Particle Swarm Optimization based Hierarchical Agglomerative Clustering for Software Modularization

A DISSERTATION
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR
THE AWARD OF THE DEGREE
OF
MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

SUBMITTED BY

Monika
Roll No. 13203014

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DR. B R AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY
JALANDHAR – 144011, PUNJAB (INDIA)
JULY, 2015
CANDIDATE DECLARATION

I hereby certify that the work, which is being presented in the dissertation, “Particle Swarm Optimization based Hierarchical Agglomerative Clustering for Software Modularization” by “Monika” in partial fulfillment of requirements for the award of degree of M.Tech. (Computer Science and Engineering) submitted in the Department of Computer Science and Engineering at Dr. B R Ambedkar National Institute of Technology, Jalandhar is an authentic record of my own work carried out during a period from August, 2014 to July, 2015 under the supervision of Dr. Paramvir Singh and Dr. Geeta Sikka. The matter presented in this dissertation has not been submitted by me in any other University/Institute for the award of any degree.

Monika
Roll No. 13203014

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Dr. Paramvir Singh  
Assistant Professor

Dr. Geeta Sikka

Associate Professor

Department of CSE  

Department of CSE

Dr. B. R. Ambedkar NIT, Jalandhar  

Dr. B. R. Ambedkar NIT, Jalandhar

The M.Tech (Dissertation) Viva-Voce examination of Monika, Roll No. 13203014, has been held on ____________ and accepted.

External Examiner  

Supervisors  

Head of Department
ACKNOWLEDGEMENTS

I would like to express my special thanks to my supervisors, Dr. Paramvir Singh, Assistant Professor, and Dr. Geeta Sikka, Associate Professor, Department of Computer Science and Engineering, Dr. B.R. Ambedkar National Institute of Technology, Jalandhar, for their guidance and constructive advice throughout my master’s program. Their suggestions and continuous encouragement at every stage has bought the entire work to the level of excellence. I consider myself extremely fortunate to have a chance to work under their supervision. In spite of their busy schedule, they were always approachable and took their time off to guide me and gave appropriate advice.

My profound thank to Dr. Renu Dhir, Associate professor and Head of the Department of Computer Science and Engineering, for her motivation and guidance.

I also wish to thank all the faculty members of the department of Computer Science and Engineering for their valuable knowledge they have imparted on me and for teaching the principles in most exciting and enjoyable way. I also extend my thanks to the technical and administrative staff of the department for maintaining an excellent working facility.

I would also like to extend the appreciation to my friends who helped me to accomplish my dissertation work with clarity.

My profound thank to my family for their constant companionship, humility, valuable lessons, grace and wonderful blessings. I feel proud to be nurtured and brought up by them. It’s their preaching which helps me stand out of the crowd. I owe this degree to them.

I am especially thankful for what the Lord has shower upon me, lessons we garner from the epics. May the Heavenly Father continue to guide throughout my life.

Place: NIT Jalandhar

Date: (Monika)
Software modularization is the process of automatically restructuring software units into modules to improve the software’s structure. Software systems need to evolve in order to extend or change their functionality, improve their performance, incorporate environment changes and so on. As software is revised to accommodate the required changes, its structure deteriorates.

In recent years, various clustering techniques have been explored to improve the architecture of such systems. These techniques can be divided into the following categories: graph theoretic techniques, Information Retrieval (IR) based techniques, data mining based techniques, pattern matching based techniques, hierarchical clustering techniques, and meta-heuristic approaches. Graph theoretical techniques, hierarchical clustering techniques, and meta-heuristic techniques are the prominently used techniques. Clustering in software modularization, groups the entities on the basis of similarities among them.

Hierarchical agglomerative clustering (HAC) algorithms have been widely used to restructure software systems, providing a multiple level architectural view. Weighted Combined Algorithm (WCA) is a hierarchical agglomerative clustering algorithm used for software modularization. Particle Swarm Optimization (PSO) is a partition based meta-heuristic search technique which has been successfully applied to solve the clustering problem in past. In this work we propose an approach for optimizing WCA using PSO for software modularization. To evaluate the performance of the proposed algorithm, a set of experiments are performed on five open source java software systems. The results of this empirical study show that proposed approach outperforms both WCA and PSO clustering techniques when applied to software modularization.
LIST OF PUBLICATIONS

CONTENTS

CERTIFICATE i
ACKNOWLEDGEMENT ii
ABSTRACT iii
LIST OF PUBLICATIONS iv
LIST OF FIGURES viii
LIST OF TABLES x
LIST OF ABBREVIATIONS xi

CHAPTER 1
INTRODUCTION

1.1 Software Modularization Process 02
1.2 Challenges and Issues 13
1.3 Motivations 15
1.4 Research Objectives 16
1.5 Thesis Outline 16

CHAPTER 2
LITERATURE REVIEW

2.1 Clustering Techniques 17
   2.1.1 Graph Theoretical Algorithms 17
   2.1.2 Hierarchical Clustering Algorithms 18
   2.1.3 Optimization Clustering Algorithms 23
2.2 Software Clustering Entities 25
2.3 Number of Clusters and Stopping Criteria 26
2.4 Similarity Measures 27
2.5 Chapter Summary 29

CHAPTER 3
METHODOLOGY AND IMPLEMENTATION

3.1 Sample Applications 30
3.2 Methodology 31
   3.2.1 Entities and Features 32
   3.2.2 Tool used 33
   3.2.3 Algorithms Used 33
      3.2.3.1 Weighted Combined Algorithm (WCA) 33
      3.2.3.2 Particle Swarm Optimization (PSO) 35
   3.2.4 Fitness Function 38
3.3 Proposed Hybrid Approach 38
3.4 Algorithmic Parameter 39
3.5 Chapter Summary 40

CHAPTER 4
EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Results Analysis using Number of Isolated Clusters 41
4.2 Results Analysis using Cohesion 41
4.3 Results Analysis using Coupling 44
4.4 Results Analysis using TurboMQ 44
CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions
5.2 Future Work

REFERENCES
# LIST OF FIGURES

| Figure 1.1 | Bunch software clustering process. | 11 |
| Figure 1.2 | The MDG of a small compiler | 12 |
| Figure 1.3 | The partitioned MDG of a small compiler | 12 |
| Figure 3.1 | Experimental Methodology | 32 |
| Figure 3.2 | Jdeps command to extract the dependencies | 33 |
| Figure 3.3 | WCA algorithm for software modularization | 34 |
| Figure 3.4 | A Sample of application of distributed Behavior of PSO | 36 |
| Figure 3.5 | Velocity and position update in PSO | 37 |
| Figure 3.6 | PSO algorithm for software modularization | 38 |
| Figure 3.7 | WCA-PSO algorithm for software modularization | 39 |
| Figure 4.1 | Number of isolated clusters formed by the algorithms | 45 |
| Figure 4.2 | Cohesion of the systems achieved by the algorithms | 45 |
| Figure 4.3 | Coupling of the systems achieved by the algorithms | 45 |
| Figure 4.4 | TurboMQ values achieved by the algorithm | 46 |
LIST OF TABLES

Table 3.1  Test Systems  31
Table 3.2  Parameter values for PSO and WCA-PSO  40
Table 4.1  Comparison of software clustering solutions found by WCA and WCA-PSO for Graphviz  42
Table 4.2  Comparison of software clustering solutions found by WCA and WCA-PSO for Launch4j  42
Table 4.3  Comparison of software clustering solutions found by WCA and WCA-PSO for MidiQuickFix  43
Table 4.4  Comparison of software clustering solutions found by WCA and WCA-PSO for Sellwin  43
Table 4.5  Comparison of software clustering solutions found by WCA, PSO, and WCA-PSO  44
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>MST</td>
<td>Minimal spanning tree</td>
</tr>
<tr>
<td>PSO</td>
<td>Particle Swarm Optimization</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>WCA</td>
<td>Weighted Combined Algorithm</td>
</tr>
<tr>
<td>MQ</td>
<td>Modularization Quality</td>
</tr>
<tr>
<td>MDG</td>
<td>Module Dependency Graph</td>
</tr>
<tr>
<td>CA</td>
<td>Combined Algorithm</td>
</tr>
<tr>
<td>LIMBO</td>
<td>scalable Information Bottleneck algorithm</td>
</tr>
<tr>
<td>LBFHC</td>
<td>LIMBO Based Fuzzy Hierarchical Clustering</td>
</tr>
<tr>
<td>UENM</td>
<td>Unbiased Ellenberg-NM</td>
</tr>
<tr>
<td>MCA</td>
<td>Maximum Cluster Approach</td>
</tr>
<tr>
<td>ECA</td>
<td>Equal Cluster Approach</td>
</tr>
<tr>
<td>MHypGA</td>
<td>Multi-objective Hyper heuristic Genetic Algorithm</td>
</tr>
<tr>
<td>CRM</td>
<td>Customer Relationship Management</td>
</tr>
<tr>
<td>ACO</td>
<td>Ant Colony Optimization</td>
</tr>
<tr>
<td>ABC</td>
<td>Artificial Bee Colony</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Software Modularization is the restructuring of the software as a set of modules with the well-defined API’s, adhering to a set of modularity principles.

Software systems are typically modified in order to extend or change their functionality, improve their performance, port them to different platforms, and so on. For developers, it is crucial to understand the structure of a system before attempting to modify it. The structure of a system, however, may not be apparent to new developers, because the design documentation is non-existent or, worse, inconsistent with the implementation. This problem could be alleviated if developers were somehow able to produce high-level system decomposition descriptions from the low-level structures present in the source code.

Design extraction from source code starts by parsing the source code to determine the components and relations of the system. The parsed code is then analyzed to produce views of the software structure, at varying levels of detail. Detailed views of the software structure are appropriate when the software engineer has isolated the subsystems that are relevant to his or her analysis. However, abstract (architectural) views are more appropriate when the software engineer is trying to understand the global structure of the software [1].

Software clustering is used to produce such abstract views. These views encapsulate source code-level components and relations into subsystems. The source code components and relations can be determined using source code analysis tools. The subsystems, however, are not found in the source code. Rather, they must be inferred from the source code either automatically, using a clustering tool, or manually (e.g., using the package/directory/class structure to define clusters) when tools are not available [2].

