CONSTRAINED CLUSTERING APPROACH TO AID IN REMODULARISATION OF OBJECT-ORIENTED SOFTWARE SYSTEMS

CHONG CHUN YONG

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2016

CONSTRAINED CLUSTERING APPROACH TO AID IN REMODULARISATION OF OBJECT-ORIENTED SOFTWARE SYSTEMS

CHONG CHUN YONG

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2016

UNIVERSITY OF MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Chong Chun Yong

Registration/Matric No: WHA130005

Name of Degree: Doctor of Philosophy

Title of Project Paper/Research Report/Dissertation/Thesis:

Constrained Clustering Approach to aid in Remodularisation of Object-oriented Software Systems

Field of Study: Software Engineering

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date:

Subscribed and solemnly declared before,

Witness's Signature Date:

Name:

Designation:

ABSTRACT

Effective execution of software maintenance requires knowledge of the detailed working of software. The structure of a software, however, may not be clear to software maintainers because it is poorly designed or, worse, there is no updated software documentation. To effectively address this issue, researchers have proposed to apply software clustering to help in recovering a high-level semantic representation of the software design by grouping sets of collaborating software components into meaningful subsystems. This high-level semantic representation serves to help bridge the dichotomy between the perceived software design from the maintainers' view and the actual code structure. However, software clustering is typically conducted in an unsupervised and rigid manner, where maintainers have no influence on the clustering results and only a single solution is produced for any given dataset. Even if maintainers possess additional information that could be useful to guide and improve the clustering results, traditional clustering algorithms have no way to take advantage of this information. These practical concerns have led the researcher to propose the idea of integrating domain knowledge into traditional unsupervised clustering algorithms, herewith referred as constrained clustering, a semi-supervised clustering technique where domain experts can explicitly exert their opinions in the form of explicit clustering constraints to restrict whether a pair of software components should or should not be clustered into the same subsystem. Apart from the explicit clustering constraints from domain experts, other sources of information to guide and improve clustering results can be derived implicitly from the source code itself. To help maintainers effectively identify and interpret the implicit information hidden in the source code, this study proposes representing software using weighted complex network in conjunction with graph theory to help in understanding and analysing the structure, behaviour, as well as the complexity of the software components and their

relationships from the graph theory's point of view. The results of the analysis can be subsequently converted into implicit clustering constraints. Hence, maintainers can make use of both the explicit and implicit constraints to help in creating a high-level semantic representation of the software design that is coherent and consistent with the actual code structure.

This thesis proposes a constrained clustering approach to aid in remodularisation of poorly designed or poorly documented object-oriented software systems. The source code of an object-oriented software system is first converted into UML class diagrams. Next, information from the class diagrams are extracted to measure the strength of cohesion among related classes together with their relationships, and then transform them into a weighted complex network with its nodes and edges associated with measured weights. Graph theory metrics are subsequently applied onto the constructed weighted complex network so that the structure, behaviour, and the complexity of software components and their relationships can be analysed. The results are then converted into sets of clustering constraints. Guided by the explicit and implicit clustering constraints, sets of cohesive clusters are progressively derived to act as a high-level semantic representation of the software design.

This research follows an empirical research methodology, where the proposed approach is validated using 40 object-oriented open-source software systems written in Java. Using MoJoFM, which is a well-established technique used to compare the similarity between multiple clustering results, the proposed approach achieves an aggregated average of 80.33% accuracy when compared against the original package diagrams of the 40 software systems, thus considerably outperforms conventional unconstrained clustering approach. The clustering results serve as supplementary information for software

maintainers to aid in making critical decisions for re-engineering, maintaining and evolving software systems. Ultimately, this research helps in reducing the cost of software maintenance through better comprehension of the recovered software design.

ABSTRAK

Penyelenggaraan perisian yang berkesan memerlukan pengetahuan tentang operasi perisian tersebut. Bagaimanapun, struktur perisian mungkin tidak jelas kepada penyelenggara perisian kerana perisian tersebut direka dengan buruk, atau lebih teruk lagi, tidak ada dokumentasi yang dikemaskini. Bagi menangani isu ini dengan berkesan, penyelidik telah mencadangkan untuk melaksanakan pengkelompokan perisian untuk membantu dalam memulihkan perwakilan semantik peringkat tinggi secara rekabentuk perisian dengan mengumpulkan komponen-komponen perisian yang bekerjasama ke dalam subsistem yang bermakna. Perwakilan semantik peringkat tinggi ini berfungsi untuk merapatkan dikotomi antara reka bentuk perisian yang dilihat dari pandangan penyelenggara dan struktur kod yang sebenarnya. Walau bagaimanapun, pengkelompokan perisian biasanya dijalankan secara tidak terselia dan tegar, di mana penyelenggara tidak mempunyai pengaruh ke atas keputusan kelompok dan hanya satu penyelesaian yang dihasilkan untuk sebarang set data yang diberikan. Walaupun penyelenggara mempunyai maklumat tambahan yang boleh membantu dan meningkatkan keputusan pengelompokan, algoritma pengelompokan tradisional tidak mempunyai cara untuk mengambil kesempatan daripada maklumat tersebut. Kebimbangan yang praktikal ini telah mendorong penyelidik kepada idea untuk menyepadukan pengetahuan domain ke dalam algoritma pengelompokan tradisional tanpa pengawasan, bersama-sama ini dirujuk sebagai pengkelompokan secara kekangan, teknik pengelompokan separuh selia dimana pakar-pakar domain boleh memberi pendapat mereka dalam bentuk kekangan kelompok untuk menyekat sama ada sepasang komponen perisian perlu atau tidak dikelompokkan ke dalam subsistem yang sama. Selain daripada kekangan pengelompokan yang jelas daripada pakar-pakar domain, sumber maklumat lain untuk membimbing dan meningkatkan hasil pengelompokan boleh diperolehi secara tersirat

dari kod sumber perisian itu sendiri. Untuk membantu penyelenggara mengenal pasti dan mentafsir maklumat yang tersirat tersembunyi dalam kod sumber secara berkesan, kajian ini mencadangkan untuk mewakili perisian menggunakan rangkaian kompleks berwajaran sempena dengan teori graf untuk membantu dalam memahami dan menganalisis struktur, kelakuan, dan juga kerumitan komponen perisian dan hubungan mereka dari sudut pandangan teori graf. Keputusan analisis boleh kemudiannya ditukar menjadi kekangan pengelompokan tersirat. Oleh itu, penyelenggara boleh menggunakan kedua-dua kekangan tersurat dan tersirat untuk membantu dalam mewujudkan perwakilan rekabentuk perisian semantik berperingkat tinggi yang koheren dan konsisten dengan struktur kod yang sebenarnya.

Tesis ini mencadangkan satu kaedah pengelompokan kekangan untuk membantu dalam remodularisasi sistem perisian berorientasikan objek yang direka secara buruk atau tidak didokumenkan. Pada mulanya, kod sumber sistem perisian berorientasikan objek ditukar kepada gambar rajah kelas UML. Seterusnya, maklumat daripada gambar rajah kelas diekstrak untuk mengukur kekuatan perpaduan di kalangan kelas yang berkaitan, dan kemudian diubahkan kepada rangkaian kompleks berwajaran dengan nod dan tepi diberatkan dengan berat yang sesuai. Metrik teori graf kemudiannya digunakan ke rangkaian kompleks wajaran yang dibina supaya struktur, tingkah laku, dan kerumitan komponen perisian dan hubungan mereka boleh dianalisis. Keputusan ini kemudiannya ditukar kepada set kekangan pengelompokan. Berpandukan kepada kekangan pengelompokan yang tersurat dan tersirat, set kluster yang padu diperoleh secara progresif untuk bertindak sebagai perwakilan semantik peringkat tinggi reka bentuk perisian.

Kajian ini mengikut kaedah penyelidikan empirikal, di mana kaedah yang dicadangkan itu disahkan menggunakan 40 sistem perisian sumber terbuka berasaskan objek ditulis dengan Java. Menggunakan MoJoFM, teknik yang mantap digunakan untuk membandingkan persamaan antara keputusan pengelompokan berbilang, kaedah yang dicadangkan mencapai purata agregat ketepatan 80.33% apabila dibandingkan dengan gambar rajah pakej asal 40 sistem perisian, dan mencapai jauh lebih baik daripada kaedah pengelompkan perisian konvensional tanpa kawalan. Keputusan pengelompokan berfungsi sebagai maklumat tambahan bagi penyelenggara perisian untuk membantu dalam membuat keputusan penting tentang kejuruteraan semula, pengekalan sistem, dan perubahan sistem. Akhirnya, kajian ini boleh membantu dalam mengurangkan kos penyelenggaraan perisian, melalui pemahaman reka bentuk perisian yang lebih baik.

ACKNOWLEDGEMENT

I would like to thank my supervisor, Prof. Dr. Lee Sai Peck for the guidance throughout my candidature. The continuous encouragement and advice from her have helped me tremendously to complete this research. She has always been patience in reading and diligently edited many of my drafts throughout this research. She has also always provided me many constructive ideas and feedback to improve the quality of my work. I greatly appreciate the time she has spent for discussion sessions. All these feedbacks help shape and make my research better.

Also, I would like to express my gratitude to everyone in Software Requirements, Architecture and Reusabilty Engineering Lab and Network Research Lab for sharing their research experiences with me. The in-depth knowledge shared by them has provided me with invaluable insight on the methods used for conducting research.

To my beloved family, wife, and daughter, I thank them for the support and motivation given. Without them, the research conducted would be a difficult and lonely journey.

Lastly, I would like to express my gratitude to those I had forgotten to mention in this section. Thanks for the assistance provided in my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AHP – Analytic Hierarchy Process

CBO - Coupling Between Object Classes

CL – Cannot-link

CLH – Cannot-link Hard

CLINK - Complete Linkage Algorithm

DIT - Depth of Inheritance Tree

GQM – Goal Question Metric

LCOM - Lack of Cohesion of Methods

MDA – Model Driven Architecture

ML – Must-link

MLH – Must-link Hard

MOOD - Metrics for Object Oriented Design

NOC - Number of Children

OO – Object-oriented

RC – Resemblance coefficient

RFC - Response for a Class

RO – Research Objective

RQ – Research Question

SLINK - Single Linkage Algorithm

SQALE – Software Quality Assessment Based on Life-Cycle Expectations

TFN – Triangular Fuzzy Number

UML – Unified Modelling Language

UPGMA - Un-weighted Pair-Group Method using Arithmetic Average

WMC - Weighted Methods per Class

CHAPTER 1: INTRODUCTION

Software requires continuous change and enhancement to satisfy new business rules and technologies. Software maintenance is a human-intensive task that requires deep understanding and comprehension of a software before any decision to modify it. Therefore, software maintainers must first gain a certain level of understanding on the structure and behaviour of the software before making any major changes.

However, if the software is poorly designed or poorly documented, the source code may be the only resource left for recovering the system's design. Without a proper mechanism to recover a high-level software design, software maintainers are often forced to make ad-hoc modifications to the source code when there is a request, without understanding the structure and behaviour of the software in advance. As such, continually adopting an ad-hoc approach to software maintenance will have a negative effect on the overall modularity of the system. Over time, the structure and modularity of the software may deteriorate to the point where it is so disorganised that the system needs to be drastically overhauled or abandoned completely (Mitchell & Mancoridis, 2001). Therefore, means other than relying on documentation to recover a high-level abstraction of the software design is needed in order to improve the modularity of software systems.

The work by (G. Canfora, Cimitile, De Lucia, & Di Lucca, 2001; Gerardo Canfora, Czeranski, & Koschke, 2000; Tonella, 2001) proposed approaches for the identification of objects or Abstract Data Types in legacy software systems to help in remodularisation of legacy software systems. Such approaches generally identify objects or Abstract Data Types in legacy source code by discovering the relationships between routines, global variables, and user-defined data types. However, the applicability of the proposed

approaches is confined to legacy software systems written in structured programming languages. Apart from that, search-based approaches are also used in several other works to aid in software remodularisation (Praditwong, Harman, & Yao, 2011; Harman, Mansouri, & Zhang, 2012). In particular, the work by Harman et al. (2012) proposed a single-objective genetic algorithm to improve the subsystem decomposition of software systems, where the fitness function is defined using a combination of software quality metrics. However, the complexity of search-based approaches is generally higher and researchers often face the NP-Complete problem when searching for the optimum solutions.

Existing studies have found that clustering analysis can help in remodularisation of poorly designed or poorly documented software systems by grouping sets of collaborating software components into meaningful subsystems to recover a high-level semantic representation of the software design. Clustering generally is based on discrete description of clustering entities (such as methods, classes, packages, etc) and is designed to cope with very complex relationships between entities. Hence, cluster analysis is arguably suitable for software remodularisation which involves taking a group of classes with complex relationships and merging them together in logically coherent groups or subsystems. Besides that, cluster analysis techniques can incorporate any number of characteristics about a piece of software, providing the relevant information about a characteristic that can be extracted from the code. The recovered high-level representation of the software design helps in narrowing down the discrepancy between the perceived software design from the maintainers' view and the actual code structure.

1.1 Motivation

Maintenance of existing software requires plenty of time in analysing and comprehending the available source code and software documentations. Successful accomplishment of software maintenance is highly dependent on how much information can be extracted by software maintainers. The time and effort spent in software maintenance could potentially be reduced through software clustering to recover a semantic representation of the software design, thus aiding in better comprehension of the structure and behaviour of the software (Maqbool & Babri, 2006; Maqbool & Babri, 2007). Software clustering has received a substantial attention in recent years because of its capability to help in improving the modularity of poorly designed or poorly documented software systems. However, software clustering is typically conducted in an unsupervised manner where software maintainers have no influence on the end results because the effectiveness of software clustering depends greatly on the algorithm used. Furthermore, unsupervised software clustering is rigid in such a way that only a single clustering result is produced for any given dataset. In the case if software maintainers do not agree with the outcome, they will need to repeat the process again using a different set of configuration and clustering algorithm.

Hence, an improvement to conventional clustering approaches was proposed in the work by Basu, Banerjee, and Mooney (2004), where the authors proposed a semi-supervised clustering technique by incorporating side information to further improve the accuracy of clustering results. The side information is commonly referred as "*clustering constraints*" which reveal the similarity between pairs of clustering entities, or user preferences about how those entities should be grouped during clustering. The clustering constraints may impose certain restrictions such as forcing a pair of clustering entities to always group

into the same cluster, or separated into disjoint clusters. These constraints are commonly referred as must-link (ML) and cannot-link (CL) constraints respectively. This type of semi-supervised clustering technique is commonly referred as constrained clustering where users have a certain degree of influence to alter the final clustering results based on the domain knowledge. It has been proven in several fields of research that constrained clustering can significantly improve the reliability and accuracy of clustering results (Davidson & Ravi, 2009). However, there is still a lack of studies on integrating constrained clustering to effectively improve the modularity of poorly designed object-oriented (OO) software systems.

In the domain of software, it is highly possible that software maintainers may have access to additional information about the software to be maintained, either explicitly or implicitly. For instance, domain experts or software developers who are involved in the early stages of software design or developments are able to provide feedbacks to indicate whether a pair of software components should be clustered into the same functional group. This type of information, which is based on the explicit opinions and feedbacks from the domain experts, are referred as explicit clustering constraints. On the other hand, implicit information refers to some extra deterministic information about the interrelationships between software components derived from the source code itself. However, software maintainers will require tool support to effectively identify and interpret the implicit information hidden in the source code because the quantity and level of granularity of the information might be too overwhelming to comprehend. For instance, representing software using weighted complex network in combination with graph theory is able to help in identifying important classes that are responsible for providing services to other classes, from the graph theory's point of view. The results of the graph theoretical analysis can then be translated into implicit clustering constraints which can help in guiding and

improving the clustering results. Thus, even if the software documentation is out of date, maintainers are still able to salvage such useful information about the implicit structure of the software. However, such information are worthless unless there is a proper way to synthesis them.

This research focuses on constrained clustering to fully exploit the clustering constraints given by the domain expert (referred as explicit constraints), or other forms of side information (referred as implicit constraints) to help create a high-level abstraction of the software system as perceived by the expert, with the purpose to bridge the dichotomy between the perceived software design from the maintainer's view and the actual code structure.

The proposed constrained clustering approach can help to improve software maintenance capability of a particular organisation, by providing a high-level semantic view of the software design. The recovered software design can aid in facilitating software maintenance work, such as when there is a request to update or remove a particular software component, maintainers can easily identify other components that are interrelated to the target component in order to avoid any unintentional service interruption.

1.2 Problem Statement

Most of the existing studies use source code as the only input parameter to perform software clustering (Anquetil & Lethbridge, 2003; Cui & Chae, 2011; Maqbool & Babri, 2007; Praditwong, Harman, & Yao, 2011; Wu, 2005). These approaches perform clustering by analysing the dependencies in the source code such as passing of messages

between methods, shared variables and shared data. However, as software becomes more and more complex, inspecting source code can be tedious (Fokaefs, Tsantalis, Chatzigeorgiou, & Sander, 2009). Besides, existing works that use source code as the sole input are often language and platform dependent due to the different style and naming convention practised by each programming language.

Besides that, deriving explicit and implicit information of the software, and subsequently translate them into clustering constraints, is a challenging task. For instance, deriving explicit constraints from domain experts or software developers who have prior knowledge regarding the software can be a very time consuming and human-intensive task because they will need to review a high-level abstraction view of the software design and all its relevant documentations, before providing the necessary information to the software maintainers. Besides that, it is possible that there are conflicting opinions among the domain experts. Thus, a way to reach a consensus among all the domain experts is needed to ensure that all the provided explicit constraints can contribute toward forming coherent clustering constraints.

Implicit constraints, on the other hand, are constraints derived from the implicit structure of the software systems without the involvement of domain experts. In order to effectively derive implicit information from the source code, in-depth understanding on the structure and behaviour of software systems is highly needed. Representing software systems using weighted complex network in combination with graph theory, for instance, is one way to help in studying and analysing the structure, behaviour, as well as the complexity of the software components and their relationships from the graph theory's point of view. Although the derived implicit information can reveal some extra deterministic information about the relationships between software components, there is a lack of

studies that addressed the problem of translating the derived information into clustering constraints. Furthermore, in representing software systems using complex network from the literature, less attention is given to measure the weights of edges, which represent the strength of inter-relationships between related software components.

Even more so, fulfilment of implicit and explicit clustering constraints remains a significant research problem. Existing studies in software clustering tend to focus on imposing only absolute constraints, i.e. constraints that must be fulfilled regardless of any situation (Davidson & Ravi, 2009). However, in the domain of software engineering, it is possible that the constraints given by domain experts are fuzzy and ambiguous in nature. For instance, domain experts or software developers who were involved in the early stage of software design might provide some constraints about the software to be maintained. However, such constraints might not be valid anymore after several phases of software updates and changes. Thus, the constraints given by the aforementioned experts or developers might be ambiguous or contain erroneous information. An effective method is needed to distinguish between absolute constraints and optional constraints, and subsequently fulfil those constraints according to their level of importance.

Evidence based on the existing studies suggested that clustering constraints are not readily available most of the time either due to out-dated software documentations or limited background knowledge on the software to be maintained (Harman, Mansouri, & Zhang, 2012). Without reliable references, software maintainers can only opt for the traditional way of relying on raw source code to manually recover a high-level abstraction of the software design, which is an inefficient use of time and resources. Wasting software maintainers' man-hours would make companies less competitive, which in turn, incurring unnecessary cost.

In summary, it is widely acknowledged in existing studies that a well-modularised software system is easier to develop and maintain (Praditwong, Harman, & Yao, 2011). However, as a software evolves with the introduction of new business and technology requirements, modularity of the software tends to degrade, which imposes demands for restructuring the software. This research focuses on using constrained clustering as one of the techniques to recover a high-level abstraction of OO software design to aid in remodularisation of software systems. The recovered high-level software design in the form of highly cohesive clusters can help in improving the modularity of software by bridging the discrepancy between the perceived software design from the maintainers' view and the actual code structure. The proposed constrained clustering approach aims to tackle several issues that have not been addressed in the existing studies: Most of the existing studies only focus on a specific programming language. How to derive explicit and implicit clustering constraints from poorly designed or poorly documented OO software systems written in any OO programming language, and utilise these constraints to aid in remodularisation of software systems through constrained clustering? Besides that, how can weighted complex network aid in analysing the structure, behaviour, as well as the complexity of software components and their relationships? If domain experts are available to provide explicit clustering constraints based on their domain knowledge, how to handle fuzzy and ambiguous constraints that might contradict with each other? Several methods and techniques are introduced in this research in order to address all the above issues.

1.3 Objectives of the Research

The objectives to be achieved by the research are as follows:

Objective 1: To propose a constrained clustering approach with the aim to recover a high-level abstraction of OO software design that is coherent and consistent with the actual code structure.

Objective 2: To propose a method that helps in deriving implicit clustering constraints from the implicit structure of OO software systems with the aid of weighted complex network and graph theoretical analysis.

Objective 3: To propose a method that is capable of deriving explicit clustering constraints from domain experts or software developers who have prior knowledge regarding the software systems.

Objective 4: To formulate an appropriate objective function that maximises the fulfilment of explicit and implicit constraints, while penalising violation of the constraints.

Objective 5: To evaluate the accuracy and scalability of the proposed approach using open-source OO software systems.

1.4 Significance of Research

Many of the software systems still remain in use after many years of commissioning. Although the maintenance of legacy systems are costly in terms of man-hour and monetary values, most organisations are not willing to substitute their legacy systems because they would need to bear high risk if the systems is of a business critical type of system. Most of the time, these aging software systems do not have up-to-date software

documentations. As a result, the structure of a software system inevitably drifts away from its original design and becomes more complex as well as harder to maintain. As discussed by Canfora, Di Penta, and Cerulo (2011), a major fraction of software life cycle's expenditure is contributed by software maintenance and support. The authors estimated that over 50% of software development budget is spent on maintaining and supporting the software itself. Hence, improving the software maintenance capability of an organisation can directly contribute toward minimising the cost of maintenance in the long run.

This research aims to use constrained clustering to aid in analysing the inherent structure, behaviour, and complexity of poorly designed or poorly documented OO software systems. Clustering constraints are derived from two different sources of information, i.e. from the domain experts who possess prior knowledge regarding the software, or from the implicit structure of the software, to aid in creating a high-level abstraction view of the software design. The interrelationships and dependencies among classes can be revealed based on the recovered software design. The recovered software design is represented as disjoint sets of clusters, such that classes that contribute toward a similar software functionality are grouped into the same cluster, while those that are dissimilar are separated to promote the notion of high intra-cluster cohesion and low inter-cluster coupling.

The recovered high-level software design can act as supplementary information for software maintainers to aid in decision making when there is a request to modify or remove a particular software component. If the classes to be modified or removed are known to be highly complex and provide plenty of services to other classes, there is a high chance that any modification on these problematic classes will cause a huge chain-

of-reaction that might be destructive to the system. Thus, with the aid of the recovered software design, software maintainers can easily comprehend the source of the problem and isolate the problematic classes during software maintenance to prevent any unintentional interruption of software services. Eventually, this research can not only help in minimising the cost of software maintenance, but also ensure that the maintained software can adapt to future requirement changes.

1.5 Outline of Thesis

This thesis consists of eight chapters. Chapter 1 describes the motivation and objectives of this research. Chapter 2 reviews the literature on software remodularisation, software clustering, and methods to represent software systems using complex network. This chapter classifies and analyses the mechanisms used in existing works. The outcome of the review is provided to highlight the shortcomings found in the literature. In Chapter 3, the research methodology used in this thesis is discussed in detail. In Chapter 4, a method to represent software systems using weighted complex networks is proposed. The method is based on a unique weighting mechanism to weight the edges of software-based complex network in order to measure the complexity of software components and their relationships. Based on the constructed weighted complex network, a way to analyse the structure and behaviour of the software is described. The results of the analysis are converted into implicit clustering constraints in the form of must-link and cannot-link constraints to aid in the subsequent clustering process. Chapter 5 presents a method to maximise the fulfilment of clustering constraints. Next, Chapter 6 discusses the experimental setup of this research. The goal of the research, selection of subjects, context, and variables, are discussed in detail. Two research hypotheses are declared with the purpose to validate between the speculated observation and the results of the proposed

constrained clustering approach. Experiments are carried out using real datasets gathered from 40 open-source OO projects. This is followed by the discussion and analysis of the experimental results in Chapter 7. Several descriptive statistics and plotting techniques are used to analyse the experimental results. The conclusion is presented in Chapter 8. A summary on the research work accomplished is provided. Then, the contribution of this thesis is highlighted. The chapter concludes with a future research direction of the proposed approach.

CHAPTER 2: LITERATURE REVIEW

This chapter provides the background information and related works. It starts by discussing the relationships between software maintenance and software remodularisation. Then, the literature in software clustering, constrained clustering, graph theory metrics, and complex networks are presented. Finally, the issues and challenges are highlighted based on the discussed literatures.

2.1 Software Maintenance and Remodularisation

Software maintenance is vital to discover and validate the relationships between technology and business models for existing operational software systems. Software maintenance efforts are strongly correlated to the efficacy of the software design. The work by Kemerer (1995) shows that software systems that exhibit high modularity, i.e. low coupling and high cohesion, and adhere to common design practices such as modular architecture are relatively less complex and easier to maintain. Changes and modifications of source code are less destructive on the rest of the system because of the low inter-module coupling advocate in modular architecture.

Modular design can be realised in forward engineering through proper planning in the early phase of software development. However, as software systems evolve with the introduction of new business and technology requirements, their structures inevitably get more complex and maintainers will find harder to maintain them (Santos, Valente, & Anquetil, 2014). Therefore, in order to improve the maintainability of existing operational software systems that undergo frequent changes, maintainers have to remodularise them, which includes reverse engineer the software into relatively independent subsystems.

The work by Santos et al. (2014) defines remodularisation as "*a major restructuring in the system's architecture, with the central goal of improving its internal quality and therefore without adding new features or fixing bugs"*. Software systems that undergo remodularisation are divided into smaller and manageable subsystems. Similar components of software system that collaborate with each other are grouped together to form a union of subsystems, while relationships between the subsystems are established. The mapping of interrelationships between software components provides a means for maintainers to easily comprehend the structure and complexity of software systems. In most of the existing studies, remodularisation is guided by understanding the structural aspects of a software, i.e. the interactions and dependencies between software components (Anquetil & Laval, 2011; Passos, Terra, Valente, Diniz, & Mendonca, 2010). Existing studies had explored the usage of software clustering as a technique to aid in remodularisation of aging software systems.

2.2 Software Clustering

Clustering can be based on either a supervised or unsupervised approach to pick from a collection of entities, then form multiple groups of entities such that entities within the same group are similar to each other, while dissimilar from entities in other groups. In the context of software clustering, entities are normally source code or classes. Similarity measures are normally common global variables used in source code or method calls made by classes. The identification of similarity is often depending on the availability of reliable information.

Generally, clustering can be categorised into partitional and hierarchical clustering. Given a collection of data, partitional clustering works by defining an initial *k* number of cluster

centroids, and assigning each entity to the nearest centroid to form *k* disjoint clusters. On the other hand, hierarchical clustering iteratively merge smaller clusters into larger ones or divide large clusters into smaller ones, depending on either it is a bottom-up or top-down approach. Merging or dividing operations are usually dependent on the clustering algorithm used in the existing studies.

The results of partitional clustering are usually presented in several disjoint clusters, with each cluster contains at least one entity and each entity belongs to only one cluster. Meanwhile, the results of hierarchical clustering are usually presented in a tree diagram, called dendrogram. A dendrogram shows taxonomic relationships of clusters produced by hierarchical clustering. Cutting the dendrogram at a certain height produces a set of disjoint clusters.

In the domain of software clustering, partitional clustering is less viable because it is almost impossible to know the initial number of clusters before performing software clustering (Chong, Lee, & Ling, 2013). According to the work by Wiggerts (1997), the working principle of agglomerative clustering (bottom-up hierarchical clustering) is actually similar to reverse engineering where the abstractions of software designs are recovered in a bottom-up manner.

Divisive clustering, on the other hand, is based on a top-down hierarchical clustering approach where the clustering process starts at the top with all data in one big cluster. The cluster is then split into smaller clusters in a recursive manner until all data resides into a single cluster. Although the complexity of divisive clustering is lower than agglomerative clustering, the complete information about the global distribution of the data is needed when making the top-level clustering decisions (Dhillon, Mallela, & Kumar, 2003). Most

of the time, software maintainers are not involved in the earlier software design phases. If the software documentations are not up-to-date, it is hard for maintainers to identify the ideal number of software packages (or the number of clusters in the context of software clustering) before any attempts to remodularise any software systems.

Therefore, the focus of this thesis is to utilise agglomerative hierarchical clustering as a remodularisation technique to help improve the modularity of poorly designed or poorly documented software systems. The next section discusses some background knowledge on the general workflow of agglomerative hierarchical clustering.

2.3 Agglomerative Hierarchical Software Clustering

Agglomerative hierarchical clustering starts by forming all entities as initial clusters. At each step, a pair of entities is merged and the algorithm ends with one big cluster. The process of agglomerative hierarchical clustering can be summarised as the following steps.

1. Identification of entities or components
2. Identification of features
3. Calculation of similarity measure
4. Application of clustering algorithm
5. Evaluation of clustering results

2.3.1 Identification of entities or components

In software clustering, the typical choices of entities are in the form of source code because they represent the basic components and functionalities of a software system. Chosen candidate entities need to be labelled or tagged properly in order to understand their purpose. The labels will also assist in evaluating the performance of output at the end of software clustering (Maqbool & Babri, 2006).

2.3.2 Identification of features

The similarities between entities are determined based on their characteristics or features extracted from the available information. An entity may possibly have many features. Various properties of an entity can be described by different features. Though the selected features must contribute to the understanding of problem domain. Features are used to analyse how closely two entities are related based on the fact that the entities are more similar if they share many common features (Lung, Zaman, & Nandi, 2004). In software clustering, there are typically two types of features identification methods: formal (global variable access, passing of messages, and shared data) and non-formal (programmer's comment) methods to identify how close a software component is to another (Maqbool & Babri, 2007).

2.3.3 Calculation of similarity measure

The next step is to ascertain the similarity between entities by referring to the features identified in the previous step. Typical ways to calculate similarity are distance measures or resemblance coefficients. Euclidean distance, for example, is the most common type of distance measure (Danielsson, 1980). It calculates the geometric distance of two entities in the given multidimensional space. Euclidean distance is suitable to use on

scenarios where the similarities between entities are quantifiable (represented in numerical values). On the other hand, resemblance coefficients are calculated based on the common attributes that two entities share. Table 2.1 shows several examples of RC_{xy} formula: resemblance coefficients between entities *x* and *y*.

Table 2.1 List of commonly used resemblance coefficients (RC)

Similarity Measure	Formula
Jaccard coefficient (Jain & Dubes, 1988)	$\frac{a}{(a+b+c)}$
Sorensen-Dice coefficient (Sørensen, 1948)	$\frac{2a}{(2a+b+c)}$
Simple Matching coefficient (Warrens, 2009)	$\frac{a+d}{(a+b+c+d)}$
Gower-Legendre coefficient (Warrens, 2009)	$\frac{(a+d)}{(a+1/2(b+c)+d}$

Variable *a* represents the number of features that have "1-1" relationship between two entities, *d* represents the number of "0-0" relationship between two entities, while *b* and *c* represent "1-0" and "0-1" relationships between two entities respectively. A "1-1" relationship indicates that both entities are correlated to each other. Correlation in this context can be referred as the existence of certain common attributes or behaviour in both entities. Meanwhile, "1-0" and "0-1" relationships represent an indirect correlation. A "0-0" relationship represents that both entities do not share any form of similarity at all. Generally, a higher coefficient value indicates higher similarity between pairs of entities. Finally, the results are stored in a pairwise matrix called the resemblance matrix which denotes the similarity or dissimilarity strength between pairs of clustering entities.

2.3.4 Application of clustering algorithm

The next step is to group similar entities based on the resemblance matrix generated from the previous step in Section 2.3.3. Generally, clustering is a sequence of operations that iteratively groups similar entities into clusters. The iteration begins with each entity in a separate cluster, which means that *n* number of candidate entities will result in *n* number of clusters. At each iteration, the two clusters that are most similar to each other are merged and the number of clusters is reduced by one. This process will continue until there is only one cluster left. At the end of the iteration, a tree-like diagram called dendrogram is formed.

A clustering algorithm is needed to decide upon how and when to merge two clusters. Depending on the algorithm used, certain algorithms merge the most similar pair first while others merge the most dissimilar first. Once the two chosen clusters have been merged, the strength of similarity or dissimilarity between the newly formed cluster and the rest of the clusters are updated to reflect the changes. It is very common that during hierarchal clustering, there exist more than two entities which are equally similar or dissimilar. In this kind of scenario, the selection of candidate entities to be clustered is arbitrary (Maqbool & Babri, 2007).

2.3.5 Evaluation of clustering result

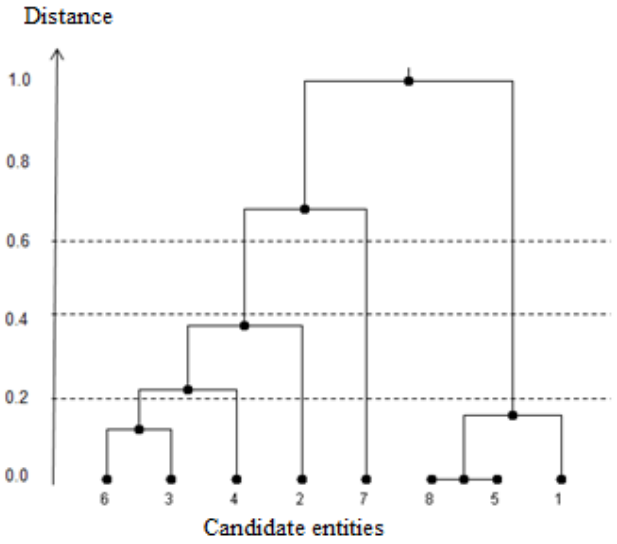


Figure 2.1: Illustration of a dendrogram

Figure 2.1 shows an example of a dendrogram. The x-axis represents the candidate entities; there are 8 in this example. The y-axis denotes the distance of the cluster pairs, where a greater value indicates a higher level of dissimilarity. The distance at which the dendrogram tree is cut determines the number of clusters formed. The dotted lines in Figure 2.1 show three tentative cuts at points 0.2, 0.4, and 0.6. At point 0.2, five clusters (6, 3), (4), (2), (7), and (8, 5, 1) are formed. On the other hand, only three clusters are formed at the cutting point 0.6. Cutting the dendrogram tree at a higher distance value usually yields a lesser number of clusters. However, this decision is at the trade-off of relaxing the constraint of cohesion in the cluster membership. Unless one is very well versed in the problem domain, it is impossible to know the correct number of clusters in advance. Defining a proximity measure is essential to identify the correct number of clusters. As a result, cluster validity index is used to find and evaluate the best partitioning formed by a particular cutting point where the actual number of clusters is unknown (Gurrutxaga et al., 2010; Saha & Bandyopadhyay, 2009). In general, the evaluation criteria of cluster validity index focus on two main aspects, which are the clustering compactness and cluster separation, as discussed by Linoff and Berry (2011).

Cluster compactness measures the similarity variance for the members of each cluster. A cluster is deemed to have high compactness if the intra-variance is low. In the context of software clustering, cluster compactness measures the cohesion of software entities within the cluster. Cluster separation, on the other hand, measures how widely separated the different clusters are. The distances among different clusters are usually large if they are well separated. The common way to measure the distance is by calculating the distance between the centroids of two clusters (Leg & Babos, 2006).

The following steps summarise a standard agglomerative hierarchical clustering algorithm.

Input: Set $T = \{x_1, x_2, \cdots, x_n\}$ of entities.

Output: Dendrogram

1. Each entity x_i forms an initial cluster G_i. The total number of clusters $K = n$. For each pair of clusters G_i and G_j, $i \neq j$, the distance between G_i and G_j is denoted by $d(G_i, G_j)$.
2. Find a pair of clusters with minimum distance, in $\{d(G_i, G_j)\}$:

   Let $d(G_a, G_b) = min\{d(G_i, G_j)\}$., where min returns the minimum distance value over the set of candidates in $\{d(G_i, G_j)\}$.

   Merge $G_c = G_a \cup G_b$ and reduce the number of clusters K= K-1.
3. If K = 1, stop the iteration; else update distance $d(G_c, G_j)$, for all other clusters G_j. (Go to Step 2)

2.3.6 Related Works on Agglomerative Hierarchical Software Clustering

Although some clustering algorithms produce a single clustering result for any given dataset, a dataset may have more than one natural and optimum clustering result. For

instance, source code can only tell very limited information about the architectural design of a software system since it is a very low-level software artifact.

The work by Deursen and Kuipers (1999) adopted a greedy search method by using mathematical analysis to analyse the structure of cluster entities and identify the features that are shared by them. The proposed approach finds all of the possible combination of clusters and evaluates the quality of each combination. Agglomerative hierarchical clustering is used in this work. The authors discovered that it is hard to analyse all possible combinations, and useful information might be missing if no attention is given to analyse all the results generated from different dendrogram cutting points.

Merits and Limitations: The proposed approach is capable of finding the most optimum clustering result in terms of cluster cohesiveness and separation since it adopted a greedy search method. However, this is at the trade-off of high computational time and complexity, which do not scale properly with huge datasets. Furthermore, the clustering results were only validated by domain experts and there is a lack of information on how the evaluations are conducted.

In contrast to the greedy search method proposed by Deursen and Kuipers, the work by Fokaefs, Tsantalis, Stroulia, and Chatzigeorgiou (2012) proposed an approach that produce multiple clustering results from which software developers and maintainers can choose the best result based on their experiences. The goal is to decompose large classes by identifying 'Extract Class' refactoring opportunities. Extract class is defined as classes that contain many methods without a clear functionality. The authors adopted the agglomerative clustering algorithm to generate a dendrogram and cut the dendrogram at several places to form multiple sets of results. The authors argued that clustering algorithms that produce a single result is too rigid and not feasible to fit into the context

of software development. According to the authors, software developers and maintainers should have the abilities to influence and pick the optimum clustering results.

Merits and Limitations: The authors allow software developers to choose the best solution from a collection of candidate clustering results. However, it is possible that the number of candidate results might be too large and difficult to reach a consensus among software developers. The authors did not discuss how to handle conflicting opinions from different software developers and maintainers.

Work by Anquetil and Lethbridge (1999a) attempted to perform agglomerative clustering on source files and found out that using source code alone to aid in software remodularisation yields poor results. In their study, clustering entities are represented in the form of source code. The authors found that the quantity of information, such as the number of variables used in the source code, the dependency between routines, the data passed and shared by functions helps in improving the reliability of clustering. This is because the authors performed their experiments on software written in structured programing languages, where most of the programs are divided into several small self-contained functions. Therefore, extracting the interrelationships between functions are not explicitly presented in the source code. Additionally, the authors suggested that domain or background knowledge from software developers who are involved in the early software design phase can greatly help improve the clustering results.

Merits and Limitations: The study covers a wide range of challenges in agglomerative software clustering, including how the cluster entities are described, how coupling between the cluster entities is computed, and the clustering algorithm used to remodularise software systems. However, as pointed out by the authors, they only experimented the proposed approach with file clustering.

Meanwhile, the work by Fokaefs, Tsantalis, Chatzigeorgiou, and Sander (2009) extends the work of Anquetil and Lethbridge (1999a) to further enhance the reliability of clustering results. Instead of just remodularising a piece of software, their technique attempts to discover classes that are completely disconnected from the system (singleton classes which have no interrelationship at all with other classes). Once the singleton classes have been discovered, the authors seek opinions from software designers whether or not to:

1. Exclude the singleton classes in future software upgrades and maintenance releases;
2. Define a new subsystem for each of these singleton classes.

Merits and Limitations: The authors tackle the issue of singleton classes, which often occurs on software systems that undergo rapid changes and maintenance. However, it is a human-intensive activity and highly dependent on the experience and skills of software developers.

Cui and Chae (2011) attempted to analyse the performance, strength, and weakness of different agglomerative hierarchical clustering algorithms using multiple case studies and setups. The authors conducted a series of experiments using 18 clustering strategies. The clustering strategies are the combination of different similarity measures, linkage methods, and weighting schemes. The case studies comprise 11 systems where source codes were used as the input parameters. The performance of each clustering result was evaluated based on a performance metric proposed by the authors. The performance metrics consist of three criteria: size of the clustering, coupling of subsystems, and cohesion among components. The authors attempted to analyse the performance of agglomerative clustering at different cutting points, by increasing the value of each cut by 5% in an iterative manner. The experiment results show that different agglomerative

clustering strategies produce various results based on different performance metrics. They found that it is difficult to identify a perfect clustering strategy which can fulfil all evaluation criteria proposed by the authors.

Merits and Limitations: Extensive work that covers a wide range of clustering strategies, evaluation criteria, and performance metrics. However, the experimental results are not conclusive because certain clustering strategies produce satisfactory performance in coupling and cohesion, but not in size.

On the other hand, the work by Chong et al. (2013) proposed a technique to enhance existing agglomerative clustering algorithms by minimising redundant effort and penalising for the formation of singleton clusters during clustering. By utilising a least-squares polynomial regression analysis, the proposed algorithm finds the optimum result that produces sets of clusters with high cohesion and low coupling. The proposed algorithm is based on a bottom-up approach, which starts by transforming source code into a flat sequence of class diagrams, and finally restructure them into a package diagram to provide a high-level semantic view of the whole system design. In one of the evaluations, the authors requested from the contributors of an open-source project, the JSPWiki project, to verify the quality of the clustering results. The authors concluded that involvement of stakeholders, even with a little amount of domain knowledge, is beneficial toward improving the modularity of software systems.

Merits and Limitations: The authors proposed a way to enhance the existing agglomerative software clustering algorithms. The proposed approach can be adapted to fit into different clustering algorithms and scale with large datasets. However, the proposed approach can only be applied on unsupervised clustering algorithm.

Table 2.2 Summary of Related Works on Software Clustering

Author	***Clustering technique***	***Input parameters***	***Objectives***	***Evaluation Methods***
(Deursen & Kuipers, 1999)	Agglomerative hierarchical clustering	Source code, routines	Find the optimum clustering result using a greedy search approach.	Validate the accuracy of clustering results by domain experts.
(Fokaefs et al., 2012)	Agglomerative hierarchical clustering	Attributes, methods, and dependencies of source code	Remodularise software by finding and decompose "Extract Class".	Retrieve ground truth from the original designer of the systems. Then, compare precision and recall between the clustering results and the retrieved ground truth.
(Anquetil & Lethbridge, 1999a)	Agglomerative hierarchical clustering	Source code	Identify and tackle various challenges in software clustering.	Compare with the results with a few agglomerative hierarchical clustering algorithms.
(Fokaefs et al., 2009)	Agglomerative hierarchical clustering	Attributes, methods, and dependencies of source code	Find singleton classes and subsequently identify solutions to refactor them.	Evaluate the effectiveness of the proposed clustering results by seeking opinions from software designers.
(Cui & Chae, 2011)	Agglomerative hierarchical clustering	Attributes, methods, and dependencies of source code	Identify a generic clustering strategy that is suitable to remodularise software systems.	Evaluate the quality of different clustering strategies by inspecting the size of clusters, coupling strength, and cohesion strength.
(Chong et al., 2013)	Agglomerative hierarchical clustering	Source code, UML class diagrams	Minimise redundant efforts when cutting dendrogram and scale with the size of datasets.	Evaluate the accuracy of the proposed approach by comparing the clustering results with the original package diagram of the software systems.

The summary of the discussed literature is shown in Table 2.2. Based on the table, it can be summarised that most of the existing studies evaluate their proposed approach from the aspect of quality and accuracy. In order to evaluate the quality of the proposed approach, the original software designers or maintainers are often involved to provide feedbacks and opinions on the clustering results. Besides that, ground truth, which in this context, a known good partition of the software systems, is used as a baseline comparison to evaluate the accuracy of the proposed approaches. Ground truth can be retrieved from domain experts or from the software documentations. Therefore, a proper way to evaluate the accuracy and quality of software clustering techniques needs to be investigated and discussed in the subsequent chapters to fit into the context of this thesis.

Besides that, the software clustering approaches discussed in the previous section are typically conducted in an unsupervised and rigid manner, where software maintainers have no influence on the clustering results and only a single solution is produced for any given dataset. However, if software maintainers do not agree with the final clustering result, they will need to repeat the whole clustering process again using different clustering algorithm or configuration, which is ineffective and time consuming. Furthermore, if software maintainers possess additional information (explicit domain knowledge, background knowledge, or other source of implicit information) that could be useful to guide and improve the clustering results, traditional unsupervised software clustering algorithms do not have the ability to take advantage of this information.

Hence, an improvement was proposed in the work by Basu et al. (2004) to incorporate domain or background knowledge into traditional unsupervised software clustering. Domain experts or software developers who are involved in the early stages of software design or development have the ability to exert their feedbacks and opinions in the form

of clustering constraints to guide and improve the clustering results. This type of semi-supervised clustering technique are also commonly known as constrained clustering.

2.4 Constrained Clustering

Given a set of data to be clustered, if the cluster entities are able to form cohesive clusters easily, there is no need for human intervention since any clustering algorithm will identify the desired clusters (Klein, Kamvar, & Manning, 2002). Similarly, if the data to be clustered are tightly coupled and no distinction can be made between the entities, then a small amount of domain or background knowledge would be useful to help improve the clustering results.

2.4.1 Formulation of Clustering Constraints

The domain or background knowledge can normally be translated into sets of explicit constraints which involve two cluster entities and impose restrictions such as determining whether the involved entities should be clustered into the same cluster or not. Constrained clustering is contrary to traditional unsupervised clustering where users have a certain degree of influence on the final clustering results. There are several ways to translate background knowledge into clustering constraints. For instance, in movie recommender systems, users may have prior knowledge that two movies belong to the same genre. Thus, the user can explicitly modify the recommendations based on his or her knowledge to improve the query results. Apart from deriving explicit clustering constraints directly from users, implicit constraints can be derived automatically from certain features or behaviour of the datasets. For instance, in the research area of wireless local area networks, clustering techniques have been used to optimise placement of access points

for better network coverage and minimise data latency (Huang, Wang, & Chang, 2005). Neighbouring access points usually form clusters and one designated access point from each cluster will have a wired connection to a switch or router. If the access points inside a cluster are operating within the same overlapping channels, they must be redeployed to other clusters to avoid frequency interferences (Eisenblätter, Geerdes, & Siomina, 2007). Thus, from a clustering's point-of-view, these observations can be treated as implicit clustering constraints, such that it enforces rules to prevent access points that operate in the same overlapping channels to be clustered into the same group. Automatically derive implicit clustering constraints from the problem domain is useful when there are no domain experts available, although such way of extracting constraints involves several complicated steps to identify and interpret them.

Several existing works (Basu et al., 2004; Davidson & Ravi, 2009; Wagstaff & Cardie, 2000) have attempted to discover the benefits of pairwise constraints in both hierarchical and non-hierarchical clustering and found that even a small amount of constraints can improve the quality of clustering when compared against those without constraints. The work by Wagstaff and Cardie defined two types of pairwise constraints, namely the must-link (ML) and cannot-link (CL) constraints, which specify that two entities must both be part of or not part of the same cluster respectively. These constraints are useful when the information of cluster entities is vague, allowing domain experts to guide the clustering process.

2.4.2 Enforcing Clustering Constraints

Enforcement of clustering constraints are divided into three major categories, namely distance based, constrained based, or the hybrid of both.

