < al.size(); j++) {
if (relClusters[j][al.size() + 1] == 0) {
relClusters[j][al.size() + 2] = 0;
}
}

/*Now making sure that the final core consists of all the entities */
int finalCoreRow = 0;
for (int j = 0; j < al.size(); j++) {
if (relClusters[j][al.size() + 1] == 0) {
finalCoreRow = j - 1;
break;
}
}
for (int j = 0; j < al.size(); j++) {
if (relClusters[finalCoreRow][j] == 0) {
//the final core does not contain all the entities
for (int h = 0; h < al.size(); h++) {
relClusters[finalCoreRow + 1][h] = 1;
}
```

```
relClusters[finalCoreRow + 1][al.size() + 1] = 0.001;
break;
}
}
}

/* This updates the degrees of all vertices connected to a vertex removed during the cores generation
(while creating kC matrix) */
private void updateConnectedVertskC(int i, int j, int mat[][]) {
for (int k = 1; k < al.size() + 1; k++) {
if (prox[j + 1][k] != 0) {
//if vertices are connected in the main graph
if (mat[i + 1][k - 1] != 0) {
mat[i + 1][k - 1]--;
if (mat[i + 1][k - 1] < i + 2 && mat[i + 1][k - 1] != 0) {
mat[i + 1][k - 1] = 0;
mat[i + 1][al.size()]--;
updateConnectedVertskC(i, k - 1, mat);
}
}
}
}
}

/* This updates the degrees of all vertices connected to a vertex removed during the cores generation
(while creating kCkwC matrix) */
private void updateConnectedVertsP2(int i, int j, int mat[][], int zeroLoc, int crntProxVal) {
for (int k = 1; k < al.size() + 1; k++) {
if (mat[i + 1][k - 1] != 0) {
if (prox[j + 1][k] != 0 && prox[j + 1][k] >= crntProxVal) {
/*Decreases the degree of the connected vertices (Only those vertices are removed which have values
>= the crntProxVal since edges with less than the currentProximityVal, i.e, the value of the current
kwC value have already been deleted.  */
mat[i + 1][k - 1]--;
if (mat[i + 1][k - 1] == 0) {
//if the vertex has degree 0, it had been removed and thus count is decreased
mat[i + 1][al.size()]--;
}
if (mat[i + 1][k - 1] < ((i + 1) / zeroLoc + 1) && mat[i + 1][k - 1] != 0) {
//remove the vertex if kC constraint is broken
mat[i + 1][k - 1] = 0;
mat[i + 1][al.size()]--;
updateConnectedVertsP2(i, k - 1, mat, zeroLoc, crntProxVal);
}
}
}
}
}

/* Identifies disconnected components in a cluster */
private void componentCreator(int m, int j, int kwCval, int compNo) {
for (int k = 1; k < al.size() + 1; k++) {
```

```
if (prox[j + 1][k] >= kwCval && relClusters[m][k - 1] != 0 && compArr[k - 1] == 0) {
compArr[k - 1] = compNo;
componentCreator(m, k - 1, kwCval, compNo);
}
}
}

/* Creates space in the relCluster matrix in order to store the disconnected components of clusters
in separate rows */
private void spaceCreator(int m) {
int lastRow = 0, thisRow = 0, max = 0;
//finding number of disconnected components
for (int i = 0; i < al.size(); i++) {
if (compArr[i] > max) {
max = compArr[i];
}
}
disconnComp = max;
if (max > 1) {
//if the number of disconnected components is greater than 1
for (int k = 0; k < al.size(); k++) {
if (relClusters[k][al.size()] == 0) {
//if the count for any row is empty
lastRow = k - 1;
thisRow = m;
break;
}
}

for (int k = lastRow; k > thisRow; k--) {
if ((k + max - 1) > al.size() - 1) {
for (int l = 0; l < al.size() + 3; l++) {
relClusters[thisRow][l] = 0;
}
break;
}
for (int l = 0; l < al.size() + 3; l++) {
relClusters[k + (max - 1)][l] = relClusters[k][l];
relClusters[k][l] = 0;
}
}
}
}

/* Splits & stores clusters with disconnected components */
private void splitCluster(int m) {
for (int k = 0; k < al.size(); k++) {
if (compArr[k] != 0 && compArr[k] != 1 && (m + (compArr[k] - 1) < al.size())) {
//the final condition signifies that stop when we have reached the last Row of RelClusters
relClusters[m + (compArr[k] - 1)][k] = relClusters[m][k];
relClusters[m + (compArr[k] - 1)][al.size()] = relClusters[m][al.size()];
relClusters[m + (compArr[k] - 1)][al.size() + 1] = relClusters[m][al.size() + 1];
```

```
relClusters[m + (compArr[k] - 1)][al.size() + 2] = relClusters[m][al.size() + 2];
relClusters[m][k] = 0;
}
}
}
}
```

Index