The reverse engineering research community has been actively investigating techniques to decompose (partition) the structure of software systems into subsystems (clusters). Subsystems provide developers with structural information about the numerous software components, their interfaces and their interconnections. Typical resources found in subsystems include modules, classes and possibly other subsystems. Subsystems facilitate program understanding by treating sets of source code resources as software abstractions.
1.1 Software Modularization Process

We now present the state of the art of software clustering research. We do so in the context of the more general framework of cluster analysis. Cluster analysis is a group of multivariate techniques whose primary purpose is to group entities based on their attributes. Entities are classified according to predetermined selection criteria, so that similar objects are placed in the same cluster. The typical stages of cluster analysis techniques are as follows:

i. fact extraction
ii. filtering
iii. similarity computation
iv. cluster creation
v. results visualization
vi. user feedback collection

The process typically repeats until satisfactory results are obtained. We discuss each stage in detail in the following.

i. **Fact Extraction**: Before applying clustering to a software system, the set of entities to cluster needs to be identified. When the software clustering method applies to design recovery problems [3–6], the entities are often software modules. Classes [7] or routines [8] can also be chosen as the entities. After entities have been identified, the next phase is attribute selection. An attribute is usually a software artifact, such as a package, a file, a function, a line of code, a database query, a piece of documentation, or a test case. Attributes may also be high-level concepts that encompass software artifacts, such as a design pattern. An entity may have many attributes. Selecting an appropriate set of attributes for a given clustering task is crucial for its success. In the following, we present the most common inputs for the artifact extraction process.

a) **Source Code**: Source code is the most popular input for fact extraction. Many researchers [1, 9, 10] are using the source code as the only trusted foundation for uncovering lost information about a software system. There are two conceptual approaches to extracting facts from source code: syntactic and semantic. The syntactic (structure-based) approaches focus on the static relationships among entities. The exported facts include variable and class references, procedure calls, use of packages, association and inheritance
relationships among classes. Semantic approaches [11] include all aspects of a system’s domain knowledge. The domain knowledge information present in the source code is extracted from comments, identifier names [12]. Syntactic approaches can be applied to any software system, whereas semantic approaches usually need to be customized to a specific software system due to domain specific assumptions, since two terms may be related in one domain knowledge and unrelated in another.

b) **Binary Code:** Some approaches work with the information available in binary modules. Depending on compilation and linkage parameters, the binary code may contain information, such as a symbol table that allows efficient fact extraction. This approach has three advantages.

- It is language independent.
- Binary modules are the most accurate and reliable information
- Module dependency information is easy to extract from binary modules (linkage information contains module dependency relations).

The main drawbacks of this approach are that binary metadata information depends on building parameters and that the implementation of the approach is compiler/ hardware dependent. Also, binary code analysis cannot always discover all relationships. In particular, the Java compiler erases type parameter information and resolves references to final static fields of primitive types (constants) [13].

c) **Dynamic Information:** Static information is often insufficient for recovering lost knowledge since it only provides limited insight into the runtime nature of the analyzed software; to understand behavioral system properties, dynamic information is more relevant [14]. Some information recovery approaches use dynamic information alone [15, 16], while others mix static and dynamic knowledge [15, 17, 18]. During the run-time of a software system, dynamic information is collected. The collected information may include the following.

- Object construction and destruction.
- Exceptions/errors.
- Method entry and exit.
- Component interface invocation.
- Dynamic type information.
- Dynamic component names.
- Performance counters and statistics:

There are various ways of collecting dynamic information, such as instrumentation methods or third-party tools (debuggers, performance monitors). Instrumentation techniques are based on introducing new pieces of code in many places to detect and log all collected events. Such techniques are language dependent and not trivial to apply. The biggest concern with these techniques is ensuring that the newly generated software system has the same run-time behavior as the original one.

d) **Physical Organization:** The physical organization of applications in terms of files, folders, packages, and so forth often represents valuable information for system understanding [19]. Physical organization is not limited to the software development structure. It may also include the deployment structure and build structure. The deployment structure often follows industrial standards. Therefore, the location of a specific module provides valuable information about its responsibilities.

It is important to consider the physical organization of the software system because it often reflects the main ideas of the system design.

e) **Historical Information:** Historical information explains the evolution of a software product. Recently, more research [20–22] using historical information to reveal software design has appeared. Historical information is collected from version management systems, bug tracking systems, release notes, emails, and so forth. Software evolution contains valuable information for the solution of the software understanding problem [23]. For example, release notes contain a lot of valuable knowledge about product features and product releases. Unfortunately, it is usually difficult to automatically/semi-automatically recover important system knowledge from historical sources due to the size and the lack of formatting of the extracted information.

f) **Software Documentation:** Software documents contain a lot of helpful information about software systems. However, they cannot be entirely trusted since they are often outdated or unsynchronized [24, 25]. Facts extracted from
software documents may not reflect the current state of the system. Therefore, the extracted facts should be validated with the current system. Hassan and Holt [26] present such an approach. The idea of the method is to collect information about a software system from existing documentation and domain knowledge. The gathered information is then verified against the current stage of the software implementation.

g) **Persistent Storage:** Persistent repositories, such as databases, and output files, contain information that can be helpful for software understanding. Developers often attempt to understand a software system by analyzing the application repositories. Software clustering methods should be able to utilize this information to their advantage as well [27].

h) **Human Expertise:** Humans may provide valuable facts based on their knowledge of requirement documents, high level design, and other sources. Every input source has different advantages and disadvantages. Even though the end result of the clustering process will likely improve if the input fact base contains information from different sources [28], the mixing of information from various sources is a challenging problem [4]. After the extraction process is finished, a filtering step may take place to ensure that irrelevant facts are removed, and the gathered facts are prepared for the clustering algorithm.

ii. **Filtering:** The filter phase is the final stage of preparing a fact base. The main goal of this stage is to discard unnecessary information, calculate facts that are a composition of existing facts, and apply a weighting scheme to the attributes. For instance, all meaningless words extracted from source comments are discarded during the filter step [12]. In an ideal situation, the fact base should be small and consist of enough information to ensure meaningful clustering. The information contained in the final fact base may be dependent on assumptions of a specific software clustering algorithm. Some algorithms expect that the fact base includes only relations between modules [6]; other algorithms do not make any assumptions about the facts [5].

a) **Utility Module Processing:** In many cases, modules containing utilities do not follow common design practices, such as high cohesion and low coupling. For example, utility modules may include drivers, commonly used methods. As a result, utilities may require special treatment. Some software clustering
algorithms are not affected by utilities [5]; others, such as Bunch [29], may be affected [25]. The research community has not reached a conclusion about the best approach to dealing with utilities.

b) **Granularity Projection**: A large number of software clustering algorithms attempt to cluster only course-grained software entities, such as modules or classes, rather than more fine-grained ones, such as functions or variables. However, facts extracted from software systems often contain dependencies only between the fine-grained software entities. The goal of a granularity projection filter is to use such low-level facts, such as function calls or variable references, in order to calculate dependencies between classes, and then remove the low-level facts from the fact base in order to retain facts only about the entities to be clustered.

iii. **Similarity Computation**: Most software clustering methods initially transform a fact base to a data table, where each row describes one entity to be clustered. Each column contains the value for a specific attribute.

Clustering algorithms are based on a similarity function between entities [30]. However, some algorithms, such as hierarchical agglomerative ones, are applying the similarity function explicitly, while others, such as search-based algorithms, are using the similarity function only implicitly. The most common type of similarity functions is resemblance coefficients. Other similarity functions include probabilistic measures and software-specific similarities.

a) **Resemblance Coefficients**: The input data matrix for a resemblance coefficient may contain different types of data, such as binary, numerical, categorical, or mixed.

- **Binary Resemblance**: The intuition behind the calculation of resemblance coefficients is to measure the amount of relevant matches between two entities. In other words, the more relevant matches there are between two entities, the more similar the two entities are. There are different methods for counting relevant matches, and many formulas exist to calculate resemblance coefficients [31–33]. In these formulas, $a$ represents the number of attributes that are “1” in both entities, $b$ and $c$ represent the number of attributes that are “1” in one entity and “0” in the other, and $d$ represents the number of attributes that are “0” in both entities. A binary resemblance coefficient that is
suitable for software clustering will ideally include the following two properties.

1. 0-0 matches are ignored; that is, \( d \) is not part of the formula. The joint lack of attributes between two entities should not be counted toward their similarity [34].

2. Heavier weight is assigned to more important factors [35].

A binary resemblance coefficient that fits these software clustering assumptions is the Sorenson coefficient [34]. Research [36] concludes that Jaccard and Sorenson have performed well, but the authors recommend using the Jaccard algorithm because it is more intuitive.

- **Categorical Resemblance:** There are similarities between binary and categorical resemblance coefficients. The calculation of a categorical resemblance coefficient, similar to that of a binary resemblance coefficient, is based on the number of matches between two entities. When categorical attributes are represented as a set of binary attributes, then the calculation of the categorical coefficient is based on the calculation of the binary resemblance coefficient. When categorical attributes are represented in compact form, then the categorical coefficient is calculated based on the simple matching formula.

- **Numerical Resemblance:** Numerical resemblance coefficients calculate distance between entities. Each entity is represented as a vector. Its distance to other vectors can be calculated using formulas such as

  1. Euclidean: \( \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \)
  2. Maximum: \( \max |x_i - y_i| \)
  3. Manhattan: \( \sum_{i=1}^{n} |x_i - y_i| \)

- **Mixed Resemblance:** An entity in the data table may be described by more than one type of attributes. At the same time, some values in the data table may be missing. For those cases, the widely used general similarity coefficient was developed by Gower [37]. Let \( x \) and \( y \) denote two entities and describe over \( d \) attributes. Then, the general similarity coefficient \( S_{\text{Gower}}(x, y) \) is defined as:
\[ SGower(x, y) = \frac{1}{\sum_{k=1}^{d} w(x_k, y_k)} \sum_{k=1}^{d} (x_k - y_k)^2 \]

Where \( s(x_k, y_k) \) is a similarity component for the \( k^{th} \) attribute and \( w(x_k, y_k) \) is either one or zero, depending on whether or not a comparison is valid for the \( k^{th} \) attribute of the entities.

b) **Probabilistic Measures**: Probabilistic measures are based on the idea that agreement on rare matches contributes more to the similarity between two entities than agreement on more frequent ones [38]. The probabilistic coefficients require the distribution of the frequencies of the attributes present over the set of entities. When this distribution is known, a measure of information or entropy can be computed for each attribute. Entropy is a measure of disorder; the smaller the increase in entropy when two (sets of) entities are combined is, the more similar the two entities are. For a more detailed discussion on probabilistic coefficients, we refer to [39].

c) **Software-Specific Similarity**: There are also similarity functions that have been developed specifically for the software clustering problem. Schwanke [40] introduced the notion of using design principles, such as low coupling and high cohesion. Koschke [41] has developed an extension of Schwanke’s metric-based hierarchical clustering technique. The Koschke similarity functions include global declarations, function calls. Also, the similarity method is considering name similarities between identifiers and filenames. Choi and Scacchi [42] also describe a similarity function based on maximizing the cohesiveness of clusters.

iv. **Cluster Creation**: At this point, all preparation steps are completed, and the clustering algorithm can start to execute. In this section, we discuss various software clustering algorithms. Wiggerts [38] suggests the following classification of software clustering algorithms:

a) Graph-theoretical algorithms
b) Construction algorithms
c) Optimization algorithms
d) Hierarchical algorithms

a) **Graph-Theoretical Algorithms**: This class of algorithms is based on graph properties. The nodes of such graphs represent entities, and the edges
represent relations. The main idea is to identify interesting subgraphs that will be used as basis for the clusters. Types of subgraphs that can be used for this purpose include connected components, cliques, and spanning trees. The two most common types of graph-theoretical clustering algorithms are aggregation algorithms and minimal spanning tree algorithms.