In distance based constrained clustering, the distance matrix of the associated clustering entities are trained beforehand to satisfy the constraints before execution of clustering algorithms. The distance matrix is a $n \times n$ two-dimensional pairwise matrix (n equals to the number of clustering entities) that contains the distance or dissimilarity strength between each pair of entities. Merging or splitting of clusters is based on the distance matrix. To provide a better illustration of how distance based constrained clustering works, given a pair of entities *x* and *y* with distance of 0.8 (1 indicates very dissimilar, 0 indicates very similar). A must-link constraint is being enforced on the pair *(x, y)* using the distance based approach, by modifying the distance between *(x, y)* from 0.8 to 0. This will allow any clustering algorithm to always group both entities into the same cluster. Thus, training the distance matrix allows certain pairs of entities to be clustered into the same group or separated if otherwise. Examples of methods to train the distance matrix include shortest path (Klein et al., 2002), expectation maximisation (Bilenko & Mooney, 2003), and convex optimisation (Shental & Weinshall, 2003).

On the other hand, constrained based methods work by modifying the cluster assignments, i.e. manually assign entities to designated clusters (Kestler, Kraus, Palm, & Schwenker, 2006). Constrained based approaches ensure that all the clustering constraints are fulfilled because the clustering assignments are manipulated by users based on the given constraints. However, studies performed by Davidson and Ravi (2009) discovered that manipulating with the clustering assignments might lead to "dead-end" situation where no pair of clusters can be merged to obtain a feasible clustering result. Hybrid of both distance based and constrained based methods is relatively more complex and harder to execute because it might result in undesired consequences if there are contradicting

clustering constraints. Thus, a proper way to ensure the fulfilment of clustering constraints must be formulated before enforcing any kind of constraints.

2.4.3 Fulfilment of Clustering Constraints

Fulfilment of clustering constraints can be classified as either hard or soft constraints associated with some cost of violation if the constraints cannot be fulfilled (Basu et al., 2004). Hard constraints are clustering constraints that cannot be violated during the clustering process regardless of any condition. Hard constraints are usually highly reliable knowledge or information given by domain experts. In general, the cost of violating hard constraints supersedes the objective function of constrained clustering. Constrained based clustering method is one of the most reliable approaches to make sure that all hard constraints are fulfilled as much as possible.

Meanwhile, soft constraints are usually associated with uncertainties and ambiguous information, such that these set of clustering constraints are good to have, and their fulfilment are not absolute (Basu et al., 2004). The cost of violating soft constraints varies depending on the importance and severity of the constraints. Clustering results will still be acceptable if some of the soft constraints are not fulfilled, with a condition that it falls within an acceptable threshold (Ares, Parapar, & Barreiro, 2012). Soft constraints are more robust against "noisy" or incorrect constraints (Hong & Yiu-Ming, 2012). As a general rule, most of the objective functions attempt to maximise the fulfilment of hard and soft constraints. However, it is to be noted that constrained clustering can fall into a NP-Complete problem if the must-link and cannot-link constraints are contradicting with each other, for instance, (Must-Link ∪ Cannot-Link) > 0. Thus, potential conflicts among constraints must be identified in advance.

The work by Davidson and Ravi (2009) examined the complexity of traditional clustering algorithms and investigated potential methods to improve the efficiency of constrained agglomerative hierarchical clustering. The authors introduced new constraints apart from the traditional ML and CL constraints to further improve the runtime of agglomerative hierarchical clustering. They discovered that small amounts of constraints not only improve the accuracy of agglomerative hierarchical clustering but also the overall runtime. However, the authors noticed that clustering under all types of constraints (ML and CL) is NP-complete, which means that creating a feasible cluster hierarchy under all types of constraints, is intractable. The NP-complete problem can be minimised if each constraint is assigned with a certain degree of importance, i.e. constraints that must be fulfilled, or optional constraints that are good to have.

2.4.4 Applying Constrained Clustering to Remodularise Software Systems

Existing studies in constrained clustering mainly focus on the domain of data mining and machine learning to cluster text documents, images, and to perform biological classifications. There is a lack of studies that focuses on applying constrained clustering in the field of software reverse engineering to aid in remodularisation of poorly designed or poorly documented software systems although it is technically feasible. For instance, experienced software developers can explicitly provide opinions with a certain degree of confidence to suggest if two classes should be clustered into the same group. However, deriving clustering constraints from software developers can be a very time consuming and human-intensive task because developers will need to review a high-level abstraction view of the software and its documentations, depending on the level of abstractions, before creating the intended clustering constraints. On the other hand, deriving clustering

constraints in the field of biological classification is relatively easier, for instance, finding organisms that share similar polygenic traits.

Therefore, other options to automatically derive clustering constraints from the implicit features and behaviours of the software systems are needed. For example, inheritance between parent and child classes suggests a strong affiliation such that changes in the parent class will directly impact on the child class. This implicit behaviour suggests a must-link constraint between the two involved classes. Additionally, two classes that provide completely different functionalities typically have little to none interactions (interactions in terms of passing parameters or sharing of variables) and are usually placed in different software packages. Each class generally handles only one responsibility to advocate the principle of single responsibility (DeMarco, 1979). This behaviour can be considered a cannot-link constraint from the perspective of constrained clustering because the two associated classes are intended to be clustered into two different disjoint clusters. However, how to identify and analyse clustering constraints, either from background knowledge or the implicit structure of software systems, remains as a huge research problem.

Strong understanding of software systems is needed in order to extract all the essential information from the source code, and subsequently convert them into potential clustering constraints. Evaluating a software system using well-established software metrics is one of the approaches used in existing studies to provide a better understanding of the software, and to prevent any faults from propagating to other parts of the software. Software components that are more prone to bugs and errors can be translated to sets of clustering constraints so that the propagation of software bugs can be controlled.

Evaluation of software systems using software metrics can be carried out at different levels of granularity in terms of classes, packages, or the entire system.

Examples of well-established software metrics are the Chidamber and Kemerer's Metrics Suite (CK) (Chidamber & Kemerer, 1994) and the Metrics for Object Oriented Design (MOOD) (Abreu & Carapuça, 1994). CK and MOOD metrics are well known for measuring the complexity of OO software as well as identifying software defects. CK metrics evaluate software at the class level by looking into factors such as class cohesion, coupling, complexity, and inheritance. MOOD metrics focus on OO characteristics such as encapsulation, polymorphism, and inheritance to provide a system-wide assessment. In spite of their wide usage, CK and MOOD metrics share the same disadvantages where they focus mainly on single classes and rarely take the interactions between classes into consideration (Zimmermann & Nagappan, 2008). In addition, several studies have found that the empirical effects of these metrics are less effective on large-scale OO software systems (El Emam, Benlarbi, Goel, & Rai, 2001; Gyimothy, Ferenc, & Siket, 2005; Olague, Etzkorn, Gholston, & Quattlebaum, 2007a; Subramanyam & Krishnan, 2003b).

As software systems become larger and more complex, software maintainers need to gain a better understanding of the macroscopic properties of these systems if they were to make critical decisions about re-engineering, maintaining and evolving such systems (Lian, Kirk, & Dromey, 2007). Large-scale industrial software systems, such as enterprise resource planning systems, usually involve multiple complex modules that are related with each other. Thus, traditional ways of analysing and characterising software systems using software metrics might not be adequate for large-scale software systems. There is therefore a need to investigate techniques from other disciplines that had successfully dealt with systems of high complexity. Graph theory used in combination with complex

network is one such suitable technique to solve the aforementioned problem. Representing software systems using complex networks enable software maintainers to gain a better understanding of the problem domain from a graph theory's point-of-view, and subsequently transform the findings into clustering constraints. The next section provides an in-depth discussion on the existing studies that represent software systems using complex networks, along with some of the challenges.

2.5 Facilitate Understanding of Software Systems with the Aid of Graph Theory Metrics

In recent years, research in software engineering in the aspect of representing software systems using complex networks has started to emerge with the aim to gain a high-level abstraction view of the analysed software systems (G. Concas, Marchesi, Murgia, Tonelli, & Turnu, 2011; Ma, He, Li, Liu, & Zhou, 2010; Tempero et al., 2010). Representing software systems using complex networks allows software maintainers to gain more insights on the studied software through the application of well-established graph theory metrics (Turnu, Concas, Marchesi, & Tonelli, 2013).

Graph theory is a field of study that looks into the formal description and analysis of graphs (Bullmore & Sporns, 2009). A graph is generally defined as a set of nodes that are connected by edges, which may or may not be weighted. If the relationships or interactions between the nodes are asymmetric, then the graph is usually presented as directed graph, as opposed to the undirected ones. When describing a real-world network, be it social network, scholarly citation network, or software system, a graph provides an abstract representation of the network's elements and their interactions. Real-world networks display fundamental topological features and patterns that are not found in

random networks (Giulio Concas, Marchesi, Pinna, & Serra, 2007; Myers, 2003). Graphs that are formed based on these real-world networks are commonly referred as *complex networks*. As discussed by Simon (1991), a complex network is an integrated sets of nodes that are organised in a hierarchical structure and interact in a non-simple way. Existing studies have found that graphs formed based on software systems often exhibit the topological features and behaviours of a complex network due to their inherent complexity (G. Concas et al., 2011).

A complex network, $G = (V, E)$, is made up of a set of nodes V, and a set of edges $E \subseteq V \times V$ that connect pairs of nodes. In general, a complex network can either directed or undirected. In both directed and undirected networks, edges may be associated with weights to denote the similarity of a pair of nodes connected by an edge or the cost of traveling through that particular edge. In a directed network $G = (V, E)$, $(i, j) \in E$ signifies that there is an edge in E that is linking node i to node j where i is the origin and j is the terminus. On the other hand, in an undirected network $G_u = (V, E)$, if $(i, j) \in E$, then edge $(j, i) \in E$ as well because the origin and terminus are not specified in an undirected network.

Both directed and undirected networks can be represented by their own adjacency matrix A. The matrix A is a $|V| \times |V|$ matrix where the rows and columns represent the nodes of the network. In an undirected network, the entry $A_{ij} = 1$, if $(i, j) \in E; \forall\, i, j \in 1, \cdots, |V|$. Value 0 indicates that there is no relationship in between nodes i and j. Meanwhile for a directed network, the value A_{ij} represents the weight associated with edge (i, j). The value of adjacency matrix A is symmetric for an undirected network such that $A_{ij} = A_{ji}$. In a directed network, however, the relation A_{ij} is asymmetrical.

In OO software systems, objects and classes are normally related through different kinds of binary relationships, such as inheritance, composition and dependency. Thus, the notion of associating graph theory to represent large OO software systems and to analyse their properties, be it structural complexity or maintainability, is feasible.

Directed network is more suitable to gain a high-level understanding of software systems because it is capable of capturing the semantic relationships between software components. The work conducted in (Anquetil & Lethbridge, 1999a; Davey & Burd, 2000) shows that software features are asymmetric in nature. Examples of such behaviour are superclass and subclass relationship, master and slave relationship in Message Passing Interface (MPI) programming, and an encapsulated object or data which has its internal behaviour or function hidden from outside of its own definition (Chong et al., 2013).

Besides that, there are several features in graph theory that can be used to analyse the structure and behaviour of software systems. Recent studies of representing objected-oriented software systems as complex networks revealed that many of these networks share some global and fundamental topological properties such as scale free and small world (Giulio Concas et al., 2007; Louridas, Spinellis, & Vlachos, 2008; Pang & Maslov, 2013; Potanin, Noble, Frean, & Biddle, 2005).

The scale free characteristic of a network is defined as follows. In graph theory, the degree k_i of a node i is the total number of its edges. In general, a node with a higher degree indicates that it possesses a higher impact with respect to the whole network. The average of k_i over all nodes is the average degree of the network, which is represented as $< k >$. The spread of nodes' degrees over a complex network is described as a distribution function $P(k)$, which is the probability that a randomly selected node has exactly k edges

(Barabasi, Albert, & Jeong, 2000). A simple network normally has a simple distribution function because all the nodes contain a similar number of edges (Barabasi & Albert, 1999). Therefore, the probability of finding a node with k edges is very high. However, studies have discovered that many complex networks derived from software systems obey the degree distribution of a power law in the form $P(k) \sim k^{-\gamma}$ where the function falls off more rapidly than an exponential function (Giulio Concas et al., 2007). This results in situations where a few nodes with very high degree (highly connected nodes) exist in these complex networks. Because the aforementioned power law is free from any characteristic scale, these complex networks are also called as scale free networks. The scale free characteristic in software systems can be interpreted as the level of reuse of important classes, or the number of dependencies between classes.

The small world property is related to the average shortest path length and clustering coefficients in graph theory. A shortest path is defined as the shortest distance between a pair of nodes in a graph. The average shortest path length, on the other hand, calculates the average number of steps along the shortest path between every pair of nodes in a network. The average shortest path length is often used to measure the efficiency of information passing and response time in OO software. Existing works that use complex networks to analyse software systems indicated that the average shortest path length of software is around 6 (Valverde & Solé, 2003). A clustering coefficient, on the other hand, is the average tendency of pairs of neighbours of a node that are also neighbours of each other. It can be used to measure the degree to which nodes tend to cluster together. In short, networks that exhibit small world property signify that the distances between nodes are relatively shorter as compared to random networks. From a graph theory's point of view, software systems that exhibit a small average path length and high clustering

coefficient signify that the analysed software obey the design principles of low cohesion and high coupling.

In short, graph theory metrics and software metrics offer different advantages for analysing the complexity of software system. Software metrics such as CK and MOOD excel in evaluating class-level complexities, particularly in the OO paradigm. Complex network, on the other hand, is capable of evaluating the impact of a particular class with respect to the whole system. The results of graph theoretical analysis can then be translated into clustering constraints to help improve the results of software clustering.

However, before applying graph theory metrics, the software systems must be transformed into its associated complex networks in advance. An OO software system is typically composed of multiple classes. At the source code level, classes in OO software may contain data structure, objects, methods, and variables. Two classes can be considered related if there are actions such as passing of messages. Due to multiple ways of representing nodes and edges, there is a need to perform an in-depth review on existing works that represent software systems using complex networks.

2.5.1 Representing Software Systems Using Un-weighted Networks

The work by Valverde and Solé (2003) discusses the usage of two graphs, namely Class Graph and Class-Method Graph, to analyse the global structure of software systems. Class Graph is derived based on UML class diagrams, where classes are represented as nodes, while relationships among classes, such as dependency and association, are depicted as edges between nodes. Class-Method Graph is modelled based on source code using the

similar concept. The nodes and edges of the constructed graph are modelled to be un-weighted.

Merits and Limitations: The authors utilise complex networks to analyse software systems from two different levels of abstractions, namely the low level Class-Method Graph, and the high level Class Graph. The two graphs are inter-related and can be analysed from different perspectives to provide a better understanding of the software. However, for both types of graphs, the complexity of nodes and edges is ignored mainly because the authors assumed that internal complexities do not change the global structure of software systems.

Myers (2003) proposed a method to represent software systems using complex networks by analysing the interdependencies of source code. A software collaboration graph based on the calling of methods is used to analyse the structure and complexity of software systems. The authors parsed source code into a software documentation generator tool called the Doxygen. The tool automatically generates documentation from the raw source code. Based on the generated documentations, the authors can easily identify all the methods declared in each class, along with the dependencies and collaborations between each method. The work by Myers was later extended in the work by LaBelle and Wallingford (2004) and Hyland-Wood, Carrington, and Kaplan (2006) to include the usage of classes and packages. However, similar to the work by Valverde and Solé, the software collaboration graphs by LaBelle et al. and Hyland et al. are constructed using un-weighted edges and nodes.

Merits and Limitations: The work by Hyland et al. explored the usage of software collaboration graph across different levels of granularity, namely package level, class level, and method level. The authors attempted to discover similar network properties from the associated networks. However, due to the way the software collaboration graph

is generated, the Doxygen tool can only identify unidirectional and symmetrical relationships between methods. This might not be representative enough to illustrate the behaviour of software systems because the work by (Anquetil & T. C. Lethbridge, 1999; Davey & Burd, 2000) shows that software features are asymmetric in nature.

Jenkins and Kirk (2007) used source code as the basis to construct a software architecture graph. Classes in the source code are identified and represented by nodes. When a class accesses or refers to data or functionality in another class, it is represented as an un-weighted edge that connects both classes. Similar way of representing a software architecture graph is also presented in the work by Louridas et al. (2008) using class interactions. The work by Louridas et al. aims to investigate the power law's behaviour in software systems written in Java, C, Perl, and Ruby.

Merits and Limitations: The work by Jenkins and Kirk proposed several metrics to measure the stability and maintainability of software through various releases. The authors discovered very strong correlation between a high growth rate and lowered maintainability. High growth rate in this context refers to classes that undergo frequent changes and increase in functionality. The proposed metrics are also applicable on nature science research to explore the mechanics of phase change in living organisms. However, although the software systems are represented as directed network, the main part of the analysis treats the software as being an undirected and un-weighted graph due to limitation in converting the directionality of graphs into quantifiable measurements.

The work by Taube-Schock, Walker, and Witten (2011) investigates the problem of high coupling in software systems. Generally, the notion of high cohesion and low coupling is a well acknowledged software design principle. However, various studies that represent software systems using complex networks have empirically discovered that all software

exhibit the scale-free behaviour, which means that in certain cases, some high coupling is unavoidable (Myers, 2003). In order to verify the aforementioned behaviour, the authors performed empirical studies using the Qualitas Corpus (Tempero et al., 2010), which is a collection of 100 independent open-source software systems written in the Java. The source code is converted into a directed graph, where nodes represent source code entities ranging from packages to variables, and edges represent connections between entities. The authors found that all the studied software systems exhibit a similar scale-free dependency structure and some high coupling might be a necessary characteristic for good software design.

Merits and Limitations: The work is evaluated on a wide range of open-source software systems which vary in terms of size, application domain, and complexity. Thus, the findings are representative enough to demonstrate the common behaviour exhibited in real-world software systems. However, the authors chose to exclude hierarchical relationships (such as inheritance and generalisation) from their analysis because they argued that inclusion of hierarchical relationships will skew the results of intra-cluster analysis.

Finally, the work by Hamilton and Danicic (2012) proposes a Backward Slice Graphs (BSGs) made up of nodes that represent program statements and un-weighted edges that represent dependencies between these statements. The main purpose of BSG is to find groups of strongly related software components by inspecting the number of dependencies between program statements.

Merits and Limitations: The BSGs are constructed at the statement level, which is the lowest level of granularity among all the discussed studies. However, since nodes and edges of the BSGs are represented based on program statement, the structure and

behaviour of the graph are highly dependent on programmers because different individuals might adopt different programming styles and preferences.

Several commonalities were found based on the discussed studies that use un-weighted edges in representing software systems. Firstly, majority of these studies use source code as a basis for representing complex networks except for the work by Valverde and Solé. Most of the studies took a black box approach when transforming source code into nodes and edges by assuming that the types of relationships between nodes do not change the global structure of the constructed network. The types of relationships, such as inheritance and method invocation, are represented by a common type of 'dependency' to signify interactions between nodes. Besides that, the complexities of nodes are ignored. Instead, most of the studies only use the frequency of in-coming and out-going dependencies as the basis of measuring the weights of edges. The authors are more concerned on the global structure and behaviour of the analysed software, either from a static or evolutionary point of view. For instance, the work by Valverde and Solé, LaBelle et al., Hyland et al. and Louridas et al. focuses on representing software using a static network abstraction and applied statistical analysis to discover the behaviour of software systems. The work by Myers and Jenkins et al., on the other hand, took an evolutionary approach by inspecting software in different releases and compared the results using a statistical approach.

2.5.2 Representing Software Systems Using Weighted Networks

Next, several related works that deal with weighted complex network representation of software systems are discussed.

Giulio Concas et al. (2007) demonstrated the use of weighted complex network to study the structure and properties of OO software called the VisualWorks Smalltalk. Nodes are made up of classes, while edges represent method calling between classes. By inspecting the source code, an edge is created toward each of the classes that implements a particular method and assigned a weightage value of 1. If the same method is called by multiple methods of a class, the weight of that corresponding edge is then multiplied by the total number of method invocations. Based on the number of method invocations, the constructed graph will be able to distinguish nodes with higher inter-dependency.

Merits and Limitations: The authors investigated a wide variety of system properties, including distributions of variables, method names, inheritance hierarchies, class sizes, method sizes, and system architecture graph. Based on the in depth evaluations, the authors concluded that certain complex network properties are highly correlated to software metrics, such that these network properties can be used to evaluate the quality of software systems. However, the effectiveness of this method is highly dependent on the programming language and style practiced by different individuals.

The work by Sun, Xia, Chen, Sun, and Wang (2009) investigates the structural properties of Linux kernel by constructing the network using C++ header files. The header files are represented as nodes, and two nodes are connected with a weighted edge if both header files are included in the same source file, i.e. using the 'include' operation. The weights of the edges are calculated based on the number of include operations in the header files.

Merits and Limitations: The constructed networks are evaluated using several un-weighted and weighted graph theory metrics, in order to provide an additional perspective on the structural organisation of the complex network. However, the experiments are performed solely based on header file interactions, which limit its applicability to software systems written in C++ programming language.

In the domain of software testing, Lan, Zhou, Feng, and Chi (2010) proposed to use complex network to study software execution process, and subsequently identified the process that is more fault-prone. Functions are represented as nodes and function calls are represented as weighted directed edges. The edges are weighted based on the number of function calls between a pair of nodes, while the directionality of the edges is based on the sequence of the execution process. Empirical validations are conducted on Linux based programs, namely tar, gedit, and emacs. Nodes are not weighted. The authors used in-degree as the metric to identify highly critical components. In-degree in this context refers to the number of times a function is being called by other processes. Thus, the weights of edges are used to measure the complexity and significance of nodes.

Merits and Limitations: Software components that are more prone to bug and errors can be easily identified. This allows software developers to take actions before software bugs are propagated to the rest of the system. However, the authors only focused on using in-degree as the only indicator to identify bug prone software components, neglecting the potential benefit of other graph theory metrics.

A hybrid approach which extracts information from both the source code and software architecture was presented by Ma et al. (2010). The authors proposed a set of metrics to measure OO software from multiple levels of granularity, at the class level, code level, graph level, and system level. Representation of nodes and edges depends on how much information is supplied during the analysis. For instance, if only the UML class diagram is provided, classes are represented as nodes, while interactions between classes such as inheritance and association are represented as edges. The edges can be weighted using several well-known software metrics depending on the choice of the user, such as Genero's metrics (Genero, Piattini, Manso, & Cantone, 2003) (number of associations,

number of aggregations, and depth of inheritance). The same concept is applied to weight the nodes by using well known software metrics such as CK and MOOD metrics. The focus of the work of Ma et al. is to measure the complexity of software, and subsequently, detect fault-prone software components.

Merits and Limitations: The proposed method allows software maintainers to measure the quality of software systems from various levels of granularity. Apart from providing a better understanding of the software systems, the constructed networks can be used to identify software components that violate common design principles. However, due to the extensive number of software metrics used in the study, software maintainers need to possess a relatively good understanding of the software systems beforehand, which might be challenging for legacy software systems.

The work by Wang and Lu (2012) presents an approach to represent complex software systems using weighted complex networks. The authors used two open-source software written in Java, namely Junit and JEDIT, to demonstrate the applicability of the approach. Classes in the source code are used to model the nodes. If a method of a particular class is dependent on other classes, a weighted edge is used to model this behaviour. The weight of an edge is measured by inspecting the number of dependencies between a pair of nodes connected by the edge.

Merits and Limitations: When comparing the statistical characteristics of weighted and un-weighted complex networks, the authors discovered that weighted network can better represent the exact dependency relationships among classes of software systems. However, the authors did not attempt to differentiate between different types of relationships, and treat all relationships as equal.

The works discussed in this sub-section use very similar technique to measure the weights of edges. Almost all of the studies use the frequency of interactions among nodes, such as number of class dependencies, number of method invocations, number of information exchanges, and number of method dependencies, to measure the weights of edges. Counting the frequency of interactions among nodes is basically based on the static structure of software systems, which might not be representative of the dependency relationship of the real-world software systems. Dynamic relationships between classes of an OO software, such as polymorphism, dynamic binding, and inheritance relationships cannot be captured easily through static analysis of source code (Arisholm, Briand, & Foyen, 2004).

2.5.3 Discussion

A summary of the discussed literature is presented in Table 2.3. There is however an exception; the work by Ma et al. uses well known software metrics to formulate the weighted edges. Most work discussed in Section 2.5.2 and this section counts the number of in-coming and out-going interactions between nodes to identify the critical nodes.

Table 2.3: Related Work in Representing Software Using Complex Networks

Related work	*Type of Network/ Graphs*	*Representation of Nodes*	*Representation of Edges*	*Weighted Edges*	*Weighted Nodes*
(Valverde & Solé, 2003)	Class Graph/ Class-Method Graph	Source code / UML class	Method dependencies / UML class relationships	No	No
(Myers, 2003)	Software collaboration graph	Source code	Calling of methods	No	No
(LaBelle & Wallingford, 2004)	Software collaboration graph	Classes and packages	Access or refer to other classes	No	No

(Hyland-Wood et al., 2006)	Software collaboration graph	Classes and packages	Access or refer to other classes	No	No
(Jenkins & Kirk, 2007)	Software architecture graph	Classes	Access or refer to other classes	No	No
(Taube-Schock et al., 2011)	Software collaboration graph	Packages, classes, methods, blocks, statements, and variables	Hierarchical containment, method invocation, and superclass	No	No
(Hamilton & Danicic, 2012)	Backward Slice Graph	Program statements	Dependencies between statements	No	No
(Giulio Concas et al., 2007)	Class Graph	Classes	Calling of methods between classes	Yes, based on frequency	No
(Sun et al., 2009)	Software collaboration graph	C++ header files	Use of include operations	Yes, based on frequency	No
(Lan et al., 2010)	Software execution process	Functions	Functions call	Yes, based on frequency	No
(Ma et al., 2010)	Hybrid Graph	Source code / UML classes	Multiple metrics	Yes, software metrics	Yes, software metrics
(Wang & Lu, 2012)	Class Graph	Classes	Calling of methods, dependencies between methods	Yes, based on frequency	No

There are potential drawbacks of relying only on counting the frequency of interactions between nodes to quantify the weightage of edges. First of all, utility classes, such as classes with static methods that are heavily called by other classes, might distort the result of statistical analysis. Generally, a high number of interactions is often observed for utility classes that are used extensively in a software system. Thus, a high out-degree or edge-weighted value is not necessarily an indication of bad design and fault proneness (Ragab & Hany, 2010). Besides that, an edge-weighted method based on the frequency of

interactions might not be suitable for software in the domain of parallel processing and software that deal with a high number of information exchanges, i.e. mail servers and database systems. Flow of information and data across all layers of the system is very common for master and slave interactions in MPI programming, and database update or query operations in a typical database management system.

The advantages of using weighted complex network over un-weighted complex network are also discussed in the work by Wang and Lu (2012) and Ma et al. (2010). Specifically, the work by Wang and Lu (2012) found that distribution of nodes with a high in-degree and out-degree in edge-weighted networks is much more concentrated than those in un-weighted networks. Thus, the group of nodes that are critical to the systems, as well as those nodes that are associated with it can be identified easily. Based on the identified nodes with a high node-strength, software developers can choose to either decompose or reuse the associated software components to improve software stability and maintainability. Ma et al., on the other hand, observed that nodes with a high out-degree generally have a low in-degree, and vice versa. Nodes with an improper ratio, i.e. high in node-strength and out node-strength, are nodes that do not adhere to high cohesion and low coupling design principle. Thus, these sets of nodes may lead to potential defects and bugs. Furthermore, Wang and Lu (2012) found that the effect of bug propagation in a weighted network is lesser compared to an un-weighted network based on the analysis of average shortest path length. Similar observations are also found in the work by Ma et al. where the average shortest path of the analysed software is around 4.86 steps. All in all, weighted networks are found to be able to capture the behaviour and characteristics of software systems in a more well-defined and detailed manner, especially when representing the relationships among nodes.

However, most of the studies discussed are working on the source code level except for the works by Valverde and Solé. and Ma et al., which also involve the software design level. At the software architecture level, UML classes are typically chosen to denote cluster nodes, providing a standardised conceptual model that represents the system's components, operations, attributes, and relationships. UML class diagram is a better choice when compared to raw source code because it is platform and language independent. UML class diagrams are also less susceptible to human factors, which in this context, refers to different programming styles practiced by different individuals. Because the structure, notations, and modelling of UML class diagrams are standardised, it is easier to construct complex networks based on class diagrams.

2.6 Challenges and Issues in Constrained Clustering

Through the in-depth reviews conducted, it can be summarised that even a small amount of clustering constraints can help in improving the quality of clustering results. Generally, the clustering constraints can be derived from two different sources, namely explicit information or implicit information of the software to be maintained. Explicit information refers to the feedbacks given by domain experts who are involved in the early stage of software development, where the information can be translated into explicit clustering constraints.

On the other hand, implicit information refers to some extra deterministic information about the interrelationships between software components that are hidden in the source code. Applying complex network in combination with graph theoretical analysis is one of the techniques used in existing studies to harness the implicit information of a software from the graph theory's point of view. The results of the graph theoretical analysis can be

subsequently converted into implicit clustering constraints to help in improving the accuracy of clustering results. However, the representation of nodes and edges of a software-based complex network for the purpose of measuring complexity and their relationships is not explicitly addressed in existing studies.

Most of the existing works that represent software systems using complex network only focus on the static representation of the software, i.e. using undirected and un-weighted complex network. Although there are several studies that attempted to use weighted network to represent software systems, almost all of them depend solely on counting the frequency of interactions among software entities, which only address the static behaviour of software systems and might not be representative enough to illustrate the dynamic dependency relationship of real-world software systems. The frequencies of interactions are highly dependent on the programming languages used and the programming skills possessed by the software developers. Valuable information such as quality and complexities of software components might be lost in the transformation process.

Furthermore, fulfilment of the derived explicit and implicit clustering constraints in agglomerative hierarchical clustering is a challenging task due to several reasons. Firstly, due to the hierarchical structure of agglomerative clustering, fulfilment of clustering constraints is relatively more complex and hard to execute compared to partitional clustering. As shown in Figure 2.1, a dendrogram needs to be cut at a certain level to form several disjoint clusters. Although there are several ways to enforce constraints, (such as distance based and constraint based method discussed in Section 2.4.2) a proper mechanism is needed to ensure that all the given must-link or cannot-link constraints are fulfilled at any cutting point. Besides that, as discussed by Davidson and Ravi (2009), manipulating with the clustering assignments without proper planning might lead to

"dead-end" situation where the clustering process might end prematurely because no pair of clusters can be merged anymore to obtain a feasible clustering result.

Even if the problem of "dead-end" can be resolved, it is almost impossible to fulfil each and every clustering constraint given by the domain expert since conflicting constraints might occur when considering both must-link and cannot-link constraints. To provide a simple illustration, given that there are two must-link constraints and one cannot-link constraint such that

- Class A must-link Class B
- Class B must link Class C
- Class A cannot-link Class C

Based on the given examples, it is obvious that the two must-link constraints are conflicting with the cannot-link constraints. This is because upon fulfilling the two must-link constraints, all three classes A, B, and C will be merged into the same cluster, causing the cannot-link constraint to be unrealisable. This is considered a NP-Complete clustering problem, as mentioned in Section 2.4.3. The problem can be mitigated if software maintainers can know beforehand which clustering constraints should be prioritised, and which constraints are optional to be fulfilled.

Furthermore, most of the existing studies only focus on a specific programming language when analysing the structure and behaviour of software systems. For instance, the work by Sun et al. (2009) only focused on C++ while the work by Wang and Lu (2012) focused on software written in Java. Since the structure, method declaration, and interactions of methods behave slightly different across different programming languages, the finding

based on one particular programming language cannot be applied to software written in other languages.

In terms of evaluation, it is discovered that most of the existing studies seek advice from the original software designers to evaluate the quality of clustering results. In order to evaluate the accuracy of the clustering results, ground truth needs to be identified beforehand to serve as a reference model. However, if the selected test subjects are open-source software systems, it is almost impossible to seek advice from the original designers since most the design decisions of open-source projects are done in an ad-hoc manner. Ground truth, on the other hand, is hard to retrieve if the software systems are not well documented. Therefore, a proper method to evaluate the effectiveness of the proposed constrained clustering approach needs to be devised.

In summary, representing software systems using weighted complex networks in combination with graph theory is an effective method to understand the implicit structure, behaviour, and complexity of software components and their relationships. The results of the graph theoretical analysis can be subsequently translated into implicit clustering constraints. Combined with the explicit constraints derived from domain experts, both the implicit and explicit constraints are beneficial to software maintainers by providing a means to guide and improve the results of software clustering. However, there are very limited studies that focus on applying clustering constraints onto software clustering and representing software systems using weighted complex networks to extract candidate clustering constraints.

2.7 Chapter Summary

This chapter has presented the literature of software remodularisation, software clustering, constrained clustering and also representing software systems using complex networks. A thorough review on the literature is done based on the approaches used to achieve better software modularity. Nonetheless, several issues and challenges are raised based on the study of existing literature.

CHAPTER 3: RESEARCH METHODOLOGY

This chapter discusses the research methodology used in this research. This thesis follows an empirical research methodology that consists of four phases, namely the Formulation Phase, Design and Conceptualisation Phase, Experimentation Phase, and Analysis and Interpretation Phase. A constrained clustering approach for clustering OO software systems is proposed in this thesis. The proposed approach is facilitated by two methods and one technique, where the first method uses complex network to represent an OO software system with the aid of a unique weighting mechanism to construct a weighted complex network for the OO software system. Next, a technique to automatically derive clustering constraints from the constructed weighted complex network and evaluate the software system based on graph theoretical analysis of the weighted complex network is introduced. Following that, a method to maximise the fulfilment of all the identified clustering constraints is presented to improve the accuracy and scalability of the clustering approach. Finally, a high-level abstraction of the software design with highly cohesive clusters is formed based on the clustering constraints derived from the implicit structure of the software.

3.1 Research Approach

The research starts with investigating existing works related to software clustering in order to identify the research gap. Subsequently, research questions and objectives are refined and reformulated based on the in-depth literature review. A constrained clustering approach supported by several methods and techniques is introduced to address the research questions and to help achieve the research objectives. Source code of an OO software system is first converted into UML class diagrams. Next, information from the

UML class diagrams are extracted to measure the cohesion strength among related classes, before transforming them into a weighted complex network. Graph theory metrics are subsequently applied onto the transformed weighted complex network so that the structure, behaviour, as well as complexity of software components and their relationships can be analysed. The result of the analysis is then converted into sets of implicit clustering constraints. If domain experts are available, they are allowed to explicitly provide feedbacks and opinions that will help in forming the explicit clustering constraints. Guided by the explicit and implicit clustering constraints derived from the previous steps, a constrained clustering algorithm is proposed to progressively derive cohesive clusters that are representative enough to serve as a high-level abstraction of the software design. Design and planning of the experiment are carried out before evaluating the proposed approach using 40 open-source OO software systems. Finally, the experimental data are analysed and interpreted to draw a general conclusion of the results. The overall framework of the research methodology is shown in Figure 3.1, where each phase consists of specific steps of the methodology.

3.2 Formulation Phase

The formulation phase consists of four major steps, beginning with the formulation of initial research questions and objectives. Next, existing studies related to software clustering, constrained clustering, and representation of software systems using weighted and un-weighted complex networks are investigated. Based on the investigation, the research questions and objectives are refined and reformulated. The following sub-sections discuss the specific steps involved in the formulation phase.

3.2.1 Formulation of Initial Research Questions and Objectives

The initial research problems are formulated in order to identify the scope of this research. The motivation behind conducting this research mainly is to help in recovering a high-level software design of poorly designed or poorly documented OO software systems through constrained clustering. This thesis focuses on utilising a constrained clustering approach that accepts pairwise constraints derived by domain experts or from the implicit structure of an OO software system to improve the accuracy and scalability of software clustering in deriving clusters with strong intra-cluster cohesiveness and inter-cluster separateness. The derived implicit and explicit constraints are formulated in the form of pairwise constraints, such that for a pair of classes (i, j), where if $(i, j) \in ML$ (respectively, if $(i, j) \in CL$), then (i, j) must belong to the same cluster (respectively, to different clusters).

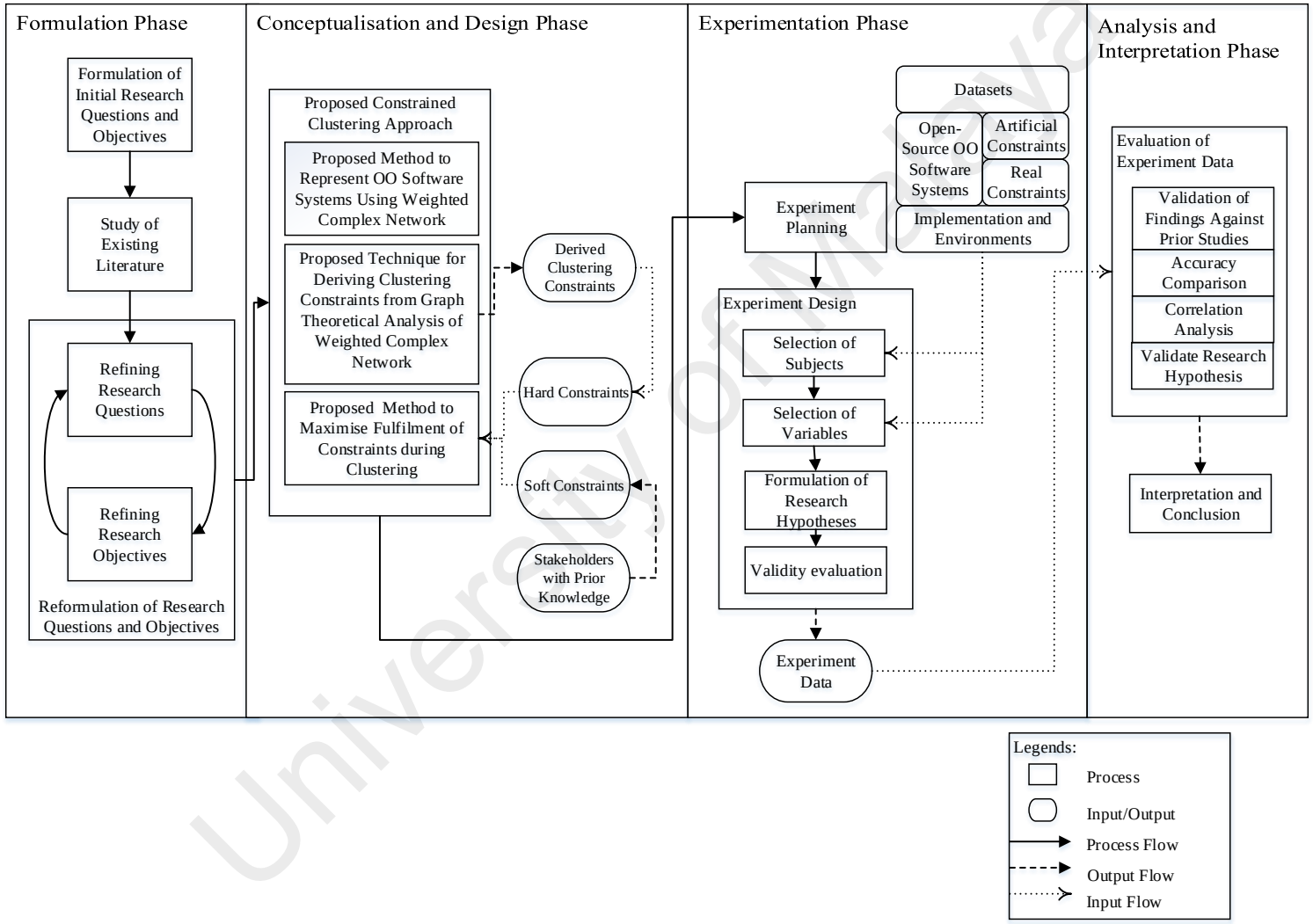


Figure 3.1: Research methodology framework

3.2.2 Study of Existing Literature

Existing studies in the area of constrained agglomerative hierarchical software clustering and representing software systems using un-weighted complex networks (LaBelle & Wallingford, 2004; Myers, 2003; Valverde & Solé, 2003) (Hamilton & Danicic, 2012; Hyland-Wood et al., 2006; Jenkins & Kirk, 2007; Taube-Schock et al., 2011) and weighted complex networks (Giulio Concas et al., 2007; Lan et al., 2010; Ma et al., 2010; Sun et al., 2009; Wang & Lu, 2012) are investigated. The challenges and issues of existing studies are highlighted in Section 2.6. Based on the studies, it can be summarised that although the idea of constrained agglomerative hierarchical software clustering is technically feasible to be used for helping in recovering a high-level abstraction view of OO software design, there are not many studies that explicitly discuss how to extract clustering constraints from software systems. One of the main reasons is because clustering constraints are not readily available most of the time and it is difficult to derive clustering constraints directly from the software itself (Harman et al., 2012). Although it is possible that software maintainers or domain experts may have prior knowledge on the software to be maintained, manually retrieving clustering constraints from them is a human-intensive and challenging task because there is a huge number of possible constraints configurations involving pairs of classes. This leads to the emergence of using weighted complex network to gain a high-level understanding of software systems. The analysis of the existing literature provides a wider perspective of the problems in constrained agglomerative hierarchical software clustering and representation of software systems using weighted complex networks.

3.2.3 Reformulation of Research Questions and Objectives

Based on the reviewed studies, the research questions are refined to focus on the challenges and issues in constrained agglomerative hierarchical software clustering. A total of six main research questions (RQ) are raised in this thesis.

RQ1: How to represent OO software systems using weighted complex networks?

- A way to represent OO software systems using weighted complex networks is part of the steps involved in the proposed constrained clustering approach before graph theoretical analysis can be performed to derive implicit clustering constraints. Hence this research question is aligned to RO1.

RQ2: When representing software systems using weighted complex networks, which measure constructs are capable of quantifying the weights of nodes and edges, while preserving the quality aspect of the software?

- This research question is aligned to RO2, where a method to automatically derive implicit clustering constraints is proposed with the aid of graph theoretical analysis. In order to ensure that the quality aspect of the software can be preserved, proper measure constructs need to be chosen to quantify the weights of nodes and edges of software-based weighted complex network.

RQ3: Is the constructed weighted complex network able to demonstrate the behaviour of real-world complex network commonly defined in existing studies?

- This research question is aligned to RO1 and RO2 to avoid intentional bias in the experimental results when constructing weighted complex networks using the proposed approach.

RQ4: How to effectively derive explicit constraints from domain experts, and implicit clustering constraints from the software itself?

- This research question is aligned to RO2 and RO3, which focus on how to derive clustering constraints from multiple sources of information, i.e. from the domain experts and from the implicit structure of the software system.

RQ5: How to handle different types of clustering constraints with various levels of importance, which might potentially conflict with each other and lead to the NP-Complete problem?

- This research question is aligned to RO4, in order to provide a way to maximise the fulfilment of explicit and implicit clustering constraints derived from domain experts and the implicit structure of the software itself.

RQ6: How to maximise the fulfilment of constraints during clustering without risking the "dead-end" situation as discussed by Davidson and Ravi (2009)?

- This research question is aligned to RO5, where the proposed approach will be evaluated using open-source OO software systems to verify if the fulfilment of constraints during clustering can be maximised without risking the "dead-end" situation.

The final stage of the Formulation Phase ends with refinement and reformulation of research objectives in order to solve the problems raised in the above-mentioned refined research questions. Sub-objectives based on the initial research objectives raised in Section 1.3 are formulated as follows.

Objective 1: To propose a constrained clustering approach with the aim to recover a high-level abstraction of OO software design that is coherent and consistent with the actual code structure.

Sub-objective 1.1: To develop a method for representing OO software systems using weighted complex network.

- Since the focus of this thesis is to recover a high-level abstraction of the software design of poorly documented OO software systems, classes are more suitable to be used as clustering entities where each class is represented as a node, while relationships between classes are represented as edges. The work by Wang and Lu (2012) and Ma et al. (2010) found that weighted representation of nodes and edges of a complex network is more capable of capturing the behaviour and characteristics of real-world software systems. However, there is a lack of research that distinguishes different types of relationships connecting two classes. Instead, most studies that use UML class diagram as a basis for representing complex networks do not differentiate between the types of relationships, but assume that all relationships are equivalent. As such, semantic information between classes may be lost when transforming UML class diagrams into complex networks. Thus, the complexity of classes and relationships must be taken into consideration when representing an OO software system by using a weighted complex network. In order to limit the scope of this research, only the maintainability and reliability of OO software systems are taken into consideration because these two software qualities contribute directly toward measuring the quality of clustering results.

Sub-objective 1.2: To identify appropriate measure constructs that are capable of quantifying maintainability and reliability of software systems represented in weighted complex networks.

- Since the focus of this research is to create a high-level abstraction view of the software design that is coherent with the actual code structure, the constructed weighted complex network must be representative enough to demonstrate the modularity of the analysed software systems. Maintainability and reliability for instance, are two software qualities that contribute directly toward estimating the modularity of a software system. Therefore, suitable measure constructs focusing on maintainability and reliability need to be chosen to quantify the weights of edges and nodes in weighted complex networks.

Sub-objective 1.3: To investigate the correlation between the statistical patterns of real-world OO software systems and their level of maintenance efforts.

- As discussed by Giulio Concas et al. (2007), several complex network properties can be observed from software-based complex networks, such as the power law and small world properties. However, the work by Concas et al. was only tested on single software, called the VisualWorks Smalltalk, which limits the generalisation of the research findings. In this thesis, the proposed approach must be designed to be generic and flexible enough to be applied on any kind of OO software systems. Since the selection of test subjects are closely related to the generalisation of the results, they must be representative enough to reflect the behaviour of different OO software systems (Wohlin et al., 2012).

Objective 2: To propose a method that helps in deriving implicit clustering constraints from the implicit structure of OO software systems with the aid of weighted complex network and graph theoretical analysis.

- Several graph theory metrics, such as in-degree, out-degree, average weighted degree, and average shortest path length, are able to reveal the quality of software systems from a graph theory's point of view. Detailed discussions have been presented in Section 2.4. A way to convert the results of graph theoretical analysis into implicit clustering constraints needs to be identified.

Objective 3: To propose a method that is capable of deriving explicit clustering constraints from domain experts or software developers who have prior knowledge regarding the software systems.

- Explicit constraints can be derived explicitly from domain experts or software developers by asking them to make judgment whether two classes should or should not be clustered into the same group. However, there are certain cases where the domain experts are not assertive enough to judge whether the given explicit clustering constraints are absolute, especially in the domain of software engineering. For instance, software developers who were involved in the early stage of software design might provide some explicit constraints about the software to be maintained. However, such constraints might not be valid anymore after several phases of software updates and changes. Thus, the explicit constraints given by the aforementioned domain experts might be misleading or contain erroneous information. Therefore, a proper method is needed to distinguish between absolute constraints and optional constraints, and subsequently fulfil those explicit constraints according to their level of importance.

Objective 4: To formulate an appropriate objective function that maximises the fulfilment of explicit and implicit constraints, while penalising violation of the constraints.

- Based on the discussed studies, clustering constraints can be originated from two sources of information, i.e. explicit feedbacks from the domain experts or implicit structure of the software itself. However, how to systematically identify and fulfil the clustering constraints remains as a research problem.
- Based on the work by (Ares et al., 2012; Basu et al., 2004; Hong & Yiu-Ming, 2012), the authors discussed that given enough information, clustering constraints can be categorised into hard and soft constraints that vary according to their level of importance. Subsequently, an objective function can be defined in order to maximise the fulfilment of all the hard and soft constraints.
- This objective involves identification of an appropriate technique to categorise clustering constraints based on their level of importance, and remove conflicting constraints. As discussed in Section 2.3.5, the following steps summarise the agglomerative hierarchical clustering algorithm.

