A New Hierarchical Clustering Technique for Restructuring Software at the Function Level

Aftab Hussain
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh
aftab.hussain46@gmail.com

Md. Saidur Rahman
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh
saidurrahman@cse.buet.ac.bd

ABSTRACT

Ill-structured code is difficult to understand and thereby, costly to maintain and reuse. 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, without changing the overall behaviour of the software. These techniques generate clustering trees, of modules, that are sliced at different cut-points to obtain desired restructurings. Choosing appropriate cut-points has always been a difficult problem in clustering. Previous HAC techniques generate clustering trees that have large number of cut-points. Moreover, many of those cut-points return clusters of which only a few lead to a meaningful restructuring of the software. In this paper, we give a new hierarchical clustering technique, the (k, w)-core clustering ((k, w)-CC) technique, for restructuring software at the function level that generates clustering trees with lower number of cut-points, which yield a lower number of redundant clusters. (k, w)-CC gives good restructurings. To establish this, we provide an experimental comparison of (k, w)-CC with four previous HAC techniques: single linkage algorithm (SLINK), complete linkage algorithm (CLINK), weighted pair group method of arithmetic averages (WPGMA), and adaptive k-nearest neighbour algorithm (A-KNN). In the experiments, the techniques were implemented on Java functions extracted from real-life industrial programs.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; I.5.3 [**Clustering**]: Algorithms; G.2.2 [**Graph Theory**]: Graph Algorithms

Keywords

Software restructuring, hierarchical clustering, cohesion, dendrogram

1. INTRODUCTION

Bad designs in software has a considerable impact on the maintenance and evolution costs of software [2, 14, 16]. This has led to vast research in software restructuring. One of the most important measures of assessing the structure of software code is cohesion. The *cohesion* of a software module is the degree to which a software module does one particular task [23]. Ill-structured software code is characterized by low cohesion. Hierarchical agglomerative clustering (HAC) algorithms have found wide utility in restructuring low-cohesive software modules primarily because of the efficiency with which they could be implemented [3, 4, 5, 6, 14]. The output of these techniques is a hierarchy of clusters, which is visualized as a dendrogram. A *dendrogram* is a two-dimensional diagram in which a scale of similarity from 1 to 0 is represented in the vertical axis and entities are indicated in the horizontal axis. Each horizontal line in the dendrogram indicates a cluster whose height indicates the level of similarity of the cluster. A *cut-point* in the dendrogram is the level of similarity at which a dendrogram is cut to obtain a partition of the entities. A dendrogram with cut-points is shown in Fig. 1, where each cluster corresponds to a cut-point, while clusters with the same level of similarity correspond to the same cut-point.

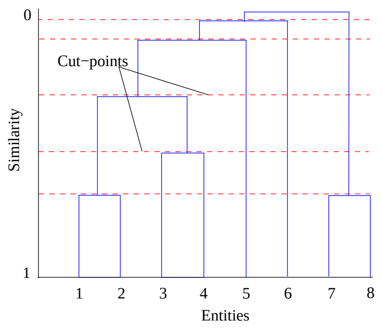


Figure 1: A dendrogram with cut-points.

Each cut-point yields a partition of clusters, which give advice on how to restructure the module. Choosing the appropriate cut-points has always been a difficult problem in clustering [6]. Previous HACs, Single Linkage Algorithm (SLINK), Complete Linkage Algorithm (CLINK), Weighted Pair Group Method of Arithmetic Averages (WPGMA), and Adaptive K-Nearest Neighbour Algorithm (A-KNN) [3, 4, 5, 6, 14], return dendrograms with a large number of cut-points, of which only a few yield clusters that lead to a meaningful restructuring. This makes it increasingly difficult for the developer to analyze the dendrogram and sieve out meaningful suggestions to perform restructuring. Pre-specifying the cut-point height or the number of clusters in the desired remodularized version of the module may seem to solve the problem. However, software code can be widely varying in structure, which is why such pre-specifications are difficult to make. In [25], a heuristic for predetermining the number of clusters in the desired restructuring is given. However, the heuristic depends on a user defined threshold value. Moreover, unfavourable results were obtained for many examples. A reasonable and intuitive way to address this problem would be to design a technique that generates clustering trees that contain a lower number of higher quality cut-points. It would also be desired if such a technique does not require any predetermined value on the height of cut-point, the cluster size, and the number of clusters. However, no such technique has been developed yet.

The main contributions of this paper are as follows,

- We have developed a new, efficient hierarchical clustering technique that gives good suggestions for restructuring software at the function level. In particular, our technique will try to give clustering trees that have a lower number of cut-points, which would be of better quality. That is, the number of redundant clusters yielded by the cut-points will be reduced.
- We have performed experiments in which we restructured functions, extracted from published papers and real-life industrial software, using the new technique and four previous hierarchical clustering techniques (SLINK, CLINK, WPGMA, A-KNN) and consequently compared the outputs of each technique.

Structure of the paper. Section 2 gives an overview of previous software restructuring techniques. Section 3 explains the overall restructuring approach that has been adopted. Section 4 discusses the clustering algorithms that has been studied in this work, and presents the new hierarchical clustering technique. Section 5 provides an experimental comparison of our approach with previous approaches. We conclude our paper in Section 6, giving directions for further research in this area.

2. RELATED WORK

Among the early works in software restructuring include that of Choi and Scacchi [10], who gave a system-restructuring algorithm, at the source file level, based on graph-theoretic computations. They map resource exchange among the source files in resource flow diagrams. Using the algorithm they derived a hierarchical description of the system. Kang and Beiman [12] restructure software modules by constructing input-output dependence graphs (IODGs) of a module. Their technique is similar to that of [13] where variable dependence graphs (VDGs) were used. The graphs indicate data dependence and control dependence relationships between input and output components of a module.

Czibula and Serban [11], proposed a partitional technique for remodularizing software at the class-level. Although their technique gave good results, it was found to take high execution time. Chatzigeorgiou *et al.* [9] suggested a partitional technique based on spectral graph clustering, without any implementation. They mentioned that the calculation of eigen vectors, a step performed in their technique, is a computationally expensive step that could lead to prohibitive execution times if implemented. Authors in [17, 19, 22] considered the partitioning problem as an optimization problem and proposed optimization algorithms (hill-climbing algorithms and genetic algorithms) to find the optimal partitions of software systems based on a cohesion criterion.