Aggregation algorithms reduce the number of nodes (representing entities) in a graph by merging them into aggregate nodes. The aggregates can be used as clusters or can be the input for a new iteration resulting in higher-level aggregates. Common graph reduction techniques are the notion of the neighborhood of a node [43], strongly connected components [44], and bicomponents [44].

Minimal spanning tree (MST) algorithms begin by finding an MST of the given graph. Next, they either interactively join the two closest nodes into a cluster or split the graph into clusters by removing “long” edges. The classic MST algorithm is not suited for software clustering due to the fact that the algorithm tends to create a few large clusters that contain many entities while several other entities remain separate [45]. Bauer and Trifu [45] suggest a two-pass modified MST algorithm. The first pass, which follows the classic MST concept, iteratively joins the two closest nodes into a cluster while the second pass assigns the remaining unclustered entities to the cluster they are the “closest” to.

b) Construction Algorithms: The algorithms in this category assign the entities to clusters in one pass. The clusters may be predefined (supervised) or constructed as part of the assignment process (unsupervised). Examples of construction algorithms include the so-called geographic techniques and the density search techniques. A well-known geographic technique is the bisection algorithm, which at each step divides the plain in two and assigns each entity according to the side that it lies on. An algorithm based on fuzzy sets was presented in [46]. An ordering is defined on entities determined by their grade of membership (defined by the characteristic function of the fuzzy set). Following this order, each entity is either assigned to the last initiated cluster or it is used to initiate a new cluster, depending on the distance to the entity which was used to initiate the last initiated cluster.
c) **Optimization Algorithms**: An optimization or improvement algorithm takes an initial solution and tries to improve this solution by iterative adaptations according to some heuristic. The optimization method has been used to produce both hierarchical [47] and nonhierarchical [48] clustering. A typical nonhierarchical clustering optimization method starts with an initial partition derived based on some heuristic. Then, entities are moved to other clusters in order to improve the partition according to some criteria. This relocating goes on until no further improvement of this criterion takes place. Examples of clustering optimization methods are presented in [49, 50].

d) **Hierarchical Algorithms**: There are two categories of hierarchical algorithms: agglomerative (bottom-up) and divisive (top-down).

*Divisive algorithms* start with one cluster that contains all entities and divide the cluster into a number (usually two) of separate clusters at each successive step. Agglomerative algorithms start at the bottom of the hierarchy by iteratively grouping similar entities into clusters. At each step, the two clusters that are most similar to each other are merged, and the number of clusters is reduced by one. According to [51], divisive algorithms offer an advantage over agglomerative algorithms because most users are interested in the main structure of the data which consists of a few large clusters found in the first steps of divisive algorithms.

*Agglomerative algorithms* perform the following steps [52].

1. Compute a similarity matrix.
2. Find the two most similar clusters and join them.
3. Calculate the similarity between the joined clusters and others obtaining a reduced matrix.
4. Repeat from step (2) until two clusters are left.

The above process implies that there is a way to calculate the similarity between an already formed cluster and other clusters/entities. This is done via what is called the update rule function. Suppose that cluster $i$ and cluster $j$ are joined to form cluster $i j$. Typical update rule functions are

1. Single linkage: \( \text{sim}(i j, k) = \min(\text{sim}(i, k), \text{sim}(j, k)) \)
2. Complete linkage: \( \text{sim}(i j, k) = \max(\text{sim}(i, k), \text{sim}(j, k)) \)
3. Average linkage: \( \text{sim}(i j, k) = (1/2)[\text{sim}(i, k) + \text{sim}(j, k)] \)
v. **Results Visualization:** The output of the cluster discovery stage needs to be presented in a user friendly manner. The challenge is to present a large amount of data with its relations in such a way that a user can understand and easily work with the data. The presentation of software clustering results is an open research question that is related to scientific visualization (the computer modeling of raw data) and human-computer interaction. The software reverse engineering community has developed various tools for the presentation of the output of software clustering. These often include the traditional software representation of a graph. Such tools include Code Crawler [53], Rigi [54], and LSEdit [55]. Rigi and LSEdit allow manual modification of the clustering result.

![Figure 1.1 Bunch software clustering process.](image)

vi. **Feedback:** During the clustering process, an expert user should be able to instruct the tool on how to improve the overall solution. Clustering algorithms that are able to process user feedback are called semiautomatic clustering algorithms. Christl et al. [56] present a semiautomatic algorithm that allows the mapping of hypothesized high level entities to source code entities. Koschke [41] created a semiautomatic clustering framework based on modified versions of the fully automatic techniques he investigated. The goal of Koschke’s framework is to enable a collaborative session between his clustering framework and the user.
The clustering algorithm does the processing, and the user validates the results. On the other hand, the result of a semiautomatic clustering algorithm may not reflect the actual state of the software system, because in many cases the software engineer may have wrong or incomplete understanding of the current state of the system. This drawback may explain the fact that the reverse engineering research community is mostly developing automatic clustering algorithms.

Figure 1.2 The MDG of a small compiler

Figure 1.3 The partitioned MDG of a small compiler
1.2 Challenges and Issues

Having presented the state of the art of software clustering methods, we go on to discuss open research challenges in software clustering.

a) *Fact Extraction*: We discussed various input sources for the attribute gathering process. Typically, a clustering method will utilize only one of these sources, such as source or binary code. However, it is possible to construct input meta-models that combine information from different sources [57]. The reason this does not happen more often in practice is the lack of an established methodology. Such a methodology has to answer various questions including the following.

1. Is it important to mix various sources? How important is a good meta-model?
2. Which source(s) contain(s) the most important information?
3. What is an effective way to represent artifacts gathered from various input sources in the same format in one meta-model? In other words, we have to define a language that allows the expression of an artifact extracted from any input source.
4. What is a reasonable weight scheme for artifacts combined from various sources? Does it vary based on factors such as the type of system or its development methodology?
5. What are other input sources for extraction of artifacts that could be integrated into the input meta-model?

b) *Cluster Discovery*: The cluster discovery process encompasses both the similarity computing and cluster creation phases. It is a complicated process that has been discussed in many research studies. Still, several key questions remain unanswered.

1. The best direction towards improving this process needs to be determined. As mentioned earlier, the cluster discovery process can be improved by developing a new software clustering algorithm or developing a new resemblance coefficient technique.
2. Several software clustering algorithms and resemblance coefficient techniques have been developed. Therefore, selecting a software clustering algorithm and a suitable resemblance coefficient is a challenging A comparative study tested on a number of systems is long overdue. It is
possible that particular combinations of clustering algorithms with
resemblance coefficients are better suited for a particular type of software
system. A categorization of algorithms based on the types of software for
which they work best would be beneficial to the software field.

3. Currently, there is no optimization-based software clustering algorithm that
can produce a nested decomposition. We know that optimization algorithms
can efficiently construct flat decompositions of a software system. It would
be worth investigating whether an optimization algorithm would be an
efficient algorithm for extracting a nested decomposition from a software
system as well.

4. Since a particular software system may have various valid decompositions,
the output of a software clustering algorithm is often not meaningful to a
software engineer. To overcome this problem, some software clustering
algorithms assign labels for each cluster. Label assignment is a challenging
task, and it is undeveloped in the context of software clustering. The
problem is that software clustering algorithms either do not assign labels or
assign labels based on simplistic rules. The development of an advanced
method for label assignment would be beneficial to the field.

c) Evaluation: Evaluation of software clustering algorithms has already been studied
from various aspects, but still there are uncharted research areas including the
following.

1. Each clustering approach uses a different set of evaluation measures from
the several measurements that have been presented in the literature.
Consequently, we cannot compare evaluation results from different
studies. To overcome this problem, the evaluation metrics have to be
compared to each other.

2. The research software community needs to define a standardized method
of evaluating software clustering. This requires the collection of reference
software systems. This collection has to contain many different types of
software systems. The construction of such a collection is complicated
because it is difficult to find good candidates. Most research studies
propose using big open source software systems as reference systems.
However typically, a big software system includes several types of
software paradigms (event based, web services, etc.). Another challenge is
that new software types constantly appear. Until now, no study has
developed a deep theoretic justification for the construction of a collection
of reference systems.

1.3 Motivations
Software clustering is the process of decomposing large software system into sub-
systems on the basis of similarity between units in the sub-systems, essentially a
depiction of the architecture. An important observation is that there are two different
conceptual approaches to developing a software clustering methodology. The first one
attempts to develop a sophisticated structure discovery approach such as ACDC and
Bunch [29]. The second approach concentrates more on developing similarity functions
[40, 41]. However, several software clustering algorithms and similarity measures have
already been developed.

Software clustering, however, is an NP-hard problem that can be efficiently
handled with help of meta-heuristic. But all the optimization techniques till date uses
random clustering as the basis for applying optimization algorithm. Particle Swarm
Optimization (PSO) has not been widely used for software modularization, but it has
been applied to a number of other research fields and results have been found to be
significant Cui et al. [58] applied hybrid PSO (combined with K-means) to the document
clustering problem and concluded that hybrid PSO produces better clustering of
documents as compared to using PSO and K-Means alone. Recently PSO was used to
optimize Genetic Algorithm (GA) for long term generation maintenance scheduling [59].
Thus, all above mentioned points constituted the motivations for this work that PSO can
be applied to optimize a software clustering methodology.

Hierarchical clustering is one of the most popular techniques for software
clustering. Hierarchical clustering algorithms produce multi-level decompositions.
During earlier phases of a typical hierarchical algorithm, a detailed view of the system’s
structure is presented and later iterations present a high-level view. Moreover,
hierarchical algorithms do not require any prior information. Weighted Combined
Algorithm (WCA) is a linkage hierarchical agglomerative clustering algorithm widely
used for software modularization.

In this work we use PSO to improve WCA. Main research question we investigate
in this work is whether optimizing WCA using PSO overcomes the shortcomings in
WCA and produces better clustering results, which has not been investigated yet in the past [61].

1.4 Research Objectives

Under the above mentioned analysis of clustering techniques, proposed objectives are:

a) To implement:
   - Automatic software modularization using hierarchical clustering based WCA.
   - Automatic software modularization using PSO.

b) To optimize hierarchical clustering based software modularization using PSO approach.

c) To compare the performance of the proposed technique with the existing WCA and PSO based approaches.

1.5 Thesis Outline

A comprehensive discussion of software modularization has been done in this chapter. Rest of the thesis is structured as follows:

- Chapter 2 presents literature surveyed on software modularization techniques and similarity measures. It also provides a summary of work done so far in related area.
- Chapter 3 of thesis discusses the research methodology and tools used.
- Chapter 4 presents the results of the experiments with a brief description about results wherever necessary.
- Conclusions with its findings and scope of future work are given in Chapter 5.
CHAPTER 2
LITERATURE REVIEW

Over the past few years there has been significant research activity in the field of reverse engineering. This chapter provides a brief insight into the techniques used for software modularization.