Input: Set $T = \{x_1, x_2, \cdots, x_n\}$ of entities.

Output: Dendrogram

1. Each entity x_i forms an initial cluster G_i. The total number of clusters $K = n$. For each pair of clusters G_i and G_j, $i \neq j$, the distance between G_i and G_j is denoted by $d(G_i, G_j)$.
2. Find a pair of clusters with minimum distance, in $\{d(G_i, G_j)\}$:

   Let $d(G_a, G_b) = min\{d(G_i, G_j)\}$,where min returns the minimum distance value over the set of candidates in $\{d(G_i, G_j)\}$.

   Merge $G_c = G_a \cup G_b$ and reduce the number of clusters K= K-1.
3. If K = 1, stop the iteration; else update distance $d(G_c, G_j)$, for all other clusters G_j. (Go to Step 2)

- The "dead-end" situation would occur in Step 2 if there is no pair of clusters with minimum distance that can be found which causes the clustering process to stop prematurely. An appropriate method needs to be identified in order to train the distance matrix prior to merging of cluster entities, in order to avoid the "dead-end" situation.

Objective 5: To evaluate the accuracy and scalability of the proposed approach using open-source OO software systems.

- Open-source software systems written in Java programming language are chosen in this thesis. In order to improve the generality of the research findings, the chosen test subjects must vary according to their application domains, number of classes, and total lines of codes.

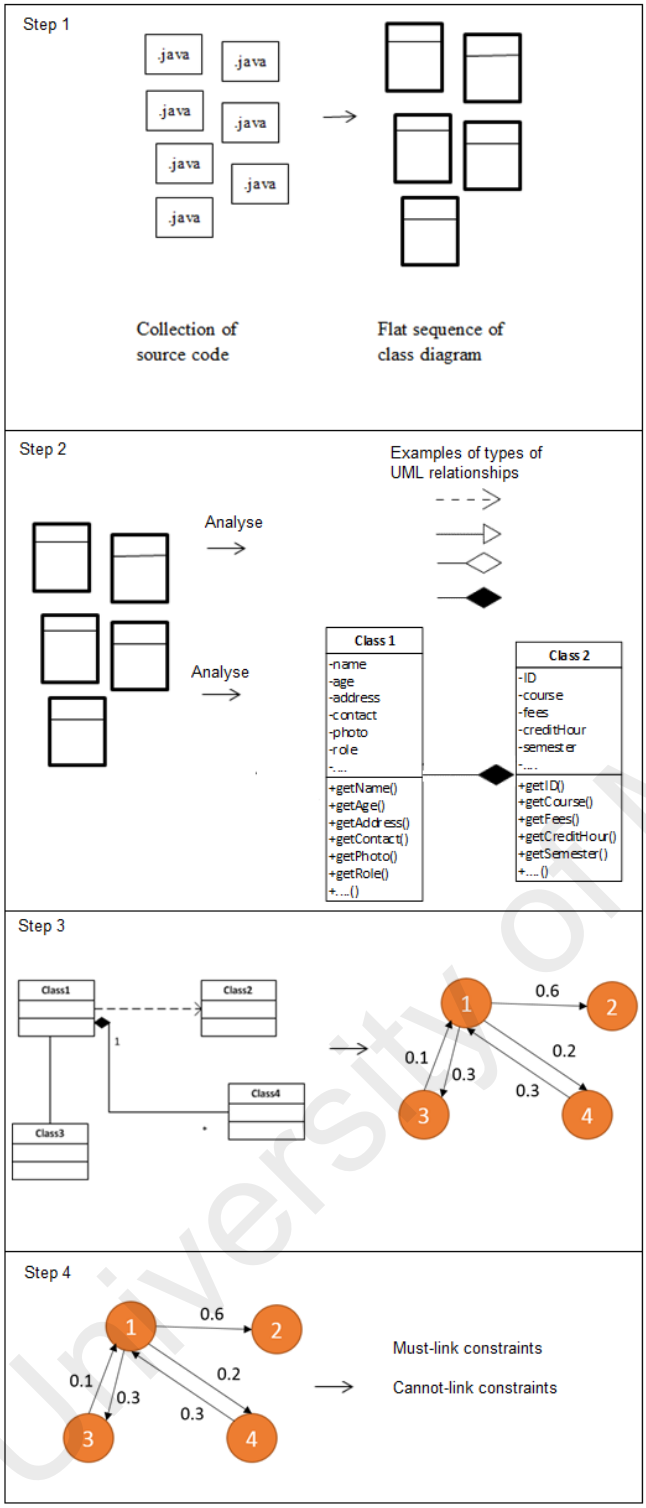
3.3 Conceptualisation and Design Phase

In this phase, a constrained clustering approach is conceptualised and designed. The constrained clustering approach is based on two methods to help derive clustering constraints and to maximise its fulfilment based on an objective function. The design decision as well as the steps involved in each method is discussed in detail in Chapter 4 and 5.

In order to achieve the research objectives (RO), a method along with a technique is proposed as illustrated in Figure 3.2. The first three steps illustrate the method to represent OO software systems using weighted complex network. Next, based on the constructed weighted complex network, a technique to derive clustering constraints based on graph theoretical analysis of the constructed network is introduced in Step 4. Collectively, both method and technique introduced are responsible for addressing RO1, RO2 and RO3. In Steps 1-3, the proposed method involves several steps to transform an OO software system into its respective weighted complex network. Next, the transformed weighted complex network is analysed and evaluated with respect to maintainability and reliability, and the results of the analysis are converted into clustering constraints in Step 4.

3.3.1 Proposed Method to Represent OO Software Systems Using Weighted Complex Network

The steps of the proposed method are briefly explained below.



Task	Method
Convert source code into UML class diagrams	Reverse engineering using off-the-shelf round trip engineering tool

Task	Method
Analyse complexity of UML relationships	Rank based on ordinal scale
Analyse complexity of UML classes	Quantify using established software metrics

Task	Method
Convert UML class diagrams into weighted complex network	Based on analysis of Step 2

Task	Method
Extract implicit constraints from the generated complex network	Apply graph theory metrics
Evaluate maintainaiblity and reliability of the analysed software	Apply graph theory metrics

Figure 3.2: Proposed method to represent OO software systems using weighted complex network along with a technique to derive implicit clustering constraints based on the constructed network.

Step 1: The software maintainer can provide either the source code or UML class diagrams as the input. If raw source code is the only reliable resource available, it is transformed into UML class diagrams using an off-the-shelf round-trip engineering tool such as Visual Paradigm or Eclipse Modelling Tool.

Step 2: The complexity of UML classes and their associated UML relationships are studied. Complexity of UML classes is quantified with the aid of established software metrics (Martin, 1994; McCabe, 1976), while the relationships are quantified based on an ordinal scale that ranks the relative complexity of each relationship. The motivation and justification of this step are discussed in the subsequent chapter.

Step 3: Next, based on the analysis performed in Step 2, the UML class diagrams are converted into a weighted and directed complex network, where each UML class is represented as a node, while each relationship is represented as an edge that connects a pair of nodes. Weighted and directed network is more suitable to represent the asymmetric behaviour of OO software systems, as discussed in Section 2.5.

3.3.2 Proposed Technique for Deriving Implicit Clustering Constraints based on Graph Theoretical Analysis of Weighted Complex Network

Next, based on the weighted complex network generated from Steps 1-3, a technique to derive implicit clustering constraints using graph theoretical analysis of the constructed network is introduced in Step 4 in Figure 3.2.

Step 4: Several graph theory metrics are chosen to analyse the software from a graph's theoretical point of view. The graph theoretical analysis is used to aid in providing a high-

level understanding of the software and evaluate it from the quality aspects of maintainability and reliability. Finally, the results are translated into a set of implicit clustering constraints that aid in the subsequent steps of the constrained clustering approach.

3.3.3 Proposed Method to Maximise Fulfilment of Implicit and Explicit Clustering Constraints

The method proposed in Figure 3.3 is responsible for fulfilling RO4 raised in Section 3.2.3, which is to maximise the fulfilment of clustering constraints. Figure 3.3 depicts the steps involved in the proposed method.

Step 1: Clustering constraints are gathered from two main sources: from the graph theoretical analysis of the software (implicit constraints) and from the domain experts who possess domain knowledge of the software (explicit constraints). Clustering constraints gathered from the graph theoretical analysis are categorised as hard constraints because they are derived from one of the most reliable sources of information (the structure and behaviour of source code). Clustering constraints from domain experts, on the other hand, are categorised as soft constraints because this set of constraints are usually imprecise and fuzzy in nature, which might contain erroneous information (Bagheri, Di Noia, Ragone, & Gasevic, 2010). A clustering result would still be acceptable if some of the soft constraints are not fulfilled, with the condition that it falls within an acceptable threshold. Fulfilling a handful of higher important soft constraints might overshadow the fulfilment of several less important ones. Thus, soft constraints must be prioritised and ranked based on their level of importance in order to create a

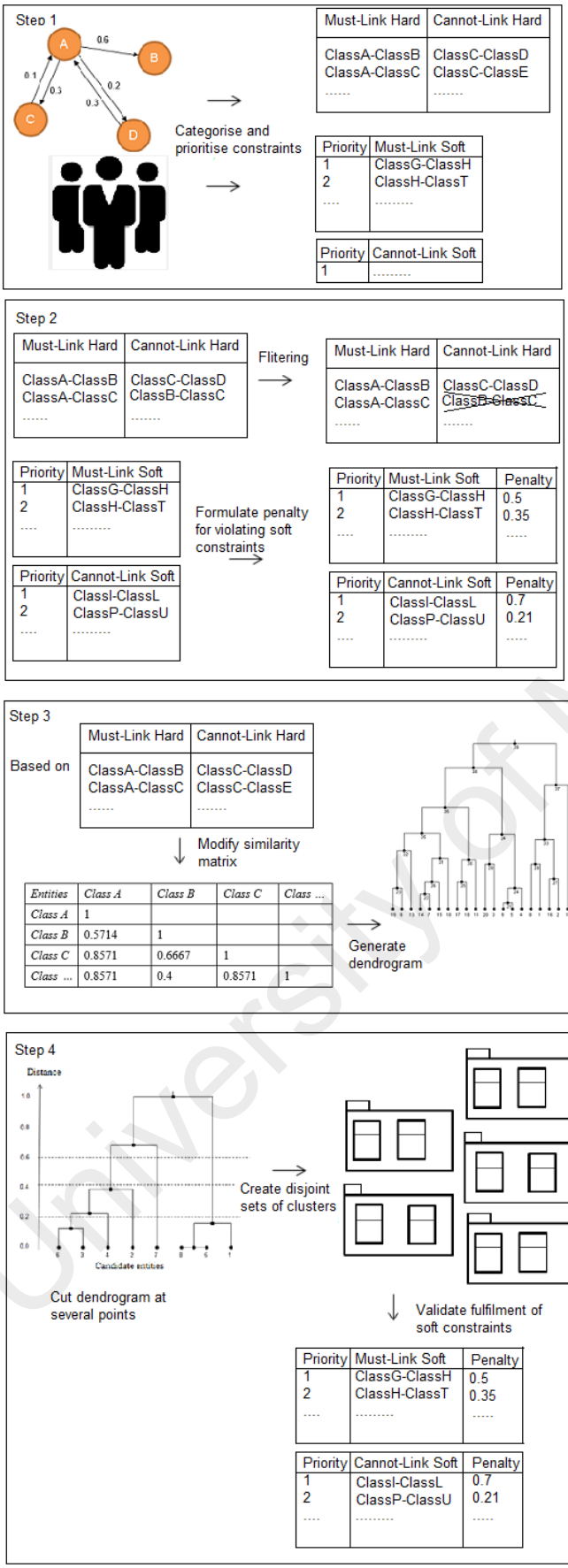
baseline for the said threshold. The soft constraints are prioritised and ranked using Multi Criteria Decision Making Method (MCDM).

Step 2: Clustering constraints that are conflicting with each other are identified and removed to prevent the NP-Complete problem. The penalty score for violating each soft constraint is formulated based on the result of MCDM in the previous step. The penalty score allows software maintainers to evaluate the quality of each clustering result with respect to the fulfilment of soft constraints.

Step 3: The similarity matrix of the associated hard constraints is modified using the distance-based method proposed by (Klein et al., 2002) in order to prevent the "dead-end" situation. Based on the modified similarity matrix, the associated dendrogram is generated using conventional agglomerative hierarchical clustering algorithm.

Step 4: The dendrogram is cut at several points to create several sets of clustering results. Each clustering result is evaluated based on intra-cluster cohesion, inter-cluster separateness, and the number of fulfilled soft constraints. Based on the evaluation criteria, the most optimum cutting point is chosen to recover a high-level abstraction of the software design. The final results are illustrated in several disjoint sets of clusters, where each set of clusters bears a resemblance to one subsystem. The clustering results can help in remodularisation of software systems through better comprehension of software design, and alert software maintainers regarding the risk of classes that violate common software design principles.

Next, based on the methods proposed in the Conceptualisation and Design phase, a proper experimental plan is drafted to evaluate its effectiveness.



Task	Method
Extract and gather clustering constraints	Retrieve from stakeholders and based on analysis of the weighted complex network
Categorise constraints into Hard or Soft constraints	Based on the confidence level
Prioritise constraints based on their importance	Multi Criteria Decision Making Method (MCDM)

Task	Method
Identify and filter conflicting constraints	Remove constraints that belong to both Must-Link Hard and Cannot-Link Hard constraints
Formulate penalty scores for violating soft constraints	Based on the result of MCDM in Step 1

Task	Method
Modify similarity matrix of cluster entities	Distance-based method to manipulate clustering process (Klein et al., 2002)
Generate dendrogram based on the modified similarity matrix	Apply agglomerative hierarchical clustering algorithm

Task	Method
Create several disjoint sets of clusters	Cut the dendrogram at several points
Maximise fulfilment of soft constraints	Evaluate quality of each cutting point Evaluate the number of fulfilled soft constraints Minimise penalty of violating soft constraints

Figure 3.3: Proposed method to maximise fulfilment of software clustering constraints

3.4 Experimentation Phase

Due to the fact that this research follows an experimental and exploratory study, the selection of subjects and variables as part of the experiment design must be representative of the real-world scenario to draw a general conclusion. Therefore, 40 open-source OO software systems that vary according to the size of project and application domain are chosen in order to address RO5. Besides, two types of clustering constraints including real constraints derived from the implicit structure of the software systems and artificial constraints are used for evaluation purposes. The artificial constraints are used to test the effectiveness of the proposed method when handling erroneous information that might compromise the accuracy of the clustering results. Besides the two aspects of the experiment design, two research hypotheses are defined to validate between the speculated observation and the results of the proposed constrained clustering approach. Finally, several threats to internal validity and external validity, along with the countermeasures to mitigate these threats are discussed.

Based on the experimental design and setup, the proposed methods are tested on the 40 open-source software systems. The resulting experimental data are the final output in the Experimentation Phase.

3.5 Analysis and Interpretation Phase

The work by Anquetil and Lethbridge (1999b) discussed that instead of recovering a software system's architecture, clustering techniques actually create a new one based on the parameters and settings used by the clustering algorithm. Thus, a way to evaluate the effectiveness of the produced result is needed. MoJoFM is a well-established technique

used to compare the similarity between the clustering results and gold standard (Wen & Tzerpos, 2004). Gold standard in this context refers to a known good clustering result or reliable reference that can act as a baseline comparison. High similarity between the clustering result and the gold standard is more desirable as it indicates that the produced clustering result resembles the gold standard. In order to evaluate its effectiveness, the results of the proposed constrained clustering approach are compared against prior studies related to software clustering and also the gold standard. MoJoFM is used as a tool to evaluate the accuracy of the proposed constrained clustering approach. Furthermore, several descriptive statistics and plotting techniques are adopted to find the correlation between several qualities attributes of software systems (in terms of maintainability and reliability) and the findings of graph theoretical analysis. The hypotheses defined in the previous phase are also validated through the experimental data. Finally, the conclusion is presented and discussed.

3.6 Chapter Summary

This chapter has explained the research methodology of this research including the proposed constrained clustering approach, guided by a technique and two supporting methods. The methods and technique are proposed based on the research questions objectives discussed in Section 3.2.3. The first method attempts to transform an OO software system into a weighted complex network, while preserving the maintainability and reliability aspects of the analysed software. Next, a technique is introduced by applying several graph theory metrics that are related to these software quality attributes onto the transformed weighted complex network in order to identify highly reusable classes, important classes that contain the main functional modules, classes that are more prone to bugs and errors, and the static and dynamic relationships between all the classes.

The graph theoretical analyses of the software are then translated into clustering constraints, which are used as the input for the second proposed method. The aim of the second method is to maximise the fulfilment of different clustering constraints (hard and soft constraints), while avoiding the NP-Complete and "dead-end" problems stated in the prior studies. In-depth details of all the proposed methods and technique will be discussed in the subsequent chapters. Furthermore, the experimental design and evaluation strategies will also be discussed in-depth in a later chapter. A wide range of datasets including 40 open-source software systems, real and artificial clustering constraints were used to evaluate the quality of the proposed constrained clustering approach.

CHAPTER 4: DERIVING IMPLICIT CLUSTERING CONSTRAINTS FROM WEIGHTED COMPLEX NETWORK TRANSFORMED FROM UML DIAGRAMS

In this chapter, a method to represent an OO software system into a weighted complex network is proposed. With the aid of the weighted complex network, the method aims to analyse the software system from the aspect of maintainability and reliability. Maintainability and reliability are chosen in this research because they contribute directly toward estimating the modularity of classes (nodes of the constructed weighted complex network) that can aid in revealing some implicit information regarding the interactions among all the associated classes. Next, a technique to automatically derive implicit clustering constraints is introduced by applying several graph theory metrics onto the constructed weighted complex network. With the aid of these metrics, software maintainers are able to analyse the software from a graph's theoretical point-of-view in order to reveal some extra deterministic information about the analysed software. Finally, the results from the analysis are converted into implicit clustering constraints.

4.1 Representing Software Systems with Weighted and Directed Complex Networks

Based on the current research scenario, the approaches for representing nodes and edges in weighted complex networks are still not well defined for software based on UML class diagrams. The edges signify direct relationships between two nodes representing classes where the edges can be associated with some weightage values to denote the communicational cohesion of the nodes connected by the edges. Communicational cohesion in this context refers to classes that share similar characteristics and behaviour,

or classes that perform a certain operation on the same input or output data (Stevens, Myers, & Constantine, 1979). The notion of communicational cohesion in complex network can be associated with UML class diagram, where two classes with high communicational cohesion indicate that they share similar functionalities and there is a high tendency for these two classes to be placed into the same software package.

For instance, in the work by Wang et al., edges are weighted based on the number of method callings between classes. A higher value of weight associated to an edge signifies a higher communicational cohesion between the associated classes because it is an indication that these two classes might belong to the same package. On the other hand, one can also use distance as the basis to derive the weights instead of using communicational cohesion to indicate the dissimilarity between the associated classes. A typical way to convert communicational cohesion to distance measure is by calculating the inverse of the strength of communicational cohesion (Cilibrasi & Vitanyi, 2007). For example, if the weight of an edge (in terms of communicational cohesion) between two nodes is in the range of [0, 1], one can convert it to a distance value by computing 1– strength of communicational cohesion. There are diverse ways to transform the information observed from UML class diagrams.

The greater the weight of an edge (in terms of communicational cohesion), the more dependency exists between the two classes. For instance, given two classes *A* and *B*, where there exists one method in class *A* that passes messages associated to three methods in class *B*. Therefore, the weight of the edge that is linking classes *A* and *B* is assigned as 3. Such approach of transforming software systems into weighted complex networks can be observed in the work by (Guoai, Yang, Fanfan, Aiguo, & Miao, 2008; Yang, Guoai, Yixian, Xinxin, & Shize, 2010; Yang, Jia, Shuai, Guoai, & Gong, 2013). However, certain

semantic behaviour and relationships of class diagrams cannot be captured using this naive transformation. For example, classes related with inheritance relationships and classes related with common association, in this case, are assumed to have equal strength of communicational cohesion, which is illogical from the software engineer's point of view.

As mentioned earlier, the work by Ma et al. (2010) explores the possibility of representing a UML class diagram as a directed complex network to analyse the relationships between classes at different levels of abstraction. However, the construction of edges does not consider the different kinds of relationships connecting multiple classes or the weighted values of edges. Instead, the authors assumed those edges to be equivalent weight.

The same assumption can also be observed in other works that relate software with complex network (Giulio Concas et al., 2007; Myers, 2003; Valverde & Solé, 2003). In the work by Giulio Concas et al. (2007), Myers (2003), Valverde & Solé, (2003), all kinds of interclass relationships such as inheritance, composition, and generalisation are simplified and represented as common dependencies. This can be a poor assumption because different types of relationships in software such as dependency, association, composition, and aggregation denote different degrees of communicational cohesion and structural complexity. Figure 4.1a shows a scenario where two classes are related with generalisation, and Figure 4.1b shows the same classes related with common association. In general, classes that are related with generalisation signify a strong parent and child class relationship. This is because any changes in the parent class will directly affect the child class. Removal of the parent class would render the child class unusable. Classes related with generalisation should have a higher degree of communicational cohesion when compared to classes related with common association. Thus, the strength of

communicational cohesion between classes in Figure 4a should behave differently when compared to that in Figure 4b. In this case, this is shown in the transformed nodes and edges with random weighted values of 0.8 and 0.2 respectively.

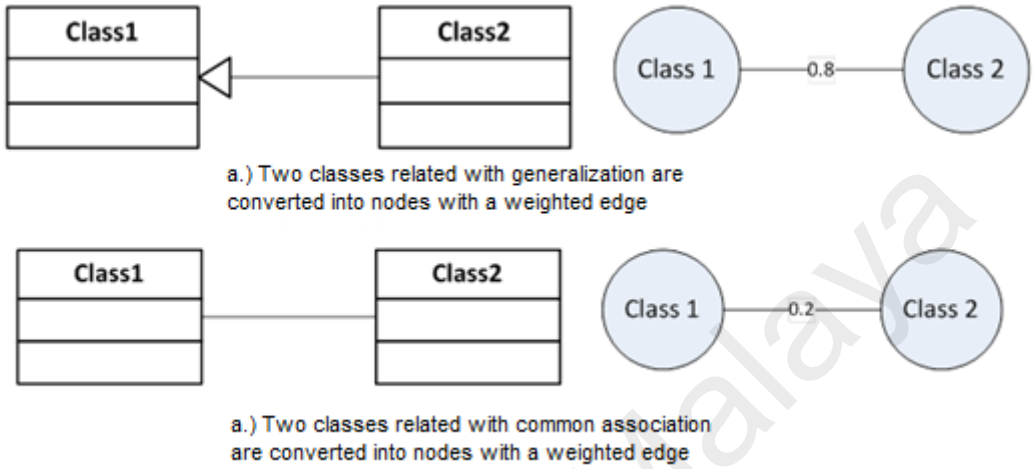


Figure 4.1: Example of UML classes related with different relationships

In addition, the complexity of a class can also affect the complexity of a relationship. For instance, two simple classes related with common association should have a different strength of communicational cohesion when compared to two complex classes related with the same type of relationship. An example is shown in Figure 4.2 to depict the aforementioned scenario. Given two complex classes shown in Figure 4.2b that consist of hundreds of methods and variables. If the interactions between these classes are simply passing a few parameters, then the strength of communicational cohesion will be insignificant. On the other hand, if the classes are very well designed, simple and only contain two methods and variables, as depicted in Figure 4.2a, then the strength of communicational cohesion between the two classes will be much stronger.

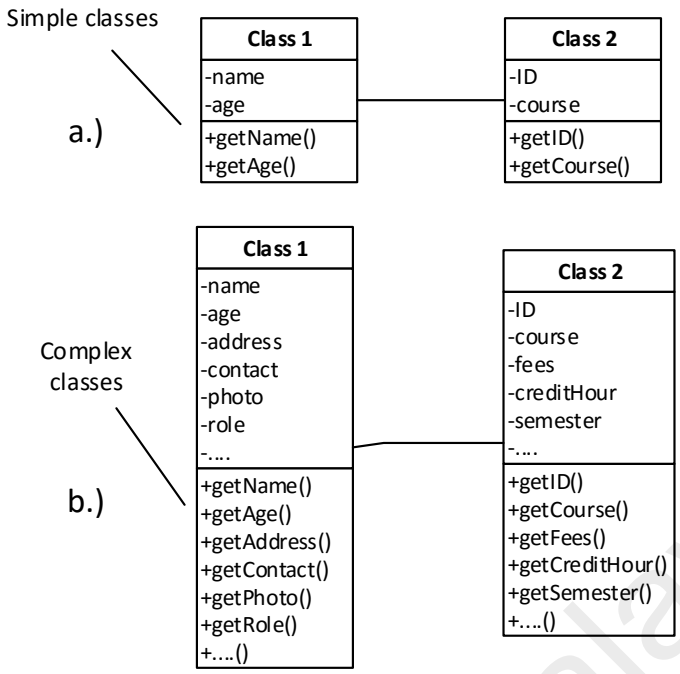


Figure 4.2: Example of UML classes with different class complexity

Thus, in general, the type of relationship and complexity of classes can influence the degree of communicational cohesion between classes. This is important because the degree of communicational cohesion can subsequently affect the weights of edges in complex networks. Based on the studies in the previous chapter, there is a lack of attention in formulating a proper way to calculate the weights of edges when transforming UML class diagrams into weighted complex networks.

4.2 Weighting Nodes and Edges in UML Class Diagram-based Complex Networks

A method is proposed to represent the weights of nodes and edges of a complex network transformed from UML Class diagrams. If only the raw source code is available, software maintainers can convert the code into UML Class diagrams using an off-the-shelf round-trip engineering tool for the ease of conversion. Justifications of the selected measure constructs are discussed before introducing the proposed method.

4.2.1 Measuring the Structural Complexity of UML Relationships and Classes

In the work by Dazhou, Baowen, Jianjiang, and Chu (2004), the authors defined "structural complexity" as the global metric to evaluate the complexity of UML class diagrams. According to the authors, structural complexity can integrate multiple class diagram metrics, such as metrics to evaluate individual classes, metrics to evaluate interaction between classes, and metrics to evaluate the whole class diagram.

In this study, the proposed weighting mechanism is based on two parameters, the complexity of classes and the complexity of relationships. Relationships (dependency, realisation, association, etc.) are taken into consideration because each end of the relationship must be linked to a certain class. This implies that the complexity of relationship has direct implication toward measuring the complexity of classes. In order to measure the complexity of relationships, the authors introduced a conceptual idea to assign different weightage depending on the type of relationship as shown in Table 4.1.

Table 4.1: Ordering of class diagram relationships proposed by Dazhou et al. (2004)

No.	Relation	Weight
1	Dependency	H1
2	Common association	H2
3	Qualified association	H3
4	Association class	H4
5	Aggregation association	H5
6	Composition association	H6
7	Generalisation (concrete parent)	H7
8	Binding	H8
9	Generalisation (abstract parent)	H9
10	Realise	H10

The table is arranged in an ascending order of weight values. Since the complexities of different relationships are relative with each other, arbitrary values of 1–10 are

respectively given to H1-H10. Based on this ordinal scale, software maintainers can compare the complexities between different kinds of relationships in UML class diagram. Empirical testing using real open-source software has been demonstrated in (Chong et al., 2013) based on the ranking in Table 4.1.

The work by Hu, Fang, Lu, Zhao, and Qin (2012) also proposes to rank UML classes and relationships using an ordinal scale based on PageRank algorithm. The purpose of the ranking is to differentiate the importance of associated UML classes based on their inherent characteristics and relationships with other classes. However, Fang et al. only addressed three types of UML relationships in the following order:

Composition > Aggregation > Association

Similarly the research conducted by (Briand, Labiche, & Yihong, 2001, 2003) also involves the ranking of relationships in UML class diagram. Briand et al. mentioned that one of the most important problems during integrating and testing OO software is to decide the order of class integration. The authors proposed a strategy to minimize the number of test stubs to be produced during software integration and testing phase. Relationships are ranked based on their complexities, where the most complex relationships (i.e. inheritance relationships) are integrated first. Common associations are perceived as the weakest links in class diagram and placed at the lowest hierarchy during software integration and testing phase. The discussed works (Briand et al., 2001, 2003; Hu et al., 2012) only compare three major types of relationships, namely inheritance (generalisation and realisation), composition (aggregation), and common association. The concept of ordering of relationships in UML class diagram based on their complexities is similar to the aforementioned work. Thus, the notion of ordering UML class relationships

in an ordinal scale, and subsequently identifying the importance or complexity of classes, is suitable to be used in this research as a basis of measuring the weights of the edges.

However, if multiple classes are related with the same kind of relationships, the weighting mechanism must be able to distinguish this difference. For example, two relatively simple classes linked with generalisation (H7) should exhibit a different weightage when compared to two complicated classes linked with the same type of relationship. Thus, the complexity of classes plays an important role to make a distinction between these two cases. However, the proposed structural complexity metric in Dazhou et al. (2004) is a conceptual idea without a proper evaluation.

This thesis attempts to integrate the concept of structural complexity in order to convert UML class diagrams into weighted complex networks. The OO abstraction from a UML class diagram will be represented as nodes and edges in a weighted complex network. The weight of an edge is calculated based on the strength of communicational cohesion between the connected nodes, which will be discussed in the next section. The directionality of edges also plays an important role to indicate a one-way relationship from the origin node to the terminus node, not vice versa. Thus, it is important that the directionality of edges is captured and analysed properly in order to provide a better understanding of the analysed software.

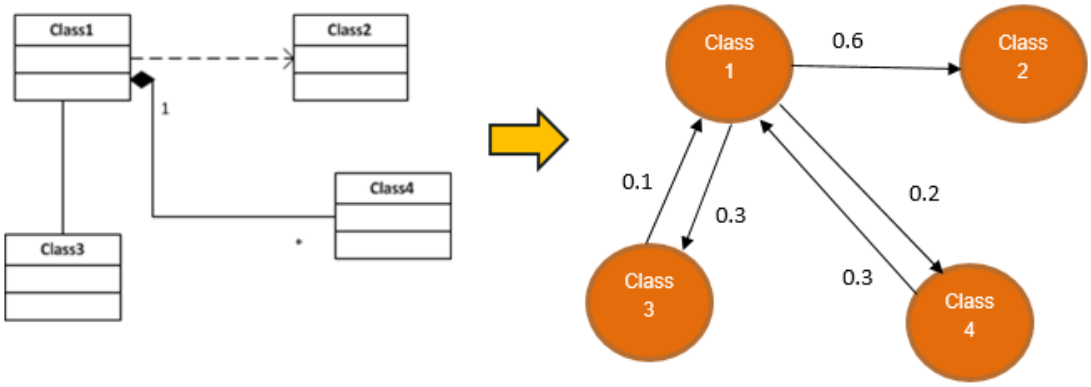


Figure 4.3: Illustration of converting a UML class diagram into a weighted complex network

As shown in Figure 4.3, a class diagram, $D = [D_1, D_2, \cdots D_n]$, consists of a set of *n* classes, $D_1, D_2, \cdots, D_n$. The aim of the proposed method is to represent OO software systems using weighted complex networks in order to provide a graph abstraction view of the software. This will facilitate the application of well-established graph theory metrics onto the constructed weighted complex network. Classes are represented as nodes while relationships among classes are represented as edges connecting a pair of nodes. Relationships in class diagrams can impose a one-way or a bidirectional relationship, which needs to be interpreted in advance. As such, the transformation rules introduced by Dazhou et al. (2004) are adopted which converts association, composition, and aggregation into bidirectional relationships in a weighted complex network. Since relationships such as generalisation, realisation, dependency and binding usually impose a one-way relationship in the model-driven architecture (MDA) perspective, they will remain as a single directed edge that links two nodes.

A relationship R connecting two classes, $R = (D_i, D_j)$, where $D_i, D_j \in D; i \neq j$, R links D_i to D_j where D_i is the origin of the relationship and D_j is the terminus. R carries a weight which denotes the strength of this relationship. The weight of relationship R,

which denotes the strength of the communicational cohesion between classes D_i, D_j depends on:

1. The complexity of relationship R
2. The complexity of classes D_i, D_j linked by R.

4.2.2 The Complexity of Relation R

The example below explains the details in calculating the weight of a given relationship R.

Given a class D_i that depends on class D_j through a one-way relationship R, such that $D_i \neq D_j$. The complexities of class D_i and class D_j are $Comp_{(i)}$ and $Comp_{(j)}$ respectively. Since this is a one-way relationship and D_i is dependent on D_j, the complexity of class D_j will affect this relationship. For a bidirectional relationship, the weight will be calculated based on the average of both directions. By referring to Table 4.1, software maintainers can identify the relative complexity of relationship R and measure the weight of the relationship R between class D_i and D_j using the proposed equation formulated in Equation (1).

$$Weight_{(R_{i\rightarrow j})} = \left(H_{R_{i\rightarrow j}} * \alpha\right) + \left[\left(Comp_{(D_j)}\right) * \beta\right] \quad (1)$$

The first operand of Equation (1) denotes the complexity of relationship R while the second operand denotes the complexity of terminus class linked by R. H_R indicates the relative complexity of relationship R (by referring to Table 4.1). The complexity of a relationship H_R is relative to the other types of relationships in Table 4.1. It is more significant to identify the ranking of this relationship in terms of influence and

complexity. This can be done by assigning a relative weight in the range of [0, 1] to each relationship H_R based on its ranking. For example, given a relationship R = Dependency (H1), a relative weight of 0.1 is assigned to this relationship. α and β, in this context, carry the meaning of preferences and risk tolerance in obtaining the relative complexity of a the terminus class D_j. The preferences and risk tolerance parameters are used to relax the constraints on obtaining the complexity of the relationships and class. Since the ranking in Table 4.1 is presented in an ordinal scale, one can assign the weight of H1-H10 based on their own preferences. If users are not confident about the weight to be given on the relationship, more emphasis can be given on the complexity of the terminus class instead. Values of α and β range between 0 and 1, in such a way that a lesser value indicates a greater uncertainty in obtaining the complexity of the relationship and the terminus class linked by it. For example, if the value of H1-H10 cannot be retrieved easily, or users are not confident regarding the weight of relationship R, value of 0.2 can be assigned to α, while 0.8 on β to indicate that the complexity of terminus class linked by R carries more significance. Value of 0.5 for α and β will be used in this study to represent a balanced environment where both values can be obtained easily.

4.2.3 The Complexity of Classes D_i, D_j Linked by R.

For a one-way relationship, given D_i is dependent on D_j, only the complexity of terminus class D_j will be calculated. For a bidirectional relationship, the complexity of both origin and terminus classes will be taken into consideration. In order to measure the complexity of a class, the three-level metrics introduced by Ma et al are adopted. In the work by Ma et al. (2010), the authors categorised their metrics into three levels, namely code-level, system-level, and graph-level, in order to analyse the OO aspect of a particular system using different levels of granularity. The authors suggested that solely relying on a

particular level of metrics is not sufficient to measure the properties of OO software derived from source code or UML class diagrams. Hence, the proposed method in this thesis combines the information from both the raw source code and UML class diagrams of a software system to model its respective weighted complex network.

The code-level metrics, which are at the finest level of granularity, measure the code complexity. Examples of metrics used are SLOC, fan in and fan out, and cyclomatic complexity (Martin, 1994; McCabe, 1976). The system-level metrics focus on OO aspect of the system by measuring characteristics such as inheritance, coupling, cohesion, and modularity. Examples of metrics used are CK metrics (Chidamber & Kemerer, 1994). The CK metrics consist of six metrics as follows:

1. Coupling Between Object Classes (CBO)
2. Weighted Methods per Class (WMC)
3. Depth of Inheritance Tree (DIT)
4. Number of Children (NOC)
5. Lack of Cohesion of Methods (LCOM)
6. Response for a Class (RFC)

CBO measures the coupling of a given class by counting the number of dependencies of that particular class on other classes. WMC is the weighted sum of all the methods in a given class. DIT is based on the inheritance hierarchy by identifying the longest inheritance path for a given class. NOC of a given class is defined as the number of immediate child classes. LCOM, on the other hand, measures cohesion of a given class by inspecting the relationships between the methods declared in the class. Finally, RFC measures the number of methods that can be used by other classes through the associated messages.

The graph-level metrics are based on complex network to measure the global features and provide an overview of large-scale software systems. Examples of graph-level metrics are degree distribution and correlation coefficient.

While the work by Ma et al. (2010) aims to discover a variety of metrics and examine each of them individually, this thesis focuses on only one metric at each level, except for the graph-level metrics. The measurement of each level is then normalised locally. This will allow software maintainers to use a standardised metric with a common scale of unit that measures complexity as a whole. In the proposed method, code-level and system-level metrics are used to measure the complexity of a particular class. On the other hand, several graph-level metrics are used to examine the overall structure of the analysed software.

In order to select an appropriate metric, a survey on the existing software metrics was conducted in this research. Several studies that performed empirical studies of CK metrics on real-world software systems are chosen.

The work by Li and Henry (1993) examined the correlations of CK metrics with software maintenance effort. The software metrics were applied on two commercial software systems in order to predict maintainability. The authors found that except for CBO, CK metrics are able to effectively predict the software maintainability of real-world software systems. Another work by Binkley and Schach (1998) applied CK metrics on four software systems. The authors found that there is no correlation between NOC and the frequency of source code changes due to field failure. The inventors of CK metrics themselves applied the proposed metrics on three commercial software systems and found

that high LCOM was associated with lower productivity, high maintenance cycle, and higher maintenance effort (Chidamber, Darcy, & Kemerer, 1998).

In terms of fault proneness, the work by Lionel C. Briand, Wüst, Ikonomovski, and Lounis (1999) applied CBO, RFC, and LCOM on an industrial software and found that the three metrics are associated with defects found in the case study. Similar observations were found in the work by Olague, Etzkorn, Gholston, and Quattlebaum (2007b), where the authors applied the complete suite of CK metrics, MOOD metrics and QMOOD (Bansiya & Davis, 2002) metrics on Mozilla's Rhino open-source software. CK and QMOOD metrics are found to be good indicators for detecting error prone classes, while MOOD metrics are less effective. The work by Mei-Huei, Ming-Hung, and Mei-Hwa (1999) applied CK metrics on three industrial-grade real-time systems to identify the correlations between the software metrics and faults found during software maintenance. DIT and NOC were found to have almost zero correlations, while WMC and RFC are strongly correlated to error prone classes. Based on the discussed literature, it is clear that not all of the CK metrics are suitable to be used in this study. DIT and NOC in particular, were found to be less effective when measuring the software maintenance effort and fault proneness of software systems. WMC and LCOM, on the other hand, were found to be suitable in identifying error prone classes and estimating software maintenance effort, as claimed in (Chidamber et al., 1998) (Briand et al., 1999; Mei-Huei et al., 1999). Thus, WMC and LCOM are chosen to be used in this study, which aligns to RO1.2 stated in Section 3.2.3.

At the code level, WMC is chosen to measure the complexity of source code. WMC measures the average cyclomatic complexity of methods inside a class. Cyclomatic complexity calculates the number of independent paths through program source code

using the concept of directed network. A class that possesses high WMC value suggests that it is very complex and does not focus on its functionality.

At the system level, the Lack of Cohesion of Methods version 4 (LCOM4) (Hitz & Montazeri, 1995) is chosen, which is an extended metric based on the LCOM included in CK metrics set. LCOM4 is used to measure the cohesion of a particular class by inspecting the relationships between the methods and variables. LCOM (Chidamber & Kemerer, 1994), LCOM2 (McCabe, 1976), and LCOM3 (Henderson-Sellers, Constantine, & Graham, 1996) are less suitable for modern OO software systems because they do not evaluate the importance of mutator and accessor methods, which are widely used to encapsulate information in the OO paradigm. In LCOM4, both shareable and non-shareable variables and methods are taken into account to cater for encapsulated variables or data. A high value of LCOM4 suggests that correlations between methods inside a class are weak, and it is undesirable in common software engineering practices.

The choice of software metrics used in this study is not random, as it is based on previous research (Basili, Briand, & Melo, 1996; Olague et al., 2007b; Subramanyam & Krishnan, 2003a) that WMC and LCOM4 are complementary with each other when used to predict faults in OO software. Furthermore, the work by Ichii, Matsushita, and Inoue (2008) has found that there is a positive correlation between WMC and LCOM4 such that an increase in WMC will lead to an increase in LCOM4. This is mainly because a class will tend to become less cohesive when more functionalities or modules are added into the class. Furthermore, the focus of this study is to capture two particular software quality attributes, namely maintainability and reliability as discussed in RO1.2. Therefore, WMC and LOCM4 are combined and use an aggregated measure to determine the complexity of classes when representing an OO software system using a weighted complex network.

An example is given below to calculate the complexity of a particular class. Given a class D_j, LCOM4 and WMC of class D_j are represented as $L(D_j)$ and $W(D_j)$ respectively. The following equation is used to quantify the complexity of D_j.

$$Comp_{(D_j)} = \left(\widetilde{L(D_j)} * \alpha\right) + \left(\widetilde{W(D_j)} * \beta\right) \quad (2)$$

where $0 \leq \alpha \leq 1,\ 0 \leq \beta \leq 1$

$\widetilde{L(D_j)}$ and $\widetilde{W(D_j)}$ represent the normalised LCOM4 and WMC values respectively over all classes in the system using a ratio scale (value range between 0 to 1). Normalisation is needed in this case because both metrics are measured using a different scale of unit. The values α and β behave similarly to Equation (1) where it denotes the preferences and risk tolerance in obtaining the two metric values. Depending on the difficulty and confidence of obtaining the values $\widetilde{L(D_j)}$ and $\widetilde{W(D_j)}$, α and β can be manipulated accordingly. Thus, higher values signify higher complexity. However, before finalising the formula, it is important to determine if there is a direct correlation between complexity of classes and their respective strength of communicational cohesion.

The work by Satuluri and Parthasarathy (2011) proposes a degree-discounting symmetrisation method to analyse the contribution of each node to the strength of communicational cohesion based on its degree (in-coming and out-going edges). Suppose that two nodes *i, j*, both point to a node *z*, which has a high in-degree (Case 1 in Figure 4.4a), and two nodes *i, j*, both point to a node *z*, which has a low in-degree (Case 2 in Figure 4.4b). The authors proposed that the strength of communicational cohesion between node *i* and node *z* together with that of node *j* and node *z* contributes more in

Case 2 as compared to in Case 1 because node z of Case 2 has a low in-degree. The communicational cohesion between nodes is inversely proportional to the in-degree.

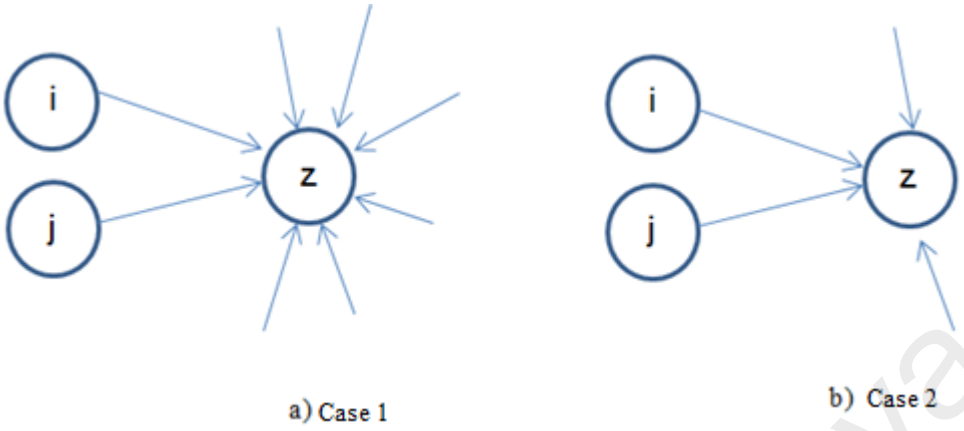


Figure 4.4: Degree-discounting symmetrisation based on Satuluri and Parthasarathy (2011)

The concept of degree-discounting symmetrisation can be applied to this study where the degree is represented as UML class complexity, $Comp_{(D_j)}$. Given $(D_i, D_j) \in R$, D_i is dependent on D_j, and the complexity of class D_j will affect the weight of R. If class D_j consists of 1000 methods and variables, and the interactions between D_i, D_j are simply passing a few parameters, the strength of communicational cohesion between D_i, D_j, or the tendency of D_i, D_j belonging to the same package will be insignificant. On the other hand, if class D_j is very well designed and focuses on its own functionality, then the strength of communicational cohesion between classes D_i, D_j will be much stronger. To provide a more concrete illustration, given for example, a complex and unorganised class *"Alpha"* which contains a lot of static methods and is constantly invoked by other classes from different software packages. As a result, cohesion of methods inside class *"Alpha"* will be very weak. Placing class *"Alpha"* into a suitable package will also be a challenging task because there are lesser distinct features and common behaviour shared by class *"Alpha"* with other classes. A common way to solve this problem is by

decomposing it into multiple modular classes that focus on their own functionality. The modular classes can then be grouped into packages that share the same functionalities. Thus, the complexity of class D_j is inversely proportional to the weight (strength of communicational cohesion) of R. Since the value $Comp_{(D_j)}$ is normalised into the value range between 0-1, the complexity can be inversed by using the formula $1 - Comp_{(D_j)}$. Thus, Equation (1) is updated to the following Equation (3) to measure the relationships between two classes.

$$Weight_{(R_{i\to j})} = \left(H_{R_{i\to j}} * \alpha\right) + \left[\left(1 - Comp_{(D_j)}\right) * \beta\right] \quad (3)$$

4.3 Overview of the Proposed Method to Represent Software Systems with the Aid of Weighted Complex Network

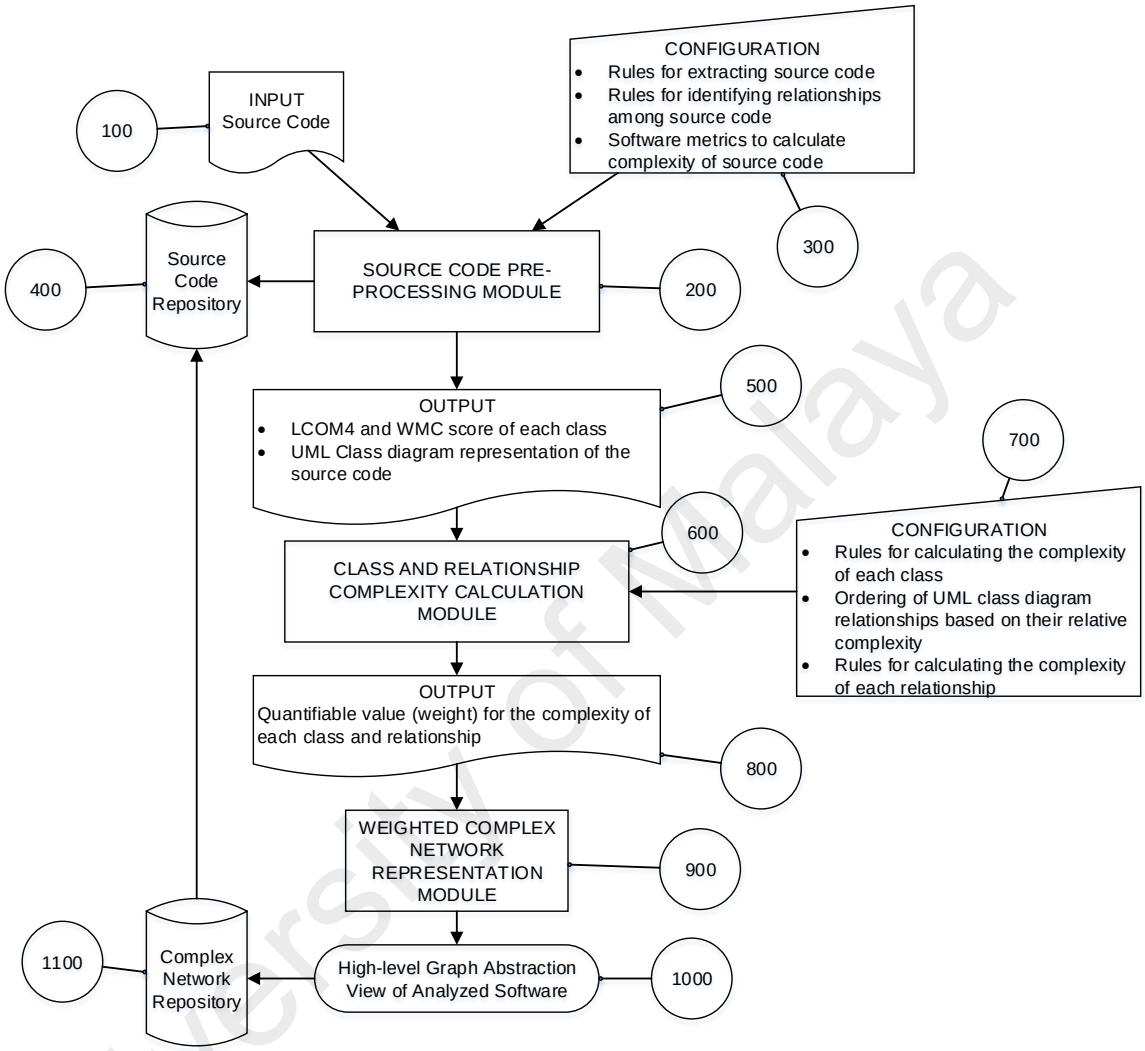


Figure 4.5: Flow chart of the proposed method to represent software systems with the aid of weighted complex network

The overall workflow of the proposed method is shown in Figure 4.5, which consists of three major software modules. The numbers ranging from 100-1100 correspond to the steps involved in the proposed method.