Hierarchical clustering techniques have been observed to work faster than optimization techniques [15]. Most hierarchical clustering techniques use hierarchical agglomerative clustering algorithms (HACs) [5, 6, 14] or algorithms *based* on HACs [3, 4, 25]. Lung et. al [14] used SLINK, CLINK, and WPGMA clustering algorithms to restructure software at the function level. Alkhalid *et al.* [3] followed the approach in [14] and proposed a more efficient clustering algorithm, the A-KNN algorithm, for restructuring software at the function level. In a later work [4], they used the A-KNN algorithm to restructure software at the package level.

3. RESTRUCTURING APPROACH

In this section we shall explain the approach of Lung *et al.* [14], which was followed in this work. Subsections 3.1 and 3.2 explains two basic components of the approach.

Fig. 2 depicts the overall framework of the approach. A function is taken as an input by the restructuring technique, which returns a dendrogram of the corresponding function. By selecting appropriate cut-points in the dendrogram, the developer can obtain desired restructurings of the function. The technique consists of three phases. In the first phase, "Entity-Attribute Matrix Generation", the presence state of attributes in the entities of the function is noted and an entity-attribute matrix is created. In the next phase, "Similarity Matrix Generation", similarity values between each pair of entities of the function are calculated using a resemblance coefficient. The values are stored in a similarity matrix. In the final phase, "Clustering", an HAC algorithm is applied on the similarity matrix and a cluster hierarchy is generated, which is represented in a dendrogram. Details on which entites and attributes are extracted, and the similarity metric used are discussed in Subsections 3.1 and 3.2, respectively. The HAC algorithms (SLINK, CLINK, WPGMA, A-KNN, and (k, w)-CC), which are finally applied on the similarity matrix are discussed in Section 4.

3.1 Entities and attributes

Entities are statements of a function which are to be clustered. Statements can be non-executable or executable. Non-executable statements include declaration and comment statements, which do not convey the functionality of the function and hence are ignored. Executable statements include assignment statements, condition statements, loop statements, which have a direct effect on the functionality of a function. Only executable statements are considered as entities. Entities are further classified into two groups: control entities and non-control entities. A control entity is an entity which corresponds to a predicate (condition/loop) statement. The remaining entities are referred to as non-control entities.

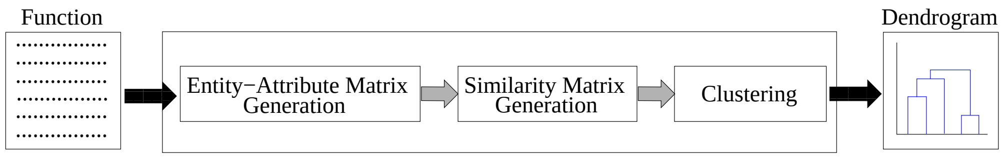

Function
Entity−Attribute Matrix Generation
Similarity Matrix Generation
Clustering
Dendrogram

Figure 2: Restructuring approach at the function-level using hierarchical clustering techniques.

In order to calculate how closely two entities are related, attributes are used. Attributes correspond to elements used in entities. Only elements that indicate a functional activity in an entity, e.g. variables and function names, are chosen as attributes. Constants, operators, loop variables and keywords are ignored. Attributes chosen in this manner reveal data dependency relationships among entities and hence are classified as data attributes. In order to obtain control dependency relationships, new attributes, control attributes, are added to the entities. Entities that belong to the same control block in the source code (e.g. `if` block), are assigned the same control entity.

Presence states of attributes in entities are stored in the entity-attribute matrix. Entities and attributes are represented in rows and columns, respectively. The value ('0', '1', or '2') of each cell of the matrix indicates the presence state of an attribute in an entity, where '0' indicates the absence of an attribute, '1' indicates the presence of a control attribute or the presence of a data attribute in a control entity, and '2' indicates the presence of a data attribute in a non-control entity.

3.2 Similarity value

The similarity value is a measure (between 0 to 1) of the degree of similarity between two entities. To calculate similarity values, Lung *et al.* [14] used a metric based on the Jaccard coefficient of similarity [5], the resemblance coefficient, which relies on similarity and dissimilarity matches between entities. Matches between entity-pairs are obtained from the entity-attribute matrix. Similarity matches include 1-1 and 2-2 matches. A 1-1 match indicates that two entities have the same control attribute, or the two entities are control entities which share a data attribute. A 2-2 match indicates that two non-control entities share a data attribute. Dissimilarity matches include 1-0/0-1 and 2-0/0-2 matches. A 1-0/0-1 match indicates that a control attribute is present in one entity and absent in the other, or, if the two entities are control entities, a data attribute is present in one and absent in the other. A 2-0/0-2 match indicates that a data attribute is present in one entity and absent in the other.

The formula for the coefficient is given below,
$coeff = (w_d a_d + w_c a_c)/(w_d a_d + w_c a_c + w_d b_d + w_c b_c)$
where, $coeff$ is the similarity value between two entities, a_c, a_d, b_c, b_d indicate the number of 1-1, 2-2, 1-0/0-1, 2-0/0-2 matches, respectively, between two entities, and w_d, w_c are non-zero weights assigned to data attributes and control attributes respectively. Because data dependency relationship is stronger than control dependency relationship in terms of functionality [13], the weights are chosen such that $w_d > w_c$. A weight ratio of 8:3 was found to give the most consistent restructuring results [14]. Using the above formula similarity values for each pair of entities are stored in the similarity matrix. The similarity matrix serves as the basis for clustering algorithms, which are explained in the following section.

4. CLUSTERING ALGORITHMS

In this section, we explain the HACs that were used for restructuring functions. In Subsection 4.1, we briefly discuss the previous HACs, and in Subsection 4.2, we present the new HAC technique that has been introduced.