2.1 Clustering Techniques

Important clustering strategies for reverse engineering and re-engineering have been identified by Sartipi and Kontogiannis [61] as follows:

a) Graph-theoretical algorithms
b) Optimization algorithms
c) Hierarchical algorithms

2.1.1 Graph-Theoretical Algorithms

One of the best known algorithms for creating a minimum spanning tree (MST) is Prim's algorithm, which is one of the simplest and perhaps most efficient [62]. Prim's algorithm, initially selects a random node. The node connected to the initial node with the smallest edge weight is then combined with the initial node to form a small tree. Next, all nodes connected to this tree are examined and the one connected to the smallest weight (that does not create a cycle) is selected and joined to the tree. This process continues until all nodes are connected to the tree.

Once the MST has been constructed, clustering of this data can be done by using any clustering method. More common, for this type of strategy, is to define a threshold factor that can be used to select to remove inconsistent edges and create sub tree clusters. An inconsistent edge is determined by comparing the weight of a selected edge to the typical weight of the edges of the connected subtree. If the weight of the selected edge is greater than this typical weight by a predetermined factor (known as the threshold value), the edge is identified as inconsistent. The size of the subtree, as well as the method for determining the typical weight of the subtree are heuristically chosen, although the use of an average weight of the subtree is more common.

Paivinen and Gronfors [63] create clusters with MST by using three different edge removal techniques. Their simplest technique is to identify the longest edges and iteratively remove them until a predefined number of clusters are created. He and Chen [64] try to automate the process by taking a more global view. Their algorithm takes into
account the outlier edge lengths within the edge length distribution, which are then used as thresholds for the edge removal process. Speer et al. [65] have incorporated Genetic Algorithms to optimize edge removal according to a global clustering objective. They also found the new algorithm to perform better than their average linkage hierarchical clustering.

Maletic and Marcus [66], they use Latent Semantic Indexing as an inexpensive way to build up an understanding of the relationships within the software. The result is the converted to an MST where nodes within a predefined degree of similarity are placed in the same cluster. Bauer and Trifu [7] use a modified MST to create more cohesive clusters. Edges in this case represent the similarity between the nodes. First, they place each node in a single cluster. Edges are considered in descending order. Clusters are joined together if the considered edge similarity is above a certain value, which is based on the average value of all edges within the clusters factored by a closeness heuristic determined by the user. They found this technique helped them to create more cohesive clusters compared with their edge removal technique.

Mancoridis [67] used graph search for software clustering with modification to min-cut in the form of a Modularization Quality (MQ) measure which depicts cohesion and coupling among software modules. They presented an optimal clustering algorithm that finds the best clustering of software’s Module Dependency Graph (MDG) by finding all partitions of the MDG and selecting the best k-partition on MQ basis. The algorithm’s complexity, however, is exponential.

Spectral graph theory deals with the algebraic representations of graphs and their properties like adjacency matrix and laplacian matrix etc. Spectral graph partitioning aims to partition a graph on basis of eigen-vectors. Shokoufandeh [69] applied Spectral Graph Theory to find clusters in the Module Dependency Graph (MDG). Modeling the problem as a relaxed optimization problem with objective function trading off cohesion and coupling between nodes of the MDG, they were able to propose a polynomial-time solution within a known factor of the optimal solution.

Problem with graph theoretical methods is that the search-space grows exponentially as the size of the software system increases [35].

2.1.2 Hierarchical Clustering Algorithms
Hierarchical Clustering Strategies are divided into agglomerative and divisive [2, 26] methods, these strategies begin by assessing the similarity between entities and use that
to progressively combine clusters (or split them in case of divisive techniques). As a result, clustering produced by these strategies is normally presented as a dendrogram. This process may also involve the user selecting the appropriate number of clusters so that the search can stop before the dendrogram inevitably reaches a single cluster.

A general disadvantage with Hierarchical clustering strategies is their lack of provision for correcting a poor initial partition. These techniques are probably best suited for biological data clustering, since it is more likely that a hierarchical structure actually exists.

Some well-known examples of hierarchical agglomerative methods are Single Linkage, Complete Linkage, Average Linkage and Ward's method. The Single Linkage method is also known as the nearest neighbor. At each stage of the algorithm, the distances between all entities in different groups are considered. A single link is all that is required to join two groups. This single link is selected by identifying the two closest entities that are not from the same group. The groups containing the selected entities are then joined regardless of the distance between any of the other individuals within these groups. The same algorithm is also applied where similarity measures between entities are used. In this case, the groups containing entities with the closest level of similarity are selected. The single linkage in practice has a tendency to form long elongated clusters made of combinations of lots of small clusters at each stage. This is also known as chaining and may be considered a weakness since most users are looking for homogeneous and compact clusters.

The Complete linkage method or furthest neighbor takes an opposite view of distance between groups or clusters. The distance between groups is here defined as the distance between the furthest entities within the groups (or with least level of similarity). The smallest distance measured in this way between groups is then used to select the groups to join at each stage. The clusters found by using this method tend to give a more compact linkage but do not show high levels of agreement when used and compared with known structures.

The Average Linkage method is a compromise between the above two methods, where the distance between clusters is some measure of average of all entities within the clusters. There are a number of variants of this method. A well-known example is the centroid method. The data consist of a matrix of variables against individuals. The distances between entities are calculated using squared Euclidean distance of all their variables. Initially the two individuals with the smallest distance are put into a group.
Their group centroid is then calculated. The centroid consists of introducing the group as a new entity into the variable matrix where the value of each variable is the mean of all entities within the group. Again the squared Euclidean distance between the centroid of the group and the other entities are calculated and the smallest distance is selected to form a new group or join another entity with the current group.

*Ward's method* is designed to provide the minimum variance within the clusters. At each stage of the algorithm, two clusters are selected to join. Clusters are selected for joining if the resulting clustering causes the minimum increases to the error function. The error function for each cluster is equivalent to the sum of squared value of each inner cluster entity's deviation from the mean of the cluster. The overall error function is the sum of all the clusters error functions.

Divisive methods as mentioned previously, this method involves splitting an initial set of entities into two groups. Unfortunately there are a large number of ways to achieve this for a large set of entities (there are $2^{n-1} - 1$ ways to split $n$ entities into two groups), making this process impractical for large sets. To overcome this, we have to reduce the number of ways considered at each step. Two main types of divisive techniques discussed by Everitt [75] are *monotheic* and *polytheic*.

*Monotheic* algorithms work on binary data and use the presence or lack of a particular attributes at each stage to divide the data into two groups. An example technique is the *Association analysis*. This technique uses a binary matrix of entities against features. All entities are initially placed in a single cluster. Features are grouped in pairs and a chi square coefficient is derived for all feature pairs. If we consider $j$ and $k$ as two features (a feature pair) Chi square is derived for feature $j$ by the formula:

$$\frac{N(ad - bc)^2}{(a + b)(a + c)(b + d)(c + d)}$$

where:
- $a$ is the sum of all instances where all entities possess both $j$ and $k$.
- $b$ is the sum of all instances where all entities contain $k$ but not $j$.
- $c$ is the sum of all instances where all entities contain $j$ but not $k$.
- $d$ is the sum of all instances where all entities contain neither $j$ or $k$.
- $N$ is the total number of entities.

Then each feature's scores are summed. The feature with the highest score is selected and used for dividing the initial cluster. The initial cluster is divided into two clusters, one with the selected feature and the other without. At each iteration the next
highest scoring feature is selected (considering only feature pairs within created groups) to divide the clusters further.

*Polytheic* algorithms divide a cluster, at each stage, according to values taken by all attributes. An example of such an algorithm is taken from a description by MacNaughton-Smith et al. This technique uses a matrix of Euclidean distances of entities. Initially, all entities are placed in the same cluster. The average Euclidean distance for each entity to all other entities is then calculated. A splinter cluster is created and the individual with the highest average distance value is moved to this cluster. At this stage, the average distance of each entity remaining in the main group is calculated with respect to the splinter group and the other entities in the main group. If cases exist, where the average distance to the main group is larger than the average distance to the splinter group, the entity with the largest distance from the main group is selected and placed in the splinter group. This process is repeated until no further entities remain that are, on average, closer to the splinter group. At this stage, if required the process of splintering can be continued on each cluster.

Hierarchical clustering is one of the most popular techniques for software clustering. Davey and Burd have used agglomerative hierarchical clustering to cluster software using data and functional dependency [36]. They have compared a variety of similarity measures and linkage techniques and found Jaccard coefficient and complete linkage to produce the most cohesive results. Maqbool et al. employed an agglomerative hierarchical clustering algorithm to cluster software functions according to feature similarity. They compared more traditional linkage distance measures to a new combined distance measure which creates a new combined similarity matrix at each stage of the algorithm. They discovered that their new combined similarity matrix improved the clustering for their test system. Hierarchical clustering has also been employed to improve graphical presentation of software systems. A recent example is an ad-hoc version of hierarchical algorithm used by Di-Lucca et al. to reduce the graphical complexity of information representing web applications.

Hierarchical clustering has also been examined as an aid for legacy code migration with mixed results. For example Deursen and Kuipers have compared the use of agglomerative hierarchical clustering and concept analysis. They found the clustering algorithm lacking due to the algorithm's inability to place constructs in multiple partitions where necessary. They also found the resulting dendrogram difficult for analysis and noticed problematic results when clustering items possessing all features. On the other
hand Phattarsukol and Muenchaisri have used hierarchical clustering techniques to cluster C program functions in order to identify candidate object with positive results [75].

Anquetil et al. [69, 70] evaluated four basic agglomerative hierarchical algorithms including Complete, Single, Weighted and Unweighted linkage. The goal of their research is to measure the impact of altering various clustering parameters when applying clustering algorithms to software remodularization problems. The authors define three quality measurements that allow the results of their experiments to be compared. Of particular interest is the expert decomposition quality measurement. This measurement quantifies the difference between two clustering results, one of which is usually the decomposition produced by an expert. The measurement has two parts: precision (agreement between the clustering method and the expert) and recall (agreement between the expert and the clustering method).

Anquetil et al. also claim that clustering methods do not “discover” the system structure, but instead, impose one. Furthermore, they state that it is important to choose a clustering method that is most suited to a particular system. We feel that this assessment is correct, however, imposing a structure that is consistent with the information hiding principle is often desired, especially when the goal is to gain an understanding of a system’s structure in order to support software maintenance. As such, we find that many similarity measures are based on maximizing cohesion and minimizing coupling. The authors claim that from experimental results it was concluded that Complete produces more cohesive clusters as compared to other algorithms.