- 100: The user provides the source code to be analysed by the proposed method.

- 200: Source Code Pre-processing Module to extract essential information and evidence from the user's input before representing it with a weighted complex network.
- 300: Configuration data for the Source Code Pre-processing Module. The configuration data includes sets of rules to extract important classes and their relationships, and a pre-weighted set of rules to calculate the complexity of source code. The rules used are the WMC and LCOM4 mentioned in Section 4.2.3.
- 400: Input source code is stored in a repository.
- 500: Consist of the output of the Source Code Pre-processing Module.
- 600: The output from Step 500 is parsed into the Class and Relationship Complexity Calculation Module.
- 700: Configuration data for the Class and Relationship Complexity Calculation Module. The configuration data includes sets of rules to calculate the complexity of each class and relationship found in the UML class diagram. The calculation is based on Equation (3) derived in Section 4.2.3.
- 800: Consist of the output of the Class and Relationship Complexity Calculation Module.
- 900: The output from Step 800 is parsed into the Weighted Complex Network Representation Module.
- 1000: Generate a corresponding weighted complex network based on the analysed software system.
- 1100: The generated weighted complex network is stored in a repository. The weighted complex network is mapped back to the corresponding source code stored in Step 400 in order for cross referencing and further analysis.

The details of each software module (Step 200, 600, 900) are explained in the following paragraphs, with the aid of diagrams.

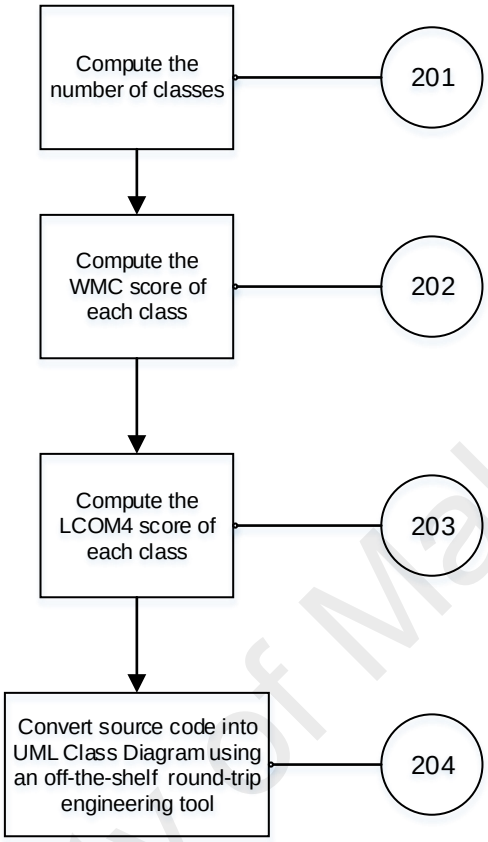


Figure 4.6: Details of Step 200 (Source Code Pre-processing Module)

- 201: The number of classes that correspond to the source code provided by the user, is recorded.
- 202: For each class in the source code, the WMC score is calculated and recorded.
- 203: For each class in the source code, the LCOM4 score is calculated and recorded.
- 204: The source code is converted into its equivalent UML class diagram using an off-the-shelf round-trip engineering tool such as the IBM Rational Rose, Eclipse UML tool or the Visual Paradigm suite of software. The final output of

Steps 201-204 are the LCOM4 and WMC score of each class, and the UML Class diagram representation of the given source code.

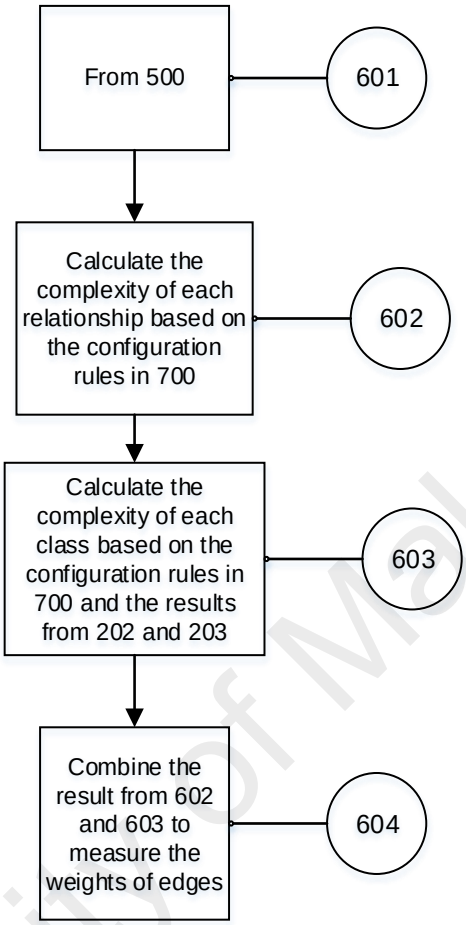


Figure 4.7: Details of Step 600 (Class and Relationship Complexity Calculation Module)

- 602: For each associated UML class relationship retrieved from Step 601, calculate the complexity of the relationship by referring to the configuration rules set in Step 700.
- 603: For each associated UML class retrieved from Step 601, calculate the complexity of the class by referring to the configuration rules set in Step 700, and the results recorded from Step 202 and 203.
- 604: Combine the results from Step 602 and 603 to measure the weight of each edge between nodes for the weighted complex network to be generated later.

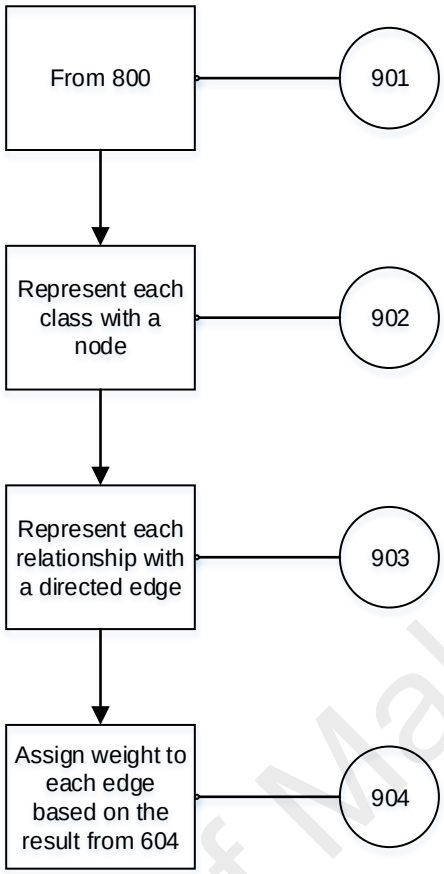


Figure 4.8: Details of Step 900 (Weighted Complex Network Representation Module)

902: For each UML class retrieved from Step 604, it is represented as a node for the weighted complex network being generated.

903: For each relationship retrieved from Step 604, it is represented as a directed edge which connects a pair of nodes.

904: Based on the output from Step 800, each calculated weight is assigned to each relevant edge. The final output of Steps 901-904 is a weighted complex network representation of the analysed software.

The flow charts in Figure 4.5-4.8 also serve as a guideline to help develop the prototype for this research. The details of the implementation and examples of software systems represented with weighted complex networks will be presented in Chapter 6.

4.4 Proposed Technique for Deriving Implicit Clustering Constraints from Graph Theoretical Analysis of Weighted Complex Network

By using the proposed Equation (3) to calculate the complexity of relationships and classes, the weights of edges (UML relationships) that connect two nodes (UML classes) in a weighted complex network are determined. Finally, at the graph level, well-established graph theory metrics are used to measure the cohesion strength among classes. The results from the graph-level metrics offer additional insights toward understanding the maintainability and reliability of the software, and subsequently, are converted into implicit clustering constraints, which addresses RO2.

4.4.1 Measuring Software Maintainability and Reliability through a Weighted Complex Network

In this thesis, six graph-level metrics are chosen, namely in-degree, out-degree, average weighted degree, average shortest path of nodes, average clustering coefficient, and betweenness centrality. These metrics are selected because prior studies have shown that they are correlated to software qualities, and can be effective to measure the maintainability and reliability of software systems (Giulio Concas et al., 2007; Jenkins & Kirk, 2007; Valverde & Solé, 2003). Besides that, average weighted degree, average shortest path, and average clustering coefficient in particular can be used to identify if a network follows the scale free and small world properties. The details of the metrics are explained in the following paragraphs.

In a generic network, the degree k_i of a node i is measured by counting the number of edges that point toward or outward from the node. The in-degree is concerned with

measuring the number of edges pointing toward the selected node. In the domain of OO software systems, in-degree of a class represents the usage of that class by other classes (Giulio Concas et al., 2007). Classes with high in-degree suggest that they contain a high degree of reusability. However, if majority of the classes exhibit very high in-degree, software bugs can propagate easily to all related classes (Turnu et al., 2013).

On the other hand, out-degree is measured by counting the number of edges pointing out from the selected node. As such, out-degree represents the number of classes used by the given class. In the OO paradigm, out-degree should be kept minimal to improve the modularity of software systems.

The average degree of a network is represented as $< k >$, where it represents the average degree of all nodes in a network. In this study, the edges are weighted. Thus, average weighted degree is used instead. Average weighted degree of a node is calculated by summing up the weights of all the edges linked to the selected node and dividing the total weight by the total number of edges. If the distribution of average weighted degree, $P(k)$, exhibits power law behaviour, it suggests that the constructed network obey the scale free characteristic. Power law characteristic also implies that there are a few important classes that are being heavily reused.

The average shortest path length used in this work calculates the average shortest path length between a source and all other reachable nodes for the weighted complex network. This will allow software maintainers to analyse the efficiency of information passing and response time of each node in the network.

A clustering coefficient measures the probability of a node's neighbours to be neighbours among themselves. A node with a high clustering coefficient indicates that there is a high tendency that the selected node will cluster together with its neighbours. The average clustering coefficient is used to represent the clustering coefficient of the whole network. In the OO point of view, a network with a high average clustering coefficient indicates high cohesion strength among groups of related functionalities. It could be also used to determine the modularity of the analysed software. Combining both the average shortest path length and average clustering coefficient allows one to examine if the network exhibits the small world characteristic.

The betweenness centrality of a node measures the number of shortest paths that pass through the selected node. It measures the importance and load of a particular node over the interactions of other nodes in the network (Yoon, Blumer, & Lee, 2006). Nodes with a high betweenness centrality often act as the communication bridge along the shortest path between a pair of nodes. Analysing the betweenness centrality allows one to comprehend the robustness and structural complexity of a given software. One can recognise in advance the potential loss of communication if nodes with high betweenness centrality are removed from the network. Table 4.2 present a summary of the selected graph theory metrics.

Table 4.2: Selected graph theory metrics and implication toward the analysed software systems

Graph Theory Metrics	Software engineering point-of-view
In-degree	Represents the usage of a particular class by other classes in the software. Demonstrate the level of reusability of a class.
Out-degree	Represents the number of classes used by the given class. High out-degree signifies that the class is composed of relatively large and complex modules. Can be refactored into several smaller classes that focus on specific responsibilities.
Average weighted degree	Identify if the analysed software obeys the power law behaviour. Provide a means to identify important classes that contribute toward a particular software functionality.
Average shortest-path length	The efficiency of information passing and response time of OO software.
Clustering coefficient	Probability of a class's neighbours to be neighbours among themselves. Helps to determine the cohesion strength of neighbouring classes.
Betweenness centrality	The number of shortest paths that pass through a particular class. Classes with high betweenness centrality indicate that they are more prone to propagating bugs and errors. In general, removal of these classes can lead to potential loss of communication between classes.

4.5 Converting Graph Theoretical Analysis into Implicit Clustering Constraints

Apart from using the graph-level metrics to analyse and evaluate the software quality aspect of software systems, the main goal of the proposed method, as mentioned in RO2, is to translate the result of graph theoretical analysis into implicit clustering constraints.

4.5.1 Identifying Community Structure of Real-world Networks

Domain or background knowledge supplied by experts are typically expressed in the form of pairwise constraint, namely must-link and cannot-link constraints to specify that two entities must both be part of or not part of the same cluster respectively. Although useful, one important and non-trivial research question remains open. How to retrieve clustering constraints if domain experts are non-existent? While various studies have shown that a

small amount of constraints can greatly improve the result of clustering, most of the studies assumed that constraints are given prior to the experiment and those constraints are absolute and without any ambiguity (Basu et al., 2004) (Kestler et al., 2006) (Klein et al., 2002).

The work by Malliaros and Vazirgiannis (2013) discussed that real-world networks have special structural patterns and properties that distinguish themselves from random networks. One of the most distinctive features in a real-world network is the community structure, such that the topology of the network is organised in several modular groups, commonly known as communities or clusters. However, in large-scale real-world networks (such as social network, power grid network, and World Wide Web), the community structure are usually hidden from users, largely due to their inherit complexity. Thus, discovering the underlying community structure of a real-world network, or commonly referred as community detection, is crucial toward the understanding of the analysed network. In this thesis, several community detection techniques that are commonly used in the field of brain network research will be adopted to discover the community structure of software systems. Next, the findings will be converted to clustering constraints in the form of ML or CL constraints to improve the results of software clustering.

4.5.2 Identify Network Hubs

Figure 4.9 shows a snippet of weighted complex network constructed using the proposed method on an open-source software written in Java, called the Apache Gora. An enlarged version of the diagram is available in Appendix A, Figure A1.

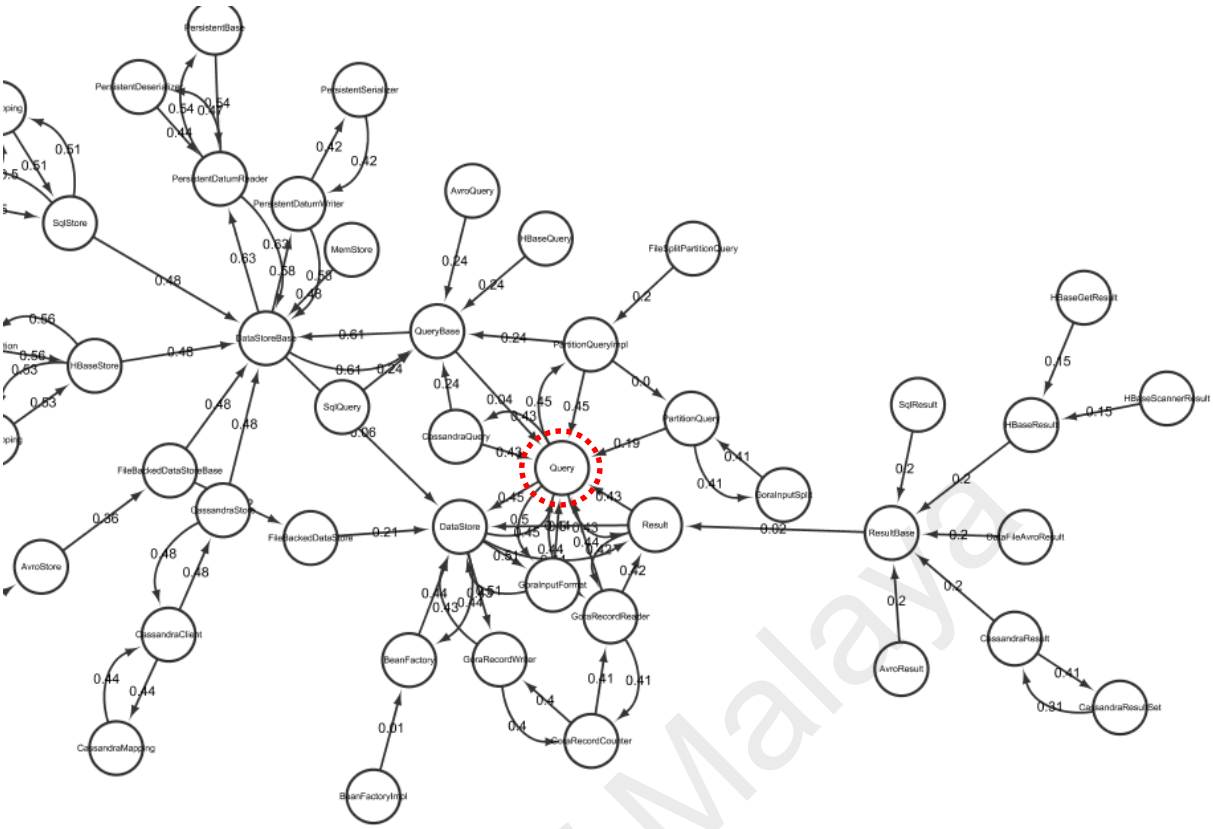


Figure 4.9: Snippet of Apache Gora project represented in weighted complex network using the proposed method

Apache Gora is a small project with 8,668 lines of code and 112 classes. Therefore, one can easily identify the community structure of the network through visual inspection. For example, the node marked with the dotted-circle possesses high in-degree because a lot of other nodes are converging and directed toward this particular node. In the field of graph theory, the presence of a high in-degree or out-degree node is usually referred as a hub. The work by Ravasz and Barabasi (2003) shows that a hub plays a very important role in complex network because it is responsible for bridging multiple small groups of clusters into a single, unified network.

From the software engineering point-of-view, hubs with high in-degree are classes that provide methods to be used by other classes. Therefore, software maintainers can view the hubs as the core functional class that contribute toward a particular software feature.

However, since hubs are directly linked to other classes, they are very vulnerable to bugs and errors propagation. The work by Turnu, Marchesi, and Tonelli (2012) shows that there is a very high correlation between the degree distribution of software-based network and the system's bug proneness. Therefore, hubs are responsible for maintaining the structural integrity of software systems against failure and it is crucial for maintainers to identify them (Liu, Slotine, & Barabasi, 2011). One simple way to identify hubs is by observing the nodes which possess high degree at the tail of the degree distribution in log-log scale (Ravasz & Barabasi, 2003). Figure 4.10 shows an example of the in-degree distribution of a sample project in log-log scale. Based on the figure, most of the nodes possess in-degree of 1, and the extreme values are roughly 60 times higher than the average in-degree. The tail of the degree distribution, as depicted by the red circle in Figure 4.10, shows that there are several nodes with exceptionally high in-degree. These nodes are usually considered as the hubs, as discussed by Ravasz et al.

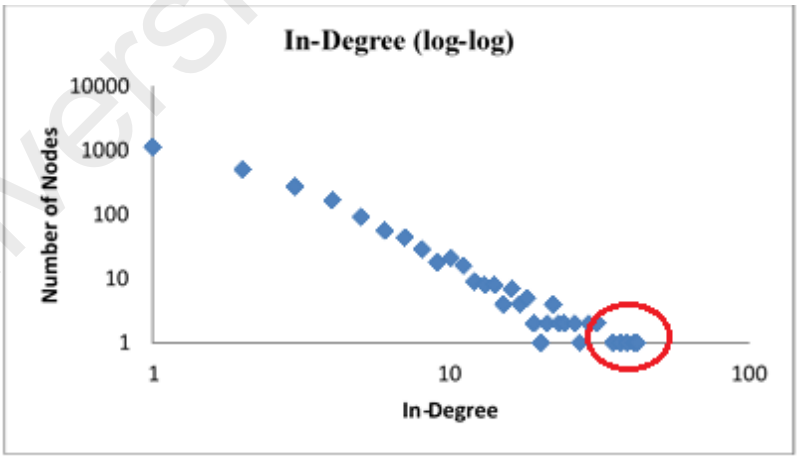


Figure 4.10: Identify hubs by observing the degree distribution of in-degree

However, it is possible that the identified nodes (classes) with high in-degree might actually be god classes. God classes are classes that are associated by a huge number of

simple data container classes, resulting in unnecessary coupling. Since god classes are tightly coupled to many other classes, maintenance of god classes are relatively more difficult compared to modular classes. Therefore, it is important to differentiate between hubs and god classes. Several studies have discovered that nodes that behave like god classes share several characteristics, especially when observed from the graph theoretical point of view (Giulio Concas et al., 2007; Turnu et al., 2013; Turnu et al., 2012). For instance, according to (Turnu et al., 2013), god classes tend to possess high in-degree and out-degree due to their "god-like" (all-knowing and all-encompassing) characteristic. Therefore in this research, when a node is found to possess exceptionally high in-degree and out-degree when compared to other classes, it is flagged as god classes instead of hubs. However, how does identifying hubs contribute toward the formulation of clustering constraints to help in constrained clustering of software systems?

4.5.3 Cannot-Link Constraints Between Hubs

In the research area of brain networks, hubs are usually neurons that are responsible for the activation of important cognitive functions and they are connected mainly to nodes in their own modules (Bullmore & Sporns, 2009). As such, hubs in brain networks usually form sub-communities that contain neurons which are correlated to the same cognitive functions.

On the other hand, network hubs in this research are considered as the core functional class that contains the methods and information of a particular software feature. It is common for other classes to invoke methods or parse parameters to and fro the hubs, resulting in high in-degree and out-degree. Therefore, this leads to a question: from a

software design's point-of-view, should the hubs be grouped into the same cluster, or separated into several disjointed clusters?

In the domain of software engineering, separation of concerns is a design principle for encapsulating software features or functionalities into separate entities to promote the notion of localisation and high modularity (Dijkstra, 1976). Thus, in order to ensure low coupling among different software functionalities, hubs should be separated into several disjoint clusters. In other words, for this research, Cannot-link constraints are established between hubs identified in the weighted complex network to promote the notion of separation of concerns. The hubs are expected to be the core class responsible for a particular software functionality. The enforcement and fulfilment of the clustering constraints are discussed in the next chapter.

4.5.4 Must-Link between Hubs and Direct Neighbours

In graph theory, clustering coefficient of a node is the average tendency of pairs of neighbours of a node that are also neighbours of each other. If all the inspected nodes are adjacent to each other, where there exists an edge that connects each pair of the neighbours, it is considered a complete clique (Watts & Strogatz, 1998). Nodes inside a complete clique are considered to be tightly connected to each other and tend to be clustered together.

Therefore, by combining the concept of hubs and clustering coefficient, software maintainers can identify the neighbouring classes that are closely related to the hubs. Neighbouring classes that form a complete clique with a hub should be always grouped

into the same cluster (Malliaros & Vazirgiannis, 2013). As such, Must-link constraints can be established between a hub and its neighbouring classes that form a complete clique, in order to ensure the formation of a baseline cluster that encompasses a group of cohesive neighbours.

4.5.5 Must-Link between Classes with High Betweenness Centrality and Their Direct Neighbours

As mentioned earlier, betweenness centrality calculates the number of shortest paths that pass through a particular class. Classes with high betweenness centrality exert relatively higher influence and impact over other neighbouring classes. Therefore, ensuring the structural stability of classes with high betweenness centrality and their neighbouring classes is important to safeguard that passing of parameters or messages is not obstructed during and after software maintenance. As such, neighbouring classes that form a complete clique with a class that possesses high betweenness centrality should be always grouped into the same cluster, similar to Section 4.5.4. The rationale behind this decision is straightforward. If software maintainers are to perform maintenance works on a class with high betweenness centrality, they would need to be notified if there are classes that are dependent on it. This is to avoid maintainers from breaking any chain of dependencies and ensure the structural stability around classes with high betweenness centrality. As such, must-link constraints can be established between classes with high betweenness centrality and the neighbouring classes that form a complete clique. The relationships between the chosen graph theory metrics and the derived clustering constraints are shown in Table 4.3.

Table 4.3: Summary of graph theory metrics and their contribution toward deriving implicit clustering constraints

Graph Theory Metric	Usage	Derived Implicit Clustering Constraint
In-degree	Identification of hubs	Cannot-link between hubs
Out-degree	Identification of hubs	Cannot-link between hubs
Average weighted degree	Identification of hubs	Cannot-link between hubs
Average shortest-path length	Calculation of betweenness centrality	-
Clustering coefficient	Identification of clique	• Must-link between a hub and its neighbouring classes that form a complete clique • Must-link between classes with high betweenness centrality and neighbouring classes that form a complete clique
Betweenness centrality	Identification of important classes	Must-link between classes with high betweenness centrality and neighbouring classes that form a complete clique

4.5.6 Identify Refactoring Opportunities as Supplementary Information

Apart from establishing clustering constraints, software maintainers can also identify potential refactoring opportunities with the aid of graph-level metrics (Al Dallal, 2015). Generally, software components that are more prone to defects and bugs should be refactored and remodularised into smaller and more manageable components. Existing studies that represent software systems using complex networks have discovered that it is possible to predict software defects with the help of graph theoretical analysis (Zimmermann & Nagappan, 2008). This information can act as supplementary information for software maintainers to aid in decision making when there is a request to modify or remove a particular software component.

For instance, Zimmermann and Nagappan (2008) discovered that there is a positive correlation between the number of defects and the value of betweenness centrality. By inspecting the faults reported in several releases of software, Zimmermann et al. showed that nodes with high betweenness centrality are usually more volatile and vulnerable to defects introduced during software changes and maintenance.

Furthermore, classes that exhibit exceptionally high value in both in-degree and out-degree are considered as fault prone classes, as discussed in the work by Turnu et al. (2012). Generally, a class with high in-degree suggests that it is a service provider, which is frequently used by other classes. A class with high out-degree, on the other hand, indicates that it is a service consumer and its operations are highly dependent on other classes. It is rare for a class to serve as both service provider and consumer. Hence, classes with high in-degree and out-degree are usually a result of poor design decision and must be refactored or remodularised into smaller modules that focus on its functionality.

In short, the chosen graph-level metrics can also be utilised to alert software maintainers regarding any potential design faults that are otherwise not noticeable using traditional software metrics. The finding of the graph theoretical analysis will be presented to the software maintainers, along with some recommendations to refactor bug and fault prone classes.

All in all, the method to represent software systems with the aid of weighted complex network proposed in this chapter aids software maintainers in several aspects:

- Provide a high-level understanding of a software system from a graph theoretical point-of-view, especially in terms of system interaction and dependencies.

- Formulation of implicit clustering constraints to aid in improving the accuracy of software clustering.
- Provide supplementary information to software maintainers such as identifying classes that more prone to bugs and errors, and those that violates common software design principles.

4.6 Chapter Summary

This chapter proposes an approach to represent OO software systems using weighted complex network in order to capture their structural characteristics, with respect to their maintainability and reliability. Nodes and edges are modelled based on the complexities of classes and their dependencies. The weighting mechanism is adopted from the three-level metrics introduced in the work by Ma et al. The choice of code-level, class-level, and graph-level metrics used in this research, along with the justification have been discussed. Graph theory metrics are applied onto the transformed weighted complex network to evaluate the software system. Finally, the observations from the graph theoretical analysis are translated into implicit clustering constraints, which help in the subsequent constrained clustering process. The enforcement and fulfilment of the clustering constraints will be discussed in the next chapter.

CHAPTER 5: MAXIMISING THE FULFILMENT OF HARD AND SOFT CONSTRAINTS

In this chapter, a method to maximise the fulfilment of clustering constraints is introduced. Unlike the work by Basu et al. (2004) where all the clustering constraints are assumed to be absolute and rigid, a method is proposed in this chapter to handle different types of constraints that vary according to their level of importance. The proposed method accepts clustering constraints from two different sources, either from the implicit structure of the software, or explicitly from domain experts who have prior knowledge regarding the software. Implicit constraints are categorised as hard constraints, while the explicit constraints are categorised as soft constraints. A unique constraints fulfilment method is proposed to maximise the fulfilment of all the derived implicit and explicit constraints. Finally, evaluations are conducted using two open-source OO software systems to evaluate the proposed constrained clustering approach.

5.1 Managing Different Types of Clustering Constraints

In Chapter 4, a method to extract clustering constraints from the implicit structure of software systems is introduced. Apart from extracting clustering constraints from the software itself, constraints can also be derived explicitly from domain experts by asking them to make judgment whether two clustering entities, or classes in this context, are similar or not to be clustered into the same group (Hong & Yiu-Ming, 2012).

Domain experts may evaluate their judgments based on their level of confidence or based on background knowledge to support their decisions. If the experts are highly confident that the provided clustering constraints are reliable and a consensus can be reached among

all the experts, the explicit constraints can be categorised as hard constraints, as discussed by Basu et al. (2004). These sets of hard constraints must be fulfilled under any kind of circumstances. On the other hand, if the domain experts are doubtful about the given constraints, it can categorised as soft constraints, such that these constraints are good to have (Basu et al., 2004). Implicit clustering constraints derived from the graph theoretical analysis are categorised as hard constraints because they are derived from one of the most reliable source of information (the structure and behaviour of source code). Explicit clustering constraints derived from domain experts, on the other hand, are categorised as soft constraints because these constraints are usually imprecise and fuzzy in nature, which might contain erroneous information (Bagheri et al., 2010).

In Section 3.2.3, the RO4 was raised based on the discussed literature to identify an appropriate technique to maximise the fulfilment of explicit and implicit constraints, while penalising violation of the constraints. It is possible that the explicit constraints derived from different domain experts might conflict with each other due to differences of opinions and experience. Therefore, in this chapter, a way to filter out conflicting clustering constraints, and to prioritise the fulfilment of constraints with high level of importance is introduced. This chapter focuses on interpreting and fulfilling explicit clustering constraints given by domain experts, as well as implicit constraints extracted from the software itself, as discussed in Chapter 4.

5.2 Constraints with High Level of Confidence

As mentioned earlier, implicit clustering constraints derived from the graph theoretical analysis in Chapter 4 are categorised as hard constraints. However, if domain experts have very high degree of confidence that a pair of classes must be grouped together or

separated, these two rules can be categorised as the Must-Link Hard (MLH) or Cannot-Link Hard (CLH) constraint as well, with the condition that these two types of clustering constraints are very clear and concise without any ambiguity.

MLH and CLH constraints are relatively easier to fulfil in k-mean clustering because clustering assignment can be manipulated easily during the clustering process. However, it is more difficult to achieve the same results for agglomerative hierarchical clustering because all clustering entities are linked together at some level of the cluster hierarchy (Bair, 2013). Therefore, when utilising hierarchical clustering algorithm to help in remodularisation of software systems, maintainers must ensure that MLH and CLH constraints are fulfilled indefinitely at all levels of the clustering hierarchy.

5.2.1 Fulfilment of Must-Link Hard Constraints

The work by Miyamoto (2012) introduced a distance based approach to impose MLH constraints by requiring entities linked by a MLH constraint to be clustered together at the lowest level of cluster hierarchy. This is done by reducing the dissimilarities between pairs of entities linked by a MLH constraint to zero.

Given a $T = \{x_1, x_2, \cdots, x_n\}$ with entities $x_1, x_2, \cdots, x_n$.

For every $(x_i, x_j) \in \{MLH\}$, the distance between x_i and x_j is modified to

$d(x_i, x_j) = 0.$

By modifying the distances between pairs of classes to zero, this will eventually form a baseline for the clustering hierarchy. Since the MLH constraints are unconditionally fulfilled at the lowest level of the hierarchy, the approach proposed by Miyamoto can

ensure that the same fulfilment can be achieved all the way through the top of cluster hierarchy. Thus in this thesis, the same technique proposed by Miyamoto (2012) is adopted to fulfil MLH constraints.

5.2.2 Fulfilment of Cannot-Link Hard Constraints

There are generally two ways to enforce CLH constraints, using either constrained based or distance based methods (Malliaros & Vazirgiannis, 2013). Constrained based method modifies the cluster assignments by inspecting the merger of two entities. Note that this thesis focuses on agglomerative hierarchical clustering algorithm, where at each step, a pair of classes with the highest similarity are chosen and merged together. If the chosen classes belong to the CLH pairs, software maintainers will need to look for the next pair of classes with the second highest similarity. However, the work by Davidson and Ravi (2009) found that the formation of dendrogram may stop prematurely in a certain scenario. The authors called this as the "dead-end" situation where unless CLH constraints are violated, there will be no more merging possible to form the final dendrogram. Thus, using constrained based approach to fulfil CLH constraints is a less viable option in this thesis.

Distance based approaches, on the other hand, modify the distance between a pair of entities linked by a CLH constraint to be a value high enough to prevent them from merging.

Given a set $T = \{x_1, x_2, \cdots, x_n\}$ with entities $x_1, x_2, \cdots, x_n$.
For every $(x_i, x_j) \in \{CLH\}$, $d(x_i, x_j) = d(x_i, x_j) + Const$
where $Const$ is a constant large enough to prevent linkage between entities x_i, x_j.

By enforcing this rule, the pairs of entities linked by a CLH constraint will not be chosen to be merged unless there are no more classes with distance more than $d(x_i, x_j) + Const.$ Classes which belong to CLH constraints will then be merged at the top of the hierarchy to form the complete dendrogram. An example is illustrated in Figure 5.1, where the circle at the top of dendrogram indicates the merging of classes linked by CLH constraints. By observing Figure 5.1 from another perspective, some CLH constraints are actually violated at the top of the hierarchy since without violating them, "dead-end" situation will occur. However, violating CLH constraints at the top of the hierarchy are negligible because it is almost impossible to cut the dendrogram at that location (Lung et al., 2004). In a typical scenario, cutting the dendrogram at the top of hierarchy will yield a very small number of clusters because this decision is at the trade-off of relaxing the constraint of cohesion in the cluster membership (Chong et al., 2013). Clusters formed when cutting the dendrogram at the top of the hierarchy are usually made up of classes with very low and fragile cohesion strength. Therefore, the distance based approach is adopted in this thesis to enforce the CLH constraints.

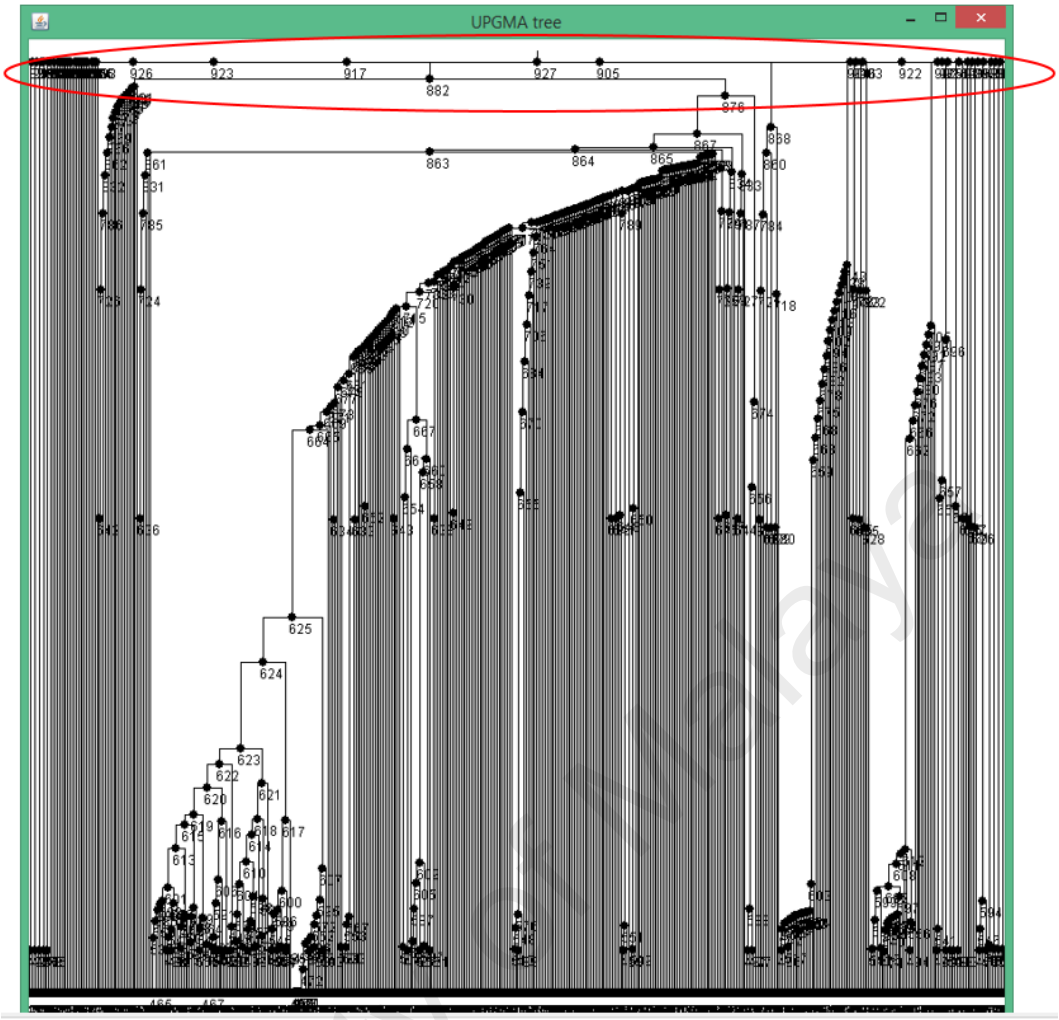


Figure 5.1: Example of imposing CLH constraints by modifying the distance measure between pairs of entities

5.2.3 Problems Associated with Enforcing MLH and CLH Constraints

However, changing the distance measure of entities involved in MLH and CLH constraints will most likely result in violating the triangle inequality of resemblance matrix – the pairwise matrix that contains the similarity or dissimilarity strengths between pairs of classes which dictate the merging of classes during the clustering process as discussed in Section 2.3.3. (Klein et al., 2002). Violating the triangle inequality of resemblance matrix means that for some classes $(x_s, x_t) \notin \{MLH\}, (x_s, x_t) \notin \{CLH\}$ with distance $d(x_s, x_t)$ apart before imposing MLH or CLH constraints, may now be

$d'(x_s, x_t) < d(x_s, x_t)$ along some path which skips through the MLH or CLH pairs. As pointed out by Klein et al. (2002), this problem can be solved by finding a new distance value with respect to the modified classes involved in MLH or CLH constraints using all-pairs-shortest-path algorithm. The algorithm will search for the shortest path between all pairs of classes after the enforcement of MLH and CLH constraints, and the results will be used to update the associated resemblance matrix. The usage of all-pairs-shortest-path algorithm can prevent the violation of triangle inequality of the resemblance matrix. For instance, Figure 5.2a shows a simple example of 6 classes, Classes A, B, C, D, E, and F. The number on the edges indicates the distance between two classes. In the figure, the shortest distance between Class A and Class C is 0.9 with the following order: A-D-E-F-C.

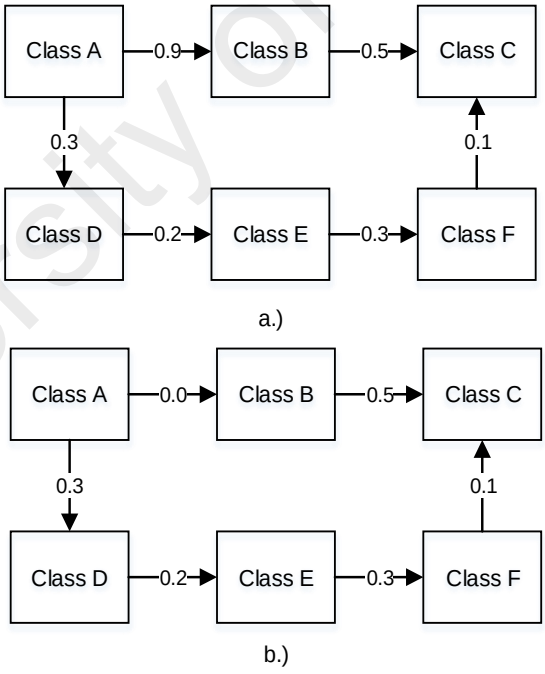


Figure 5.2: Potential triangle inequality problem when imposing MLH and CLH constraints

After several discussions, the original developers discovered that Class A and Class B in fact are very closely related and impose a MLH constraint onto the two classes. Thus, the

distance between A and B is now 0.0 to reflect the MLH constraint, as illustrated in Figure 5.2b. In this case, the shortest path between Class A and Class C after the imposition of the MLH constraint is now 0.5, with the following order: A-B-C. If the resemblance matrix is not updated accordingly to reflect the changes, the final clustering result might be erroneous. Therefore, the proposed constrained clustering method addresses this violation in fulfilling both MLH and CLH constraints using the following algorithm:

Input: A set of entities S = $\{x_1, x_2, \cdots, x_n\}$, a set of MLH (must-link hard constraints) and a set of CLH (cannot-link hard constraints)

Output: A modified resemblance matrix

1. Calculate the distance between each pair of entities and store it in a resemblance matrix *D* where $D_{i,j} = D_{j,i}$
2. Let *D'* = *D* (create a clone resemblance matrix to modify the original one)
3. while $\neg [(\forall(x_p, x_q) \in \{MLH\}) \cap (\forall(x_r, x_s) \in \{CLH\}) > 0]$
    i. for every $(x_i, x_j) \in \{MLH\}$, the distance between x_i and x_j is modified to $d(x_i, x_j) = 0$.

        run all-pairs-shortest-path algorithm to prevent violation of triangular inequality

    ii. for every $(x_i, x_j) \in \{CLH\}$, the distance between x_i and x_j is modified to $d(x_i, x_j) = d(x_i, x_j) + Const$ where $Const$ is a constant large enough to prevent linkage between entities x_i, x_j

        run all-pairs-shortest-path algorithm to prevent violation of triangular inequality
4. Return D' as the new updated resemblance matrix.

5.3 Constraints with Low Level of Confidence

In scenarios where domain experts are not confident enough to judge whether the given clustering constraints are absolute, these sets of constraints will be categorised as soft constraints. Since soft constraints are not definite, clustering results with partial fulfilment of soft constraints are still acceptable in most cases (Bair, 2013). However, soft constraints might be derived with a different level of importance and ranking, subject to the information provided by domain experts. Fulfilling a handful of higher importance soft constraints might overshadow the fulfilment of several less important ones. Soft constraints are typically assigned with a penalty score. The penalty score is used to evaluate the quality of clustering results where minimisation of the penalty score is preferred. Thus, a prioritisation and ranking mechanism of soft constraints is introduced in this study.

The nature of prioritising a given set of clustering constraints is a multi-criteria decision-making (MCDM) problem. MCDM is a research of methods and procedures by which it concerns about evaluating multiple conflicting criteria and derive a way to come to a compromise. This set of criteria often differs in the degree of importance. Examples of methods to handle MCDM problems are analytic hierarchical process (AHP), fuzzy AHP, goal programming, scoring methods, and multi-attribute value functions.

In this thesis, ranking and prioritising the importance of soft constraints are achieved using the fuzzy AHP technique. Fuzzy AHP is capable of handling the fuzziness of users' opinions with respect to the importance of soft constraints (Chong, Lee, & Ling, 2014). The results gathered from fuzzy AHP will be represented in a table which shows a list of candidate criteria (soft constraints) associated with weightage (importance toward the

analysed software), where a higher weightage value represents higher priority. The result acts as a baseline to evaluate the penalty score of each soft constraint. Must-Link Soft (MLS) and Cannot-Link Soft (CLS) constraints are evaluated separately because the notion of Must-Link (ML) and Cannot-Link (CL) is opposing to each other.

In both traditional and fuzzy AHP, the process starts by modelling a hierarchy of decisions based on the problem domain. The top of the hierarchy consists of the goal for conducting the test, followed by a group of possible choices to achieve that particular goal. The choices can be further divided into sub-criteria if required.

A pairwise comparison among all the possible choices is conducted to justify the importance between them. Each choice is associated with a weightage in order to reflect the priority of each choice toward the ultimate goal. Domain experts will perform a pairwise comparison and give weightage using a nine-point scale ranging from 1-9, where a greater value represents higher importance. In order to reach a consensus among the domain experts, triangular fuzzy number (TFN) is used. TFN is capable of aggregating the subjective opinions of all the decision makers through fuzzy set theory. Figure 5.3 shows an example of TFN denoted as (*L, M, H*) which represents the lowest possible value, most ideal value, and highest possible value respectively.

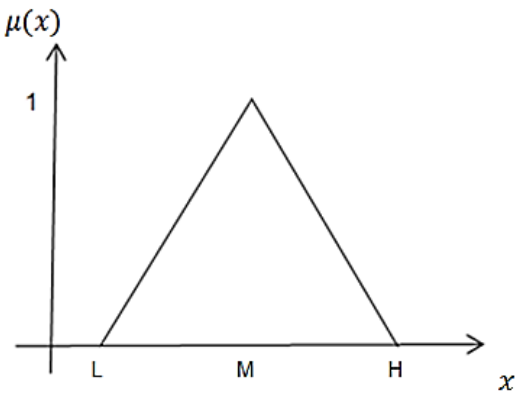


Figure 5.3: Triangular fuzzy number

The triangular fuzzy number T_{xy} is constructed using the following formula:

$$T_{xy} = (L_{xy}, M_{xy}, H_{xy})$$

$$L_{xy}, M_{xy}, H_{xy} \in (1/9, 9)$$

$$M_{xy} = \sqrt[n]{J_{xy1} \cdot J_{xy2} \cdot J_{xy3} \cdots \cdot J_{xyn}}$$

where *xy* represent a pair of criteria being judged by domain expert. J_{xy1} represents an opinion of stakeholder "*1*" toward the relative importance for criteria C_x - C_y. Value M_{xy} is produced by calculating the geometric mean of domain experts' scores for a particular comparison. The geometric mean is capable of accurately aggregating and representing the consensus of decision makers (Saaty, 1980).

After getting the TFN value for every pair of comparison, a fuzzy pairwise comparison matrix is established in the form of *n* x *n* matrix. Table 5.1 illustrates an example of the matrix.

Table 5.1: Fuzzy pairwise comparison matrix

$\tilde{F}_{xy} =$

	C_a	C_b	.....	C_n
C_a	1	p_{ab}	.....	p_{an}
C_b	$1/p_{ab}$	1	.....	p_{bn}
⋮	⋮	⋮	⋮	⋮
C_n	$1/p_{an}$	$1/p_{bn}$	.....	1

p_{ab} represents the triangular fuzzy number for the comparison between criteria C_a and C_b. Comparison between criteria C_b to C_a is the reverse of C_a to C_b, thus making the TFN value for C_b to C_a to be represented as $1/p_{ab}$. $\tilde{F}_{xy}$ denotes the TFN values derived from the formula.