4.1 SLINK, CLINK, WPGMA, A-KNN

SLINK, CLINK, and WPGMA are HACs that find a hierarchical clustering of a set of entities. They start by considering individual entities as clusters and proceed by successively merging the closest (most similar) two clusters until only one cluster remains. Entity-pair proximity (similarity) values are obtained from the similarity matrix. After each merge, the similarity matrix is recalculated in order to obtain proximites between newly formed clusters. SLINK defines the similarity between two clusters as the similarity between the closest pair of entities, taking one from each cluster. For the same purpose, CLINK uses the similarity between the farthest pair of entities, taking one from each cluster, and WPGMA uses the average pairwise similarity between all pairs of entities from different clusters.

A-KNN [3] is a variant of a traditional HAC. It avoids the recalculation of the similarity matrix at every merge and is more efficient than the previous HACs. It classifies an entity based on the majority classifications of its nearest k-neighbours. A-KNN was designed for k=3, which starts by uniquely labelling each entity. It then finds the three nearest neighbours (clusters) to the entity that will be clustered. If at least two out of the three clusters have the same label, the current entity is labelled with the same label of those two clusters. If all cluster have different labels, the entity is labelled with the label of the closest cluster. The process continues repeatedly until all entities have the same label. (For detailed mechanisms of A-KNN, see [3]).

4.2 (k, w)-Core clustering ((k, w)-CC)

The previous HACs merge clusters pairwise, and only take into account cluster/entity proximity (similarity) when making the merge decisions. As a consequence, the hierarchies produced by those techniques tend to have a large number of small clusters, which lead to a large number of cut-points in the corresponding dendrograms. During analysis of the dendrogram, the small clusters returned by the cut-points mostly turn out to be redundant in the restructuring context. This wastes analysis time and makes it difficult to find meaningful clusters. (k, w)-CC intuitively generates larger, more meaningful, clusters by considering other structural properties of clusters. Since, (k, w)-CC is based on (k, w)-core decomposition, a graph theoretic algorithm, we shall provide some necessary graph theory definitions before explaining the technique.

4.2.1 Preliminary graph definitions

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 [21]. An edge joining two vertices u and v of the graph $G = (V, E)$ can be denoted by (u, v). If $(u, v) \in E$ then the two vertices u and v of the graph G are said to be *adjacent* and the edge uv is said to be *incident* to the vertices u and v. The *degree* of a vertex v in G, $deg_G(v)$, is the number of edges incident to v in G. A graph G is called a *connected* graph 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 *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'.

We now give the definitions and lemmas of two key elements of (k, w)-CC: the *k-core*, first introduced by Seidman [24], and the (k, w)-*core*, introduced in this paper.

Definition 1. 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 [7].

LEMMA 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}.*

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

Definition 2. 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

LEMMA 2. *A (k, w)-core of a weighted graph, G, is a subgraph of a k-core of G.*

4.2.2 Clustering technique

Fig. 3 illustrates the overall approach of our novel clustering technique. In the first step, (Fig. 3(a)) a weighted graph is realised from the similarity matrix obtained from the "Similarity Matrix Generation" phase. In this graph, vertices represent entities and edges represent the presence of some similarity between the vertices (entities) joined by the edges. Each edge carries a weight equal to the similarity value between the vertices joined by the edge. Entity-pairs that have a similarity value of 0 in the similarity matrix have no edges between their representative vertices in the weighted graph. The similarity matrix therefore, serves as an adjacency matrix for this weighted graph. In the next step (Fig. 3(b)), all possible (k, w)-cores are generated from the weighted graph. In the final step of the approach (Fig. 3(c)), cores are systematically selected to form clusters, which together form a cluster hierarchy. We now discuss these last two steps in detail.

(k, w)-Core Decomposition. Batagelj *et al.* [7] 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.

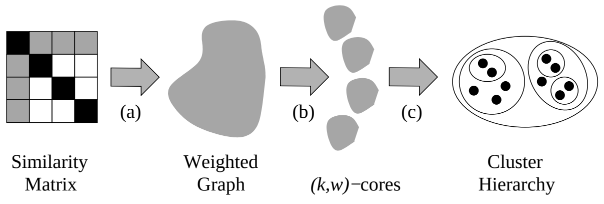


Figure 3: Clustering approach using (k, w)-Core Clustering.

Algorithm 1 shows the steps for generating all the k-cores of a weighted graph $G(V, E)$, where k belongs to the ordered (in ascending sequence) set, 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 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 2 (Steps 2, 3 in Algorithm 2). In case the degrees of any of v's adjacent vertices, denoted by u where $u \epsilon N_P(v)$, fall below k as a result of the decrements, those vertices are deleted and the update function is called again, recursively (Steps 4-7 in Algorithm 2). In this manner, the first k-core is generated (Step 9 Algorithm 1). For generating the k-core for the next value of k in ordered set D, it is sufficient to check the previous k-core rather than the entire graph (See Lemma 1). Likewise, all the subsequent k-cores are generated.

Algorithm 1 *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 2)
 7:       end if
 8:    end for
 9:    $H_k \leftarrow P(V_P, E_P)$
10:    return $H_k$
11: end for
```

Algorithm 2 *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
   }
```

By Lemma 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 3. A k-core, H_k, is taken as input in P (Step 1). Let w belong to the ordered (in ascending sequence) set, 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 set I. In this way, intermediate graphs for all $w \epsilon W$ is obtained. The deletion of an edge (Step 6) decrements degrees of vertices on which the edge was incident. As a result of this, the degrees of some vertices may fall 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 2) is called, just as was done in Algorithm 1. The output is a set of all (k, w)-cores of G for the value of k, determined by the k-core. (For e.g, if a 3-core of G is input, Algorithm 3 will generate all the $(3, w)$- cores of G.) Thus, in order to obtain all the (k, w)-cores of G, each k-core obtained from Algorithm 1 is input to Algorithm 3.