Anquetil et al. make another strong claim, namely that clustering based on naming conventions often produce better results than clustering using information extracted from source code structure. Clustering algorithms that are based on naming conventions group entities with similar source code file names and procedure names. Anquetil and Lethbridge [69,70] presented several case studies that show promising results (high precision and high recall) using name similarity as the criterion to cluster software systems.

Saeed et al. [71] developed a new linkage algorithm called Combined Algorithm (CA) and proved it better than Complete Linkage. Maqbool et al. [52] developed the WCA and proposed the Unbiased Ellenberg similarity measure for non-binary features that aims to reduce the formation of non-cohesive clusters. They compared it with
Complete Linkage and CA and suggested that WCA outperforms both of them especially with Unbiased Ellenberg similarity measure.

Andritsos et al. [5] developed an algorithm called LIMBO (scLable InforMation BOttleneck algorithm). They compared the results with ACDC, NAHC-lib, SAHC, SAHClib, Single Linkage, Complete Linkage, Weighted Average Linkage and Unweighted Average Linkage algorithms. They concluded that LIMBO usually outperforms all other algorithms.

Maqbool et al. [72] presented a comprehensive review of hierarchical clustering algorithms for software modularization and analyzed the behavior of similarity and distance measures for binary and non-binary features. Wang et al. [73] introduced an improved hierarchical clustering algorithm LBFHC (LIMBO Based Fuzzy Hierarchical Clustering) and concluded that LBFHC produces larger number of clusters with lesser number of arbitrary decisions. Naseem et al. [32] proposed a new similarity measure called Unbiased Ellenberg-NM (UENM) for non-binary features and concluded that although LIMBO performs better than WCA-UE, it is outperformed by WCA-UENM [15].

Cooperative approach based on hierarchical clustering [60], employed cooperation between more than one similarity measures. Cooperation between measures is thus a promising approach to make use of the strengths of the multiple similarity measures rather than using just a single similarity measure.

### 2.1.3 Optimization Clustering Algorithms

Optimization clustering strategies are also presented as switching [76], Partitioning [75] and Iterative Partitioning [74] strategies but they all agree on the following procedural steps:

- A method of initiating clusters
- A method for placing entities in initial clusters
- A method of reallocating entities to other clusters

K-Means clustering is one of the better known optimization techniques. A general outline of this technique involves estimating a number of initial values as values for the cluster center, which also implies that the number of clusters is selected prior to the procedure. These cluster center values may also be determined by randomly selecting a predetermined number of data entities and using them as cluster centers. Each remaining data entity is then placed in the nearest (least distant) cluster. The values of the cluster
center may, again, be calculated after each entity is added. This process continues until all entities are placed into clusters. At this stage, each entity is examined to find whether, after all initial allocation, it has become closer to another cluster. In this case the entity is reallocated. The reallocation process usually involves the recalculation of the cluster center. The reallocation process may continue as many times as necessary, or until no further improvements are possible.

Hill Climbing is another way for relocating entities after the initial clustering process of assigning all entities has taken place. The hill climb selects entities randomly and decides whether their relocation affects a global fitness criterion. A recommended fitness criterion involves minimizing inner cluster distances and optimizing distances between clusters [67]. An entity is relocated if its relocation improves the current fitness. This process continues until no further improvements are possible.

Recent examples are the work done on hill climb clustering by Mitchell et al. [67] in their clustering tool Bunch.

Everitt [75], Aldernderfer [74] and Hartigan [76] agree on the uncertainty in initial partitioning and suboptimal solutions as one of the problems with optimization strategies: There is always the possibility to find another initial partition that may result in a better final clustering. The problems with local optima or suboptimal solution arise from the fact that these strategies sample a very small proportion of all possible clustering.

Harman et al. [77] developed an automatic software modularization tool called BruBunch by re-implementing the hill climbing approach proposed by Mancoridis et al. [67]. Other search based techniques such as simulated annealing [78], multiple hill climbing [3] etc. have also been used for software modularization. However, experimental results indicate that hill-climbing algorithms produce better and stable results in reasonable time [79].

Praditwong et al. [79] modeled software modularization as a multi-objective search based problem instead of the single-objective problem as illustrated in the previous research. These multiple objectives are combined in Maximum Cluster Approach (MCA) and Equal Cluster Approach (ECA). They compared their two-archive algorithm[25] based multi-objective approaches with the hill climbing algorithm based single-objective approach of Bunch tool, and showed that the multi-objective approaches outperform the single objective approach for weighted graphs. Deepika et al. [80] also stated that the multi-objective approach is better than the single-objective approach. Barros [81] studied the effects of ECA and MQ on software modularization, and concluded that by
suppressing MQ, the multi-objective Genetic Algorithm (GA) produces high quality solutions in lesser computational time. Kumari et al. [82] proposed a fast Multi-objective Hyper heuristic Genetic Algorithm (MHypGA) for software module clustering. MHypGA produced high quality solutions with a computational time of nearly one-twentieth of the time taken by the two-archive multi-objective evolutionary algorithm [83].

Hussain et al. [84] presented a novel approach based on a relatively new and more efficient meta-heuristic technique i.e. the real valued Particle Swarm Optimization (PSO) for software clustering and claimed that PSO outperforms GA.

Koschke’s Ph.D. thesis [42] examines 23 different clustering techniques and classifies them into connection-, metric-, graph-, and concept-based categories. Of the 23 clustering techniques examined, 16 are fully-automatic and 7 are semi-automatic. Koschke also created a semi-automatic clustering framework based on modified versions of the fully-automatic techniques discussed in his thesis. The goal of Kochke’s framework is to enable a collaborative session between his clustering framework and the user. The clustering algorithm does the processing, and the user validates the results.

### 2.2 Software Clustering Entities

In carrying out architecture recovery, re-engineering and reverse engineering one need to create a different or more abstract representation of the software. This requires the identification of software entities at different levels of granularity. A useful guide for identifying these levels is the Horseshoe model [85]. This framework looks at different levels of software as a model for re-engineering. Levels of interest for extracting useful entities are the code level, functional level and architectural level.

Code level represents the actual code, which may also include non-formal features. Non-formal features refer to data that has no influence on running of code such as the function names and comment. Examples of the use of these features are the identification of semantically similar components for clustering [86] and tracing the links between documentation and source code. Functional levels are normally used to describe the formal features of software. The entities in this case would comprise of formal features, such as global variables and functions. Formal features are information that have direct impact on or are a direct consequence of software behavior", such as function calls and program variable sharing. Architectural level entities correspond to abstract components
of software. Examples of entities at this level are objects and classes. For example Mitchell et al. [67, 78] use component level for their clustering.

Functional and architectural level entities lend themselves more naturally to software analysis techniques, although techniques such as Latent Semantic indexing [22], traditionally used for document analysis, have also been adapted and used to extract non formal features of code and there are numerous tools directed at functional and architectural.

2.3 Number of Clusters and Stopping Conditions

There are no definitive techniques to predict the correct number of clusters for a set of entities. In many cases this is decided by plotting some selected clustering criterion and comparing it against the number of groups. A sharp increase or decrease (depends if we are maximizing or minimizing the criterion) may be an indication of the correct number of clusters. In case of hierarchical strategies, large changes between fusions [75] may be of interest in selecting the appropriate point of interest in the dendrogram.

For other strategies, in particular switching (optimization), finding an appropriate number of clusters is often decided by what is known as a stopping condition. When using techniques such as Hill Climbing, the fitness function acts as the stopping condition as well as a search guide. Stopping conditions can be generally divided into two categories, global and local [87]. Global rules evaluate a measure of goodness for the overall clustering. They normally involve a comparison of within cluster" (cohesion) and between cluster" (coupling) ratios or the difference of their calculated values.

Local rules on the other hand examine whether an existing cluster should be manipulated, which involves the movement of entities, splitting a cluster or amalgamating two clusters. One of the more detailed comparative studies was carried out Milligan and Cooper [88], involving the study of 30 published stopping conditions. Three global and two local conditions were found to be the best for their simulated set of data entities that contained clear-cut cluster structures. Although these rules performed best for this well organized data, other stopping conditions may be more suitable for different shapes of data density. Weakness of global stopping conditions lies in the difficulty in finding a natural definition of a global measure and local stopping conditions normally require the specification of some threshold value 40 or significance level, which depends on the properties of the entities that are mostly unknown. It seems that the best stopping
conditions need to be selected for individual distributions of data and it has not been possible to come up with a stopping condition for all data landscapes.

2.4 Similarity Measures

Once the type of entities and relations for a software system are determined, the “similarity” criteria between pairs of entities must be established. Similarity measures are designed so that larger values generally indicate a stronger similarity between the entities. For example, the Rigi tool establishes a “weight” to represent the collective “strength” of the relationships between two entities. The Arch tool uses a different similarity measurement where a similarity weight is calculated by considering common features that exist, or do not exist, between a pair of two entities. Wiggerts classifies the approach used by the abovementioned tools as association coefficients. Additional similarity measurements that have been classified are distance measures, which determine the degree of dissimilarity between two entities, correlation coefficients, which use the notion of statistical correlation to establish the degree of similarity between two entities, and probabilistic measures, which assign significance to rare features that are shared between entities (i.e., entities that share rare features may be considered more similar than entities that share common features). Example Similarity Measurements

1. Arch: The Arch similarity measurement [69] is formed by normalizing an expression based on features that are shared between two software entities, and features that are distinct to each of the entities:

\[
sim(A, B) = \frac{W(a \cap b)}{1 + c \times W(a \cap b) + d \times (W(a - b) + W(b - a))}
\]

In Arch’s similarity measurement, c and d are user-defined constants that can add or subtract significance to source code features. \(W(a \cap b)\) represents the weight assigned to features that are common to entity A and entity B. \(W(a-b)\) and \(W(b-a)\) represent the weights assigned to features that are distinct to entity A and entity B, respectively.

2. Rigi: Rigi’s primary similarity measurement, Interconnection Strength (IS), represents the exact number of syntactic objects that are exchanged or shared between two components. The IS measurement is based on information that is contained within a system’s resource flow graph (RFG). The RFG is a directed graph \(G = (V,E)\), where \(V\) is the set of nameable system components, and \(E \times V\) is a set of pairs \(u,v\) which indicates that component \(u\) provides a set of
syntactic objects to component v. Each edge \(uv, v'\) in the RFG contains an edge weight (EW) that is labeled with the actual set of syntactic objects that component u provides to component v. For example, if class v calls a method in class u named \(\text{getCurrentTime}()\), then \(\text{getCurrentTime}()\) would be included in the EW set that labels the edge from u to v in the RFG. Given the RFG, and the EW set for each edge in the RFG, the IS similarity measurement is calculated as follows:

\[
Prv(s) = \bigcup_{x \in v} EW(s,x) \quad \text{Req}(s) = \bigcup_{x \in v} EW(s,x) \\
ER(s,x) = \text{Req}(s) \cup Prv(x) \\
EP(s,x) = prv(s) \cup \text{Req}(s) \\
IS(u,v) = |ER(u,v)| + |EP(u,v)|
\]

Thus, the interconnection strength IS between a pair of source code entities is calculated by considering the number of shared exact requisitions (ER) and exact provisions (EP) between two modules. ER is the set of syntactic objects that are needed by a module and are provided by another module. EP is the set of syntactic objects that are provided by a module and needed by another module. Both ER and EP are determined by examining the set of objects that are provided by a module, \(Prv(s)\), and the set of objects that are needed (requisitioned) by a module \(\text{Req}(s)\).