Following the construction of comparison matrix, defuzzification will take place to produce a quantifiable value based on the calculated TFN values. The defuzzification method adopted in this thesis is derived from Liou and Wang (1992), which is based on the alpha cut manner.

$$\mu_{\alpha,\beta}(\tilde{F}_{xy}) = [\beta \cdot f_{\alpha}(L_{xy}) + (1-\beta) \cdot f_{\alpha}(H_{xy})]$$

and

$$0 \leq \alpha \leq 1, \qquad 0 \leq \beta \leq 1$$

such that $f_{\alpha}(L_{xy}) = (M_{xy} - L_{xy}) \cdot \alpha + L_{xy}$, which represents the left-end boundary value of alpha cut for $\tilde{F}_{xy}$. On the other hand, $f_{\alpha}(H_{xy}) = H_{xy} - (H_{xy} - M_{xy}) \cdot \alpha$ represents the right-end boundary value of alpha cut for $\tilde{F}_{xy}$.

α and β in this context carry the meaning of preferences and risk tolerance of domain experts. These two values range between 0 and 1, in such a way that a lesser value indicates greater uncertainty in decision making. Since preferences and risk tolerance are not the focus of this research, value of 0.5 for α and β will be used to represent a balanced environment. This indicates that decision makers are neither extremely optimistic nor pessimistic about their judgments.

The next step is to determine the eigenvalue and eigenvector of the fuzzy pairwise comparison matrix. The purpose of calculating eigenvector is to determine the aggregated weightage of a particular criterion. Assuming that δ denotes the eigenvector while λ denotes the eigenvalue of fuzzy pairwise comparison matrix $\tilde{F}_{xy}$,

$$[(\mu_{\alpha,\beta}(\tilde{F}_{xy}) - \lambda I] \cdot \delta = 0$$

The formula above is based on the linear transformation of vectors, where *I* represents the unitary matrix. Using the above formula, the weightage of a particular criterion with respect to all other possible criteria can be acquired. The results gathered from fuzzy AHP will be represented in a table form. The table shows a list of candidate soft constraints associated with weightage value, where a higher weightage represents higher priority and higher penalty score upon violating these clustering constraints. This table will act as a baseline to assist in formulating the objective function of MLS and CLS constraints.

The objective function of MLS and CLS constraints is shown below.

Given a set S = $\{x_1, x_2, \cdots, x_n\}$ with classes $x_1, x_2, \cdots, x_n$, a set of MLS (must-link soft constraints) and a set of CLS (cannot-link soft constraints). The objective function is to maximise the number of satisfied MLS and CLS constraints:

$$max\ f(Z) = \frac{1}{n_c}\sum_{i=1}^{m}\gamma(x_i) - \frac{1}{2}\sum_{i=1}^{m}\delta(x_i) \quad (4)$$

subject to $\gamma(x_i) \geq 0, i = 1, \cdots m$

$0 \leq \delta(x_i) \leq 1, i = 1, \cdots m$

Where n_c is the total number of available soft constraints (including MLS and CLS) and $\gamma(x_i)$ is the number of satisfied soft constraints involving pairs of classes with x_i as one of the classes. The first operand is the ratio of fulfilled soft constraints over the total number of soft constraints. Meanwhile, $\delta(x_i)$ is the penalty score for violated clustering constraints involving pairs of entities with x_i as one of the classes. The penalty score is based on its importance toward the overall software system using fuzzy AHP technique. The cumulative weightage (penalty score) of either MLS or CLS constraints is equal to 1. Thus, a scaling constant of 1/2 is used to normalise the second operand of the equation

when adding both the MLS and CLS constraints. Maximisation of function $f(Z)$ is the goal of this objective function. The evaluation of soft constraints fulfilment is performed after the formation of dendrogram. The dendrogram needs to be cut at a certain height to produce a set of disjoint clusters. Evaluation of soft constraints can then be done by inspecting the set of disjoint clusters, to check whether or not the soft constraints are violated. A few cutting points can be executed to compare and contrast the quality of each cut with respect to the minimisation of soft constraints' penalty.

5.4 Overview of the Proposed Constrained Agglomerative Hierarchical Software Clustering Method

All in all, the complete constrained agglomerative hierarchical software clustering method is shown below.

Given a set of clustering entities *S*, the distance for each pair of entities *x* and *y* in *S* is $1 \geq d(x, y) \geq 0$ and a set of clustering constraints $\alpha = \{MLH\}, \{MLS\}, \{CLH\}, \{CLS\}$.

1. Construct the baseline clusters from MLH constraints resulting in *n* number of initial clusters $M_1, M_2, \cdots M_n$.
2. If there is a pair of entities (x, y) in $M_1, M_2, \cdots M_n$ and CLH$(x, y) \in \alpha$, then this is a NP-Complete problem with no solution.
3. Construct an initial clustering with t_{max} clusters consisting of the *n* clusters $M_1, M_2, \cdots M_n$ and a singleton cluster for each entity. t_{max} is the maximum number of clusters for the set of entities *S*. Initialise $t = t_{max}$.
4. while $t \neq 1$
    a. Find the pair of entities (S_p, S_q) with minimum distance.

b. Merge $S_r = S_p \cup S_q$ at the level of dissimilarity.

c. Remove S_p, S_q.

d. $t = t - 1$.

e. Repeat Step 4.

5. Generate a dendrogram tree based on the clustering results.
6. Cut dendrogram at several points.
7. Evaluate the fulfilment of MLS and CLS with respect to the objective function proposed in Equation (4).

The overall workflow of the proposed algorithm is as follows:

i. The software maintainer provides the UML class diagrams of the software to be analysed. If class diagrams are not available, source codes are converted into class diagrams using an off-the-shelf round-trip engineering tool.

ii. Based on the method proposed in Chapter 4, the software is represented in a weighted complex network to identify and derive clustering constraints.

iii. Methods related to the formation of clustering entities, identification of features, construction of resemblance matrix, and formation of dendrogram are discussed in the following sub-section.

iv. The software maintainer and/or the original developer can then provide domain knowledge to aid in the software clustering process. Based on the confidence level of the maintainer and/or developer, each clustering constraint is categorised into either hard or soft constraint. A dendrogram is formed based on all the retrieved information.

v. The dendrogram is cut based on the available clustering constraints. Each cutting point is evaluated using the proposed objective function and a cluster validity index, as discussed in Step iii.

vi. The cutting point that can fulfil the most clustering constraints and the best cluster validity index is preferred and chosen as the optimum cutting point that forms highly cohesive clusters.

In this thesis, UML classes are represented as nodes while interrelationships between pairs of classes are represented as edges. Due to the uniqueness of the proposed constrained clustering method, an enhanced software clustering algorithm is proposed to fit into the context of this research. Details of the enhanced software clustering algorithm are discussed in Section 5.4.1.

All in all, the proposed constrained clustering method will not only fulfil RQ5, but also address the question raised in RQ6: *How to maximise the fulfilment of constraints during clustering without risking the "dead-end" situation as discussed by Davidson and Ravi (2009)?*

5.4.1 Enhanced Software Clustering Algorithm

As mentioned in Section 2.3, software clustering involves 5 main steps as follows:

1. Identification of entities or components
2. Identification of features
3. Calculation of similarity measure
4. Application of clustering algorithm
5. Evaluation of clustering results

However, conventional software clustering algorithms are not capable of addressing the issue of using UML classes as the basic clustering entities, as well as the introduction of

clustering constraints. Therefore, an enhanced software clustering algorithm is proposed in this research.

5.4.2 Identification of Entities or Components

In this research, UML class diagrams are used to represent the clustering entities instead of relying on source code alone. UML class diagrams provide a standardised conceptual model that represents the system's components, operations, attributes and relationships among classes. Interrelationships among classes are used to specify either through the presence of abstraction or accessing features of another class. This gives software engineers a static view of the structural connections being designed. Therefore, class diagrams can provide an informative summary of many design decisions about the system's organisation. In this research, it is assumed that software maintainers are provided the UML class diagrams of the software to be analysed. If class diagrams are not available, source codes are converted into class diagrams using an off-the-shelf round-trip engineering tool.

5.4.3 Identification of Features

Feature identification is used to analyse how similar or closely related two entities are based on certain common attributes. UML classes are used to represent the clustering entities in this research. As such, class relationships such as realisation, aggregation, and association are the best indicator to observe whether a class is related to another class.

5.4.4 Calculation of Similarity Measure

Selection of similarity measure is an important step in software clustering because it is used to construct the resemblance matrix for the clustering entities. Resemblance matrix is a matrix that contains the similarity strengths between all pairs of clustering entities. In this research, selection of similarity measure, and subsequently, construction of resemblance matrix are supported by the usage of weighted complex network, as discussed in Chapter 4. Analysing the structure and behaviour of software systems using graph theoretical metrics had been proven to be useful to capture the dynamic relationships and dependencies between software components (Louridas et al., 2008). Shortest path algorithm, in particular, enables software maintainers to identify how closely related two classes are based on the type of weighting mechanism used in quantifying the weights of edges. In this research, the weight of a particular edge is measured using a unique weighting mechanism that takes into account the complexity of UML relationship (edges) and the complexity of classes (nodes) linked by the specific relationship. The proposed weighting mechanism provides a means to estimate the similarity strengths between pairs of classes, such that a lower value signifies higher similarity between a pair of classes, as shown in Equation (3), Section 4.2.3.

$$Weight_{(R_{i\rightarrow j})} = \left(H_{R_{i\rightarrow j}} * \alpha\right) + \left[\left(1 - Comp_{(D_j)}\right) * \beta\right] \quad (3)$$

By using Dijkstra's shortest path algorithm (Dijkstra, 1976), software maintainers can identify how closely related two classes are, and provide a means to indicate whether they belong to the same functional group. As such, in this research, shortest path algorithm is used to construct the resemblance matrix of the clustering entities.

5.4.5 Application of Clustering Algorithm

Based on the resemblance matrix produced, merging of entities will then take place. Depending on the algorithm used, certain algorithms merge the pair with highest similarity first, while others merge the most dissimilar first. This process is performed iteratively until all entities are merged into a single cluster.

There are three main types of clustering algorithm:

1. Single Linkage Algorithm (SLINK)
2. Complete Linkage Algorithm (CLINK)
3. Un-weighted Pair-Group Method using Arithmetic Average (UPGMA)

Single linkage, or commonly known as nearest neighbour method, defines the similarity measure of two chosen clusters as the maximum similarity strength among all pairs of entities in the two clusters. Complete linkage or furthest neighbour method, on the other hand, is the opposite of single linkage. The minimum similarity strength among all pairs of entities is used instead of the maximum similarity strength.

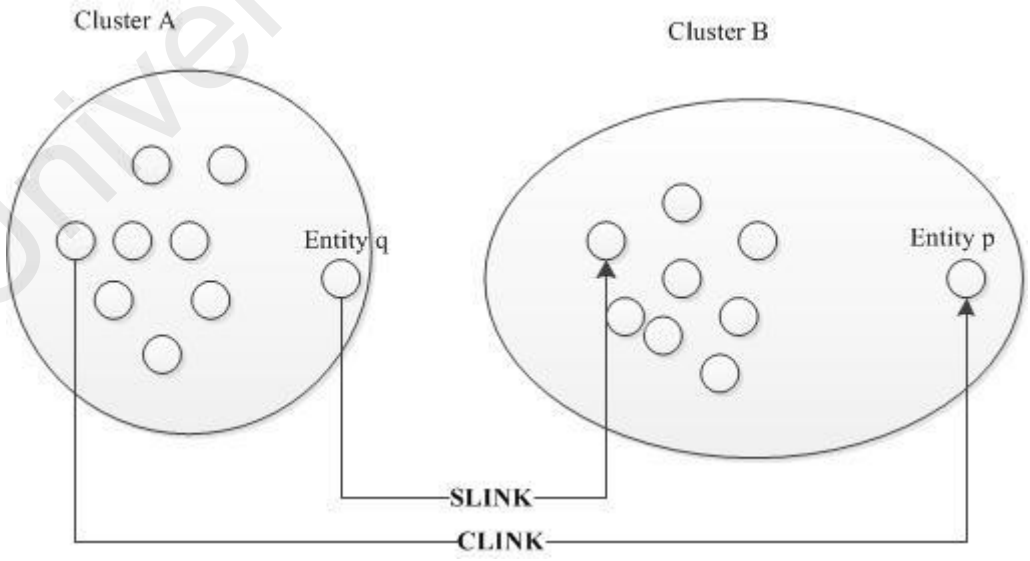


Figure 5.4: Illustration of SLINK and CLINK linkage algorithms

However in condition where outliers exist, SLINK and CLINK methods are less effective. Take Figure 5.4 for example, there is an outlier *"Entity p"* in Cluster B where it is very far away from the other members of the group. If CLINK method was used, the outlier will drastically affect the accuracy. The same goes to *"Entity q"* in Cluster A if SLINK was used to merge the cluster. This would eventually cause the formation of low cohesive clusters if the number of outliers is considerably high.

Instead, UPGMA defines the similarity measure between two clusters as the arithmetic average of similarity strengths among all pairs of entities in the two clusters. Given a dissimilarity matrix $DM = [DM(C_x, C_y)]$ from a set of clusters *C*:

1) Select a pair of clusters $(C_i, C_j) \in C$ such that $RC_{i,j}$ is the minimum similarity strength.
2) Remove (C_i, C_j) from cluster *C* and substitute with a new cluster $C_k = (C_i \cup C_j)$
3) The similarity strength RC_k of the new cluster C_k is calculated by using the arithmetic average (similarity strength) of the old cluster (C_i, C_j).
4) Repeat the process until all elements are connected in a single cluster.

UPGMA is the most popular clustering method due to the fact that it is less sensitive toward the effect of outlier as compared to SLINK and CLINK (Gronau & Moran, 2007; Lung & Zhou, 2008).

In this research, UPGMA will be used to merge clusters and form a dendrogram. After applying the clustering algorithm, the output is in the form of dendrogram. Further analysis of clustering results can then be performed based on the dendrogram. However, since clustering constraints are an important focus to be considered and addressed in this research, the dendrogram is not generated immediately after the application of clustering

algorithm. Instead, all MLH and CLH constraints will be imposed to modify the resemblance matrix using the algorithm shown in Section 5.2.3, before generating the dendrogram.

5.4.6 Evaluation of Clustering Results

One of the popular cluster validity indices is the Davies-Bouldin index (Davies & Bouldin, 1979). Davies-Bouldin index is a function of the ratio of the sum of within-cluster scatter to between clusters separation. Because a low scatter and a high distance between clusters lead to a low value index, a minimisation of Davies-Bouldin index is preferred. However, the algorithm is highly computational intensive because the index is an average over the *n* number of clusters. When the number of entities in the dataset is extremely huge, the number of clusters formed will also increase relatively. In return, the computational efforts of Davies-Bouldin index increase because it needs to average out the index values over the *n* number of clusters. Thus, this index does not scale well when the search space is extremely huge.

An enhanced version of the Davies-Bouldin validity index, which focuses on scalability, is introduced in this research to validate the performance of different cutting points of the dendrogram. Given a dataset, $X = \{X_1, X_2, \dots, X_n\}$, which is a set of n entities. The dendrogram, D, is the hierarchy of clusters that are formed by the dataset X. A cutting point D^y on the dendrogram results in the partitioning of the dataset into a set of clusters, $C = \{C_1, C_2, \dots, C_N\}$. $C \subseteq X$ is a subset of entities in the dataset. Let C_k be one of the clusters that is formed after applying cutting point D^y, and it contains m entities with their similarity strengths $\{X_{1d}, \dots, X_{md}\}$. S_k, the centroid of cluster C_k, is the average of all similarity strengths between all pairs of entities in the cluster.

$$S_k = \frac{1}{m}\sum_{i=1}^{m} X_{id}$$

S_k is referred to as the centroid of cluster C_k, $k \leq$ N. The enhanced Davies-Bouldin index to validate the performance of cutting point D^y is defined as:

$$Index(D^y) = \frac{1}{N}\sum_{i=1}^{N} \frac{intra\ (C_i)}{inter_{min}}$$

where *N* is the total number of clusters that are formed at cutting point D^y.

$intra(C_i)$ is the average distance between each entity in C_i and its cluster centroid S_i.

$inter_{min}$= $min(inter\ (C_i,\ C_j))$, where $C_i, C_j \in C; i \neq j$, is the nearest distance between the two centroids of clusters C_i and C_j, i.e., it represents the minimum distance between the centroids S_i and S_j of C_i and C_j, respectively.

For each cutting point, the enhanced Davies-Bouldin index is evaluated separately, and all of the partial calculations are averaged by its weighted mean. $intra(C_i)$ validates the cohesion strength among all members of the same cluster C_i. A lower value of $intra(C_i)$ signifies a higher cohesion, whereas $inter_{min}$validates the coupling strength among the neighboring clusters, C_i and C_j. A higher $inter_{min}$ value indicates that the clusters formed are well separated.

The major difference between the original Davies-Bouldin index and the proposed enhanced Davies-Bouldin index is that the enhanced version evaluates partial calculations instead of averaging over all of the clusters. In cases where there are outliers (i.e., singleton clusters), the original Davies-Bouldin index is not capable of detecting them because the outliers that are involved are also averaged out. A singleton cluster here refers to a cluster that contains only one entity. Furthermore, when calculating the clusters'

separation for a specific cluster, the proposed enhanced index only inspects the cluster centroid that is closest to the evaluated cluster. The original Davies-Bouldin index calculates cluster separation of a cluster C_i by measuring the distance from its centroid to an average over all *N* clusters. The time complexity of this operation is linear O(N) to the size of data. Because the merging points of a dendrogram are formed in an ascending order, the enhanced version only needs to inspect the cluster formed before and the cluster formed after the current one in order to identify its nearest neighbour instead of averaging out its distance with all other clusters. The rationale behind this decision is that if there is a cluster centroid that is very close to the evaluated centroid, then there is a very high chance that the two clusters are strongly coupled. Using this approach, the complexity of calculating cluster separation can be reduced from O(N) to O(1). Through the application of this enhanced Davies-Bouldin index, a cutting point that produces the lowest index value is considered to have the best balance in terms of the cohesion, coupling, and similarity constraint.

However, in the case of a singleton cluster, the index score should be penalised. It is always a good practice to minimise the occurrences of clusters with a single element (Mirkin, 2004). Many cluster validity index tend to incorrectly favour clustering that generates singleton clusters because they do not have a mechanism to detect it. Singleton clusters are assumed to be removed by users manually. To avoid this bias, a penalty mechanism to penalise singleton clusters is introduced, where if a single entity is found inside a cluster C_i, $intra(C_i)$ score will be adaptively increased to penalise the final $Index(D^y)$ of that specific cutting point.

The initial concept of penalty is by penalising the cluster cohesion value on singleton clusters. The proposed penalty mechanism will assign a very high value to $intra(C_i)$ for

every singleton cluster C_i, which increases the validity index D^y. This scenario forces the algorithm to continue to search for a better partition point that will produce a lower validity index value. Eventually, the penalty mechanism will minimise the probability of cutting the dendrogram at an undesirable position.

The simplest way to penalise a singleton cluster is by assigning a static number (for example, 1, 2, 3) to increase the overall validity index value. However, in a certain worst case scenario, the penalty effect might become insignificant. This extreme scenario refers to the case where the $inter$ value is only a few times larger than the $intra$ value. When the value of $inter$ grows exponentially, the effect of the static penalty mechanism of $intra$ (C_i) will be negligible. Ultimately, this strategy will cause the calculation to incorrectly give a low $Index(D^y)$ value to a cutting point that forms singleton clusters. Even worse, the validity index will assume that the erroneous result is the best cutting point. Figure 5.5 shows an example of the worst case scenario, in which the average $inter$ value is large because of the very large distances between the pairs of cluster centroids.

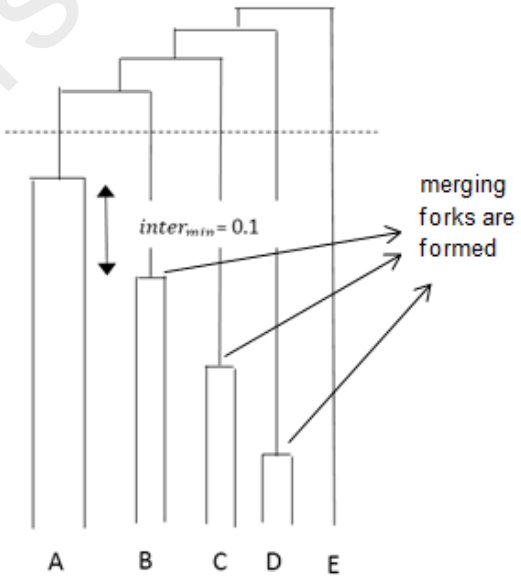


Figure 5.5: Example of a worst case high $inter$ score

To create a more concrete example, assume the case in Figure 5.5, where the dotted line cuts the dendrogram and forms 5 clusters, A, B, C, D, and E. The average $inter_{min}$ is 0.1 and a singleton cluster is found (Cluster E). If a static penalty of 1 is given to an $intra\ (C_E)$ score, the $\frac{intra\ (C_E)}{inter_{min}}$ will be equal to 10. This value is not sufficiently significant to signify that this cut produces a bad clustering result because the effect of a penalty is nullified by the denominator (note that the enhanced Davies-Bouldin index favours a low score).

Thus, the imposition of a penalty mechanism should be adaptive instead of assigning a static penalty value. The factor that increases the $inter_{min}$value is the wide gap between the pair of merging forks. $inter_{min}$ calculates the nearest distance between the two centroids of clusters, S_i and S_j of C_i and C_j, respectively, among all pairs of clusters. When the gap between the merging forks is relatively large, it will cause the index value to increase proportionally.

To address this problem, an adaptive penalty mechanism based on the relative change between the maximum $inter$ and average $inter$ values is introduced. Let D^y be the cutting point that forms a set of clusters $C = \{C_1, C_2, \dots, C_N\}$. The average $inter$ is given as $inter_{mean}$, while the maximum $inter$ value is given as $inter_{max}$.

The relative change between the average $inter$ and maximum $inter$ is

$$Relative\ Change = \frac{|inter_{mean} - inter_{max}|}{inter_{mean}}$$

If $inter_{max}$ is x times larger than the $inter_{mean}$value, then the equivalent value of the penalty should be imposed. Thus, the adaptive penalty P is given as

$$P = Relative\ Change$$

As such, if a dendrogram cutting point produces singleton clusters, penalty *P* is assigned as the cluster validity index value in order to penalise the formation of singleton clusters. Otherwise, the enhanced Davies-Bouldin index is used to evaluate the quality of each cluster.

- Finding the Optimum Dendrogram Cutting Point

For a small dataset for which the search space is small, it is very easy to identify the optimum cutting point through visual inspection. The optimum cutting point in this context refers to the cutting point that produces highly cohesive sets of clusters. However, when a large dataset is involved, visual inspection is infeasible. Finding out the optimum cutting point can be highly computationally intensive. One way to find the optimum cutting point is by using the exhaustive cut method, which cuts the dendrogram at each possible point to find the best possible clustering result. This method is not feasible, and duplicate effort will definitely occur because the possibility of producing repeating index values is much higher. Therefore, a solution to adaptively search for the optimum point in a large-scale dataset is proposed in this research.

Scalability is an important issue in software clustering, for recovering from a software system that has a very large number of entities. If the proposed approach were to run on a larger and more complex system, then what is the average number of cuts that is needed? To address this problem and to improve the scalability of the dendrogram cutting method, the cutting technique introduced in (Fokaefs et al., 2012) is adapted, where the dendrogram is only cut at every unique merging fork instead of setting a threshold value.

In order to verify the performance of this method, a series of tests and simulations were conducted, and the details of the simulations are presented in Appendix B.

Based on the results presented in Appendix B, cutting the dendrogram after every unique merging fork produces the best result. The reason for this choice is that there is usually a significant change in the validity index value after a merging fork. At the same time, cutting at unique merging forks can minimise redundant effort. The redundant effort in this context refers to the repetitive effort that is required to find optimum clustering results. In order to further enhance the accuracy of finding the optimum cutting point, a least-squares polynomial regression analysis is introduced in this research.

In a polynomial equation, one can identify the highest and lowest points by finding the derivative of the polynomial function. Using this same concept, given a distribution of different cutting points and their validity index values, the algorithm can find the cutting point that produces the lowest validity index value if the polynomial equation can be formed.

Table 5.2 shows an example of a dendrogram that has points cut at 0.2, 0.4, 0.6, 0.8, and 0.99, which produces different cluster validity index values. Using the information that is retrieved from this observation, a cutting point-validity index graph can be plotted to observe the trend with which different cutting points can affect the results of the validity index.

Table 5.2: Example of validity index values retrieved from different cutting points

Cutting point (x-axis)	Enhanced Davies-Bouldin validity index (y-axis)
0.2	15
0.4	7
0.6	4
0.8	24
0.99	27

Figure 5.6 depicts the graph that is plotted using the information that is retrieved from Table 5.2. An estimation of the polynomial equation can then be formed using the least-squares method (Wolberg, 2006).

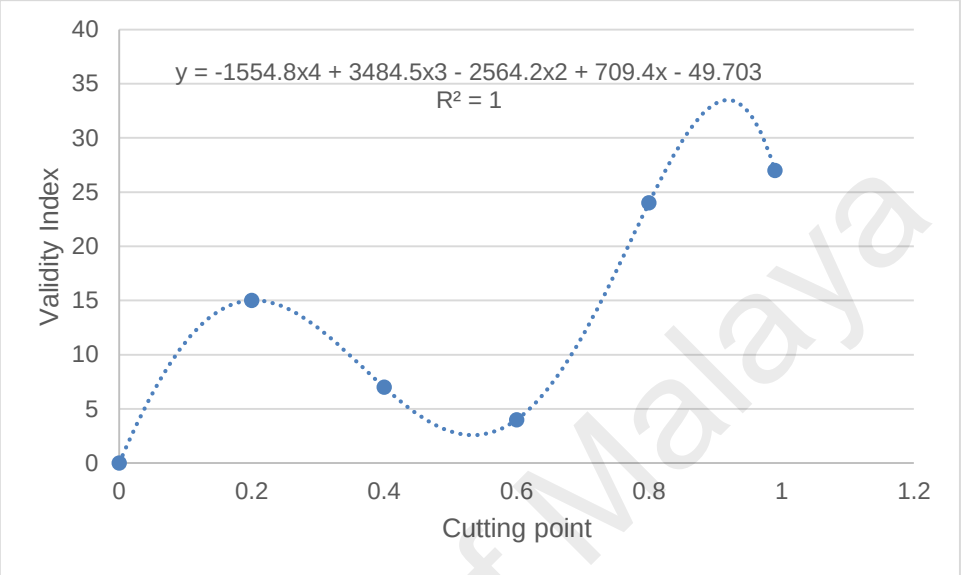


Figure 5.6: Polynomial regression based on the data from Table 5.2

The R^2 value indicates the correlation coefficient of the estimated polynomial equation. The correlation values that are equal to 1 or -1 correspond to data points that lie exactly on the estimated line which indicates that the estimation is very reliable. The next step is to find the root of the polynomial equation. This step provides identification of the minimum point of the x-axis, which in this context, is the cutting point that produces the lowest validity index value.

The example in Figure 5.6 is a quartic function. The roots of $y = -1554.8x^4 + 3484.5x^3 - 2564.2x^2 + 709.4x - 49.703$ are [x= 1.080, x= 0.104, x= 0.529, x= 0.454]. It is to be noted that the curve represented in Figure 5.6 is estimated based on polynomial interpolation of the data points, which can only be served as a visual guidance. Using the four roots, one can find the corresponding y-axis values, which resemble the validity index values. The

root that produces the lowest y-axis value will be deemed to be the best cutting point because it yields the lowest validity index value. Through this method, the cutting point x=0.529 can easily be identified, and it forms an optimum set of reliable clusters.

Eventually, the proposed adaptive penalty mechanism and the least-squares polynomial regression technique can be applied to different types of cluster validity index and clustering algorithms. These two techniques are generic in the sense that they do not alter the working principles of the existing algorithms but instead improve their efficiency and accuracy.

5.5 Preliminary Evaluation of the Proposed Constrained Clustering Approach

In order to test the applicability and performance of the proposed method, a preliminary evaluation has been conducted using two open-source software systems. The evaluation provides a concrete example to illustrate the general workflow of the proposed constrained clustering approach.

Numerous remodularisation techniques have been proposed in the literature to aid in software architecture recovery but it is hard to evaluate their performance and applicability toward real-world software development. The work by Ducasse and Pollet (2009) presented a comprehensive taxonomy for the state-of-the-art software architecture recovery approaches. The taxonomy is designed in a process-oriented manner such that it allows a potential reverse engineer who wants to reconstruct the architecture of an existing software to clearly understand the existing approaches. Thus, software practitioners can choose the most appropriate approaches that can fit into their problem domain. The taxonomy proposed by (Ducasse & Pollet, 2009) is shown in Figure 5.7.

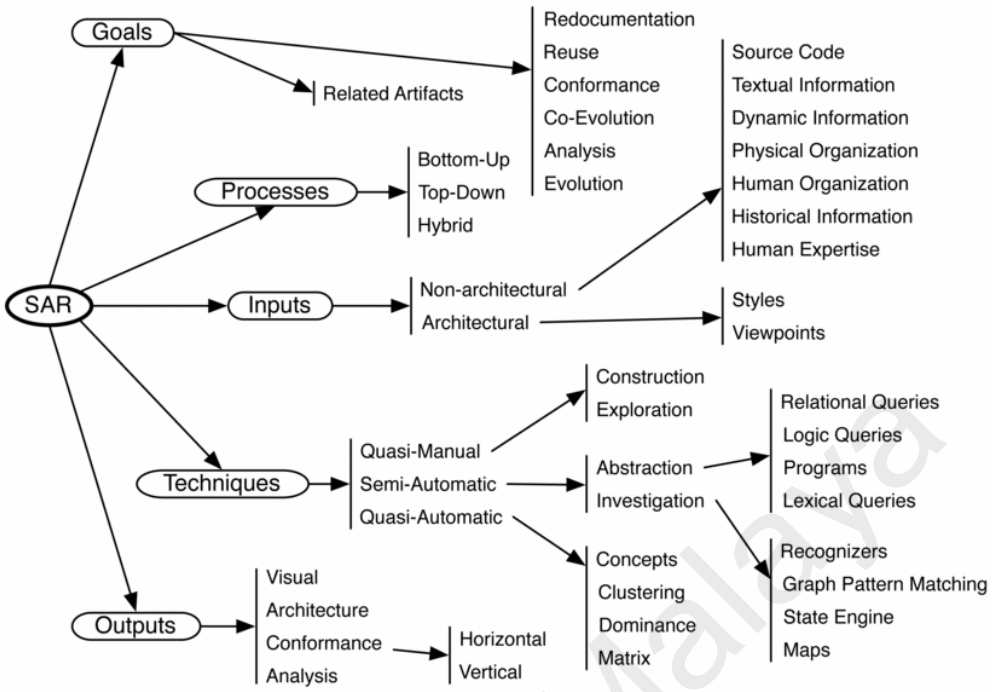


Figure 5.7: Process-oriented taxonomy extracted from Ducasse & Pollet (2009).

The taxonomy consists of several important affiliations, which are the goals, the processes, the inputs, the techniques, and the outputs of software architecture recovery approaches. For each affiliation, it is further categorised into several sub-groups. The taxonomy allows researchers to claim the applicability of their proposed software architecture recovery approaches on real-world problems and provide clear explanations in a process-oriented manner. Thus, the proposed constrained agglomerative hierarchical software clustering approach is described using the same taxonomy.

- Goal: The goal of this study is to recover a high-level abstraction of OO software systems to aid in software maintenance phase. The result of clustering can aid in understanding and analysing the underlying structure of the analysed software. Thus, the goal of this research falls in the category of ‘Redocumentation’ and ‘Analysis’.

- Process: A hybrid constrained agglomerative hierarchical software clustering technique, following a bottom-up approach.

- Input: From the architectural aspect, UML class diagrams of the analysed software are used as the input, which consist of both architecture style and viewpoints.

- Technique: Semi-automatic approach which combines software clustering and domain knowledge in the form of hard and soft constraints.

- Output: Horizontal conformance. The proposed constrained clustering result is verified with the normal software clustering result (without involvement of clustering constraints) and also the original package diagram of the analysed software system. MoJoFM (Wen & Tzerpos, 2004) is used in this research to evaluate the quality of the proposed technique.

As discussed in Section 3.5, MoJoFM is a suitable tool used to compare the similarities between two clustering results. However, Mitchell and Mancoridis (2001) discussed that often time, gold standard does not exist. The authors suggested another approach by clustering the analysed software using different clustering algorithms. Then, the similarities between the results of different algorithms are compared with each other. This will allow software maintainers to identify not only the quality of the clustering results, but also the stability of the clustering algorithm, when compared against other prior studies.

Thus, in this research, the evaluation of the proposed constrained clustering approach is conducted in the following manner:

1. Perform conventional clustering approach that do not make use of any clustering constraints, or commonly referred as unconstrained clustering approach.
2. Perform constrained clustering using the proposed approach by incorporating hard and soft constraints.
3. Retrieve the original package diagram of the analysed open-source software from the project website or repository. Based on the work by Beck & Diehl (2012), a reference decomposition or gold standard can be created by using the current, factual architecture of the system created by the developers (e.g., the package structure of an object-oriented system). The package structure or diagram is by no means the gold standard since there is no way to verify the quality of the decomposition. However, it can be treated as a guideline to evaluate and compare between the results produced by the proposed technique and the documented artifact.
4. Use MoJoFM to calculate the similarities between all three results (unconstrained clustering, constrained clustering, and package diagram).

Two evaluations were carried out to assess the feasibility of the proposed method. First, a university research project, MathArc ("MathArc - Ensuring Access to Mathematics Over Time," August 2009), is chosen for the evaluation. This project is aimed at creating a system that is capable of the long-term preservation and dissemination of digital journals in mathematics and statistics. This system is a joint project by Cornell University Library and Göttingen State University Library, which took two years to develop. The system contains 33 classes with an average of 8 attributes and 4 methods per class. The system's

functional modules can be visually represented as in Figure 5.8. Dotted black boxes represent the original UML packages. There are a total of six subsystems in this software.

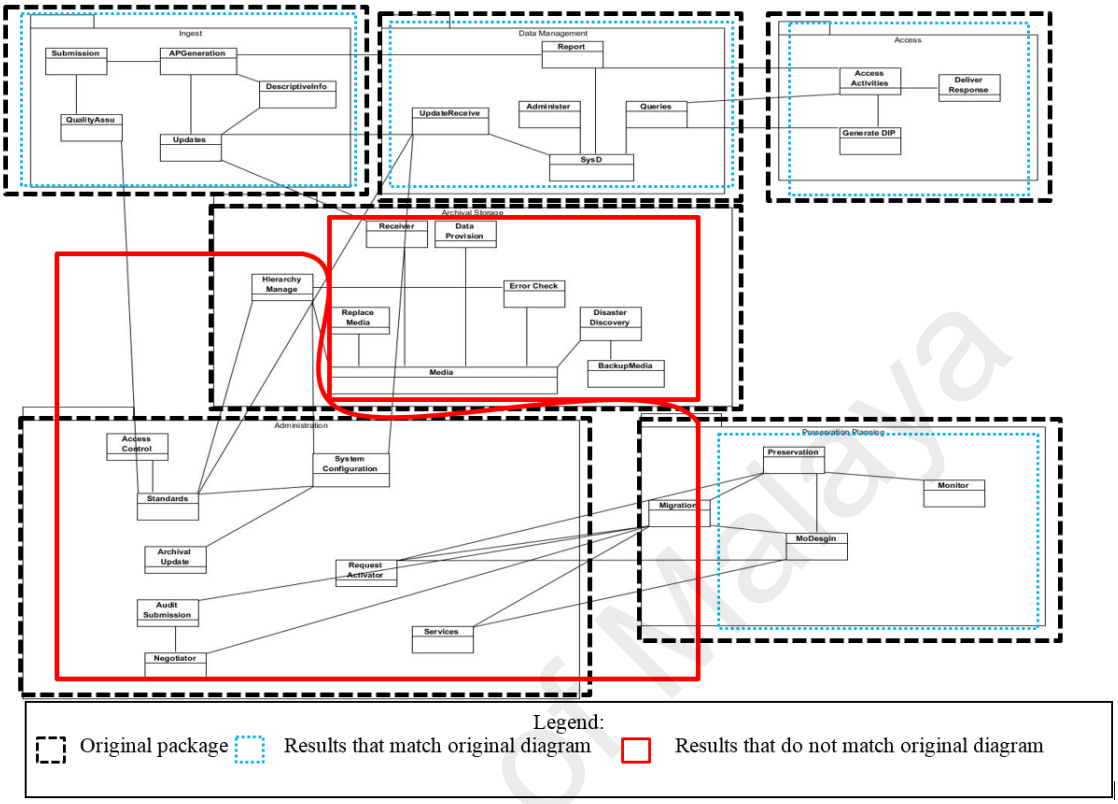


Figure 5.8: Overview of visual representation of the original package diagram and the constrained clustering results

The steps involved in performing the evaluation are as follows:

1. Prior to the evaluation, all the classes are assumed to be scattered around and not grouped in their respective packages.

2. Based on the original UML package diagram, several MLH, CLH, MLS, and CLS constraints are extracted. For instance, based on Figure 5.8, it can be observed that class “Monitor” and “Preservation” must be grouped into the same cluster because they are from the same subsystem. Thus, a MLH constraint “Monitor-Preservation” is generated in Table 5.3.

3. For MLS and CLS constraints, penalty scores for violating the soft constraints are generated randomly. Besides that, an erroneous clustering constraint was created

intentionally, but it is assigned with a very low penalty score to see how the proposed approach handles the fake constraint. For instance, although the original package diagram indicates that "Media" and "Standards" classes belong to different packages, a MLS constraint with low penalty score of 0.1 is created. This MLS constraint simulates the situation where domain experts are not very confident about the given clustering constraint.

4. Apply the proposed constrained clustering algorithm to restructure the class diagram, so that similar classes are grouped into the same package, while dissimilar ones are separated from each other.

5. Use MoJoFM to compare the result of the proposed constrained clustering approach with the original packages to evaluate the accuracy of the proposed approach.

Table 5.3: Generated clustering constraints for MathArc system

Clustering Constraints			
MLH	***CLH***	***MLS(penalty)***	***CLS(penalty)***
Submission-QualityAssu	AccessControl-Submission	Report-SysD(0.3)	Monitor-Negotiator (0.5)
Monitor-Preservation	Report-Services	Standards-AccessControl(0.3)	Submission-Services(0.5)
ErrorCheck-Media		Updates-APGeneration(0.3)	
ReplaceMedia-Media		Media-Standards(0.1)	

5.5.1 Accuracy and Scalability of the Proposed Clustering Approach

However, it is also important to validate the proposed clustering approach with prior studies to evaluate its accuracy and scalability. The proposed dendrogram cutting technique is compared to well-known hierarchical clustering algorithms with a different

linkage method, specifically SLINK and CLINK combined with exhaustive and static dendrogram cutting methods (Greenacre, 2012). The exhaustive cut method cuts the dendrogram, starting from the 0.001 level of similarity and making increases of 0.001 for every subsequent cut, until the cutting point is equal to the maximum height. A static cut method means cutting the dendrogram at a predefined similarity threshold. However, if there is no information on the correct number of clusters and the number of entities inside of a cluster, a static cut is not a feasible choice. Another form of static cut is to cut the dendrogram at the highest gap, which would indicate that those clusters are naturally clustered (Mirkin, 2004). The highest gap refers to cutting the dendrogram at the largest distance between two successive entities. Thus, the highest gap cut is chosen to compare with the proposed dendrogram cutting technique. Table 5.4 shows the result of using SLINK combined with an exhaustive cut, a highest gap cut and the proposed dendrogram cutting method.

Table 5.4: Simulation using SLINK with 3 different dendrogram cutting methods

	SLINK + Exhaustive Cut	SLINK + Highest Gap Cut	SLINK + proposed cutting method
Runtime (in second)	21	3	8
Suggested cutting point	0.38	0.475	0.35
Cluster validity index (enhanced Davies-Bouldin index)	20.15	150.35	21.65

From the table, it can be observed that although SLINK + Exhaustive Cut discovers a cutting point that produces the lowest index value, the runtime is significantly higher compared to the other two methods. SLINK + Highest Gap discovers that the highest gap locates very near to the top of the dendrogram tree. Cutting the dendrogram near the top

of the tree results in a bad cluster validity index values. This situation is very common because these clusters usually have the highest dissimilarity between them in the dendrogram. Figure 5.9 shows the dendrogram tree that is produced by using the SLINK algorithm, which indicates that the highest gap is very near to the top of the dendrogram. SLINK + the proposed cutting method, on the other hand, managed to discover a cutting point that is very close to the exhaustive method but that has a relatively fast runtime.

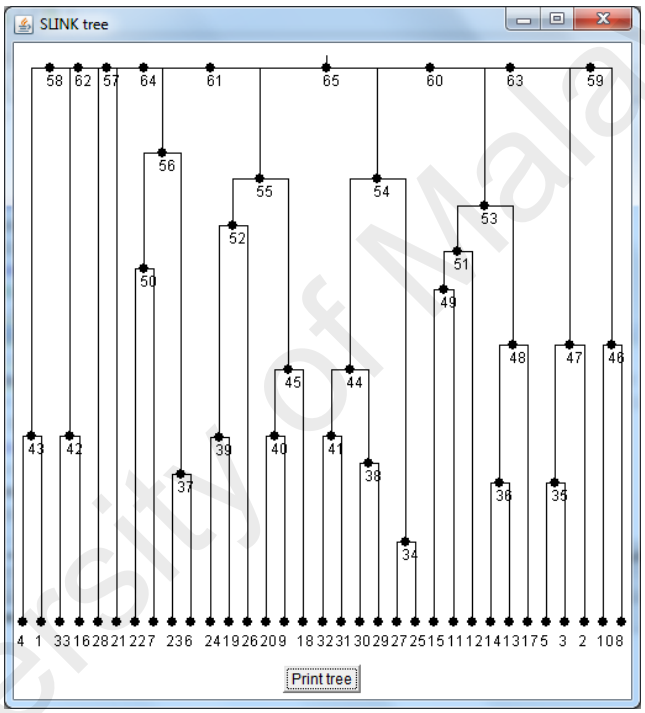


Figure 5.9: Dendrogram tree generated using SLINK

Table 5.5 shows the result of using CLINK with 3 different dendrogram cutting methods.

Table 5.5: Simulation using CLINK with 3 different dendrogram cutting methods

	CLINK + Exhaustive Cut	CLINK + Highest Gap Cut	CLINK + proposed cutting method
Runtime (in seconds)	28	6	11
Suggested cutting point	0.43	0.05	0.44
Cluster validity index (enhanced Davies-Bouldin index)	23.66	5035.68	23.87

Similar to the simulation with SLINK, an exhaustive cut yields the highest runtime. The highest gap using the CLINK method cuts the dendrogram near to the base of the tree and produces a bad result because it forms a substantial number of singleton clusters. Figure 5.10 depicts the dendrogram tree that is generated from CLINK. The proposed dendrogram cutting method, on the other hand, detects a cutting point that is very close to an exhaustive method while the runtime is more than two times faster.

The results shown in Table 5.4 and Table 5.5 match the outcomes of Table B1 through Table B3 (available in Appendix B), which are generated from artificial datasets. These results demonstrate that the accuracy of the proposed dendrogram cutting method is on par with the exhaustive cut method while having the capability of minimizing the efforts that are required to find the optimum cutting point.

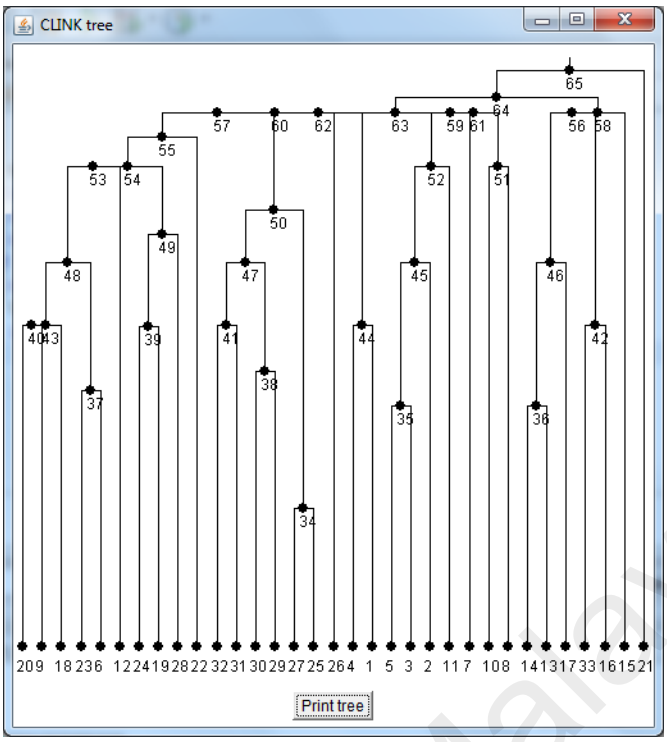


Figure 5.10: Dendrogram tree generated using CLINK

By observing Figure 5.9 and Figure 5.10, it is found that the dendrogram trees generated from SLINK and CLINK produce a completely different hierarchy. SLINK and CLINK are based on opposite philosophies, where SLINK defines the similarity measure of two clusters as the maximum coefficient between all of the pairs of entities, while the reverse is true for CLINK. Based on the work by (Hughes, 1979), if two different clustering methods generate an entirely different tree, the clusters are said to be weakly defined. This result indicates that the MathArc case study does not demonstrate a well-defined architecture, and the grouping of clusters is not as distinct. Thus, evaluation using the UPGMA method will be more appropriate.

Next, to evaluate the effectiveness of the penalty mechanism, evaluations were conducted using two other well-known cluster validity indices, namely the Davies-Bouldin index and Dunn's index (Dunn, 1973). Dunn's index is calculated by dividing the minimum

inter-cluster separation by the maximum intra-cluster cohesion. A higher Dunn's index indicates a better clustering result, which is opposite from the Davies-Bouldin index. The results gathered from all three validity indices were analysed to find out the effectiveness of the penalty mechanism that was introduced.

To compare all of the three indices fairly, the dendrogram tree is constructed using UPGMA. The dendrogram is then cut using the proposed dendrogram cutting method to form a cutting point-validity index table. The table allows one to observe the variation in the index scores at different cutting points. Table 5.6 shows the index scores of the cutting point-validity index of all three indices.