Algorithm 3 *Generating all (k, w)-cores of G for a certain value of k.*

```
 1: $P(V_P, E_P) \leftarrow H_k(V_k, E_k)$
 2: for each weight $w$ in $W$ do
 3:     for each vertex $v$ in $V_P$ do
 4:         for each vertex $u$ in $N_P(v)$ do
 5:             if $W(u, v) < w$ then
 6:                 delete edge $(u, v)$
 7:             end if
 8:         end for
 9:     end for
10:     $I \leftarrow \{I \cup H'_k(V_k, E_k - E') | E'$ is the set of edges in $H_k$ with
        $W < w\}$
11: end for
12: for each intermediate graph $I'(V_{I'}, E_{I'})$ in $I$ do
13:     for each vertex $v$ in $V_{I'}$ do
14:         if $deg_{I'}(v) < k$ then
15:             $deg_{I'}(v) \leftarrow 0$
16:             update $(v, k, I')$ //Completes deletion of $v$ in $I'$ (Algorithm 2)
17:         end if
18:     end for
19:     $H_{k,w} \leftarrow P(V_P, E_P)$
20:     return $H_{k,w}$
21: end for
```

Core Selection and Clustering Tree Generation. Since our main objective is to hierarchically cluster entities represented by the vertices in G, the (k, w)-cores obtained are systematically selected as clusters. The selection depends on a new metric, *relatedness*, which has been introduced in this paper. The metric gives a quantitative measure of the level of similarity, between 0 to 1, among the vertices (entities) of a core. We formulate the relatedness, $R(H_{k,w})$, of a (k, w)-core, $H_{k,w}$, as,

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

where $strength_k (= k/degree_{max})$, resembles the *structural relatedness* of $H_{k,w}$, and $strength_w (= w/weight_{max})$, 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). $share_k$, $share_w$ are the percentage contributions to the overall relatedness of the core by the structural and weight relatedness parameters, respectively.

Because R considers the structural relationship, $strength_k$, of vertices in cores, cores whose vertices have a high interconnectivity can also have high R values. Such cores, having high k values, have a larger number of vertices and as a result are larger in size. Selecting such cores would, thus, yield larger clusters. However, evaluating cores solely on their k-values would lead to unrealistically large clusters that will not lead to any restructuring of a module. For this reason, R also considers the inter-entity similarity, $strength_w$, of the entities in 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). We now explain the selection process in detail.

R values of 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 have the same R value as that of the removed core, $H_{k,w}$. Next, by Algorithm 4, 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 last clusters the common entities were present in (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}.

Algorithm 4 *Generating clusters from cores of $G(V, E)$*

```
 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 $e \epsilon F_i$ 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
```

Note that initially, cores with $k = 1$ are ignored (by condition in Step 9). Regardless of their R values, $(1, w)$-cores are given less preference because they have the lowest structural relatedness among all the cores. $(1, w)$-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 4 is repeated without the condition in Step 4.

4.2.3 Modified attribute selection

We have observed that using the attribute selection criteria of [14] (discussed in Subsection 3.1), some attributes distort clustering results obtained by (k, w)-CC. Such attributes tend to be present in most of the function's statements and are called *omnipresent attributes.* The notion of omnipresent software components and their negative impact on software clustering was first identified in [20]. A common example of omnipresent attributes 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 code segments are for widely contrasting purposes. Thus, considering system-related attributes would be of no help in separating these attributes.

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 undesirably leads to the generation of large clusters having high relatedness values, which ultimately provides no restructuring advice.

We give a new approach for detecting omnipresent attributes based on a new categorization of attributes; we cite that in a function an attribute can be *dependent*, i.e. it directly uses the value of another attribute in a statement, or *independent*, i.e. it does not directly use the value of any other attribute in any statement of the function. Attribute dependency is identified only for variables, through the `=` assignment operator. For example, in the statement `a=b+c`, `a` is a dependent attribute that directly uses attributes `b` and `c`. We observed that omnipresent attributes mostly tend to be independent. Thus, we discard independent attributes in (k, w)-CC.

In order to formally define attribute dependency of attributes within a statement, we need to give a simplified form for the various types of statements we may encounter. Since we are concerned with detecting dependency, only those non-control statements of the function which contain `=` as an assignment operator are considered. In order to present a suitable form of such statements, we shall consider a *refined version* of the statement 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.). (Traditional variable characterization in programs based on variable 'definition' and 'use' [18] may have been used here. However, those characterizations are more suited for data flow analysis in programs, whereas our objective here is to determine inter-attribute dependency at the statement level.)

Let S be the set of all non-control statements 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 statement $s \epsilon S$, 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 statement $s \epsilon S$. Before giving a complete definition of a dependent attribute, it is necessary to understand the ramifications of the various of statements we may encounter.

We have seen most of these statements to have two basic characteristics. We now explore each of these two characteristics, and show how they affect the process of finding attribute dependencies in statements.

1. Variables assigned to variables. This trait resembles a statement where the value of a certain variable is assigned to another variable, e.g,

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

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

```
  a = b c d
```

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

2. Functions assigned to variables. There are many statements in which 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`$\}\epsilon R$. 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 statement, except the `=` operator, we get,

```
   title = title application getName()
```

By segregating the attributes, we have `title`ϵL and $\{$`title`, `application`, `getName()`$\}\epsilon R$. 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 statement $s \epsilon S$, provided that neither of x and y are function names or array index variables.

Definition 3. 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 statement $s \epsilon S$, for any attribute $y \epsilon A$, provided that $x \neq y$ and neither of x and y are function names or array index variables.

Definition 4. 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 statement $s \epsilon S$, for any attribute $y \epsilon A$, provided that $x \neq y$ and neither of x and y are function names or array index variables.

Based on the above definitions, we shall 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 will improve the results obtained by (k, w)-CC. Only dependent attributes will be used. We shall refer to the attribute selection mode based on these criteria as *Selective (S) Attribute Selection Mode* and that based on Lung *et al.'s* criteria as *Normal (N) Attribute Selection Mode.*

5. EXPERIMENTAL RESULTS

5.1 Experimental Design

In our experiment functions extracted from published papers and real-life software were restructured using SLINK, CLINK, WPGMA, A-KNN, and (k, w)-CC. The HACs returned dendrograms that were analyzed to restructure the functions. In order to compare the techniques, we recorded the **number of cut-points** (N_{cp}) and the **number of bad clusters** (N_{bc}) that were observed in their respective dendrograms. We also measured the **execution time** by each technique to generate the dendrograms and the **maximum cohesion improvement** that was attainable through each technique.