3. Other classical similarity measurements: Wiggerts presents a few similarity measurements in his software clustering survey [83]. These measurements are based on classifying how a particular software feature is shared, or not shared between two software entities. Consider the table shown below.

<table>
<thead>
<tr>
<th>Entity (j)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entity (j)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Similarity measurements can be formed by considering the relative importance of how features are shared between two software entities. An interesting example is d, which represents the number of 0-0 matches, or the
number of features that are absent in both entities. According to Wiggerts, the
most commonly used similarity measurements are:

\[
simple(i,j) = \frac{a + d}{a + b + c + d} \quad jaccard(i,j) = \frac{a}{a + b + c}
\]

Thus, the simple measurement treats 0-0 (type d) and 1-1 (type a) matches
equally. The Jaccard measurement does not consider 0-0 matches. In a recent
paper, Anquetil et al. [2] applied the simple and Jaccard similarity measurements to
cluster Linux, Mosaic, gcc, and a large proprietary legacy telecommunication
system. Using their decomposition quality measurement, the authors determined
that the Jaccard tends to gives better results than the simple measurement.

2.5 Chapter Summary
This provides insight into different techniques available for software modularization.
Literature provides evidence that graph theoretic techniques, hierarchical techniques and
optimization techniques are mostly used for software modularization. It also focuses on
other important elements required for software modularization like entities to be clustered, number of clusters to be formed, stopping criteria etc.
CHAPTER 3

METHODOLOGY AND IMPLEMENTATION

This chapter provides an overview of the methodology followed to carry out the performance analysis of our proposed approach. It provides a description of set applications chosen to test the proposed approach. It provides an insight into the algorithms implementation to carry out the comparison with an overview of the fitness function chosen.

3.1 Sample Applications

For conducting our experiments, we selected five open source object oriented software systems written in java. Test systems cover small to moderate size software systems. Our test systems are chosen from a variety of domains to make the empirical study more applicable. A brief description of these systems is provided in Table 3.1. Sample applications chosen on the basis of above discussed selection criteria are described as following:

**Graphviz**

Graphviz (short for *Graph Visualization Software*) is a package of open-source tools initiated by AT&T Labs Research for drawing graphs specified in DOT language scripts. It also provides libraries for software applications to use the tools. Graphviz is free software licensed under the Eclipse Public License.

**Launch4J**

Launch4j is a cross-platform tool for wrapping Java applications distributed as jars in lightweight Windows native executable. The executable can be configured to search for a certain JRE version or use a bundled one, and it's possible to set runtime options, like the initial/max heap size. The wrapper also provides better user experience through an application icon, a native pre-JRE splash screen, and a Java download page in case the appropriate JRE cannot be found.

**MidiQuickFix**

MidiQuickFix is a software utility developed in Java that enables you to play MIDI files found on your computer, as well as brings a wide array of modifications to their properties and adjusts their characteristics.
JHawk
JHawk is a static code analysis tool - i.e. it takes the source code of the project and calculates metrics based on numerous aspects of the code - for example volume, complexity, relationships between class and packages and relationships within classes and packages. JHawk's customers operate at all levels - from small developers, through consultancies and multinationals.

SellWin
SellWin is an open source CRM Sales application. SellWin is built upon a multi-tier component architecture using the latest Java technology. Since SellWin is open source, you can custom tailor every aspect of the application to suit your unique sales process. SellWin comes completely functional with contact management, activity scheduling, data synchronization, administration dialogs, quote generation, quote-to-order conversion, and many other features. SellWin includes two user interfaces currently: a J2ME Wireless User interface that enables users to retrieve data via their phone or other J2ME capable device and a full-featured Swing client.

Table 3.1 Test Systems

<table>
<thead>
<tr>
<th>Software Systems</th>
<th>Description</th>
<th>Number of Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphviz [89]</td>
<td>Graph Visualization Software</td>
<td>31</td>
</tr>
<tr>
<td>Launch4J [90]</td>
<td>Java Executable Wrapper</td>
<td>109</td>
</tr>
<tr>
<td>MidiQuickFix [91]</td>
<td>Editor and Player</td>
<td>160</td>
</tr>
<tr>
<td>JHawk [92]</td>
<td>Java Metrics Tools</td>
<td>314</td>
</tr>
<tr>
<td>Sellwin [93]</td>
<td>CRM Sales Applications</td>
<td>322</td>
</tr>
</tbody>
</table>

3.2 Methodology
This section provides the methodology adopted to analyze the performance of proposed approach.

Stepwise Methodology:

- Firstly some java dependency analyzer tool was selected to list down the dependencies between the packages/classes of software systems in the dot format.
- Open source software systems were selected for empirical validation of proposed approach.
- A dot parser was designed to extract dependencies from dot file and encode those into numerical format. After this, the dependencies between classes of software...
systems were converted in the form of numerical matrix.

- Hierarchical agglomerative clustering technical was applied to group the classes in to different clusters according to their similarities.
- Particle swarm optimization technique was applied to optimize the results produces by hierarchical clustering.
- Once the clusters were formed, the next step was the evaluation of results by using one or more assessment criteria like modularization quality.

3.2.1 Entities and Features

Since all systems are object-oriented, we selected class as an entity. From different relationships that exist between entities, the static dependencies among classes were selected as features for finding the similarity among the entities.
3.2.2 Tool Used

Jdeps [94] is a static class dependency analyzer tool for java that is useful for quickly identifying static dependencies of applications and libraries. Jdeps can extract package level as well as class level dependencies. The input class can be a path name to a .class file, a directory, a JAR file, or it can be a fully qualified class name to analyze all class files. The options determine the output. By default, Jdeps outputs the dependencies to the system output. It can generate the dependencies in DOT language. Its output can be exported in ‘dot’ format. Jdeps comes as a built-in command line tool with JDK 8. This Jdeps command to give class level dependencies is as follows:

![Jdeps Command Example](image.png)

Dependencies of jar file Graphviz is imported in a file Graphviz.jar.dot in the directory c:\jdeps\output.

3.2.3 Algorithm Used

Clustering algorithms are techniques for searching through the set of all possible clustering to find a best clustering according to a strategy and fitness criterion, although this criterion might not be explicit. The set of all possible clustering grows very rapidly producing search spaces far too large for a complete search. All strategies try to find ways to effectively tackle this large search space. In this section, we describe algorithms used in our work for software modularization.

3.2.3.1 Weighted Combined Algorithm (WCA)

WCA is a hierarchical clustering based algorithm given by Maqbool et al. [52]. It combines entities on the basis of similarity among them. It adopts a novel two-step approach to determine the similarity of a cluster with existing entities clusters. As the first step, they associate a new feature vector with the newly formed cluster. This new
feature vector is based on the feature vector of the constituent entities. At the second step, similarity between the cluster and existing entities is recomputed. This approach allows useful feature-related information to be retained, thus reducing the number of arbitrary decisions and improving clustering results. When classes are joined to form clusters, information about number of entities accessing a feature is lost. WCA maintains this information by creating a new feature vector for newly formed cluster. When the WCA is applied, the feature vector no longer remains binary as clustering proceeds. Thus, an appropriate measure must be employed to calculate similarity, keeping in mind the non-binary nature of features.

Unbiased Ellenberg (UE) is a Jaccard like measure but for non-binary features. WCA-UE uses Unbiased Ellenberg as given in equation 1.

\[
Unbiased\ Ellenberg = \frac{0.5 \times Ma}{0.5 \times Ma + b + c}
\]

Unbiased Ellenberg-NM is a similarity measure used for non-binary features. WCA-UENM outperforms other Hierarchical Agglomerative Clustering (HAC) techniques and produces better clusters with lesser arbitrary decisions [33]. WCA-UENM uses UENM similarity measure to calculate similarity among entities.

\[
Unbiased\ Ellenberg - NM = \frac{0.5 \times Ma}{0.5 \times Ma + b + c + \pi}
= \frac{0.5 \times Ma}{0.5 \times Ma + b + c + (a + b + c + d)}
= \frac{0.5 \times Ma}{0.5 \times Ma + 2(b + c) + a + d}
\]

Where \( Ma \) represents the sum of features present in both the entities, \( a \) represents the number of features that are “1” in both entities, \( d \) represents the number of features that are “0” in both entities, whereas \( b \) and \( c \) represent the features that are “1” in one entity and “0” in the other.

\[\text{Algorithm 1: WCA Algorithm}\]

1. Associate each entity \( E_i \) with a feature vector \( f_i \) having components \( f_{i1}, f_{i2}, \ldots f_{ip} \).
2. Calculate similarity between each pair of entities within the system using (2).
3. Repeat
   a. Cluster the two most similar entities.
b. Associate a new feature vector with the newly formed cluster. Let E_i and E_j be the entities to be clustered, with feature vectors f_i and f_j, respectively. Let the number of entities in E_i be denoted by n_i and the number of entities in E_j by n_j.

The feature vector f_ij is calculated as:

\[ f_{ij} = \frac{(f_i + f_j)}{(n_i + n_j)} \]

\[ = \{ \frac{(f_{ik} + f_{jk})}{(n_i + n_j)} \} \text{ where } k = 1, 2, \ldots, p. \]

c. Treat the newly formed non-singleton cluster as a new entity and re-calculate similarity between this newly formed cluster and all other entities.

d. Calculate TurboMQ using (4).

Until

The required number of clusters is formed.

Figure 3.3. WCA algorithm for software modularization.

A disadvantage in case of WCA is that it requires a cutoff point to be selected, which means number of steps after which to stop the algorithm has to be decided in advance.

3.2.3.2 Particle Swarm Optimization (PSO)

PSO is a meta-heuristic search based technique proposed by Kennedy et al. [95]. It is based on the bird and fish flock movement behavior. In PSO, a set of randomly generated solutions (initial swarm) propagates in the design space towards the optimal solution over a number of iterations (moves) based on large amount of information about the design space that is assimilated and shared by all members of the swarm.

(a) Initial configuration
The basic PSO algorithm consists of three steps, namely, generating particles’ positions and velocities, velocity update, and finally, position update. Here, a particle refers to a point in the design space that changes its position from one move (iteration) to another based on velocity updates. Each individual entity is treated as a particle. Each particle’s position in the swarm is affected both by its own most optimist position so far and the position of the most optimist particle in the swarm.