Table 5.6: Index scores of the cluster validity indices for the MathArc system

Cutting point	Davies-Bouldin	Dunn's Index	Enhanced Davies-Bouldin (with the proposed penalty mechanism)
0.072	0.021	∞	94.958
0.126	0.133	0.203	88.333
0.134	1.088	0.187	85.571
0.144	1.484	0.18	81.984
0.168	59294.83	0.129	59350.22
0.198	46914.07	0.129	46967.53
0.215	33354.08	0.135	33410.08
0.226	36865.2	0.132	36916.78
0.251	38913.63	0.12	38957.19
0.296	4360.97	0.123	4401.322
0.301	4637.941	0.098	4668.566
0.306	4949.625	0.087	4975.759
0.318	5743.269	0.065	5758.346
0.324	6224.793	0.05	6232.959
0.329	6751.137	0.054	6760.046
0.332	7435.395	0.077	7439.595
0.373	8254.413	0.099	8259.08
0.389	9283.049	0.096	9288.299
0.408	8.051	0.157	14.051

0.41	77.156	0.198	80.682
0.421	93.987	0.053	93.987
0.438	109.906	0.078	109.906
0.462	144.017	0.053	144.017
0.466	5.326	0.168	120.572
0.481	9.22E+15	0	9.22E+15

It should be noted that the Davies-Bouldin index and the enhanced Davies-Bouldin index favour the minimisation of the index value while the Dunn's index favours the maximisation of the index value. As can be seen from the table, the original Davies-Bouldin index and the Dunn's index tend to favour cutting the dendrogram at low levels of similarity. These cutting points range from 0.072 to 0.144, which yield a low Davies-Bouldin index and a high Dunn's index. However, these four cutting points are in fact forming a substantial number of singleton clusters. Because of the lack of singleton cluster detection, both of the indices incorrectly assume that those are the optimum cutting points. The proposed enhanced Davies-Bouldin index, on the other hand, penalises the formation of singleton clusters to prevent software maintainers from choosing those four cutting points. As such, the evaluation shows that the proposed enhanced Davies-Bouldin index coupled with the penalty mechanism are able to effectively prevent the formation of singleton clusters, when compared to two other cluster validity indices.

5.5.2 Evaluation Result for MathArc System

Figure 5.8 shows the clustering results using the proposed approach. The blue and red boxes represent the results of the evaluation, with each box representing one subsystem. The blue boxes indicate the clustering results that match the original package diagram, while the red boxes indicate the mixture of results that match and do not match the original

package diagram. The diagram was redrawn to normalise all of the association, aggregation, and generalisation into the form of normal association notation.

Note that all the MLH and CLH constraints are fulfilled in the result. However, the MLS constraint of "Media-Standards was violated. This is because based on Davies-Bouldin index, fulfilling the MLS constraint of “Media-Standards” will result in low cohesion strength among the associated clusters. Since the cost of violation is relatively smaller, selecting another cutting point that violates this MLS constraint is a better option. The objective function of soft constraints in this evaluation is $f(Z) = [(5/6) - (0.05)] = 0.7833$. The first operand signifies that 5 out of 6 soft constraints are fulfilled. The value of 0.05 is calculated based on the penalty score of violating the constraint “Media-Standards” and multiplying it with scaling constant of 1/2.

By using the MoJoFM tool provided by Wen and Tzerpos (2004), the evaluation result using the proposed constrained clustering method managed to achieve MoJoFM metric of 92.59% when compared against the original package diagram. The MoJoFM favours high metric value where a 100% score is given if both clustering results are identical. However, as mentioned earlier, the original package diagram is by no means the ‘gold standard’ because there is no way to verify if it is the best abstraction to represent the software design of MathArc system. Thus, another evaluation is performed by comparing the results without imposing any constraints. The result can be visually represented as shown in Figure 5.11.

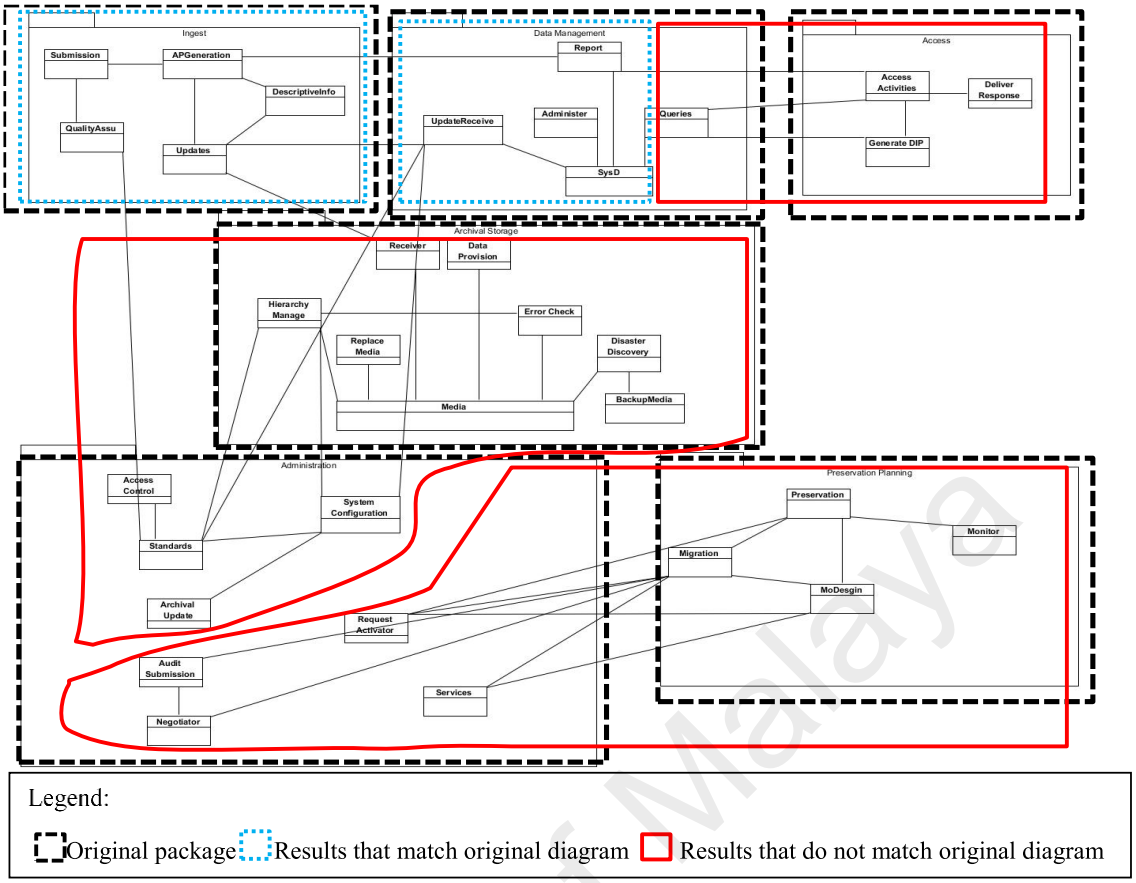


Figure 5.11: Overview of visual representation of the original package diagram and the clustering results without clustering constraints

Based on Figure 5.11, it can be observed that the 'Administrator' package (lower left hand side) contains classes from two other packages. This is because these classes behave similarly to utility classes, for which the association strengths within the same package are relatively weak compared to the other packages. When compared with the original package diagram, the MoJoFM achieves value of 88.89%. Although there are slight improvements when using the proposed constrained clustering technique, it is not significant enough. Thus, another evaluation is performed using a larger software.

5.5.3 Evaluation Using JSPWiki Project

An open-source project, the JSPWiki which is a Wiki engine written in J2EE component is chosen. Wiki engines are used to host and manage Wiki web pages. JSPWiki contains 42560 lines of code and 425 classes with an average of 5.5 methods per class. A total of 326 out of these 425 classes were removed in this evaluation because it was discovered that there were some classes that do not have any direct dependency with other classes. These classes are either standalone features or classes with very specific functions that do not have any interaction with other components in the system.

A total of 15 MLH and CLH constraints, and 5 MLS and CLS constraints were extracted from the original package diagram of JSPWiki. The constraints are listed in Table 5.7.

Table 5.7: Clustering constraints derived from the JWPWiki project

Clustering Constraints			
MLH	***CLH***	***MLS(penalty)***	***CLS(penalty)***
GroupCommand-AbstractCommand	Workflow-TemplateDirTag	Tast-Outcome (0.3)	Command-WikiEventUtil(0.2)
AbstractCommand-WikiCommand	MailUtil-Entry	WatchDog-RSSThread(0.3)	WikiPrinciple-WikiPage(0.3)
UserCheckTag-WikiServletFilter	Workflow-CommandResolver	PageManager-EditorManager(0.2)	Step-ParseException(0.3)
AdminBeanManager-WikiEngineEvent	PageRenamer-Entry	Feed-RSS20Feed(0.1)	UserBean-Editor(0.1)
UserDatabase-WikiSession	Workflow-WikiRPCHandler	Editor-RSSGenerator(0.1)	BlogUtil-FileUtil(0.1)
Entry-AclImpl	MessageTag-Denounce		
WikiSession-UserProfile	MessageTag-Entry		
FormClose-FormSelect	FileUtil-RPCCallable		
FormElement-FormSet	Heading-MarkupParser		
FormOutput-FormOpen	Heading-ProviderException		
FormInput-FormTestArea	SecurityVerifier-WikiException		
InsertPage-TableofContents	FileUtil-ClassUtil		

Clustering Constraints			
MLH	***CLH***	***MLS(penalty)***	***CLS(penalty)***
Entry-FileSystemProvider	BasicPageFilter-CoreBean		
InitializablePlugin-Plugin	Util.PageSorter-Outcome		
TemplateDirTag-WikiRPCHandler	Outcome-Feed		

However, due to the large number of classes exist in the project, the size of the class diagram is too large to be displayed. Only the MoJoFM metrics are reported. The full details of the evaluation results can be accessed by the following URL: http://sourceforge.net/projects/umltocomplexnetwork/files/.

MoJoFM metric: Constrained clustering compared to original package = 76.25%

MoJoFM metric: Unconstrained clustering compared to original package = 62.45%

The improvement by imposing pairwise constraints, observing from the results of MoJoFM metric, is more significant in larger software systems. The same observation was also found in the work by Davidson and Ravi (2009), where the author discovered that when performing on large datasets, a small number of clustering constraints can significantly improve the results of agglomerative hierarchical clustering.

5.6 Chapter Summary

This chapter presents a method to integrate explicit and implicit constraints with agglomerative hierarchical software clustering. The proposed method is capable of handling four types of constraints, namely MLH, CLH, MLS, and CLS constraints. Hard constraints are fulfilled throughout the whole clustering process while soft constraints are optional constraints associated with some validation of penalty if they are violated.

The proposed approach has been successfully implemented on two projects, the MathArc and JSPWiki system. Comparisons against prior studies were also conducted to evaluate the effectiveness of the proposed dendrogram cutting method and the penalty mechanism. Several MLH, MLS, CLH, and CLS constraints were generated to test the proposed technique. When compared against unconstrained clustering approach, the proposed approach managed to achieve better results measured using MoJoFM metric. The proposed method is designed to be generic and flexible enough to be applied on different domains. For instance, the fulfilment of MLH and CLH constraints is not domain specific and can be adapted to be used on a different field of study. The fulfilment of MLS and CLS constraints, on the other hand, can be extended to other MCDM resolution techniques, such as goal programming, scoring methods, and multi-attribute value functions. The penalty score of violating a particular soft constraint can be adjusted to fit into the domain of study. In the next chapter, the full-scale experiment, along with the design decisions are discussed in detail.

CHAPTER 6: EXPERIMENTAL DESIGN AND EXECUTION

In this chapter, experiments to evaluate the accuracy and scalability of the proposed approach are discussed in detail. This chapter adopts the experimental design and framework discussed by Wohlin et al. (2012), which follows a systematic approach to conduct software engineering research. The chapter starts by discussing the implementation plan of the proposed approach. Next, implementation of the prototype is carried out using real datasets gathered from open-source projects. The experimental results are evaluated and discussed in the next chapter.

6.1 Experiment Scoping

This research follows an empirical research methodology where the proposed approach is validated using real-world OO software systems. Therefore, the scope of the experiment must be pre-determined in order to ensure that the experiments conducted are aligned to the goal of the research.

6.1.1 Goal Definition

The first step is to decide whether an experiment is a suitable way to analyse the problem at hand. In this thesis, the objective of the empirical study is to determine the suitable measure constructs to represent OO software systems using weighted complex network, and perform the graph theoretical analysis to reveal some extra deterministic information about relationships among classes. This information is then used to support the subsequent constrained clustering approach to form cohesive clusters and improve the overall effectiveness of software clustering.

The experiment is motivated by the need to understand how constrained clustering can help in recovering a high-level abstraction of OO software systems, as compared to unconstrained approaches. It is well known in existing studies that even a small amount of domain knowledge of the software in the form of pairwise constraints can help improve the clustering results. However, there has been much less attention focused on how to automatically derive clustering constraints without human intervention and ways to fulfil clustering constraints in the domain of software engineering. As such, it is important to formulate a proper method to derive explicit constraints from domain experts, and also implicit constraints from the implicit structure of software systems to form highly cohesive clusters that are representative enough to illustrate a high-level abstraction view of the OO software systems. The goal of the experiment can be expressed as follows:

Object of study – The object of this research is to propose a constrained clustering approach facilitated by the use of weighted complex network analysis.

Purpose – The purpose of the experiment is to evaluate the accuracy and scalability of the proposed approach when compared to an existing unconstrained clustering approach.

Perspective – The perspective of the experiment is from the point of view of software maintainers. Software maintainers can identify if there are any statistical differences when domain knowledge in the form of pairwise constraints are integrated into software clustering. They can also observe how graph theoretical analysis can reveal some extra deterministic information about relationships among all the involved classes, with the purpose to provide a high-level abstraction of the analysed software.

Quality focus – The main result studied in the experiment is the cohesiveness of the clusters formed by the proposed constrained clustering approach. The formation of clusters is based on the explicit constraints derived by domain experts and also implicit

constraints derived automatically using the proposed method. In terms of software quality, two specific aspects are emphasised, which are the maintainability and reliability of the analysed software system.

Context – A total of 40 open-source Java software systems are chosen in this study. The sizes of the software systems vary from 128 to 2,408 classes and 7,436 to 216,093 lines of code. The software systems are chosen to reflect some representative distribution on the population of open-source OO software available in the market, based on the following class count categories:

- less than 250 classes – 7 projects
- between 250-500 classes – 11 projects
- between 501-1000 classes – 14 projects
- more than 1000 classes – 8 projects

As this research is based on an exploratory study, the selected software systems must be of high quality and reputable among the open-source communities. As it is, all the 40 software systems are being actively developed and maintained by a large number of open-source contributors.

6.1.2 Summary of Scoping

In order to provide a clear and concise summary of the scope for this research, the Goal Question Metric (GQM) approach proposed by Van Solingen, Basili, Caldiera, and Rombach (2002) is adopted. GQM approach emphasises eliciting goals and research questions to find necessary metrics for addressing the identified goals and questions. The GQM approach is usually presented in the following template as follows:

Analyse <Object(s) of study>
for the purpose of <Purpose>
with respect to their <Quality focus>
from the point of view of the <Perspective>

in the context of <Context>.

Hence, the template is adopted and modified based on the research goal discussed in Section 6.1.1.

Analyse *the outcome of the proposed constrained clustering approach*
for the purpose of *evaluation*
with respect to *the maintainability and reliability of the analysed software, and the cohesiveness of the clusters formed by the proposed approach*
from the point of view of *software maintainers*
in the context of *open-source OO software systems.*

6.2 Experiment Planning

The next phase is the planning of the experiment study. Details such as selection of context, formation of hypothesis, selection of variables, and the experimental design are discussed in detail.

6.2.1 Context Selection

The use of open-source OO software systems as an experimental context provides several benefits to simulate real-world scenario when maintaining aging and poorly designed software systems. First of all, development of open-source projects is usually based on an ad-hoc basis, such that contributors write their own code to fulfil certain new requirements. Thus, the quality of the source code delivered by contributors usually varies depending on the programming skills of each individual. In return, the maintenance efforts of the selected test subjects are highly dependent on the skills and experience of the open-source contributors. This behaviour is aligned to RO1.3, *To investigate the correlation between the statistical patterns of real-world OO software systems and their level of maintenance efforts.*

Furthermore, in order to improve the generalisation of the experiments and eventually formulate more conclusive experiment results, the test subjects chosen in this research vary depending on their application domain, class count, and lines of code to reflect some representative distribution on the population of open-source OO software systems. All the configuration files, input files, and test results are uploaded to a public domain to allow for replication of the experiments.

6.2.2 Hypothesis Formulation

An important aspect of experiment is to formally state clearly what is going to be evaluated in the experiment. This leads to the formulation of hypotheses. In this research, two hypotheses are formulated.

Hypothesis 1: Given any number and size of test subjects, the constructed weighted complex network based on the proposed approach should be able to demonstrate common statistical patterns of real-world OO software systems. When the test subjects are grouped and compared based on their levels of maintenance efforts, their statistical patterns are more distinguishable.

Hypothesis 2: The proposed constrained agglomerative hierarchical software clustering approach is able to form relatively more cohesive clusters as compared to the unconstrained clustering approach.

6.2.3 Variables Selection

The independent variables are the level of maintenance efforts of the chosen test subjects, and the number of hard and soft constraints derived from the software systems. The dependent variables are the accuracy of software clustering results and the number of fulfilled clustering constraints.

6.2.4 Selection of Subjects

The selection of test subjects greatly affects the results of empirical testing. In this research, quota sampling is used to select OO software systems from various elements of population, such as application domains, lines of code, and number of classes. The chosen software systems have to demonstrate a certain level of quality in terms of maintainability and reliability to allow for baseline evaluations and comparisons. Thus, the number of defects and maintenance costs of the chosen software systems have to be identified to allow for baseline evaluations and comparisons. However, as the selected software systems are open-source projects, it is hard to accurately measure the maintenance costs of the selected software systems in terms of man-day. One alternative to measure the quality of open-source software is by the means of technical debt (Izurieta, Griffith, Reimanis, & Luhr, 2013).

Technical debt, as discussed by Sterling (2010), is related to the issues in software that will hinder future development if left unresolved. Software systems with a high technical debt are at the risk of high maintenance cost. Curtis, Sappidi, and Szynkarski (2012) discussed that it is hard to measure technical debt using a generic measurement because identification of technical debts is based on the structural flaws that software developers intend to fix. Certain developers might just ignore the flaws or fail to recognise the flaws.

Curtis et al. claimed that it is hard to quantify technical debt using a generic algorithm or technique.

The work by Heitlager, Kuipers, and Visser (2007) proposes a model to measure the maintainability of software based on ISO/IEC 9126 standard. A Maintainability Index (MI) is used to quantify the maintainability of software systems by analysing the source code. The algorithm to calculate MI is based on several software metrics, including cyclomatic complexity and average number of lines of code per module. However, this technique only focuses on one particular non-functional requirement stated in ISO/IEC 9126 standard.

The work by Letouzey and Ilkiewicz introduces a method to estimate the technical debt of software systems by examining the source code (Letouzey & Ilkiewicz, 2012). The authors proposed the Software Quality Assessment Based on Life-cycle Expectations (SQALE) method that provides a systematic model to estimate technical debt, and subsequently ranks the severity of debts using five scales, ranging from A to E. Although there are no existing studies that attempt to demonstrate how SQALE ratings can correspond to actual development cost or effort, the work by Lim, Taksande, and Seaman (2012) did demonstrate a considerable finding on how software practitioners view technical debts, and how technical debts are relevant toward the maintainability of software systems. The authors showed that technical debts do play an important role in commercial projects and widely recognised by software practitioners.

The SQALE method uses eight non-functional requirements, namely Testability, Reliability, Changeability, Efficiency, Security, Maintainability, Portability, and Reusability adapted from ISO/IEC 9126, as a reference to estimate technical debt of

software. Software components that do not comply with the non-functional requirements are treated as debts. For each non-functional requirement, there is an estimation of time needed to fix the debt generated from the requirements. The sum of all the identified debts, along with the time estimated to solve them, is quantified as the total technical debt of a software system. As mentioned earlier, the focus of this research is to measure and analyse the maintainability and reliability of the software systems when represented using weighted complex network, which are also software quality attributes covered by the SQALE method. Thus, the inclusion of SQALE method as a basis of measuring the maintenance costs of the selected software systems will allow for better comparative analysis.

To give a more concrete example on how to measure the maintenance effort of software, given a software system with 5000 lines of code. The average cost of developing 1000 lines of code is 100 days, resulting in 500 days for the overall development cost. While analysing the software using the SQALE method, it was discovered that there are several parts of the source code that do not conform to the reliability requirement. The rule "Switch cases should end with an unconditional break statement" is part of the reliability requirement. Given that there are 5 occurrences in source code that violate this rule, and the cost to fix this violation is approximately 1 day each. Thus, the total technical debt for Reliability is 5 days because of the violations. In order to measure the Reliability rating, the technical debt is divided by the total development cost, which is 5days/500days = 1%.

There are several rules for each of the non-functional requirements, such that each of the rules contributes toward estimating the technical debt associated with each non-functional requirement. Hence, eight indices are produced, namely SQALE Testability Index, SQALE Reliability Index, SQALE Changeability Index, SQALE Efficiency Index,

SQALE Security Index, SQALE Maintainability Index, SQALE Portability Index, and SQALE Reusability Index, which estimate the amount of technical debt associated with each of the ISO/IEC 9126 non-functional requirements. In order to provide a high-level indicator based on the ratio between the estimated technical debt and the development cost, all the aforementioned indices are aggregated into a single index called the SQALE rating. A SQALE rating, ranging from "A" to "E", where A signifies high conformance of requirements, is rated. Thus, the overall SQALE rating for all eight non-functional requirements provides a systematic evaluation of the analysed software.

In order to estimate the maintenance efforts of the selected test subjects, all the software systems are evaluated using the SQALE rating method. The evaluations are performed using the SonarQube (SonarQube, 2014) tool, with SQALE plugin installed. In the evaluation, software systems with overall SQALE rating of 0 to <2% are rated as A, while 2% to <4% are rated as B, 4% to <8% as C, 8% to 16% as D, and E for any rating higher than 16%. Below are the results of evaluations extracted from Table 6.1.

Software systems that achieve SQALE rating of A – Apache Maven Wagon, Apache Tika, openFAST, Apache Synapse, IWebMvc, JEuclid, Jajuk, Apache Mahout, Fitnesse, Apache Shindig, Apache XBean, Apache Commons VFS, and Apache Tobago.

Software systems that achieve SQALE rating of B – Apache Karaf, Apache EmpireDB, Apache Log4j, Apache Gora, Eclipse SWTBot, Apache Deltaspike, JFreeChart, Titan, Jackcess, Apache Pluto, Apache Roller, jOOQ, Apache Sirona, Apache Hudson, Apache JSPWiki, Apache Wink, Apache Commons Collections, and Apache Commons BCEL.

Software systems that achieve SQALE rating of C – Apache Rampart, Kryo, Apache Abdera, ApacheDS, Apache Archiva, Apache Helix, Apache Strusts, Apache Falcon, and Apache Mina.

Since most of the selected software systems fall into the range of A-rated and B-rated SQALE rating, it is assumed that the selected software can reveal some of the properties and characteristics of good OO software.

Table 6.1: Summary of selected software systems

No.	Name	Number of classes	Lines of Code	Technical Debt	SQALE Rating
1	Apache Maven Wagon	128	14582	89 days	A
2	Apache Gora	131	8668	112 days	B
3	IWebMvc	178	7436	23 days	A
4	Apache Rampart	191	20585	235 days	C
5	JEuclid	230	12664	20 days	A
6	Apache Falcon	235	20362	276 days	C
7	openFAST	236	11656	63 days	A
8	Apache Commons VFS	280	23059	34 days	A
9	Jackcess	302	21452	180 days	B
10	Apache Sirona	345	57736	428 days	B
11	Kryo	346	23908	339 days	C
12	Apache Pluto	375	25888	193 days	B
13	Apache Commons BCEL	396	28966	325 days	B
14	Apache XBean	401	26845	77 days	A
15	Apache JSPWiki	411	40738	398 days	B
16	Apache Commons Collections	441	26371	321 days	B
17	Apache Tika	457	34558	200 days	A
18	Apache EmpireDB	470	41775	307 days	B
19	Apache Archiva	506	75638	535 days	C
20	Apache Roller	528	55395	532 days	B
21	Titan	532	35415	350 days	B
22	Jajuk	543	57029	58 days	A
23	Apache Mina	583	36978	723 days	C
24	Apache Abdera	682	50568	783 days	C
25	Apache Log4j	704	32987	209 days	B
26	Apache Helix	710	51149	1561 days	C
27	Eclipse SWTBot	731	52841	302 days	B
28	Apache Wink	740	54416	930 days	B
29	Apache Karaf	773	46544	662 days	B

30	Fitnesse	852	47818	112 days	A
31	Apache Tobago	873	53024	239 days	A
32	Apache Shindig	950	54975	98 days	A
33	Apache Deltaspike	1002	31504	502 days	B
34	JFreeChart	1013	95396	670 days	B
35	jOOQ	1106	96520	656 days	B
36	Apache Mahout	1130	82002	143 days	A
37	Apache Synapse	1276	84266	165 days	A
38	Apache Hudson	1492	119005	1173 days	B
39	Apache Strusts	1646	120025	2259 days	C
40	ApacheDS	2408	216093	3664 days	C

6.2.5 Experiment Design

This section discusses the design decisions of the experiment in order to address all the research objectives and hypotheses as clearly and efficiently as possible. Design decisions such as the pre-processing treatment of the datasets along with the rationale behind each decision are discussed in detail. The general design principles are as follows:

Pre-processing: Since the proposed approach transforms software systems from UML class diagrams to weighted complex networks, pre-processing of the source code is needed. A round-trip engineering tool provided by Visual Paradigm is used to transform raw source code into UML class diagrams. Although it is arguably that converting source code into UML class diagrams might result in loss of information, this risk is mitigated by incorporating two software metrics, namely WMC and LCOM4 discussed in Section 4.2.3 toward representing weighted complex networks. Figure 6.1 shows an example of how Visual Paradigm converts Java source code into a UML Class diagram.

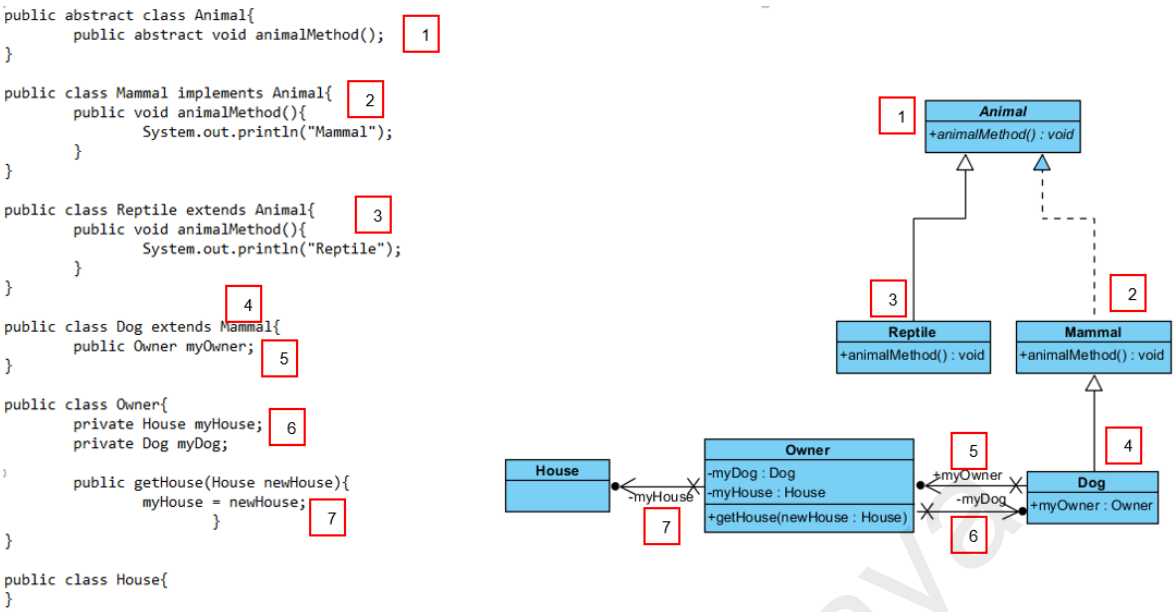


Figure 6.1: Example of Java to UML Class Diagram transformation

1. Abstract class ‘Animal’ with one abstract method.
2. Class ‘Mammal’ implements abstract class ‘Animal’ to create a concrete class. Realisation notation is used to represent the relationship.
3. Class ‘Reptile’ extends ‘Animal’. Generalisation notation is used to represent the relationship.
4. Class ‘Dog’ extends ‘Mammal’. Generalisation notation is used to represent the relationship.
5. An object myOwner of class ‘Owner’ is created. Association notation is used to represent the relationship.
6. An object myDog of class ‘Dog’ is created. Association notation is used to represent the relationship.
7. An input parameter newHouse of class ‘House’ is parsed into the method getHouse(). Association is used to represent this relationship.

Note that Visual Paradigm is unable to reverse engineer composition and aggregation relationships from the source code. Thus, in this research, composition and aggregation relationships in the source code are carefully studied and manually extracted to be

represented in the corresponding weighted complex network. One way to automatically differentiate between composition and aggregation relationships is by checking if deep cloning or shallow cloning is used in the clone() and equal() methods declared in the software.

Shallow copy refers to methods that create a new object that has an exact copy of the values in the original object. If the original object contains references to other object, shallow copy will only copy the associated memory addresses. On the other hand, deep copy copies the complete data structure of the original object recursively and allocates new memory addresses in a different location. From the modelling perspective, deep cloning implies composition relationship while shallow cloning suggests aggregation between two classes (Karsai, Maroti, Ledeczi, Gray, & Sztipanovits, 2004; Porres & Alanen, 2003).

According to the Java API documentation (Oracle), Java provides a <Cloneable> interface which allows cloning of objects. Implementing this interface allows programmers to duplicate objects by calling the clone() method in java.lang.Object class. By default, the clone() method creates a new object instance of the class and initialises all the fields of the new object with exactly the contents of the corresponding fields of the original object. As such, the contents of the newly created object are not cloned directly, which is commonly referred as shallow copy. If the programmer wants to perform a deep copy, he/she has to override the clone() method and give his/her own definition of the cloning operation for all the variables, methods, and constructors, declared in the original object.

However, most of the equal() and clone() methods declared in the chosen test subjects use the default java implementation, which is the shallow clone method. It is possible that the programmers do not actually differentiate between deep and shallow cloning. Therefore, it is impossible to automatically extract all the composition relationships using a simple program parser.

There is a growing interest among the research community to formally identify composition and aggregation relationships for UML class diagrams from raw source code (Milanova, 2005, 2007; Yann-Ga et al., 2004). The work by Milanova has proven that their proposed approach can achieve perfect accuracy when capturing composition relationships from raw source code. However, Milanova suggested that there is no definitive conclusion or solution that can be drawn from the limited and small-scale experiment setup.

According to the UML specification documents, composition and aggregation are specific forms of association relationship between two objects or classes. Composition is referred as the type of association when one object owns another object, as depicted in Figure 6.1 when 'Owner' class owns an object of 'House'. Aggregation, on the other hand, is described as a whole-part relationship between source and target classes (Grand, 2003). The work by Kollmann, Selonen, Stroulia, Systa, and Zundorf (2002) discussed that the best way to recover composition and aggregation relationships from source code is to acquire sufficient knowledge of the software architecture. However, without involving in the development of the selected software systems, one can only rely on the clone() and equal() methods to differentiate the types of relationships. Unless otherwise indicated, the programmers did not differentiate between shallow clone and deep clone methods and it

is unable to distinguish between aggregation and composition relationships, therefore aggregation is used to represent both scenarios.

Figure 6.2 shows an example of how Visual Paradigm converts C++ header files (.h) into a UML Class diagram.

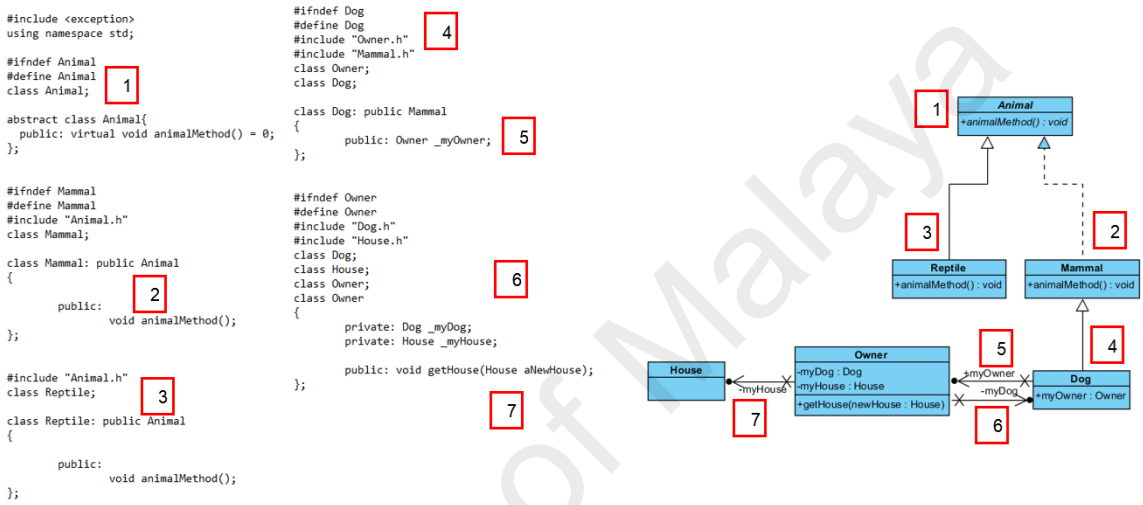


Figure 6.2: Example of C++ to UML Class Diagram transformation

Figure 6.3 illustrates the result of transformation from UML class diagrams to a weighted complex network using the proposed technique, facilitated by Cytoscape, which is an open-source software tool to visualise complex networks (Shannon et al., 2003). Figure 6.3 is represented based on Apache Synapse system which consists of 1276 classes, where each class in the software system is represented by one node. The experimental subjects involve all the maximal connected subgraphs of the software system. However, it is emphasised that the proposed approach can also be applied for forward engineering in software development to provide a better understanding of the software during the early stage of development. The Cytoscape and Visual Paradigm data files of all 40 software systems, along with the retrieved graph-level metrics can be accessed by the following URL: http://sourceforge.net/projects/umltocomplexnetwork/files/

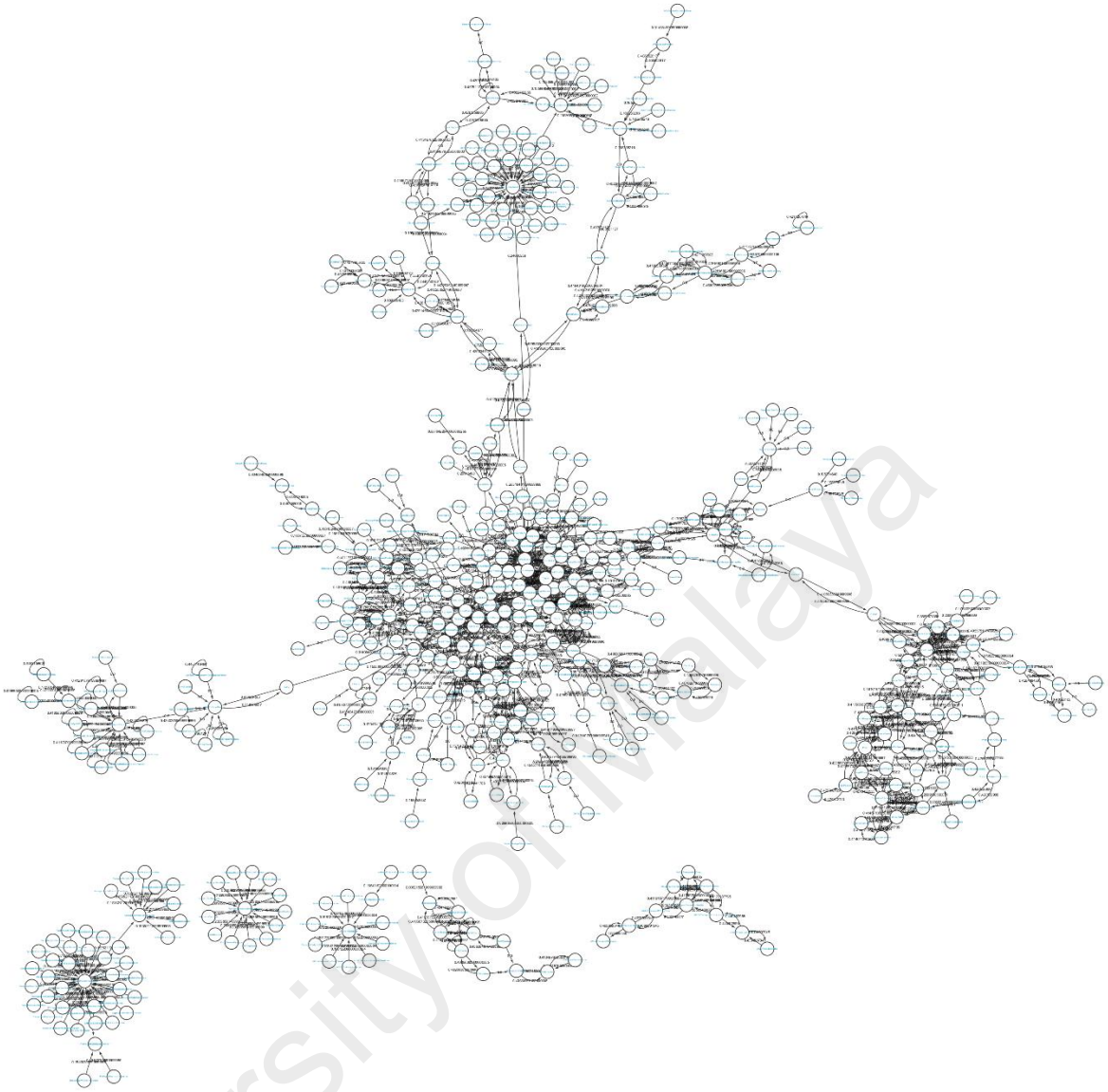


Figure 6.3: Apache Synapse system represented in a weighted complex network using Cytoscape

Randomisation: To recall, MLH and CLH constraints are clustering constraints derived from the implicit structure of the software systems with the aid of graph theoretical analysis. MLS and CLS constraints, on the other hand, are explicit constraints formed based on the domain knowledge provided by the domain experts who have prior knowledge of the analysed software systems. However, due to the fact that there are no prior involvement and development of the chosen test subjects, it is impossible to provide

domain knowledge in this research. Therefore the steps involved to retrieve MLS and CLS constraints are similar to the small experiment done in Section 5.5.

1. Prior to the experiment, all the clustering entities (classes) are assumed to be scattered around and not grouped in their respective packages.
2. Based on the original UML package diagram extracted from the project documentation, several MLS and CLS constraints are extracted.
3. For all MLS and CLS constraints, penalty scores for violating the soft constraints are generated randomly.

While the clustering constraints are generated from an oracle/golden standard, the penalty scores for violating the software constraints are randomised within the range of 0-1 to simulate the importance of the soft constraints.

Blocking: Isolated classes, such as classes without any interactions with other classes are considered as outliers in this research. The isolated classes are removed from the analysis to avoid inconsistent results.

Balancing: It is preferable to have a balanced dataset in order to improve the generalisation of the experimental results. In order to achieve a balanced dataset, 40 software systems were selected in order to reflect some representative class-count distribution on the population of software systems available in the open-source community. In terms of SQALE rating, 13 software systems are A-rated, 18 are B-rated, and 9 are C-rated in order to perform a comparative evaluation between software systems that possess different levels of maintenance efforts.

Standard Design Type: The next step is to decide on the type of experimental design used in this research. The experiment includes one factor of primary interest (quality of

clustering results in terms of intra-cluster cohesiveness and inter-cluster separation) with two treatments (proposed constrained clustering approach and traditional unconstrained approaches). A *completely randomised design* where each test subject randomly uses either the constrained or unconstrained clustering approach to generate a corresponding clustering result is infeasible to be adopted in this research. This is because randomised design does not allow a fair comparison between the two approaches to validate the research hypotheses. Since the goal of the experiment is to investigate if the proposed constrained clustering approach produces clustering results with better intra-cluster cohesiveness and inter-cluster separateness, the example of an experiment design type is better suited in this research. Thus, all the 40 test subjects will undergo two treatments, i.e. the proposed constrained clustering approach and the traditional unconstrained clustering approach, and the clustering results will be compared using several statistical analysis techniques to identify the strength and weakness of the proposed approach.

6.2.6 Instrumentation

In general, there are three types of instruments for an experiment, namely objects, guidelines and measurement instruments (Wohlin et al., 2012).

The experiment objects in this research are the source code and the project documentations of the selected open-source software systems. Source code is used as the input for the proposed constrained clustering approach, while the project documentations are studied to extract the package diagrams of software systems. The package diagrams are used as the benchmark to evaluate the quality of clustering results with the aid of MoJoFM metric.

The MoJoFM metric defines the distance between two different clustering results of the same software system as the minimum number of Move and Join operations to transform one to the other.

- Move: Remove an object from a cluster and put it in a different cluster.
- Join: Merge two clusters into one cluster.
- Split: Split one cluster into two clusters, simulated by the Move operations.

To provide a more illustrative example, Figure 6.4 shows two different clustering results, Decomposition A which is the gold standard, and Decomposition B as the clustering result of the same software. MoJoFM calculates the number of Move and Join operations needed for Decomposition B to match Decomposition A.

- Move node 4 to the cluster <1, 2, 3>
- Split the cluster <5, 6, 7, 8> into two clusters, containing node <5, 6> and <7, 8> respectively

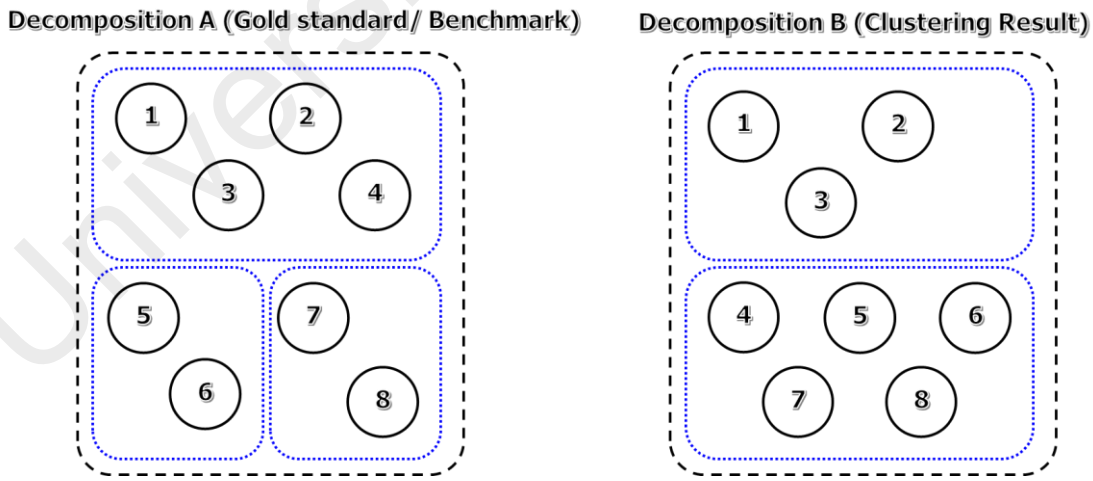


Figure 6.4: Example of MoJoFM operations

Thus, low number of Move and Join operations indicates that the clustering result shows high resemblance with respect to the gold standard. Achieving 100% MoJoFM metric

value indicates that a clustering result is identical to the gold standard. Hence, maximisation of MoJoFM metric values is preferred.

Furthermore, SQALE rating is used to measure and inspect the maintenance efforts of all the software systems in order to provide a means to perform comparative analysis between software of different maintenance efforts.

On the other hand, documented guidelines such as checklists and process descriptions are not required in this research because no participants are involved in the experiments. Instead, with the aid of weighted complex networks, clustering constraints are extracted to help in the subsequent constrained clustering process. The clustering constraints derived from the implicit structure of software systems can be deemed as the guidelines to support the proposed constrained clustering approach.

A prototype to generate and cut dendrogram is developed to serve as the measurement instrument. The prototype takes the input from the graph theoretical analysis of the weighted complex network, and generates the corresponding dendrogram. Two important inputs are taken from the weighted complex network, which are distance between a pair of classes (using Dijkstra's shortest path algorithm discussed in Section 5.4.4) and the clustering constraints derived from the implicit structure of the analysed software. The whole process of generating and cutting dendrogram is automated using the prototype. The algorithm to form a dendrogram is cited from the work by Durbin (1998). The interface class UPGMACluster is shown below to list down several important classes, methods, and attributes used to develop the prototype.

```
interface UPGMACluster:
/**constructor of UPGMACluster
*  id = cluster id, height = cluster height, distance = distance to lower numbered node
*/
public UPGMACluster(int id, UPGMACluster left, UPGMACluster right, double height,
        double[] distance)

/** method to check whether the cluster is empty */
public boolean live()

/** method to find and join the closest live clusters */
void findAndJoin()

/** method to join cluster i,j to form new cluster k */
public void join(int i, int j)

/** methods to draw the dendrogram */
public void draw(Graphics g, int w, int h)

/** methods to find the root of cluster */
public UPGMACluster getRoot()

/** method to write the content of cluster to a file */
public void printRoot()throws IOException

/** method to cut the dendrogram at a specific point */
public void cutTree(double c)
```

Based on the inputs, the prototype will find a pair of the most similar clusters and merge the pair. This process will continue until all the classes are merged into a single cluster, which is performed by using the findAnJoin() and join() methods.

Figure 6.5 shows an illustration of a dendrogram formed by the prototype using 20 classes. The classes (black dots) are tagged in an ascending manner. Values from 1 to 20 represent the classes to be clustered, while values higher than 20 represent the fork nodes

(merging points of clusters). The gaps between clusters are directly proportional to the distance between clusters.

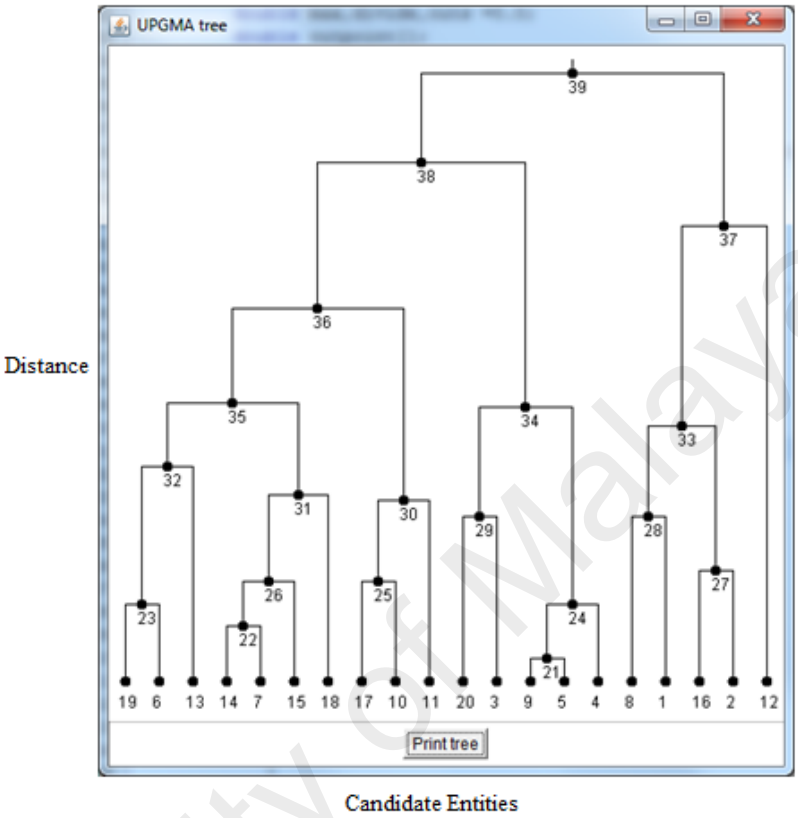


Figure 6.5: Output example of a dendrogram tree with 20 classes

6.2.7 Validity Evaluation

This section discusses threats to the internal validity and external validity. Countermeasures against the threats to the validity were taken and are described below. The internal validity is examined with respect to three aspects, which are the regression toward the mean, the selection of subjects, and the confounding variables.