Functions analyzed include, low cohesive functions extracted from published papers and an industrial open source Java application, Sweet Home 3D [1], a popular[1] cross platform interior design application for drawing 2D plans 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 1,

Table 1: Functions analyzed.

List of Functions Restructured			
Function Id.	Function Name	LOC	C
1	sum_max_avg [14]	11	0.191
2	sum_and_prod [14]	9	0.139
3	sale_pay_profit [3]	19	0.1524
4	sum1_or_sum2 [8]	14	0.073
5	prod1_and_prod2 [8]	8	0.121
6	fiboAvg [8]	10	0.28
7	deleteLevel [1]	41	0.13
8	displayView [1]	22	0.053
9	exitAfter3dError [1]	37	0.1022
10	updateFrameTitle [1]	37	0.0587
11	updateSunLocation [1]	40	0.113

[1]As of September 2012, the application was downloaded 82,689 times from SourceForge, receiving more than 3,000 recommendations.

As was mentioned earlier, for (k, w)-CC we have used the S-Attribute Selection Mode for attribute selection due to reasons mentioned in Subsection 4.2.3. 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 N-Attribute Selection Mode, and the S-Attribute Selection Mode. Therefore, effectively, we have implemented nine different restructuring techniques for each function: (k, w)-CC with S-Attribute Selection Mode only and SLINK, CLINK, WPGMA, A-KNN with both N- and S-Attribute Selection Mode. Henceforth, for the 11 functions mentioned above we generated and analyzed a total of 99 dendrograms by implementing these clustering techniques.

For the calculation of the similarity values (refer to Subsection 3.2), a weight ratio of 8:3 for data to control attributes was used.

5.2 Results

5.2.1 Number of cut-points and bad clusters

N_{cp} and 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. We determined bad clusters as clusters returned by a cut-point partition, which did not lead to a meaningful restructuring of the function, e.g. clusters with only control entities, clusters splitting conditional constructs, clusters with extreme sizes, clusters without related entities, etc. Note that a partition obtained from a cut-point may also yield singleton clusters. Since, singleton clusters do not give any grouping information they were ignored. Only non-singleton clusters were classified as good or bad, as they required inspection and thus consumed analysis time.

Stacked charts representing the total number of cut-points and bad clusters generated by the techniques with their prescribed attribute selection modes are given in Figures 4 and 5, respectively.

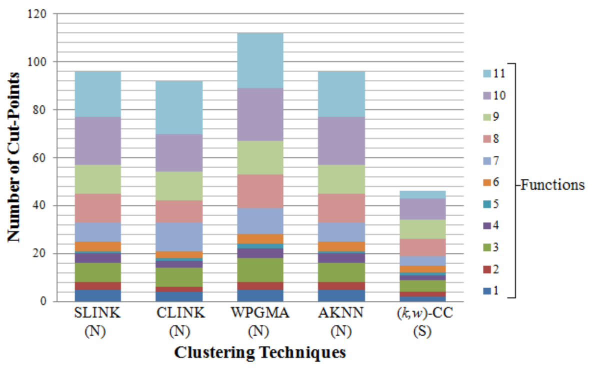


Figure 4: Total number of cut-points generated for (k, w)-CC with S-Attribute Selection Mode and SLINK, CLINK, WPGMA, A-KNN with N-Attribute Selection Mode.

As can be seen, overall, (k, w)-CC was found to give both a lesser number of cut-points and a lesser number of bad clusters than all the other techniques. For many of the functions,

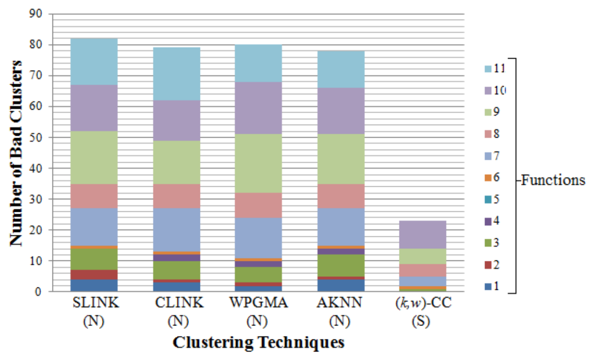


Figure 5: Total number of bad clusters generated for (k,w)-CC with S-Attribute Selection Mode and SLINK, CLINK, WPGMA, A-KNN with N-Attribute Selection Mode.

(k,w)-CC gave almost zero bad clusters. On average, (k,w)-CC gave 52.08%, 50.00%, 58.93%, 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 71.95%, 70.89%, 71.25%, 70.51% fewer number of bad clusters than did SLINK(N), CLINK(N), WPGMA(N), and A-KNN(N), respectively.

From the stacked charts in Figures 4 it can be seen that the S-Attribute Selection Mode did improve the results of the clustering techniques in this regard. However, their results were still inferior to (k,w)-CC's. is also superior in this regard. 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 58.18%, 59.65%, 70.13%, 53.06% fewer number of bad clusters than did SLINK(S), CLINK(S), WPGMA(S), and A-KNN(S), respectively.

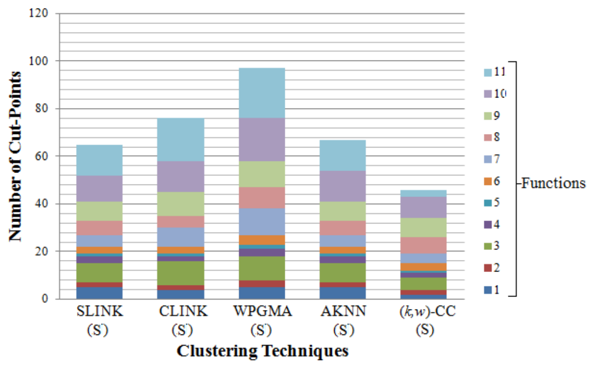


Figure 6: Total number of cut-points generated for all techniques with S-Attribute Selection Mode.