First, the positions, $x_k^i$, and velocities, $v_k^i$, of the initial swarm of particles are randomly generated using upper and lower bounds on the design variables values, $x_{\text{min}}$ and $x_{\text{max}}$, as expressed in Equations 1 and 2. The positions and velocities are given in a vector format with the superscript and subscript denoting the $i^{th}$ particle at time $k$. In
Equations 3 and 4, \( rand \) is a uniformly distributed random variable that can take any value between 0 and 1. This initialization process allows the swarm particles to be randomly distributed across the design space.

\[
x_k^i = x_{min} + rand(x_{max} - x_{max})
\]  

(3)

\[
v_k^i = \frac{x_{min} + rand(x_{max} - x_{max})}{\Delta t}
\]  

(4)

PSO has many variants available. PSO with inertia weights is an improved version of PSO [96]. The velocity and position of each particle change according to (5) and (6)

\[
v_{t+1} = w * v_t + c_1 r_1(l_{best} - x_t) + c_2 r_2(g_{best} - x_t)
\]  

(5)

\[
x_{t+1} = x_t + v_{t+1}
\]  

(6)

\( v_{t+1} \) and \( x_{t+1} \) are particle’s velocity and position after \( t+1 \) iterations. \( r_1 \) and \( r_2 \) are uniform random numbers in the range [0,1]. \( g_{best} \) is the best global value in the current swarm and \( l_{best} \) is the best position of each particle over the iterations. During each iteration, firstly the velocity of particle is updated and then according to the newly calculated velocity, the position of particle is changed. This process continues till the desired value of fitness function is achieved.

Figure 3.5. Velocity and position updates in PSO.
Algorithm 2: PSO Algorithm

1. For each particle:
   a. Initialize particle position equal to random value.
   b. Initialize particle velocity to random value.
   c. Set lbest to 0.
   End.
2. Set gbest to 0.
3. Repeat
   a. Calculate fitness value using (4).
   b. If fitness value is better than lbest fitness value
      i. For each particle:
         Set lbest to current position of the particle.
      End.
   c. Set gbest to lbest of the cluster having maximum fitness value.
   d. For each particle:
      i. Calculate velocity using (2)
      ii. Calculate position using (3)
   End
   Until
   A maximum iterations criterion is not attained.

Figure 3.6. PSO algorithm for software modularization.

3.2.4 Fitness Function

The fitness function used in this work to assess the software clustering results is TurboMQ. The TurboMQ [1], measurement for an MDG partitioned into k clusters is calculated as:

\[
TurboMQ = \sum_{i}^{k} CF
\]

\[
CF_{i} = \begin{cases} 
0 & \mu_{i} = 0 \\
\frac{\mu_{i}}{\mu_{i} + \frac{1}{2} \sum_{j=1}^{k} (e_{i,j} + e_{j,i})} & \mu_{i} > 0 
\end{cases}
\]  \quad (7)

Where CF is Component Factor which is based on cohesion and coupling for the cluster [1]. \( \mu_{i} \) is the cohesion or intra-connectivity of a cluster, and \( e_{ij} \) is the inter-connectivity between cluster i and j.

3.3 Proposed Approach

This study aims to optimize WCA-UENM hierarchical clustering algorithm using PSO in order to achieve better software modularization. We used real valued PSO with inertia weights [35], an improved version of PSO.
Algorithm 2: WCA-PSO Algorithm

1. Associate each entity \( E_i \) with a feature vector \( f_i \) having components \( f_{i1}, f_{i2}, \ldots, f_{ip} \).
2. Calculate similarity between each pair of entities within the system using (2).
3. Repeat
   e. Cluster the two most similar entities.
   f. Associate a new feature vector with the newly formed cluster. Let \( E_i \) and \( E_j \) be the entities to be clustered, with feature vectors \( f_i \) and \( f_j \) respectively. Let the number of entities in \( E_i \) be denoted by \( n_i \) and the number of entities in \( E_j \) by \( n_j \).
   The feature vector \( f_{ij} \) is calculated as:
   \[
   f_{ij} = \frac{f_i + f_j}{(n_i + n_j)} = \frac{(f_{ik} + f_{jk})}{(n_i + n_j)} \quad \text{where } k=1, 2, \ldots, p.
   \]
   g. Treat the newly formed non-singleton cluster as a new entity and re-calculate similarity between this newly formed cluster and all other entities.
   h. Calculate TurboMQ using (4).
   Until
   The required number of clusters is formed or TurboMQ stops increasing.
4. For each particle:
   d. Initialize particle position equal to clustering result obtained from Step 3.
   e. Initialize particle velocity to 0.
   f. Set \( lbest \) to 0.
   End.
5. Set \( gbest \) to 0.
6. Repeat
   e. Calculate fitness value using (7).
   f. If fitness value is better than \( lbest \) fitness value
      ii. For each particle:
          Set \( lbest \) to current position of the particle.
      End.
   g. Set \( gbest \) to \( lbest \) of the cluster having maximum fitness value.
   h. For each particle:
      iii. Calculate velocity using (5)
      iv. Calculate position using (6)
      End
   Until
   A maximum iterations criterion is not attained.

Figure 3.7. WCA-PSO algorithm for software modularization.

3.4 Algorithmic Parameter

To cluster the most similar entities together, WCA, PSO and our proposed hybrid approach (WCA-PSO) were selected. Maximum number of iterations for WCA is \( n-1 \) (where \( n \) is the number of classes present in an entity) because after \( n-1 \) iterations WCA starts to merge all the clusters to form a single cluster. For this study, out of \( n-1 \) iterations for WCA, the iteration resulting in the highest TurboMQ is chosen.
Parameter values for PSO and WCA-PSO are shown in Table II. The output of WCA is used as an initial seed to the result optimizing part (PSO) of the proposed algorithm, which generates the final result.

Table 3.2. Parameter values for PSO and WCA-PSO

<table>
<thead>
<tr>
<th>Parameters</th>
<th>PSO</th>
<th>WCA-PSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>Equal to number of entities</td>
<td>Equal to number of entities</td>
</tr>
<tr>
<td>Iterations</td>
<td>100</td>
<td>WCA’s iteration + 100</td>
</tr>
<tr>
<td>Initial position</td>
<td>Random</td>
<td>Results obtained from WCA</td>
</tr>
<tr>
<td>Initial velocity</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Self-confidence (c1)</td>
<td>1 - 2</td>
<td>1 – 2</td>
</tr>
<tr>
<td>Swarm-confidence (c2)</td>
<td>1.5 - 2.5</td>
<td>1.5 - 2.5</td>
</tr>
<tr>
<td>Inertia weight(w)</td>
<td>0.4 - 0.9</td>
<td>0.4 - 0.9</td>
</tr>
</tbody>
</table>

3.5 Chapter Summary

This chapter provides a description of the applications selected for carrying out experimentation along with the explanation of algorithms used. It also focuses on algorithmic parameter and fitness function used for study.
CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents the experimental results of algorithms WCA, PSO, and our proposed approach (WCA-PSO). For each test system, total 30 trials of each algorithm are executed. The clustering providing the highest TurboMQ is chosen to be the best clustering solution for each execution and then an average is taken over all the executions. The decompositions obtained automatically using above mentioned clustering algorithms are evaluated using following criteria:

1. Number of isolated clusters
2. Cohesion
3. Coupling
4. TurboMQ

4.1 Results Analysis using Number of Isolated Clusters

Number of clusters produced by an algorithm during the clustering process is an important factor for software modularization. A good clustering algorithm should not group all the entities in one cluster and not produce too many singleton (cluster having only one class) clusters as well. Table 4.4 and Figure 4.5 shows number of isolated clusters formed by all the three algorithms under study. It is clear from Table 4.5 that WCA-PSO does not form any isolated clusters for the 3 out of 5 test systems i.e. Launch4J, MidiQuickFix and JHawk. But WCA forms isolated clusters for all test systems except JHawk. PSO also forms isolated clusters for 2 test systems i.e. Launch4J and MidiQuickFix.

4.2 Results Analysis using Cohesion

Cohesion is the number of intra-edges between classes in a single cluster. Cohesion should be high in a well modularized software system. In this work cohesion values is calculated as aggregation of the cohesion values taken over all the individual clusters. Table 4.5 and Figure 4.2 shows that WCA-PSO results in stronger cohesion in comparison to that resulted by WCA for all the test systems. WCA-PSO steadily increases the number of clusters formed (or at least keeps it constant in the worst case), except for Launch4J, which is a good sign for better software modularization. PSO also increases cohesion for all test systems except for Sellwin.
Table 4.1-4.4 shows the clustering results achieved by WCA and WCA-PSO for all the test systems. Clustering results are taken over a range of iterations for every test system.

Table 4.1. Comparison of software clustering solutions found by WCA and WCA-PSO for Graphviz

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iterations</th>
<th>Clusters</th>
<th>Isolated Clusters</th>
<th>Cohesion</th>
<th>Coupling</th>
<th>TurboMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCA</td>
<td>25</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>49</td>
<td>0.41</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>25+100</td>
<td>3</td>
<td>0</td>
<td>17</td>
<td>46</td>
<td><strong>0.58</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>26</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>49</td>
<td>0.41</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>26+100</td>
<td>3</td>
<td>1</td>
<td>17</td>
<td>46</td>
<td><strong>0.54</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>27</td>
<td>2</td>
<td>1</td>
<td>36</td>
<td>28</td>
<td>0.72</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>27+100</td>
<td>2</td>
<td>1</td>
<td>26</td>
<td>31</td>
<td>0.62</td>
</tr>
<tr>
<td>WCA</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>28+100</td>
<td>2</td>
<td>1</td>
<td>54</td>
<td>3</td>
<td><strong>0.97</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>29</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>29+100</td>
<td>2</td>
<td>1</td>
<td>54</td>
<td>3</td>
<td><strong>0.97</strong></td>
</tr>
</tbody>
</table>

Table 4.2. Comparison of software clustering solutions found by WCA and WCA-PSO for Launch4j

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iterations</th>
<th>Clusters</th>
<th>Isolated Clusters</th>
<th>Cohesion</th>
<th>Coupling</th>
<th>TurboMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCA</td>
<td>65</td>
<td>17</td>
<td>12</td>
<td>53</td>
<td>182</td>
<td>2.69</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>65+100</td>
<td>18</td>
<td>0</td>
<td>82</td>
<td>188</td>
<td><strong>3.02</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>71</td>
<td>11</td>
<td>12</td>
<td>83</td>
<td>152</td>
<td>2.63</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>71+100</td>
<td>11</td>
<td>3</td>
<td>129</td>
<td>142</td>
<td><strong>2.86</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>72</td>
<td>10</td>
<td>12</td>
<td>89</td>
<td>147</td>
<td>2.79</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>72+100</td>
<td>13</td>
<td>3</td>
<td>125</td>
<td>137</td>
<td><strong>3.29</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>76</td>
<td>6</td>
<td>12</td>
<td>120</td>
<td>120</td>
<td>2.37</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>76+100</td>
<td>6</td>
<td>1</td>
<td>135</td>
<td>123</td>
<td><strong>2.53</strong></td>
</tr>
<tr>
<td>WCA</td>
<td>77</td>
<td>5</td>
<td>12</td>
<td>120</td>
<td>113</td>
<td>2.08</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>77+100</td>
<td>5</td>
<td>0</td>
<td>158</td>
<td>113</td>
<td><strong>2.29</strong></td>
</tr>
</tbody>
</table>