With respect to the threat from regression toward the mean, the risk is mitigated by evenly selecting test subjects that vary according to their size, class-count and maintenance

efforts to reflect some representative distribution on the population of open-source OO software systems. If the selected test subjects are representative of the population mean, then the threat to regress towards the mean is no longer a valid concern.

To address the threat from subject selection, 40 open-source software systems are selected in this experiment. The test subjects are categorised into four groups – projects with less than 250 classes, between 250-500 classes, between 501-1000 classes, and more than 1000 classes. The chosen software systems are well known projects that are actively developed and maintained by the open-source community. Although it is impossible to guarantee that these software systems are the best examples, good software should exhibit similar behaviour when analysed from a graph-level abstraction.

The choice of code-level, system-level, and graph-level metrics used in this study might impose the threat of confounding variables. The chosen code-level and system-level metrics are WMC and LOCM4 respectively. Both metrics are originated from the CK OO metrics suite and proven to be complimentary (Chidamber & Kemerer, 1994). An in depth analysis of CK metrics is presented in Section 4.2.3, which discussed the development of CK metrics in the past decade, along with its effectiveness in predicting software maintenance cost and software bug prediction. Besides that, the preferences and risk tolerance parameters are introduced to provide more flexibility in obtaining the values of WMC and LCOM4. The chosen graph-level metrics are selected based on their interpretation toward the behaviour of OO software systems. The details of explanations have been discussed in Section 4.2.

Besides that, the choice of cluster validity index, must-link constraints, and cannot-link constraints might also impose the threat of confounding variables. The chosen cluster

validity index in this research is based on the Davies-Bouldin index, which is the ratio of cluster cohesion to cluster separation. According to the work by (Kim & Ramakrishna, 2005; Maulik & Bandyopadhyay, 2002), ratio type cluster validity indices, specifically the Davies-Bouldin index, is among the best indices in terms of performance, reliability, and computational cost. As for the choice of clustering constraints, the steps to extract must-link soft and cannot-link soft constraints are outlined in Section 6.2.5. It is to be stressed that the extraction of must-link soft and cannot-link soft constraints are not arbitrary, but actually based on the original UML package diagrams of the chosen software systems. Thus, this will allow a proper examination of the effectiveness of the proposed approach without bias.

In order to mitigate the threat to construct validity, measure constructs that focus on measuring the maintainability and reliability of software systems are selected in this research. Besides that, the SQALE rating is used to estimate the maintenance costs of the selected software systems in order to facilitate the validation of research hypotheses.

The external validity threats are concerned with the pre-test assumption of removing the isolated classes before performing software clustering, which might result in a biased outcome. There have been claims in several existing studies on software clustering (Patel, Hamou-Lhadj, & Rilling, 2009; Pirzadeh, Alawneh, & Hamou-Lhadj, 2009; Wen & Tzerpos, 2005) that isolated utility classes which can result in ambiguity in the organization of a software system. The study in Patel et al. (2009) also makes a pre-test assumption by removing all of the utility classes before the initiation of a clustering process.

6.3 Experiment Execution

Experiments were carried out based on the design and setup discussed in the previous subsections. Due to the scale of the study and number of test subjects involved in this research, it is impossible to report all the data in this thesis alone. All the raw data are uploaded to a public domain for ease of reading and providing a means to replicate the experiments if necessary. The files are accessible at http://sourceforge.net/projects/umltocomplexnetwork/files/

The analysis and interpretation of the experiment data will be discussed in the next chapter to provide an in-depth analysis of the experiment results.

6.4 Chapter Summary

This chapter discussed the experiment design and setup used in this research. The presentation of this chapter follows the framework discussed by the work of Wohlin et al. (2012), where the authors propose a systematic approach to conduct software engineering research. The goals of this research are formulated using the GQM approach discussed in Section 6.1.2. The selection of context, subjects, and variables, along with the justification of each decision are discussed in detail. Two research hypotheses have also been declared in this chapter. Finally, the experiment design, instrumentation and validity evaluations are presented to provide a clear picture on the flow and design decisions of the experiment. The analysis and interpretation of the experiment data will be discussed in Chapter 7 using descriptive statistics to identify the correlation between the research hypotheses and experiment results.

CHAPTER 7: ANALYSIS AND INTERPRETATION OF EXPERIMENT EVALUATION

After collecting experimental data from the previous phase, the results need to be analysed and interpreted in order to draw a valid conclusion with respect to the research hypotheses and objectives. In this chapter, the experimental results are analysed using descriptive statistics to identify the correlations between research hypotheses and the experimental data. The results are also validated against prior studies in order to evaluate the effectiveness of the proposed methods. Finally, with the aid of MoJoFM metric, the accuracy of the proposed constrained clustering approach is compared against an existing unconstrained clustering approach and the golden standard of the test subjects.

7.1 Graph Theoretical Analysis of Software-based Weighted Complex Network

The first phase of the experiment is to represent OO software systems using weighted complex network based on the approach proposed in Chapter 4. Based on the constructed weighted complex network, software systems are analysed from the aspect of maintainability and reliability using graph theoretical analysis to derive clustering constraints. The graph theoretical analysis provides a means to improve program comprehension from the perspective of software maintainers.

All the 40 OO software systems are represented with their associated weighted complex network. However, due to the scale and size of the diagram, it is impossible to illustrate all the complex networks. Instead the Cytoscape (open-source program used to construct and visualise the weighted complex network) source files are hosted in a public domain which can be accessed using the following URL http://sourceforge.net/projects/umltocomplexnetwork/. The Visual Paradigm source files

which include the class diagrams of all the 40 software systems are also available at the same URL. An example of the weighted complex network is shown in Figure 7.1, which represent a close-up snippet of Apache Gora project. The labels on the nodes are the class names, while the labels on the edges are the weights of all respective edges.

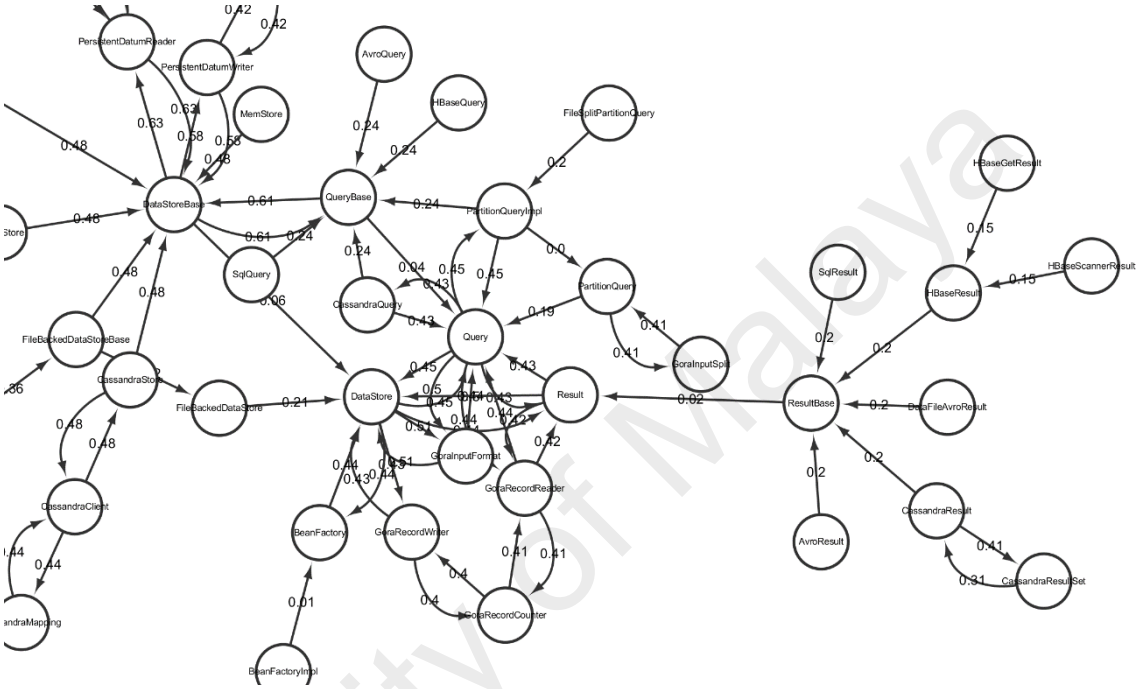


Figure 7.1: Close-up snippet of Apache Gora represented in weighted complex network

The next step is to analyse the constructed weighted complex networks with respect to the selected six graph-level metrics (in-degree, out-degree, average weighted degree, average shortest path length, average clustering coefficient, and betweenness centrality). By analysing the statistical distribution of the graph-level metrics, several common patterns of the selected software systems can be captured. The identified patterns are able to represent certain structural characteristics of the software systems with respect to their maintainability and reliability. The metrics are calculated using the Network Analyser plugin in Cytoscape tool, illustrated in Figure 7.2.

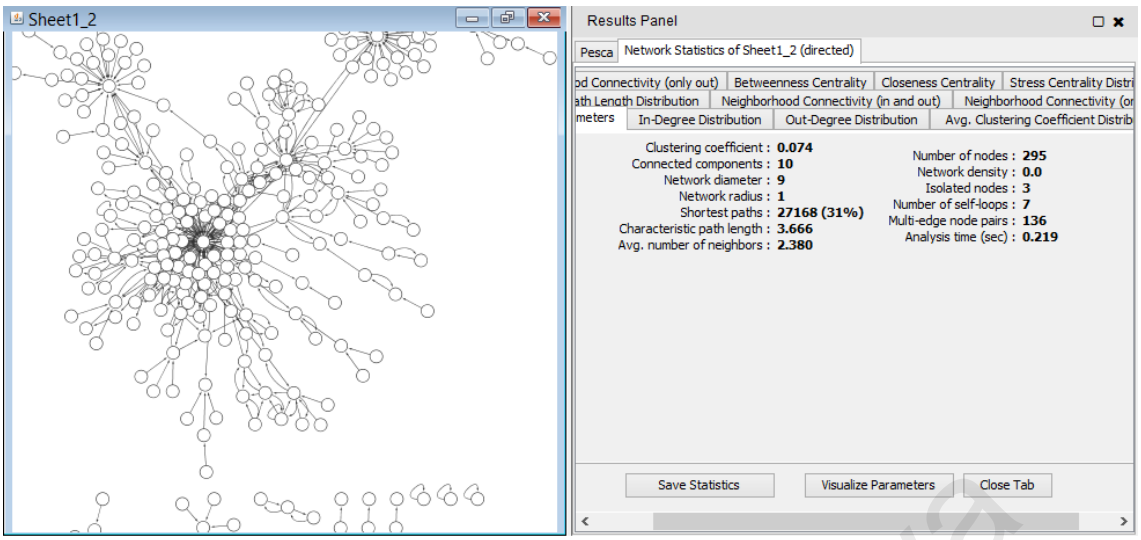


Figure 7.2: Illustration of Graph-Level Metrics extracted from weighted complex network using Network Analyser plugin

7.2 Validation of Findings Against Prior Studies

In Section 6.2.2, the first hypothesis declared in this research is, "*given any number and size of test subjects, the constructed weighted complex network based on the proposed approach should be able to demonstrate common statistical patterns of real-world OO software systems. When the test subjects are grouped and compared based on their levels of maintenance efforts, their statistical patterns are more distinguishable"*. It is important to show that the constructed networks using the proposed approach corroborate with the findings of existing research, such as the expected distribution of in-degree and out-degree of nodes in software-based weighted complex network. This is because validation of experimental results cannot proceed without proof of intentional bias in the interpretation of these results (Kumar & Phrommathed, 2005). Therefore, a way to identify and analyse the statistical distribution of the graph-level metrics retrieved from the constructed weighted complex networks is needed to address the first hypothesis.

7.2.1 Dataset Distribution Fitting

In order to facilitate a better understanding and interpretation of the graph-level metrics, the work by Ferreira, Bigonha, Bigonha, Mendes, and Almeida (2012) is adopted in this research. Ferreira et al. (2012) discussed that although there is a large collection of software metrics available in existing studies, it is hard to evaluate the quality of software systems using software metrics alone because for most metrics, the range of expected or reference values are not known. Therefore in their work, the authors proposed a systematic approach to derive thresholds for six software metrics. The approach is designed to be flexible and generic enough to derive thresholds, and analyse the statistical distribution of these software metrics. Therefore, in this research, the approach proposed by Ferreira et al. (2012) is adapted to identify the statistical distribution of graph-level metrics. The following sections discuss the approach to identify the thresholds of graph-level metrics in detail.

Firstly, the best fit probability distribution of each of the selected graph-level metrics is identified. These best fit distributions are able to show common patterns and structural characteristics of the chosen software systems, and subsequently identify the correlation between graph-level metrics and the associated software quality attributes, i.e. maintainability and reliability.

In terms of the statistical distribution of complex network based on software system, the work by Giulio Concas et al. (2007) found that in-degree follows a Pareto distribution while out-degree follows a log-normal distribution. Both distributions exhibit a power law distribution, which corroborates with several works (Louridas et al., 2008; Valverde & Solé, 2003; Zimmermann & Nagappan, 2008). The reason why in-degree and out-

degree are distributed in a power law manner was further analysed in the work by Chatzigeorgiou and Melas (2012). Chatzigeorgiou and Melas discovered that software follows a 'preferential attachment' where some classes tend to interact with the classes that belong to a similar community or functional groups. The authors claimed that important nodes (high in-degree and out-degree) in a software-based complex network tend to act as attractors for new members that join an existing network.

A distribution fitting tool, EasyFit (MathWave, 2014) is used to fit the datasets into various probability distributions. Once the best fit probability is found, the probability density function (pdf), $f(x)$, is calculated to identify the continuous random variables. Based on the data retrieved from all the 40 chosen test subjects, the Generalised Pareto distribution and the Normal distribution have shown to be best-fitted in this research.

The pdf of Generalised Pareto distribution, $f_g(x)$, is defined in Equation (5) in Hosking and Wallis (1987). The parameters k, σ and μ denote the shape, scale, and location respectively.

$$f_g(x) = \begin{cases} \frac{1}{\sigma}\left(1 + k\frac{(x-\mu)}{\sigma}\right)^{-1-(1/k)} & k \neq 0 \\ \frac{1}{\sigma}\exp\left(-\frac{(x-\mu)}{\sigma}\right) & k = 0 \end{cases} \quad (5)$$

The scale parameter, σ, defines the height and spread of the distribution. The larger the σ value, the more spread out the distribution is.

The pdf of Normal distribution, $f_n(x)$, is defined in Equation (6) in (Stein, 1981). The parameters σ and μ denote the scale and location respectively.

$$f_n(x) = \frac{exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)}{\sigma\sqrt{2\pi}} \quad (6)$$

The analysis of graph-level metrics is done by examining all datasets as a whole (combining all the data retrieved from the 40 test subjects). Two diagrams are generated for each of the graph-level metrics. First, a scatter plot in log-log scale is drawn to show the frequency of each graph-level metric and to identify if the distribution shows a power law behaviour. Existing studies have shown that a candidate power law distribution should exhibit right-skewed properties and an approximately linear relationship on a log-log plot (Stumpf & Porter, 2012). Hence, log-log plots are used in this research to detect the existence of power law behaviour. In the second diagram, the best fit probability distribution for each graph-level metric is illustrated using the EasyFit tool.

7.2.2 Result of distribution fitting for all datasets as a whole

In this section, the results of the computed graph-level metrics and the best fit probability distributions are shown and discussed. Note that the metrics are computed based on all datasets as a whole. The results of each graph-level metric are analysed and discussed with respect to its software maintainability and reliability.

- **In-Degree**

Figure 7.3(a) shows that majority of the classes in the analysed open-source software have in-degree of less than 5, with the mean value at 1.998. The figure also shows an almost linear behaviour, which is the characteristic of power law distribution. The in-degree is best fitted with Generalised Pareto distribution shown in Figure 7.3(b). The goodness of fit based on Kolmogorov–Smirnov (KS) test (Smirnov, 1948) is also shown in Figure

7.3(b). However, it is to be noted that the KS test results show a relatively low significance level due to the large sample size, N>10,000. This is a well-known issue discussed in the existing studies such that for a large sample size, goodness of fit test becomes sensitive to very small and insignificant deviation from a distribution (Bollen, 1990; Inman, 1994; Tanaka, 1987). The parameters k, σ and μ are 0.409, 1.258, and -0.125 respectively. The parameters k, σ and μ are shown in order to provide a means to replicate the experiment if necessary, where the diagram in Figure 7.3(b) can be redrawn using these 3 parameters. Based on the data distribution diagram shown in Figure 7.2, it can be interpreted that majority of the classes do not provide services to other classes because 80% of the classes possess in-degree of less than 3. Low in-degree signifies that most of the tasks are handled locally to promote loose coupling between classes. Although there are some classes that contain significantly higher in-degree values, those classes are typically utility classes that are designed to be reused frequently in the system. All in all, the observed behaviour in terms of high modularity and loose coupling contributes toward improving the maintainability of the software.

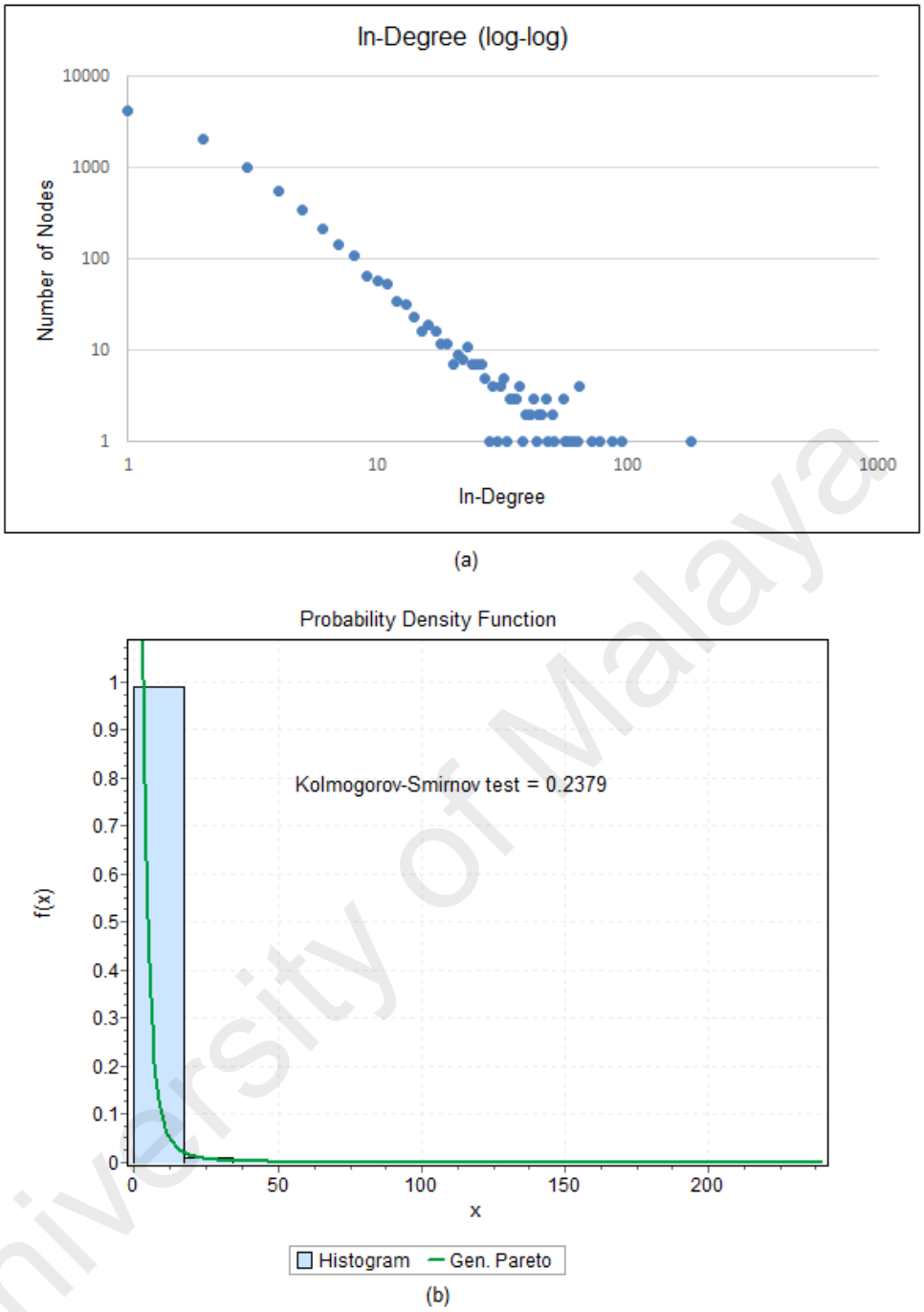


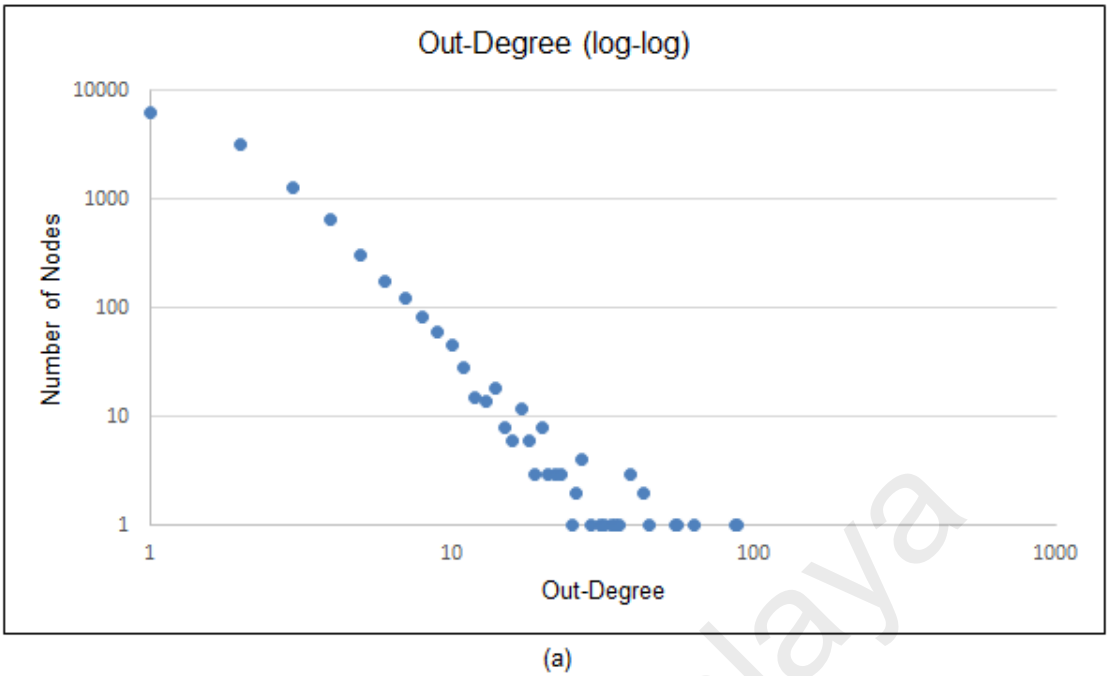
Figure 7.3: In-Degree (a) frequency in log-log scale, (b) fit into Generalised Pareto Distribution

- **Out-Degree**

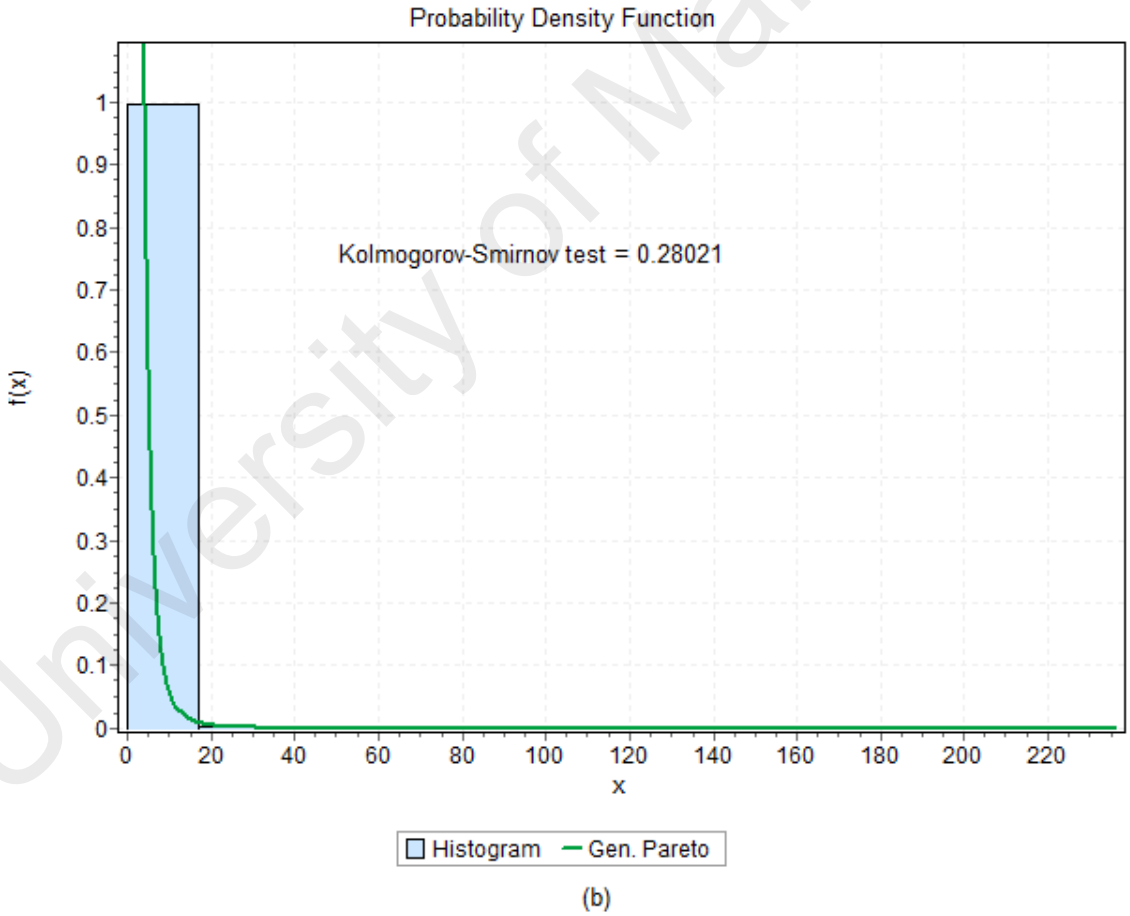
The frequency plot of out-degree in log-log scale is shown in Figure 7.4(a). The average out-degree of the analysed open-source software is at 1.991. The maximum out-degree is roughly 44 times higher than the average value. Figure 7.4(a) also exhibits the characteristic of a power law distribution. The results of data fitting, depicted in Figure

7.4(b), have shown that out-degree is best fitted in Generalised Pareto distribution, with parameters k, σ and μ at 0.2663, 1.161 and 0.420 respectively.

In the in-degree analysis, the maximum in-degree is roughly 90 times higher than the average in-degree. The maximum out-degree is roughly 44 times higher than the average out-degree, which is relatively lower when compared to in-degree. This observation shows several important behaviours of the analysed software systems. Calling classes outside of their package should be minimised to avoid unnecessary coupling. The work by Valverde, Cancho, and Sole (2002) suggests that making use of large hubs, or in this context, nodes with high in and out-degree are bad software design practices, or also known as anti-patterns in existing software engineering literature. If the node becomes too complex, it might become a burden during software maintenance. It is better to break large hubs into smaller and more modular components that focus on specific functionalities, as discussed in the work by Myers (2003). Thus, theoretically the maximum in-degree and out-degree should not deviate too much from its mean value. However, it is unavoidable in certain scenarios where important classes are being repeatedly reused.



(a)

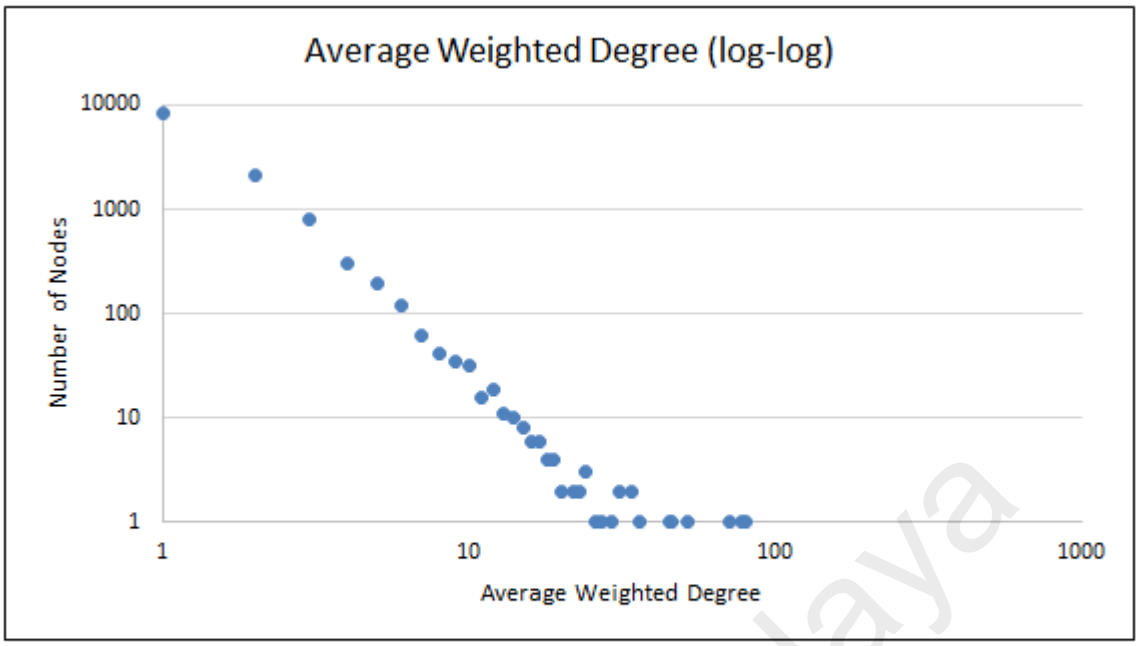


(b)

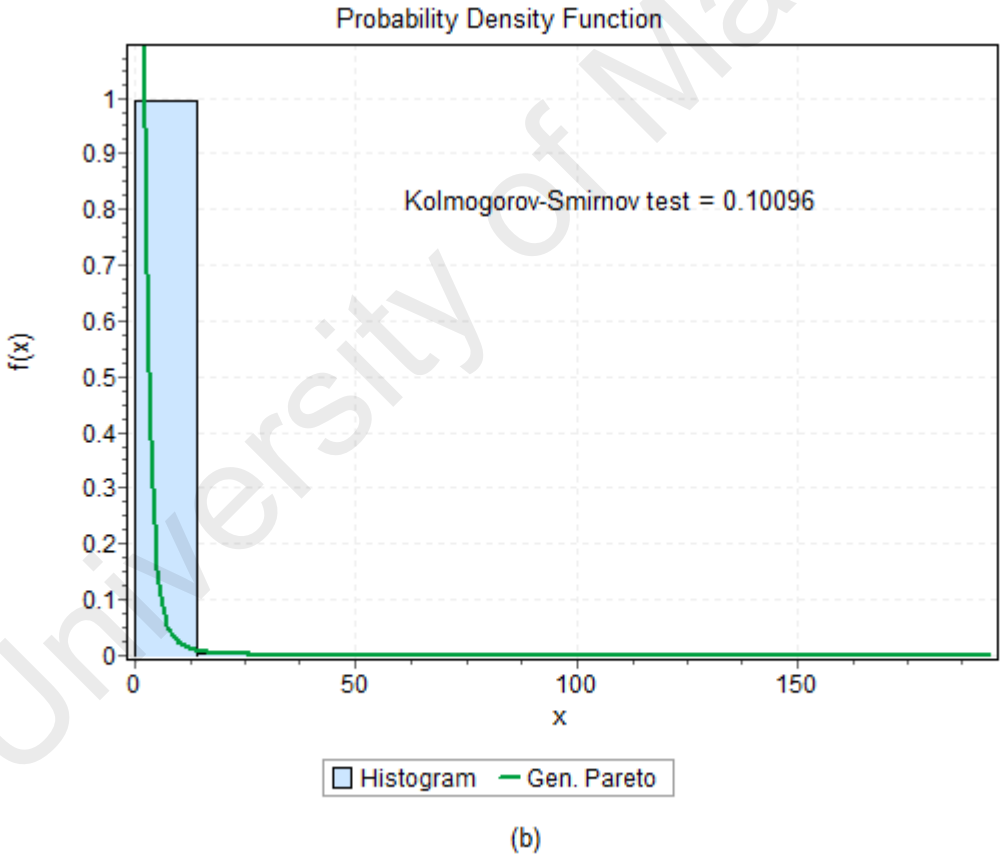
Figure 7.4: Out-Degree (a) frequency in log-log scale, (b) fit into Generalised Pareto Distribution

- **Average Weighted Degree**

Figure 7.5(a) shows the frequency plot in log-log scale, where the mean average weighted degree is 1.071. The maximum average weighted degree derived from all datasets is around 80, which is about 75 times larger than the mean value. Similar to in-degree and out-degree, average weighted degree shows the characteristic of power law distribution. Figure 7.5(b) shows that the average weighted degree of all data can be modelled by the Generalised Pareto distribution. The parameters k, σ and μ are 0.402, 0.658, and -0.002 respectively. The observation shows that majority of the analysed software systems have low average weighted degree, where the probability of classes having a value of less than 5 is very high. Although there are a few nodes with a very high average weighted degree, these classes are usually utility or main system classes that supply services to other classes. Examples of utility classes that exhibit a high average weighted degree are WikiEngine.java of Apache JSPWiki (77.589), Hudson.java of Apache Hudson (51.75), Field.java of jOOQ (70.535) and Logger.java of Strusts (77.23).



(a)

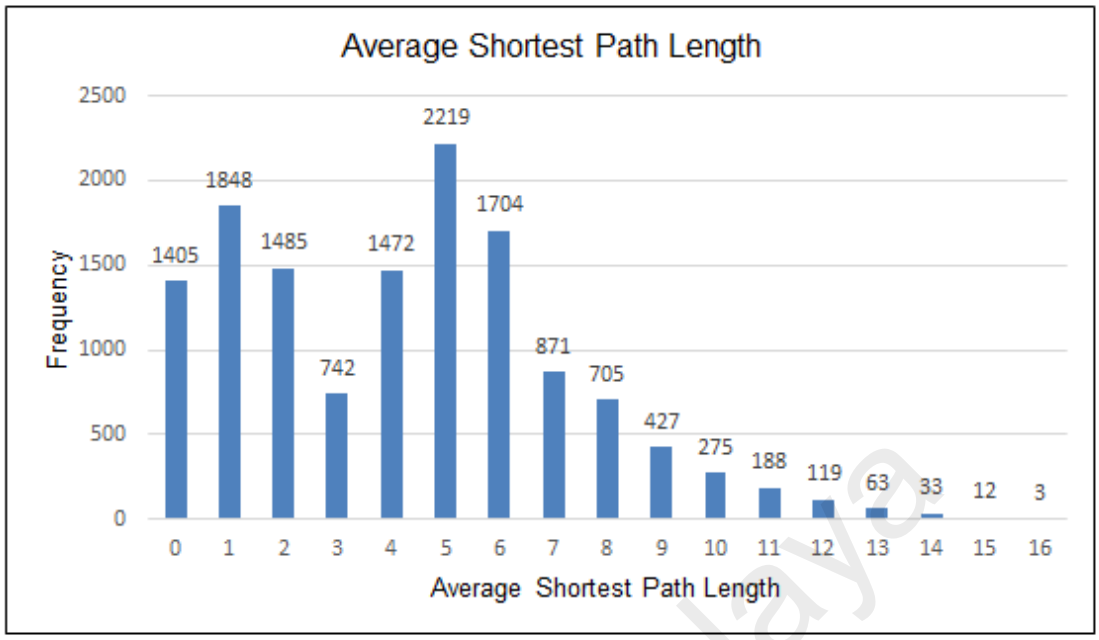


(b)

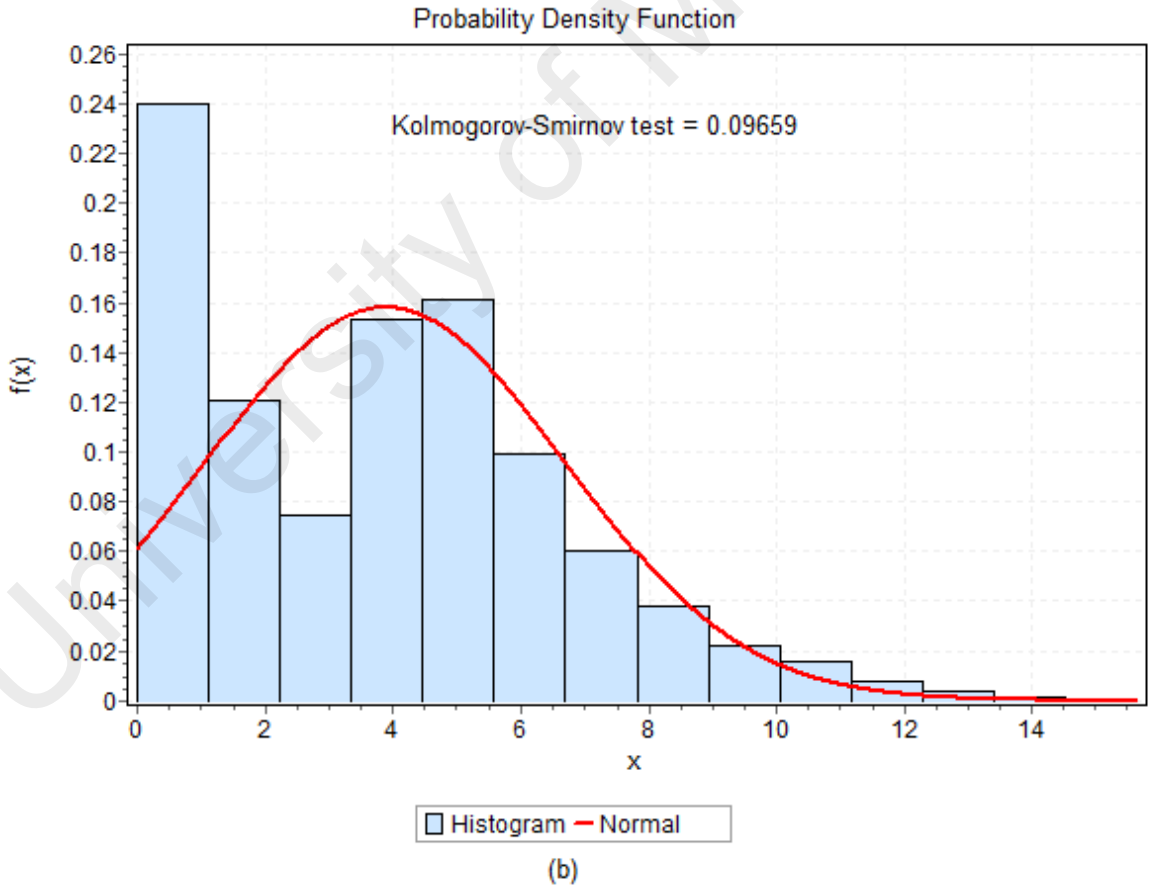
Figure 7.5: Average weighted degree (a) frequency in log-log scale, (b) fit into Generalised Pareto distribution

- **Average Shortest Path Length**

The frequency plot in histogram for average shortest path length is shown in Figure 7.6(a). The mean average shortest path for all datasets is around 3.8 steps, which corroborates with the work by Valverde and Solé (2003) who found that the average shortest path in software is less than 6 steps. The log-log frequency plot for average shortest path length is not included because the average shortest path length is best fitted in normal distribution with parameters σ = 2.816 and μ = 3.877 (Figure 7.6(b)). Figure 7.6(a) shows that the diagram is positively skewed, where the distribution is concentrated on the left of the figure. Typically, power law behaviour is not shown in log-log plot for Normal distributed data. This result demonstrates that most classes in OO software can communicate with each other easily. Low average shortest path length also contributes toward high response capability of the analysed software, especially Apache Deltaspike and Apache Synapse, because they are usually deployed on a web-based environment.



(a)



(b)

Figure 7.6: Average shortest path length (a) frequency in histogram, (b) fit into Normal distribution

- **Average Clustering Coefficient**

Clustering coefficient is a graph-level metric that measures if a given node's neighbours are neighbours among themselves. Average clustering coefficient provides the average score of clustering coefficients of all nodes for the whole network. If the average clustering coefficient of the network is equal to 1, the network is called a clique where each pair of nodes is connected by an edge. Analysing the frequency plot of clustering coefficient is difficult because it does not translate directly into any OO behaviour. Thus, a different approach to analyse the average clustering coefficient is adopted.

Small world properties of a complex network can be determined by looking into the average shortest path length and clustering coefficient. A complex network with a high clustering coefficient has a strong characteristic of small world property. Many studies have found that the average clustering coefficients of OO software are much higher than those of random networks constructed based on the same node property (Giulio Concas et al., 2007; Louridas et al., 2008; Pang & Maslov, 2013; Potanin et al., 2005). This behaviour suggests that software systems possess a higher degree of cohesion with respect to random networks.

The work by Newman (2006) has further proven that clustering coefficients of real-world networks should be higher than what would be expected if edges were randomly placed, using a 'graph modularity' measurement. Newman shows that in real-world networks, the number and density of interactions among nodes belonging to a community are higher than expected in a random network of the same size.

In order to test this particular behaviour discovered by Newman, a Cytoscape plugin developed by Mcsweeney (2008) is used in this research. The plugin is capable of

randomising an existing network, while preserving the number of nodes and edges in the original network. The algorithm used to generate a random network from an existing one is shown below.

1. A random edge *(u,v)* is selected from the network.
2. A second random edge *(s,t)* is selected with the constraints that:
    - $u \neq v \neq s \neq t$
    - *(u,t)* and *(s,v)* do not already exist in the network
3. Edges *(u,v)* and *(s,t)* are removed and edges *(u,t)* and *(s,v)* are inserted into the network.
4. Repeat Steps 1-3 *n* times (where $n = 100$ to generate 100 random networks).

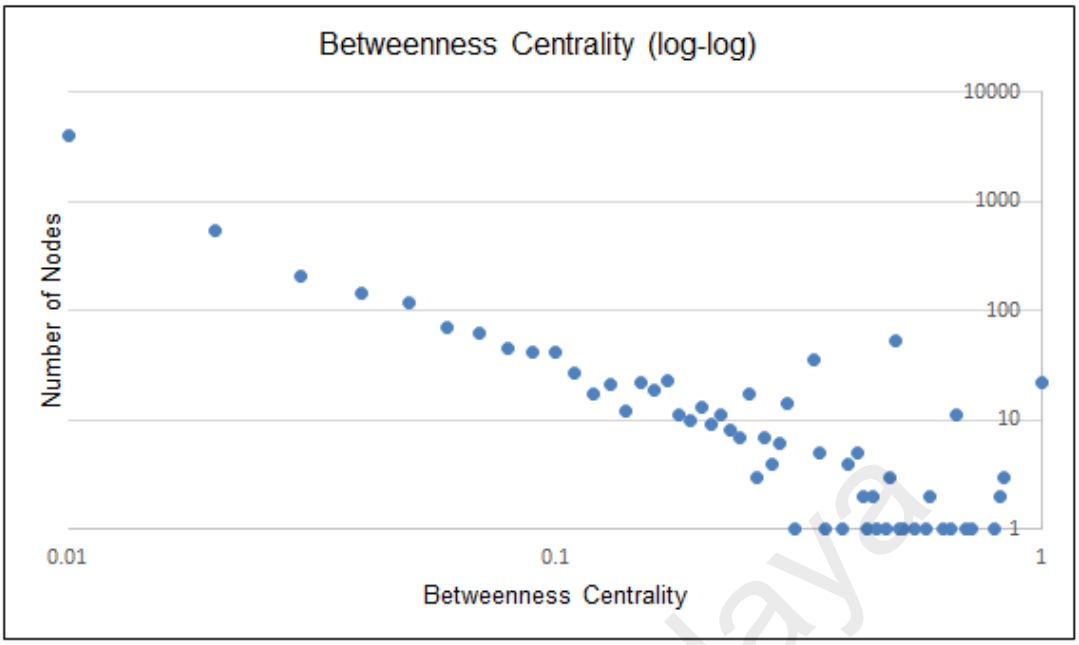
First, the average clustering coefficients of all datasets as a whole are calculated. Next, based on the nodes' properties of the original real dataset (such as number of nodes, number of edges, weights of edges, and directions of edges), 100 random networks are generated. The average clustering coefficients of all the 100 randomised network are then calculated. Finally, the average clustering coefficients of all the real datasets are compared against the average clustering coefficients of all the random networks. Hence, in total, 4000 random networks are generated for all the 40 test subjects.

Based on the test results, the average clustering coefficient of all the 40 test subjects is reported to be 0.048, while the average clustering coefficient for 100 random networks is 0.0012. The test results show that the constructed software-based weighted complex networks do behave like a real-world network and substantiate with the findings of Newman (2006). Combined with the observation from average shortest path length, the

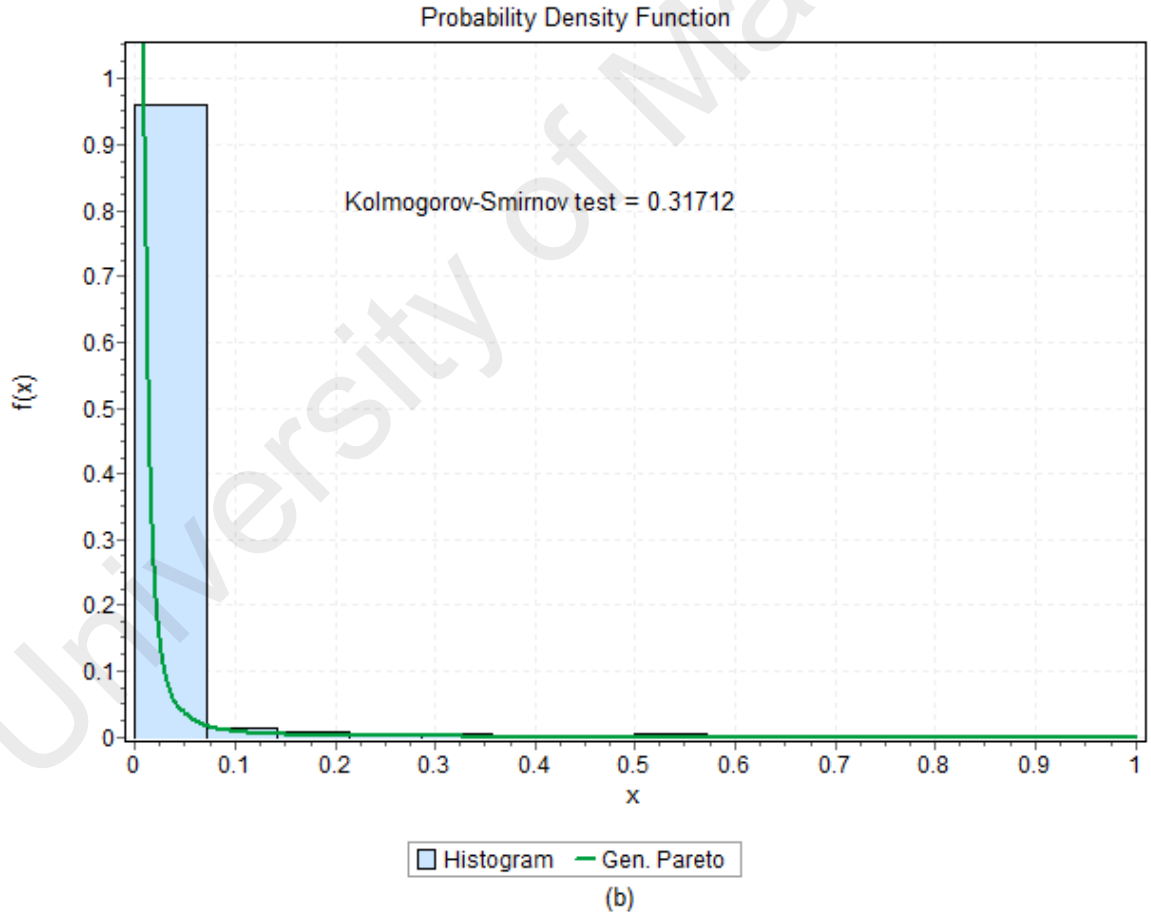
constructed networks using the proposed technique do adhere to small world properties commonly found in the existing literature.