5.2.2 *Maximum Cohesion Improvement*

In order to demonstrate that (k,w)-CC retained quality restructuring suggestions, despite reducing cut-points, we compared the *maximum cohesion improvement* that was attainable by each technique, for each function. Cohesion was measured using the metric of Lung *et al.* [14]. They define the cohesion of a function as the average resemblance coefficient between any two entities of the function. The formula

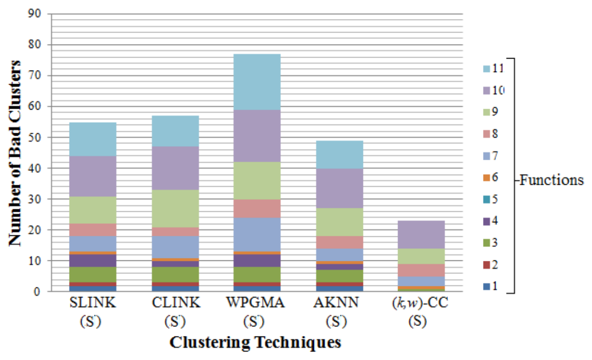


Figure 7: Total number of bad clusters generated for all techniques with S-Attribute Selection Mode.

for cohesion, C, of a function, F, is given as under,
$$C_F = \frac{\sum_{i=1}^{m}\sum_{j=1}^{m} coeff(i,j)}{m^2},$$ 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 3.2). 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 consisting of n functions is given as,
$$C_R = \frac{\sum_{i=1}^{n} C_{F_i}}{n}.$$
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 (N) and WPGMA(N) to give the best results in this aspect. A comparison of (k,w)-CC with the two techniques based on percentage improvements in cohesion for each function is shown in Figures 8.

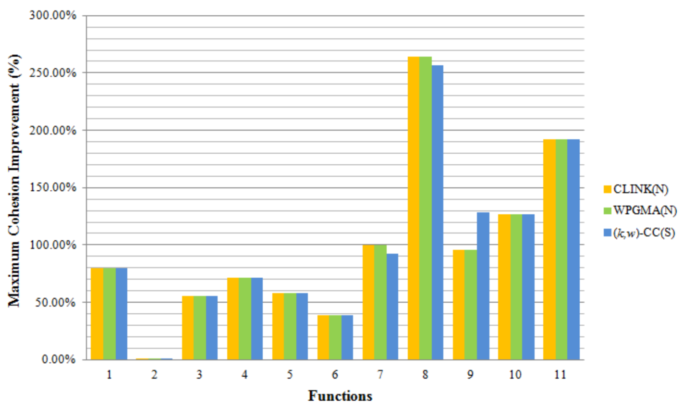


Figure 8: Maximum Cohesion Improvement through (k,w)-CC, CLINK(N), WPGMA(N)

We see that (k,w)-CC provides competitive results in terms of the quality of 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.

5.2.3 *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 our thesis. The execution time was measured only for the Clustering phase (refer to Figure 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).

It was observed that changing the attribute selection mode did not make any noticeable difference in the execution time of the techniques, which is why we implemented the algorithms with 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. Moreover, SLINK, CLINK, and WPGMA consumed the same execution times. The reason for this is due to the fact that they all deploy exactly the same algorithmic mechanism except for minor differences in the way they calculate cluster similarity. Figure 9 shows the corresponding graph for this information, with the functions presented in increasing order by LOC. All executions were carried out in a system with a 2.4 GHz processor and a 4096 Mb RAM.

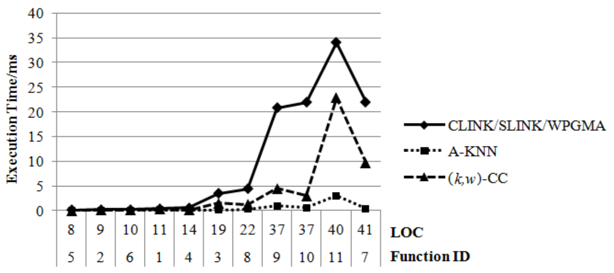


Figure 9: Execution times of the clustering techniques for each function, with functions arranged by LOC.

As can be seen, SLINK, CLINK, and WPGMA consumed the greatest execution times, while, A-KNN was found to perform the fastest. (k, w)-CC 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.

5.3 Discussion and Limitations

From our experiments we have established that (k, w)-CC produces both a lesser number of cut-points and bad clusters in its dendrogram outputs. Consequently, the dendrograms are easier to analyze and more readily usable for the purpose of restructuring codes 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 the redundant results, but also to discard meaningful results which were of inferior quality.

Nevertheless, (k, w)-CC was found to suffer in producing dendrograms for functions which contained widely present 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 significantly faster than SLINK, CLINK, and WPGMA, (k, w)-CC was much slower than A-KNN.

6. CONCLUSION AND FUTURE WORK

In this work, we developed a new hierarchical clustering technique based on (k, w)-core decomposition, (k, w)-Core Clustering ((k, w)-CC), for restructuring functions in order to improve cohesion, one of the most crucial aspects of software quality. We compared the performance of (k, w)-CC with four previous HACs (SLINK, CLINK, WPGMA, and A-KNN), that were known to give good restructuring solutions. The techniques were implemented on functions extracted from published papers and real-life software. Our technique gave the same restructuring solutions as those of the other techniques through better dendrograms. In particular, (k, w)-CC generated dendrograms that contained a lesser number of cut-points and a lesser number of bad clusters. As a result, (k, w)-CC's dendrograms were easier to analyze, from which meaningful suggestions were more readily retrievable. An important characteristic of (k, w)-CC is that it considers the structural relationship (interconnectivity) of entities, in addition to their inter-similarities. As a consequence, (k, w)-CC intuitively produces larger and more meaningful clusters. Performance-wise, although (k, w)-CC was slower than A-KNN, it was considerably faster than SLINK, CLINK, and WPGMA.

Future works on this area can entail the design of efficient software restructuring techniques that consider other properties of entity relationships with the aim of obtaining more meaningful suggestions on restructuring. In addition to that, 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. to class, package, or even architecture levels.

7. ACKNOWLEDGMENTS

We would like to thank Mohammad Tanvir Parvez, Abdulaziz Alkhalid, Mohammad Alshayeb, and Sabri Mahmoud for providing the implementation details of the A-KNN algorithm.