Table 4.1 and Table 4.2 show that WCA-PSO gives improved results for Graphviz and Launch4j. In case of Graphviz, WCA sometimes outperforms WCA-PSO because for small sized test (classes<=50) systems it can cluster classes effectively. But for large sized test WCA-PSO always outperforms WCA such as Launch4j.
Table 4.3. Comparison of software clustering solutions found by WCA and WCA-PSO for MidiQuickFix

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iterations</th>
<th>Clusters</th>
<th>Isolated Clusters</th>
<th>Cohesion</th>
<th>Coupling</th>
<th>TurboMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCA</td>
<td>124</td>
<td>8</td>
<td>10</td>
<td>78</td>
<td>189</td>
<td>0.74</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>124+100</td>
<td>8</td>
<td>0</td>
<td>130</td>
<td>185</td>
<td>1.24</td>
</tr>
<tr>
<td>WCA</td>
<td>125</td>
<td>6</td>
<td>10</td>
<td>110</td>
<td>157</td>
<td>1.14</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>125+100</td>
<td>5</td>
<td>0</td>
<td>151</td>
<td>164</td>
<td>1.30</td>
</tr>
<tr>
<td>WCA</td>
<td>126</td>
<td>7</td>
<td>10</td>
<td>94</td>
<td>173</td>
<td>1.05</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>126+100</td>
<td>4</td>
<td>0</td>
<td>135</td>
<td>179</td>
<td>1.28</td>
</tr>
<tr>
<td>WCA</td>
<td>127</td>
<td>9</td>
<td>10</td>
<td>79</td>
<td>189</td>
<td>0.77</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>127+100</td>
<td>8</td>
<td>0</td>
<td>130</td>
<td>185</td>
<td>1.24</td>
</tr>
<tr>
<td>WCA</td>
<td>128</td>
<td>5</td>
<td>10</td>
<td>133</td>
<td>135</td>
<td>1.52</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>128+100</td>
<td>6</td>
<td>0</td>
<td>151</td>
<td>163</td>
<td>1.66</td>
</tr>
</tbody>
</table>

Table 4.4. Comparison of software clustering solutions found by WCA and WCA-PSO for SellWin

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iterations</th>
<th>Clusters</th>
<th>Isolated Clusters</th>
<th>Cohesion</th>
<th>Coupling</th>
<th>TurboMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCA</td>
<td>233</td>
<td>26</td>
<td>30</td>
<td>179</td>
<td>670</td>
<td>3.38</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>233+100</td>
<td>36</td>
<td>0</td>
<td>188</td>
<td>837</td>
<td>3.80</td>
</tr>
<tr>
<td>WCA</td>
<td>234</td>
<td>25</td>
<td>30</td>
<td>181</td>
<td>668</td>
<td>3.40</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>234+100</td>
<td>35</td>
<td>3</td>
<td>192</td>
<td>835</td>
<td>3.83</td>
</tr>
<tr>
<td>WCA</td>
<td>235</td>
<td>24</td>
<td>30</td>
<td>183</td>
<td>666</td>
<td>3.42</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>235+100</td>
<td>34</td>
<td>4</td>
<td>209</td>
<td>819</td>
<td>3.93</td>
</tr>
<tr>
<td>WCA</td>
<td>236</td>
<td>23</td>
<td>30</td>
<td>185</td>
<td>664</td>
<td>3.43</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>236+100</td>
<td>35</td>
<td>1</td>
<td>188</td>
<td>839</td>
<td>4.01</td>
</tr>
<tr>
<td>WCA</td>
<td>237</td>
<td>22</td>
<td>30</td>
<td>194</td>
<td>655</td>
<td>3.21</td>
</tr>
<tr>
<td>WCA-PSO</td>
<td>237+100</td>
<td>28</td>
<td>1</td>
<td>236</td>
<td>790</td>
<td>3.75</td>
</tr>
</tbody>
</table>

Table 4.3 and Table 4.4 show that WCA-PSO always outperforms WCA for MidiQuickFix and SellWin. In some cases WCA-PSO gives higher coupling in comparison to WCA but as long as TurboMQ is increasing, we can ignore those cases because increase in TurboMQ is a sign of good modularization.
Table 4.5. Comparison of software clustering solutions found by WCA, PSO, WCA-PSO

<table>
<thead>
<tr>
<th>Test Systems</th>
<th>Algorithm</th>
<th>Iterations</th>
<th>Clusters</th>
<th>Isolated Clusters</th>
<th>Cohesion</th>
<th>Coupling</th>
<th>TurboMQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphviz</td>
<td>WCA</td>
<td>26</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>49</td>
<td>0.410</td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>100</td>
<td>4</td>
<td>0</td>
<td>48</td>
<td>19</td>
<td>1.934</td>
</tr>
<tr>
<td></td>
<td>WCA-PSO</td>
<td>26+100</td>
<td>3</td>
<td>1</td>
<td>17</td>
<td>46</td>
<td>0.541</td>
</tr>
<tr>
<td>Launch4J</td>
<td>WCA</td>
<td>72</td>
<td>10</td>
<td>12</td>
<td>89</td>
<td>147</td>
<td>2.790</td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>100</td>
<td>12</td>
<td>2</td>
<td>156</td>
<td>118</td>
<td>1.385</td>
</tr>
<tr>
<td></td>
<td>WCA-PSO</td>
<td>72+100</td>
<td>8</td>
<td>0</td>
<td>104</td>
<td>165</td>
<td>3.060</td>
</tr>
<tr>
<td>MidiQuickFix</td>
<td>WCA</td>
<td>125</td>
<td>8</td>
<td>10</td>
<td>78</td>
<td>189</td>
<td>0.744</td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>100</td>
<td>10</td>
<td>2</td>
<td>138</td>
<td>179</td>
<td>1.217</td>
</tr>
<tr>
<td></td>
<td>WCA-PSO</td>
<td>125+100</td>
<td>8</td>
<td>0</td>
<td>130</td>
<td>185</td>
<td>1.241</td>
</tr>
<tr>
<td>JHawk</td>
<td>WCA</td>
<td>263</td>
<td>10</td>
<td>0</td>
<td>343</td>
<td>764</td>
<td>2.922</td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>100</td>
<td>11</td>
<td>0</td>
<td>249</td>
<td>990</td>
<td>1.572</td>
</tr>
<tr>
<td></td>
<td>WCA-PSO</td>
<td>263+100</td>
<td>14</td>
<td>0</td>
<td>391</td>
<td>843</td>
<td>3.168</td>
</tr>
<tr>
<td>Sellwin</td>
<td>WCA</td>
<td>237</td>
<td>22</td>
<td>30</td>
<td>194</td>
<td>655</td>
<td>3.215</td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>100</td>
<td>20</td>
<td>0</td>
<td>99</td>
<td>934</td>
<td>1.302</td>
</tr>
<tr>
<td></td>
<td>WCA-PSO</td>
<td>237+100</td>
<td>25</td>
<td>1</td>
<td>236</td>
<td>790</td>
<td>3.750</td>
</tr>
</tbody>
</table>

4.3 Results Analysis using Coupling

Coupling is the number of inter-edges between clusters of software system. Coupling should be low in a well modularized software system. In this work coupling values are calculated as aggregation of the coupling values of all the individual clusters. As shown in Table 4.5 and represented by Figure 4.3, WCA-PSO results in higher coupling as compared to WCA for 3 out of 5 test systems (i.e. Launch4J, JHawk, Sellwin). PSO also does the same for 2 out of 5 test systems (i.e. JHawk, Sellwin).

4.4 Results Analysis using TurboMQ

TurboMQ is a measure of cohesiveness of the system in terms of ratio between cohesion and coupling of the clusters formed. It is clear from Table 4.4 and Figure 4.5 that WCA-PSO gives highest TurboMQ values for all test systems except Graphviz.
Figure 4.1. Number of isolated clusters formed by algorithms during clustering process.

Figure 4.2. Cohesion of systems achieved by different algorithms during clustering process.

Figure 4.3. Coupling of systems achieved by different algorithms during clustering process.
Figure 4.4 TurboMQ value achieved by different algorithms during clustering process.

### 4.5 Result Summary

WCA groups classes based on similarity among them. But once clusters are formed, it does not change the position of classes and keeps on merging clusters (singleton/others). Towards the ending phase of algorithm, it combines all the clusters in a single cluster which is disastrous for software modularization.

But in case of WCA-PSO, PSO optimizes the results of WCA by changing classes’ positions according to global optima, and hence helps provide better overall clustering.

It is clear from the results that PSO alone also reduces number of clusters and increases cohesion than the WCA. But it reduces TurboMQ as well. So it answers our research question that when we optimize WCA using PSO, better software clustering is achieved. The results also provide evidence that rather than applying optimization techniques alone, hybrid approaches promise better results.
CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this chapter, we will look briefly at results from experiments in chapter 4 and draw some overall conclusions. We will also look at some of the shortcomings and possible ways to overcome them and improve the search.

5.1 Conclusions

This work presents a hybrid multi-objective technique for software module clustering. It proposes a PSO based optimization of hierarchical clustering algorithm WCA and compares the results with WCA and PSO. The comparison is based on four main objectives TurboMQ, isolated clusters, cohesion and coupling.

- WCA and PSO alone give poor clustering results.
- The results indicate that WCA-PSO outperforms WCA and PSO both in all the assessment criteria except coupling
- WCA-PSO forms a large number of clusters and keeps the number of isolated clusters minimal as well.
- The results indicate that hybrid approach produces better software clustering than individual algorithms.

The primary threat to any research in software engineering is related to generalizing the research results to a wider range of software systems. Hence a threat to our study is related to number and type of test systems. We have considered five test systems for our study. However, to make our results more affordable, we have used open source Java software systems. Our test systems are chosen from a variety of domains to make the empirical study more applicable. Using limited features (only static dependencies) for the proposed approach could be considered as another threat, however as the results are satisfactory with these limited features, the utilization of more features could be deferred.

In case of optimization technique, selecting parameters’ values is very crucial and important for producing better results. To mitigate this challenge, we have experimented with each possible value of parameters used in PSO and selected the best ones.
5.2 Future Work

In this work, we have provided evidence that optimization technique improves the software clustering results of hierarchical algorithm. Future work that can be done further in this direction is as follows:

- Additional features of entities like inheritance, global variables, and user defined types etc. could be consider.
- It should also consider effect of other optimization techniques such as Ant Colony Optimization (ACO); Artificial Bee Colony (ABC) optimization etc. for improving hierarchical clustering based software modularization.
- Another future direction is to consider the application of PSO technique to cooperative based hierarchical clustering for software modularization.
REFERENCES