8. REFERENCES

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

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

[3] 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.

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

[5] 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.

[6] 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.

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

[8] J. M. Bieman and B.-K. Kang. Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):111–124, 1998.

[9] 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.

[10] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, 1990.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] A. P. Mathur. *Foundations of Software Testing.* Pearson, New Delhi, 2008.

[19] 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.

[20] 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.

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

[22] 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.

[23] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 6th edition, 2004.

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

[25] 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.

APPENDIX

A. PROOFS OF LEMMAS IN SUBSECTION 4.2.1

Proof of Lemma 1. (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, (see Definition 1), 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}.

Proof of Lemma 2. (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 (see Definitions 1, 2), 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}.

B. SAMPLE RESTRUCTURING RESULT

In this section, we shall give the complete results obtained while restructuring the exitAfter3dError() function (function no. 9 in Table 1.) Below is the code for the function,

```
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 }
```

Table 2, shows the dendrograms that were obtained when the above code was restructured using the different clustering techniques. The entity numbers in the horizontal axes of the dendrograms refer to the line numbers of above code. The number of cut-points and the number of bad clusters returned by each technique have also been indicated in the tables. The following notation were used in the tables,
1. $\{X, Y, Z\}$ - 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.
2. (x, y) - a cluster consisting of the entities x and y.
3. $(x \leftrightarrow y)$ - a cluster in the dendrogram consisting of all the entities listed on the horizontal axis of the corresponding dendrogram that are between x to 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.

While analyzing the clusters obtained from the dendrograms, there were several patterns observed in the clusters. Many of the clusters were found to be of extreme sizes. For example clusters, (25,24), (33,32), (22,20), (28,13), returned by most of the techniques, were too small to be considered as the constituents of individual functions, and clusters like $(18 \leftrightarrow 11)$ in the dendrogram of AKNN(N) were too big to form any meaningful restructuring. There were also several clusters which did not include all related entities, e.g. cluster (17,16) which omits the related entity 18. There were also clusters which only contained control entities, which did not meaningfully suggest on how to restructure the code. Examples include (20,15), (32,2), and (13,12). Finally, we found clusters which suggested in splitting conditional constructs and violating the original execution sequence of the code. An example is cluster $(25 \leftrightarrow 12)$ which is returned by SLINK(N). It suggests in 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. Also, the cluster breaks the intended execution sequence of the function by leaving out the `if` block across entities 15-18.

Based on the meaningful clusters that were output by the dendrograms, the following two restructured versions of the function were obtained. The versions are represented by indicating the lines which should be extracted from the original function. In particular, lines that are highlighted with the same colour are extracted into a separate 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()) {
```

```
deleteHome(home);
}
System.exit(0);
}
```

Restructured Version 2

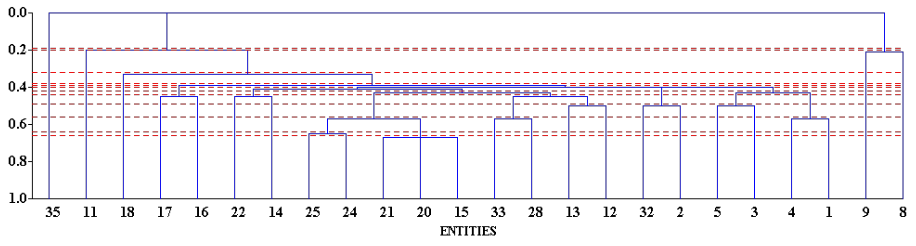
```
private void exitAfter3dError() {
boolean modifiedHomes = false;
for (Home home : getHomes()) {
if (home.isModified()) {
modifiedHomes = true;
break;
}
}
if (!modifiedHomes) {
show3DError();
}
else if (confirmSaveAfter3DError()) {
for (Home home : getHomes()){
if (home.isModified()) {
String homeName = home.getName();
if (homeName == null) {
JFrame homeFrame = getHomeFrame(home);
homeFrame.toFront();
homeName = contentManager.showSaveDialog((View) homeFrame.
            getRootPane(),null,ContentManager.ContentType.
             SWEET_HOME_3D,null);
}
if (homeName != null) {
try {
getHomeRecorder().writeHome(home, homeName);
}
catch (RecorderException ex) {
ex.printStackTrace();
}
}
deleteHome(home);
}
}
}
for (Home home :  getHomes()) {
deleteHome(home);
}
System.exit(0);
}
```

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 1.) The first version was suggested only by SLINK(N), CLINK(N), WPGMA(N), and A-KNN(N). The second version was suggested by SLINK(S), CLINK(S), WPGMA(S), A-KNN(S), and (k, w)-CC.

As can be seen, (k, w)-CC gave both a lower number of cut-points and a lower number of bad clusters than did all the other techniques.

Table 2: Cluster outputs from dendrograms obtained for the exitAfter3dError() function.

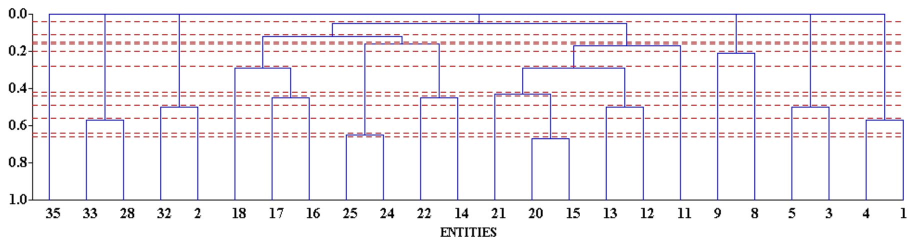
SLINK (N)



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)~~}

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

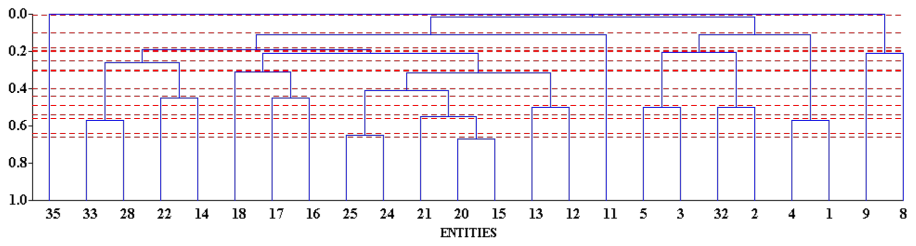
CLINK (N)



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)}

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

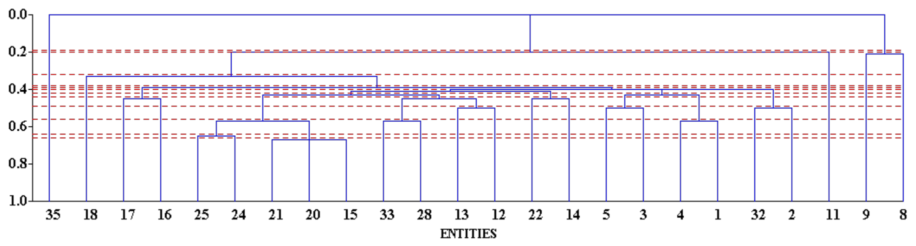
WPGMA (N)



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)~~}

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

A-KNN (N)

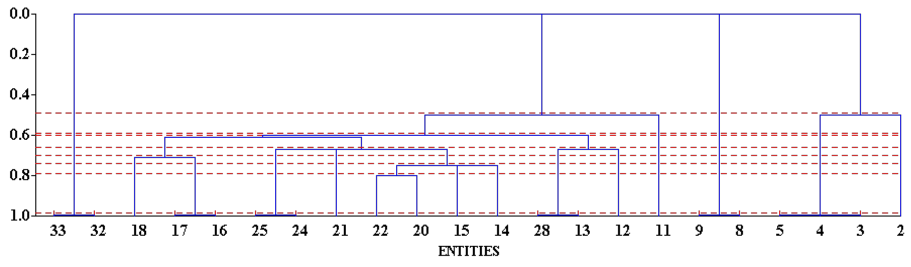


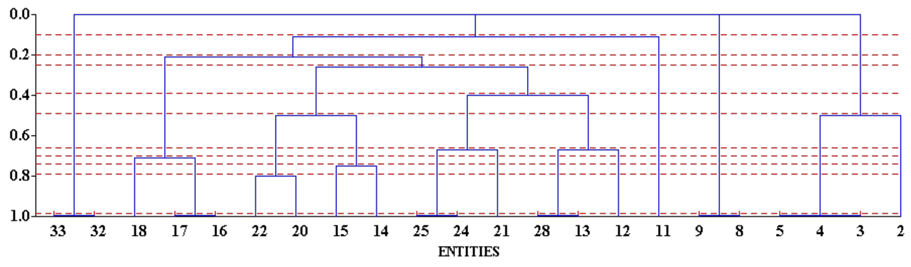
Cut-Point Partitions:
{~~(21↔15)~~}, {~~(24,24)~~}, {~~(25↔15)~~,~~(32,28)~~,(4,1)}, {~~(5,3)~~,~~(32,2)~~,~~(13,12)~~}, {~~(33↔12)~~,~~(17,16)~~},
{~~(25↔12)~~,(5↔1)}, {~~(25↔14)~~}, {~~(25↔2)~~}, {~~(17↔2)~~}, {~~(18↔2)~~}, {~~(9,8)~~}, {~~(18↔11)~~}

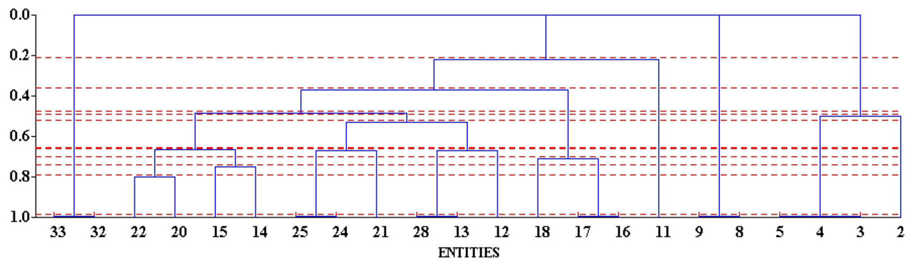
$N_{cp} = 12$, $N_{bc} = 16$

Continued on next page

Table 3: Continued from previous page

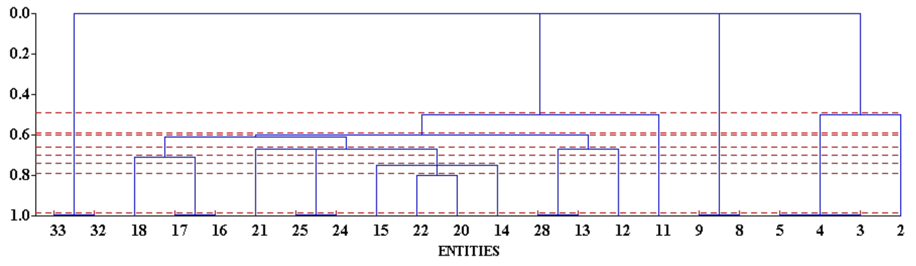
SLINK (S)

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

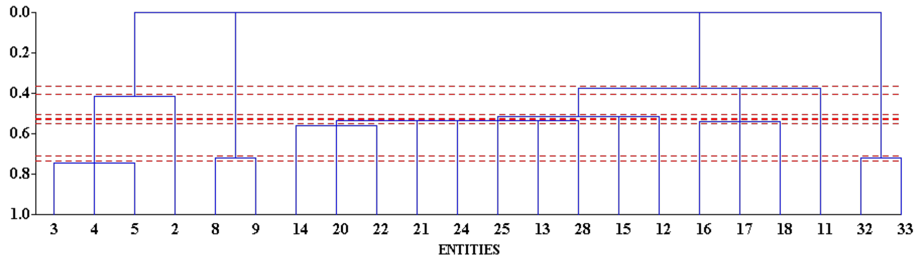
CLINK(S)

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

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)} $N_{cp} = 11$, $N_{bc} = 12$

Continued on next page

Table 4: Continued from previous page

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

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