- **Betweenness Centrality**

Figure 7.7(a) shows the distribution of betweenness centrality for all analysed software systems in log-log scale. Note that the betweenness centrality for a given node *n* is normalised by dividing by the number of node pairs in the network, excluding *n*. Thus, the values of betweenness centrality range from 0 to 1. The scatter plot shows that majority of the nodes have value of less than 0.1, which indicates that they do not control the flow of information in the network. Figure 7.7(a) suggests that power law characteristic is present. The data is best fitted in Generalised Pareto distribution as shown in Figure 7.7(b), where the parameters k, σ and μ are 0.887, 0.0016, and -6.63E-4. This observation reveals that most of the classes in OO software do not have dominant power that dictates the flow of information and data. Removing certain components from the software will have minimal impact on the structure stability. Although there are a few nodes with very high centrality value, those nodes are normally interface classes that act as the 'authority classes', as discussed by the work by Ovatman, Weigert, and Buzluca (2011). Ovatman et al. found that classes in UML class diagrams show distinctive recurring patterns in terms of dependencies between each other and to other classes. 'Authority classes' is one of the patterns where a large number of classes are coupled with one another.



(a)



(b)

Figure 7.7: Betweenness Centrality (a) frequency in log-log scale, (b) fit into Generalised Pareto distribution

In summary, the results and discussions presented in this sub-section are able to address the sub-objective 1.3 discussed earlier, where all the statistical patterns of real-world OO software systems can be identified and analysed through dataset distribution fitting. However, certain patterns and behaviour might not be directly visible using the aforementioned approach (Hyndman & Fan, 1996; Williamson, Parker, & Kendrick, 1989). Thus, in the next section, the empirical distributions of the datasets are analysed by examining the quartiles of each respective dataset using boxplot.

7.2.3 Comparative Analysis

Boxplot analysis is one of the statistical methods used to visually identify patterns that may otherwise be hidden in a dataset (Hyndman & Fan, 1996; Williamson et al., 1989). In this study, boxplot analysis is used to perform a comparative analysis of the selected software systems and eventually address the research hypothesis. Recall that the first hypothesis mentioned that *when the test subjects are grouped and compared based on their levels of maintenance efforts, their statistical patterns are more distinguishable*. SQALE rating is used in this research to provide a means to estimate the levels of maintenance efforts of all the chosen test subjects. If the test subjects can be further grouped and analysed based on their respective SQALE rating, it will provide more insight toward understanding the statistical behaviour of software systems, and ultimately address the research hypothesis.

By combining all the data of A-rated projects, a boxplot can be produced to identify the median, the approximate quartiles, spread, and symmetry of the distribution (Williamson et al., 1989). The results of A-rated projects can then be compared against B-rated and C-

rated projects in order to identify the differences in statistical behaviour for these three groups of datasets.

Figure 7.8 depicts the boxplots of all the graph-level metrics for A-rated, B-rated, and C-rated software systems except for Betweenness Centrality. The boxplot of Betweenness Centrality is separated as shown in Figure 7.9 due to the difference in the scale of values.

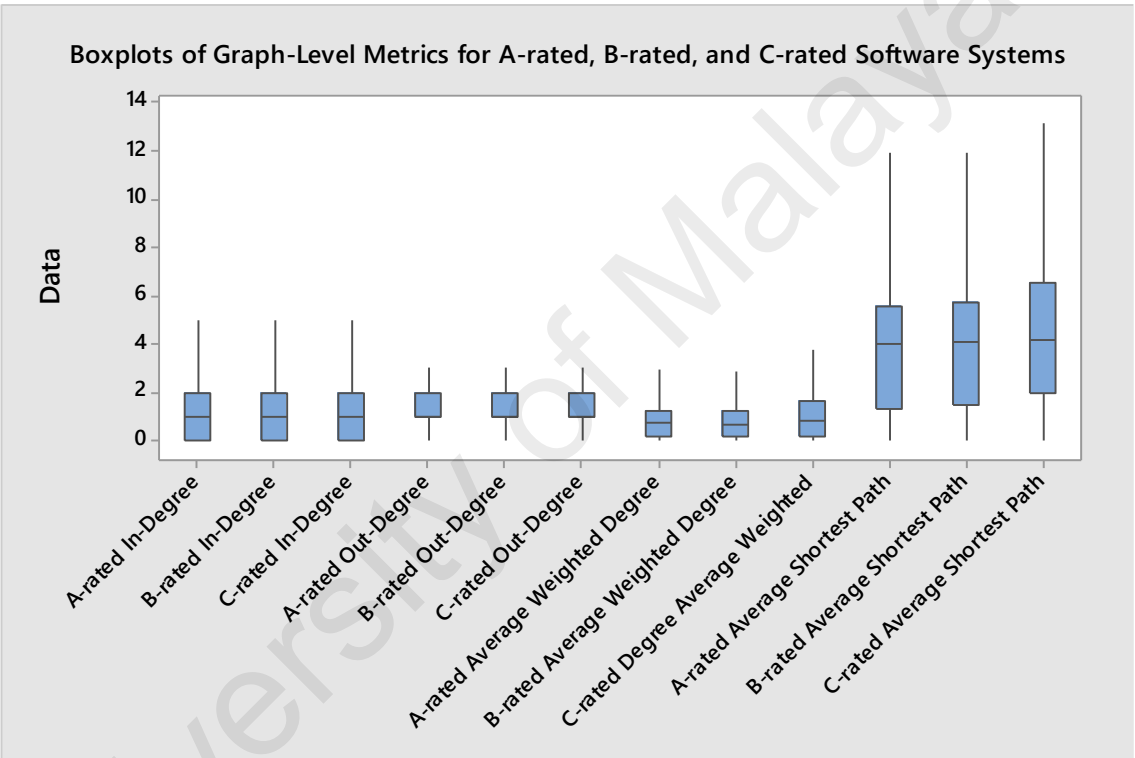


Figure 7.8: Boxplots of In-Degree, Out-Degree, Average Weighted Degree, and Average Shortest Path for A-rated, B-rated, and C-rated software systems

Table 7.1: Analysis of boxplots from Figure 7.8

Metrics	1st Quartile	Median	3rd Quartile	Interquartile Range	Whiskers
A-rated In-Degree	0	1	2	2	0, 5
B-rated In-Degree	0	1	2	2	0, 5
C-rated In-Degree	0	1	2	2	0, 5
A-rated Out-Degree	1	1	2	1	0, 3
B-rated Out-Degree	1	1	2	1	0, 3
C-rated Out-Degree	1	1	2	1	0, 3

A-rated Average Weighted Degree	0.161	0.75	1.282	1.121	0, 296
B-rated Average Weighted Degree	0.158	0.701	1.232	1.075	0, 2.844
C-rated Average Weighted Degree	0.157	0.810	1.618	1.461	0, 3.799
A-rated Average Shortest Path	1.333	4.039	5.562	4.229	0, 11.898
B-rated Average Shortest Path	1.5	4.120	5.690	4.190	0, 11.933
C-rated Average Shortest Path	2	4.187	6.5	4.5	0, 13.098

Table 7.1 presents a summary of analysis including first quartile, median, third quartile, interquartile range, and whiskers of the boxplots from Figure 7.8. Based on the table, there is no variation between the in-degree and out-degree boxplots of A-rated, B-rated, and C-rated software systems. The boxplots of in-degree are positively skewed due to the power law behaviour observed earlier in the frequency distribution plot. The out-degree shows lesser variability, with an interquartile range of 1. Low interquartile range of out-degree is consistent with the observation in Figure 7.4, where it was observed that most of the classes have a similar out-degree value except for a few exceptional cases. This observation shows that developers of A-rated, B-rated and C-rated software systems adhere to the high modularity concept when developing and updating the software systems, as discussed in the work by Myers (2003).

The boxplots of average weighted degree are also positively skewed due to the power law behaviour. When the edges and nodes are weighted, there is a slight variation from the observed quartile, where the whisker of C-rated software systems is slightly higher at 3.799. This indicates that the coupling strength of C-rated software systems is relatively higher when compared to A-rated and B-rated software systems, which might contribute toward their higher maintenance efforts. In terms of average shortest path, the whiskers of C-rated software systems are slightly higher at 13.098 when compared to A-rated and

B-rated software systems (11.898 and 11.933 respectively), which could indicate that several classes have high communication costs. Upon further investigation, it was being discovered that several classes in ApacheDS, namely AvlTreeImpl.java, AvlTreeSingleton.java, ArrayTree.java, KeyTupleArrayCursor.java, and KeyTupleAvlCursor.java, which have an average shortest path length of 12-13 steps. These classes contain methods that depend on the class Index.java, which is located separately in another package.

The boxplots of betweenness centrality for A-rated, B-rated, and C-rated software systems are shown in Figure 7.9, along with the analysis in Table 7.2.

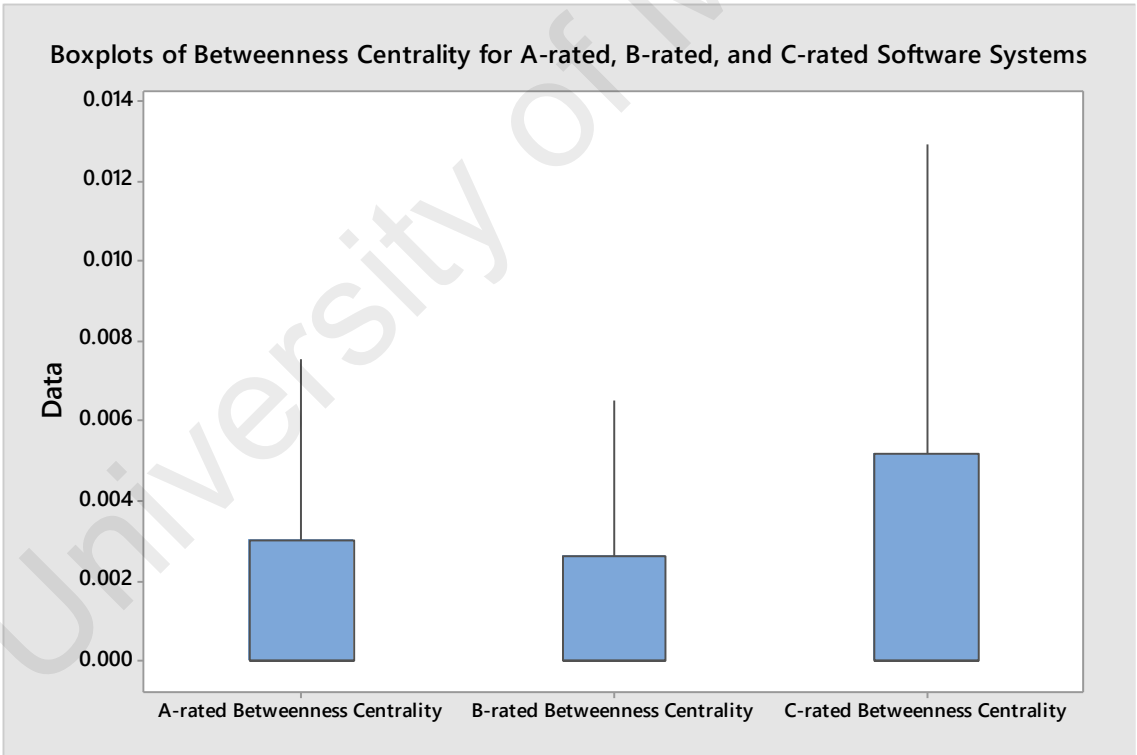


Figure 7.9: Boxplots of Betweenness Centrality for A-rated, B-rated, and C-rated software systems

Table 7.2: Analysis of boxplots from Figure 7.9

Metrics	1st Quartile	Median	3rd Quartile	Interquartile Range	Whiskers
A-rated Betweenness Centrality	0	0	0.00301	0.00301	0, 0.00754
B-rated Betweenness Centrality	0	0	0.00261	0.00261	0, 0.00651
C-rated Betweenness Centrality	0	0	0.00517	0.00517	0, 0.0129

Betweenness centrality measures the number of shortest paths that pass through a selected class. Classes with high betweenness centrality signify that they are important because they usually act as the communication bridge. On the flip side, if these classes are highly error prone, it can easily propagate bugs due to their behaviour (G. Concas et al., 2011). As can be observed from Table 7.2, C-rated software systems have much higher value of betweenness centrality when compared to A-rated and B-rated software systems. Upon further investigation, it was discovered that the ApacheDS project contains a few utility classes that possess high betweenness centrality. These classes are AvlNode.java, Marshaller.java, KeyIntegrityChecker.java, NtpService.java, PasswordPolicyConfiguration.java, PasswordValidator.java, NtpMessage.java, NtpMessageModifier.java, and LdapServer.java. These classes should be given more attention as they might be highly error prone.

- **Comparison of Node-Weighted and Edge-Weighted Approaches**

In Chapter 4, Equation (3) was proposed as a means to measure communicational cohesion-based weights by looking into the complexity of classes and relationships.

$$Weight_{(R_{i\rightarrow j})} = \left(H_{R_{i\rightarrow j}} * \alpha\right) + \left[\left(1 - Comp_{(D_j)}\right) * \beta\right] \quad (3)$$

The equation is based upon a hybrid of node (class) weighted and edge (relationship) weighted approach similar to the work presented by Ma et al. (2010). There are different strategies to represent the weights of nodes or edges, along with their own advantages and disadvantages. Thus, in this subsection, the comparative analysis is further expanded by contrasting the performance of different representation strategies, i.e. node-weighted and edge-weighted approaches.

The first operand of Equation (3) represents the complexity of a relationship, while the second operand represents the complexity of the terminus class linked by the associated relationship. In order to compare the performance of different representation strategies, the following steps are performed.

1. For each software system, calculate the weights of edges by using only the first operand of Equation (3), $\left(H_{R_{i\rightarrow j}} * \alpha\right)$, and repeat the process by using only the second operand of Equation (3), $[(1 - Comp_{(D_j)}) * \beta]$.
2. Reconstruct the weighted complex network. Since there are two representations of weights, there will be two weighted complex networks for each software – node-weighted only network and edge-weighted only network.
3. Recalculate the value of average weighted degree for each network.
4. Perform a comparative analysis of different representation strategies by grouping the software based on their SQALE rating.

Figure 7.10 depicts the result of our analysis and the details of the analysis are presented in Table 7.3.

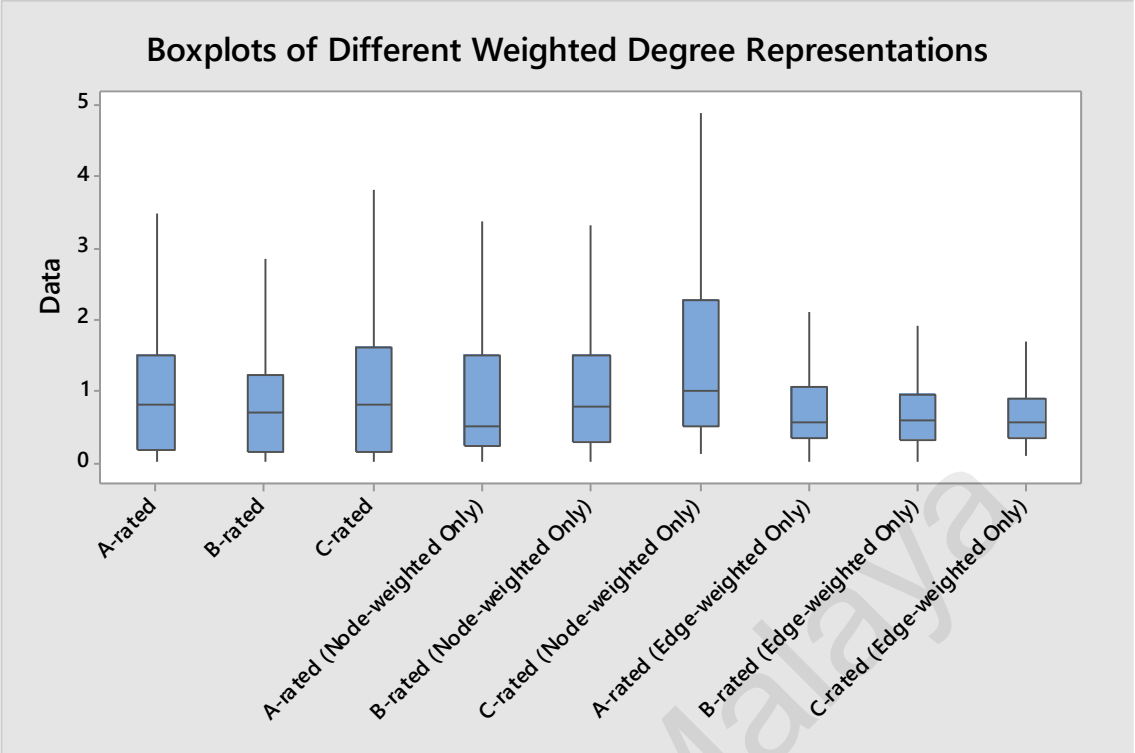


Figure 7.10: Boxplots of different weighted degree representations for A-rated, B-rated, and C-rated software systems

Table 7.3: Analysis of boxplots from Figure 7.10

Metrics	1st Quartile	Median	3rd Quartile	Interquartile Range	Whiskers
A-rated	0.173	0.813	1.495	1.322	0, 3.476
B-rated	0.158	0.701	1.232	1.075	0, 2.844
C-rated	0.157	0.810	1.618	1.461	0, 3.799
A-rated (Node-weighted Only)	0.234	0.5	1.485	1.251	0, 3.360
B-rated (Node-weighted Only)	0.285	0.767	1.498	1.213	0, 3.317
C-rated (Node-weighted Only)	0.499	1	2.258	1.758	0.125, 4.892
A-rated (Edge-weighted Only)	0.35	0.55	1.05	0.7	0, 2.1
B-rated (Edge-weighted Only)	0.3	0.6	0.95	0.65	0, 1.9
C-rated (Edge-weighted Only)	0.35	0.55	0.9	0.55	0.1, 1.7

The first three boxplots represent the average weighted degree calculated using the exact Equation (3). The 'Node-weighted' and 'Edge-weighted' boxplots, on the other hand, represent the calculation based on class complexity $[(1 - Comp_{(D_j)}) * \beta]$ and relationship complexity $\left(H_{R_{i \to j}} * \alpha\right)$ respectively.

From the 'Node-weighted' boxplots, it can be observed that the median values for B and C-rated software systems are much higher compared to A-rated software systems. Several reasons contributed toward this observation. Firstly, when calculating the weights for classes related with unidirectional relationships (generalisation and realisation), only the complexity of parent or supplier class is considered. Thus, for interface or utility classes that are heavily reused, the occurrence of duplicate weightage increases significantly. Secondly, highly reusable classes tend to be more complex and possess higher LOC values, especially in B-rated and C-rated software systems. For example, AbstractBTreePartition.java class in ApacheDS project which contains 1980 LOC and WMC of 399 is a highly complex interface class. Another example is the Registries.java class in ApacheDS which contains 1711 LOC and WMC of 392. The Registries.java acts as a utility class and it is implemented by many other classes.

As for the 'Edge-weighted' boxplots, the occurrence of duplicate weightage is even more distinguishable because only the type of relationship between two classes is considered to measure the weights of edges. Recall that the weights of relationships are based on the ordinal scale shown in Table 4.1. Thus, the variability of weightage is very limited and very little information can be extracted from the boxplot. Not much useful information related to software maintainability and reliability can be extracted based on the network represented with 'Edge-weighted' weightage.

All in all, the 'Node-weighted' strategy to represent communicational cohesion-based weight can be useful to identify highly reused and fault-prone classes. The comparative analysis in Figure 7.10 shows that software systems with a higher level of maintenance effort tend to have a higher average weighted degree. Software with a higher average weighted degree indicates that most of the software components are entangled and depend on each other (Giulio Concas et al., 2007; Louridas et al., 2008). Thus, more efforts are needed to maintain this group of software systems. One disadvantage of 'Node-weighted' strategy is that it is unable to clearly distinguish classes and relationships that possess higher strength of communicational cohesion over other classes because it is assumed that all relationships are even. Furthermore, duplicate of weightage occurs very often in 'Node-weighted' strategy. As for the 'Edge-weighted' strategy, it does not provide much useful information toward understanding the maintainability and reliability of software systems.

Besides that, classes that violate common software design principles, or those that are more prone to bugs and errors can be easily identified with the aid of graph theory metrics and weighted complex networks. For instance, by using the betweenness centrality metric, it is found that HasCurrentMarkup.java and HasIDBindingAndRendered.java of Apache Tobago project are very vulnerable toward bug propagation. These two classes possess a very high betweenness centrality value, which means that plenty of communications between classes (including passing of variables or parameters) need to go through them. From a software engineering perspective, unless these classes are purposely designed as an interface or mediator class, it is risky to have multiple classes that dictate the flow of communications because the failure or removal of these classes will cause a system-wide service interruption.

Besides that, the results shown in Figure 7.10 and Table 7.3 have shown that a hybrid of node and edge-weighted strategy is more appropriate because it is able to represent the dynamic interactions between software features. Both the classes and their interactions play equally important roles to quantify the communicational cohesion of classes. Furthermore, the proposed hybrid weighting strategy can also be used in forward engineering phase to evaluate the effectiveness of the software design.

7.2.4 Addressing Research Objectives and Hypothesis

In Section 3.2.3, RO1 outlines the goal to propose a constrained clustering approach, which is further breakdown into 3 sub-objectives. Sub-objective 1.1 outlines the needs to develop a method for representing OO software systems using weighted complex networks. Based on the discussed literature, it was found that weighted and directed network is more suitable to be used in the context of software engineering because not all software features are symmetrical in nature.

On the other hand, sub-objective 1.2 outlines the goal to identify appropriate measure constructs that are capable of quantifying maintainability and reliability of OO software systems represented in weighted complex networks. The focus of this research is to recover a high-level abstraction view of OO software design that is coherent with the actual code structure. Thus, the constructed weighted complex network must be representative enough to demonstrate the modularity of the analysed software system. Maintainability and reliability, for instance, are two software qualities that contribute directly toward estimating the modularity of a software system. Therefore, in order to address this sub-objective, a unique weighting function which is capable of capturing the maintainability and reliability of software systems, is proposed to quantify the weights of

edges and nodes in the constructed weighted complex network. Software systems are first converted into UML class diagrams in order to standardise the transformation rules. The proposed weighting function focuses on the complexity of UML classes and their associated relationships.

Finally, sub-objective 1.3 is about finding the correlation between statistical patterns of real-world OO software systems and their level of maintenance efforts. In Section 7.2.2, a distribution fitting tool called the Easyfit is used to identify the best fit distribution of all datasets. It was discovered that most of the selected graph theory metrics show the power law behaviour, which is a typical characteristic of complex networks. A large majority of the test subjects are shown to have low in-degree and out-degree. From a software engineering perspective, it can be concluded that most of the classes from the pool of test subjects are designed to be focused on their own functionalities and easy to maintain. Classes that are frequently reused or called (i.e. utility classes and interface classes) can be easily identified through the inspection of distribution fitting diagrams.

With respect to research hypothesis, the first hypothesis discussed that, *given any number and size of test subjects, the constructed weighted complex network based on the proposed approach should be able to demonstrate common statistical patterns of real-world OO software systems. When the test subjects are grouped and compared based on their levels of maintenance efforts, their statistical patterns are more distinguishable*. In order to address the first hypothesis, the statistical patterns of all the selected graph theory metrics are compared based on their associated SQALE rating. The datasets are grouped into three different categories, namely A-rated, B-rated, C-rated software systems. If the proposed weighting function and the selected graph-level metrics are able to represent the

maintainability and reliability of software systems, the resulting comparison should exhibit a certain degree of correlation.

Based on the results shown in Section 7.2.3, it was discovered that when software systems are grouped according to their maintenance effort, their statistical patterns are consistent with the findings of the existing literature. For example, the average weighted degree and shortest path length of B-rated and C-rated software systems are relatively higher than those of A-rated software systems. A higher weighted degree is associated with high coupling, while a high average shortest path length signifies poor communication between classes. Both observations contribute toward low maintainability and reliability of software systems, which is consistent with the experimental setup, and eventually provide a concrete answer for addressing the first research hypothesis and all sub-objectives under RO1 that concern with developing a method to represent OO software systems using weighted complex network, while preserving the quality aspects of the software and ensure that the constructed weighted complex network adheres to the certain statistical behaviour commonly found in existing studies.

Next, the proposed method to derive clustering constraints based on graph theoretical analysis is executed, followed by fulfilment of the clustering constraints using the proposed dendrogram cutting method to address the second research hypothesis - *The proposed constrained agglomerative hierarchical software clustering approach is able to form relatively more cohesive clusters as compared to the unconstrained clustering approach*. In this experiment, the proposed constrained clustering approach is compared against an unconstrained clustering approach. To recall, agglomerative software clustering consists of five main steps, as outlined in Section 2.3.

1. Identification of entities or components

2. Identification of features
3. Calculation of similarity measure
4. Application of clustering algorithm
5. Evaluation of clustering results

Different configuration for each of the five steps above will result in different clustering results. For instance, using Jaccard coefficient and Sorensen-Dice coefficient to measure the similarity between cluster entities (Step 3) will produce two distinctively different clustering results. Thus, in order to perform a fair comparison, the configuration (validity index used, clustering algorithm used, similarity measure used, etc.) used by the proposed constrained clustering approach and the unconstrained clustering approach must be identical. The only difference between the two clustering approaches is that the unconstrained approach does not make use of any clustering constraints. However, this particular setting does not compromise the generalisability of the research because all the design decisions (choice of similarity measure, choice of clustering algorithm, choice of validity index, etc.) have been discussed in detail in the previous chapters, i.e. to choose the most suitable clustering algorithm, similarity metric, and validity index to be used in this research. For instance, the decision of using UPGMA as the clustering algorithm (instead of SLINK and CLINK) have been discussed and analysed in Section 5.4.5. Similarly, an in-depth comparison between Davies-Bouldin index, Dunn's index, and the proposed enhanced Davies-Bouldin index was conducted in Section 5.5.1 in order to find the best cluster validity index to evaluate the quality of clustering results.

7.3 Executing the Proposed Constrained Clustering Approach

First, clustering constraints are derived based on graph theoretical analysis of the weighted complex network generated from the previous step.

7.3.1 Deriving MLH and CLH Constraints from the Implicit Structure of Software

Due to size and page constraints, all the clustering constraints derived from the 40 test subjects are presented in Table C1 in Appendix C. Some examples of Table C1 are illustrated in Table 7.4, which shows the clustering constraints derived from Apache Gora, openFAST, and Apache Tika.

The second column in Table 7.4 and Table C1 shows the hubs found in each test subject, while the third column shows the neighbouring classes that form a complete clique with each corresponding hub in the second column. Note that cannot-link constraints are established for each pair of hubs in order to promote the notion of separation of concerns. The fourth column lists down the classes that possess high betweenness centrality (high BC), while the last column shows the list of neighbouring classes that form a complete clique.

For hubs and high BC classes in C-rated software, fewer cliques can be identified, resulting in less constraint derived from these test subjects. The main reason behind this observation is due to the existence of god classes in software with a higher level of maintenance efforts. God class in the context of software engineering refers to classes that contain many instance variables and perform a lot of system operations on its own (Perez-Castillo & Piattini, 2014). As a software evolves and is updated, a god class will become denser as new classes are associated with it, causing the software to become more and more complex.

Table 7.4: Clustering constraints derived from Apache Gora, openFAST, and Apache Tika

Projects	**Hubs (Cannot-link between all pairs of hubs)**	**Classes that form a complete clique with hub (Must-link)**	**Classes with high betweenness centrality (high BC)**	**Classes that form a complete clique with high BC (Must-link)**
Apache Gora	DataStoreBase	MemStore	Where	-
	Query	GoraInputFormat	DataStoreBase	-
openFAST	Session	-	XMLMessageTemplateSerializer	-
	Context	-	Scalar	Operator
	MessageTemplate	FieldSet StaticTemplateReference	TemplateRegistry	NullTemplateRegistry FastMessageReader TemplateExchangeDefinitionEncoder AbstractTemplateRegistry
	TemplateRegistry	NullTemplateRegistry FastMessageReader TemplateExchangeDefinitionEncoder AbstractTemplateRegistry		
	Scalar	Operator		
Apache Tika	MediaType	-	LinkContentHandler	LinkBuilder Link
	Property	MetadataHandler Geographic ElementMetadataHandler MSOffice HttpHeaders TIFF	MediaType	-
	XHTMLContent Handler	XHTMLClassVisitor PagesContentHandler PDF2XHTML	CharsetRecognizer	CharsetMatch CharsetDetector
	Parser	-		
	Matcher	NamedAttributeMatcher		

In particular, hubs and high BC classes in JFreeChart, Apache Falcon, and Apache Archiva do not have neighbouring classes that can form a complete clique. The work by Singh (2013) and Chatzigeorgiou and Melas (2012) has shown that the modularity of JFreeChart project decreases over time due to frequent and unmanaged incremental updates. Chatzigeorgiou et al. reported that several classes in JFreeChart became denser with each incremental update. Based on the experimental findings in Table C1, classes that behave like god classes are XYItemRenderer.java, Plot.java, XYDataset.java, and Range.java. Refactoring and remodularisation of these classes should be done to minimise unnecessary coupling and dependencies in order to improve its overall maintainability.

The results in Table 7.4 and Table C1 show that the graph theoretical analysis managed to automatically derive clustering constraints from the implicit structure of software systems. Existing studies in constrained clustering often assumed that user feedbacks are always reliable and accessible prior to the clustering process, which is unrealistic in software development especially when dealing with poorly designed or poorly documented software systems. The proposed method has succeeded in deriving a number of clustering constraints without the need for user feedback to help facilitate in the subsequent constrained clustering process.

Table 7.5 lists down the number of clustering constraints derived from each test subject, sorted according to SQALE rating.

Table 7.5 Number of clustering constraints derived from each test subject

Project	**Number of Clustering Constraints**	**Number of Classes**	**SQALE rating**
Apache Maven Wagon	13	128	A
IWebMvc	4	178	A
JEuclid	15	230	A
openFAST	17	236	A
Apache Commons VFS	18	280	A
Apache XBean	10	401	A
Apache Tika	24	457	A
Jajuk	27	543	A
Fitnesse	30	852	A
Apache Tobago	30	873	A
Apache Shindig	3	950	A
Apache Mahout	36	1130	A
Apache Synapse	54	1276	A
Apache Gora	3	131	B
Jackcess	11	302	B
Apache Sirona	9	345	B
Apache Pluto	12	375	B
Apache Commons BCEL	23	396	B
JSPWiki	22	411	B
Apache Commons Collections	10	441	B
Apache EmpireDB	41	470	B
Apache Roller	14	528	B
Titan	10	532	B
Apache Log4J	41	704	B
Eclipse SWTBot	32	731	B
Apache Wink	11	740	B
Apache Karaf	69	773	B
Apache Deltaspike	53	1002	B
JFreeChart	6	1013	B
jOOQ	39	1106	B
Apache Hudson	23	1492	B
Apache Rampart	7	191	C
Apache Falcon	3	235	C
Kyro	12	346	C
Apache Archiva	15	506	C
Apache Mina	11	583	C
Apache Abdera	5	682	C
Apache Helix	7	710	C
Struts	39	1646	C
ApacheDS	14	2408	C

The experimental results show that the number of derived clustering constraints is not positively correlated to the size of the projects. Instead, more clustering constraints were derived from projects with lower level of maintenance effort such as those in A-rated and B-rated projects. Due to the complexity of C-rated projects, their structural behaviour are relatively more vague and entangled compared to A-rated and B-rated projects, resulting in a lesser number of clustering constraints can be derived automatically. For instance, the projects with highest number of classes in B-rated and C-rated projects, namely Apache Hudson (1492 classes) and ApacheDS (2408 classes), only managed to derive 23 and 14 clustering constraints respectively. When compared to a relatively small-sized A-rated project, both Apache Hudson and ApacheDS actually yield a lesser number of clustering constraints compared to Apache Tika (457 classes, with 24 constraints derived automatically).

After all the clustering constraints are automatically retrieved using the proposed method, the next step is to fulfil these constraints by altering the distance between pairs of MLH and CLH constraints using the distance based approach discussed in Section 5.2.

7.3.2 Fulfilment of Must-Link and Cannot-Link Constraints

First, the resemblance matrix of each project is constructed based on Dijkstra's shortest path algorithm discussed in Section 5.4.4. One resemblance matrix is created for each project. Next, conflicting MLH and CLH constraints are identified. If there is a pair of classes (x, y), such that (x, y) belongs to both MLH and CLH, then this is a NP-Complete problem with no solution, as discussed by Davidson and Ravi (2009). Software maintainers can choose to randomly omit one of the conflicting constraints from the system to avoid the NP-Complete problem.

Then, each MLH constraint is fulfilled by changing the distance between a pair of classes to zero, indicating that these classes must be grouped into the same cluster regardless of any condition. The pair of classes involved in the MLH constraint will eventually form the base of the dendrogram. Next, each CLH constraint is fulfilled by changing the distance between a pair of classes to a large enough constant that prevents them from clustered into the same group. The constant is determined by calculating $l + 1$, such that l = largest distance exist in a particular resemblance matrix.

To recall, the proposed constrained clustering approach allows domain experts to explicitly provide their domain knowledge in the form of MLS and CLS constraints to further improve the clustering results. However, without directly involved in the design and development of the selected test subjects, it is difficult to find experts who possess a certain level of understanding on all the test subjects and are willing to take part in the experiment. Therefore, the approach for generating and fulfilling the MLS and CLS constraints is similar to the evaluation conducted as discussed in Section 5.5.

The steps involved are as follows:

1. Prior to the experiment, all the classes are assumed to be scattered around and not grouped in their respective packages.
2. Based on the original UML package diagram, several MLS and CLS constraints are extracted. The number of soft constraints is limited to 50% of the total number of hard constraints derived from the proposed method. This is to prevent biasness in the results because the soft constraints are extracted directly from the original package diagram and will definitely improve the accuracy of clustering results once they are imposed in the experiment. Hence, limiting the number of soft

constraints is needed to prevent such biasness when interpreting the experiment results.

3. For MLS and CLS constraints, penalty scores for violating the soft constraints are generated randomly.

However, solely relying on extracting soft constraints from the original package diagram is at the risk of causing biasness in the experiment results. Hence, another approach is taken to evaluate the accuracy of the proposed method for fulfilling MLS and CLS constraints in Equation (4).

Five participants were recruited to take part in the experiment, where each of them has at least 5 years of industrial experience in developing and maintaining software systems. While it is impossible to request the participants to provide feedbacks in the form of soft constraints on all the 40 test subjects, 3 projects with different levels of maintenance efforts were chosen namely Apache XBean (A-rated), Apache Gora (B-rated), and Apache Rampart (C-rated).

The five participants then provided their feedbacks in the form of soft constraints and ranked their judgement using the fuzzy-AHP method discussed in Section 5.3. In order to reach a consensus among the participants, they were instructed to perform pair-wise comparison of the identified soft constraints and ranked their judgement based on the relative scores of 1-9, where a greater value represents higher importance. Next, the triangular fuzzy numbers (TNF) values were calculated using the formula in Section 5.3. The results were tabulated into a comparison matrix shown in Table 7.6 to Table 7.8 for all the 3 test subjects.

Table 7.6: Fuzzy pair-wise TFN values for MLS and CLS constraints derived from Apache XBean Project

	BundleClassLoader-DelegatingBundleReference (MLS)	ClassPath-SunURLClassPath (MLS)	ConstructionException-UrlResourceFinder(CLS)	Command-AnnotatedMember(CLS)
BundleClassLoader-DelegatingBundleReference (ML)	**1**	0.2, 1.508, 6	0.2, 1.675, 9	0.14, 1.304, 2
ClassPath-SunURLClassPath (MLS)		**1**	0.33, 1.52, 8	0.33, 1.508, 9
ConstructionException-UrlResourceFinder(CLS)			**1**	0.2, 1.16, 6
Command-AnnotatedMember(CLS)				**1**

Table 7.7: Fuzzy pair-wise TFN values for MLS and CLS constraints derived from Apache Gora Project

	Persistent-StateManager (MLS)	SqlStore-InsertUpdateStatement (MLS)	HBaseMapping-HBaseTableConnection (MLS)	SqlResult-SqlMapping (CLS)	StatefulMap-StatefulHashMap (MLS)	CassandraColumn-CassandraSubColumn (MLS)
Persistent-StateManager (MLS)	**1**	0.5, 1.488, 4	0.1, 1.2 , 2	0.2, 1.28, 6	0.25, 0.84, 5	0.15, 0.861, 6
SqlStore-InsertUpdateStatement (MLS)		**1**	0.2, 0.35, 4	0.11, 1.38, 4	0.2, 0.72, 5	0.2, 0.88, 3
HBaseMapping-HBaseTableConnection (MLS)			**1**	0.25, 0.72, 3	0.33, 1.5, 4	0.33, 0.72, 4
SqlResult-SqlMapping (CLS)				**1**	0.12, 1.32, 6	0.12, 0.45, 4
StatefulMap-StatefulHashMap (MLS)					**1**	0.2, 0.38, 6
CassandraColumn-CassandraSubColumn (MLS)						**1**

Table 7.8: Fuzzy pair-wise TFN values for MLS and CLS constraints derived from Apache Rampart Project

	EncryptedKeyToken-Token (MLS)	TokenStorage-SimpleTokenStore (MLS)	AbstractIssuerConfig-SCTIssuerConfig (MLS)	RahasData-Binding (CLS)	Token-Layout (CLS)	SCTIssuer-Wss10 (CLS)
EncryptedKeyToken-Token (MLS)	**1**	0.1, 0.5, 2	0.1, 0.878, 7	0.12, 1.032, 4	0.25, 1.32, 8	0.12, 0.861, 5
TokenStorage-SimpleTokenStore (MLS)		**1**	0.3, 1.052, 5	0.33, 0.897, 5	0.2, 1.025, 4	0.33, 0.52, 4
AbstractIssuerConfig-SCTIssuerConfig (MLS)			**1**	0.12, 1.16, 6	0.2, 1.02, 5	0.15, 1.03, 6
RahasData-Binding (CLS)				**1**	0.14, 1.218, 5	0.14, 0.98, 4
Token-Layout (CLS)					**1**	0.11, 0.33, 5
SCTIssuer-Wss10 (CLS)						**1**

Next, defuzzification was performed to produce a quantifiable value based on the calculated TFN values. Recall that the defuzzification method adopted in this research is based on the alpha cut method proposed by Liou and Wang (1992), as discussed in Section 5.3.

An example of calculation is shown below for the comparison between BundleClassLoader-DelegatingBundleReference (MLS) and ClassPath-SunURLClassPath (MLS) for the Apache XBean project:

$$f_{0.5}(L_{xy}) = (1.508 - 0.2) \cdot 0.2 + 0.2 = 0.4616$$

$$f_{0.5}(H_{xy}) = 6 - (6 - 1.508) \cdot 0.2 = 5.102$$

$$\mu_{05,0.5}(\tilde{F}_{xy}) = [0.5 \cdot 0.4616 + (1 - 0.5) \cdot 3.244] = 1.853$$

$$\mu_{05,0.5}(\tilde{F}_{Usability-Reliability}) = 1/1.853 = 0.54$$

Table 7.9 – Table 7.11 show the result of fuzzy pair-wise comparison after the defuzzification process.

Table 7.9: Fuzzy pair-wise TFN values for MLS and CLS constraints derived from Apache XBean Project

	BundleClassLoader-DelegatingBundleReference (MLS)	ClassPath-SunURLClassPath (MLS)	ConstructionException-UrlResourceFinder(CLS)	Command-AnnotatedMember(CLS)
BundleClassLoader-DelegatingBundleReference (MLS)	**1**	1.853	4.015	1.103
ClassPath-SunURLClassPath (MLS)	0.54	**1**	3.292	3.624
ConstructionException-UrlResourceFinder(CLS)	0.249	0.304	**1**	2.712
Command-AnnotatedMember(CLS)	0.907	0.276	0.369	**1**

Table 7.10: Fuzzy pair-wise TFN values for MLS and CLS constraints derived from Apache Gora Project

	Persistent-StateManager (MLS)	SqlStore-InsertUpdateStatement (MLS)	HBaseMapping-HBaseTableConnection (MLS)	SqlResult-SqlMapping (CLS)	StatefulMap-StatefulHashMap (MLS)	CassandraColumn-CassandraSubColumn (MLS)
Persistent-StateManager (MLS)	**1**	1.869	1.065	2.736	2.179	2.743
SqlStore-InsertUpdateStatement (MLS)	0.535	**1**	1.476	1.98	2.224	1.458
HBaseMapping-HBaseTableConnection (MLS)	0.939	0.677	**1**	1.4	1.946	1.688
SqlResult-SqlMapping (CLS)	0.365	0.505	0.714	**1**	2.851	1.947
StatefulMap-StatefulHashMap (MLS)	0.459	0.450	0.514	0.351	**1**	2.556
CassandraColumn-CassandraSubColumn (MLS)	0.365	0.687	0.592	0.514	0.391	**1**

Table 7.11: Fuzzy pair-wise TFN values for MLS and CLS constraints derived from Apache Rampart Project

	EncryptedKeyToken-Token (MLS)	TokenStorage-SimpleTokenStore (MLS)	AbstractIssuerConfig-SCTIssuerConfig (MLS)	RahasData-Binding (CLS)	Token-Layout (CLS)	SCTIssuer-Wss10 (CLS)
EncryptedKeyToken-Token (MLS)	**1**	0.995	3.283	1.937	3.424	2.461
TokenStorage-SimpleTokenStore (MLS)	1.005	**1**	2.171	2.082	1.885	1.622
AbstractIssuerConfig-SCTIssuerConfig (MLS)	0.305	0.461	**1**	1.885	2.284	2.768
RahasData-Binding (CLS)	0.516	0.480	0.531	**1**	2.38	1.917
Token-Layout (CLS)	0.292	0.531	0.438	0.420	**1**	2.31
SCTIssuer-Wss10 (CLS)	0.406	0.617	0.361	0.522	0.423	**1**

The next step is to determine the eigenvalue and eigenvector of the fuzzy pair-wise comparison matrix, as shown in Section 5.3. The eigenvector can help in determining the aggregated weightage, or in other words, relative importance of a particular soft constraint. Assume that δ denotes the eigenvector while λ denotes the eigenvalue of fuzzy pair-wise comparison matrix $\tilde{F}_{xy}$,

$$[(\mu_{\alpha,\beta}(\tilde{F}_{xy}) - \lambda I] \cdot \delta = 0$$

The above formula is based on the linear transformation of vectors, where *I* represents the unitary matrix. The eigenvectors of all the associated soft constraints were then calculated using the formula. The example below shows the calculation for Apache XBean project.

$$[(\mu_{\alpha,\beta}(\tilde{F}_{xy}) - \lambda I] = \begin{bmatrix} 1 & 1.853 & 4.015 & 1.103 \\ 0.54 & 1 & 3.292 & 3.624 \\ 0.249 & 0.304 & 1 & 2.712 \\ 0.907 & 0.276 & 0.369 & 1 \end{bmatrix}$$

Multiplying eigenvalue λ with unitary matrix *I* produces an identity matrix that cancels out each other. Thus, the notation λI is discarded in this case.

$$\begin{bmatrix} 1 & 1.853 & 4.015 & 1.103 \\ 0.54 & 1 & 3.292 & 3.624 \\ 0.249 & 0.304 & 1 & 2.712 \\ 0.907 & 0.276 & 0.369 & 1 \end{bmatrix} \begin{bmatrix} \delta_{\text{BundleClassLoader-DelegatingBundleReference}} \\ \delta_{\text{ClassPath-SunURLClassPath}} \\ \delta_{\text{ConstructionException-UrlResourceFinder}} \\ \delta_{\text{Command-AnnotatedMember}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \delta_{\text{BundleClassLoader-DelegatingBundleReference}} \\ \delta_{\text{ClassPath-SunURLClassPath}} \\ \delta_{\text{ConstructionException-UrlResourceFinder}} \\ \delta_{\text{Command-AnnotatedMember}} \end{bmatrix} = \begin{bmatrix} 0.3795 \\ 0.331 \\ 0.154 \\ 0.135 \end{bmatrix}$$

The aggregated result in terms of weightage is tabulated in Table 7.12. The results obtained are ordered as follows: BundleClassLoader-DelegatingBundleReference

(0.3795), ClassPath-SunURLClassPath (0.331), ConstructionException-UrlResourceFinder (0.154), Command-AnnotatedMember (0.135).

Table 7.12: Weightage and priority of soft constraints derived from Apache XBean

Priority	Derived Soft Constraints	Weightage/Penalty Score
1	BundleClassLoader-DelegatingBundleReference (MLS)	0.3795
2	ClassPath-SunURLClassPath (MLS)	0.331
3	ConstructionException-UrlResourceFinder(CLS)	0.154
4	Command-AnnotatedMember(CLS)	0.135

Note that the weightage in Table 7.12 is equivalent to the penalty score for violating a particular soft constraint, as in Equation (4). The results for Apache Gora and Apache Rampart are illustrated in Table 7.13 and Table 7.14 respectively.

Table 7.13: Weightage and priority of soft constraints derived from Apache Gora

Priority	Derived Soft Constraints	Weightage/ Penalty Score
1	Persistent-StateManager (MLS)	0.272
2	SqlStore-InsertUpdateStatement (MLS)	0.201
3	HBaseMapping-HBaseTableConnection (MLS)	0.180
4	SqlResult-SqlMapping (CLS)	0.151
5	StatefulMap-StatefulHashMap (MLS)	0.109
6	CassandraColumn-CassandraSubColumn (MLS)	0.087

Table 7.14: Weightage and priority of soft constraints derived from Apache Rampart

Priority	Derived Soft Constraints	Weightage/ Penalty Score
1	EncryptedKeyToken-Token (MLS)	0.29
2	TokenStorage-SimpleTokenStore (MLS)	0.231
3	AbstractIssuerConfig-SCTIssuerConfig (MLS)	0.166
4	RahasData-Binding (CLS)	0.136

5	Token-Layout (CLS)	0.097
6	SCTIssuer-Wss10 (CLS)	0.08

The ranking and weighting of the identified soft constraints are able to aid in penalising the violation of soft constraints. The penalty score will be taken into account when evaluating the quality of clusters formed by a particular cutting point. Hence, maximising the fulfilment of soft constraints is preferred.

7.3.3 Forming and Cutting of Dendrogram

Now that both the explicit and implicit clustering constraints are derived, the next step is to generate a dendrogram for each of the associated test subjects. Since all the MLH constraints are unconditionally fulfilled at the bottom of the dendrogram, users do not have to worry about the fulfilment of these hard constraints. Due to the size and scale of the experiment, one example is chosen and shown in Figure 7.11, where it depicts the dendrogram generated from Apache JSPWiki project.

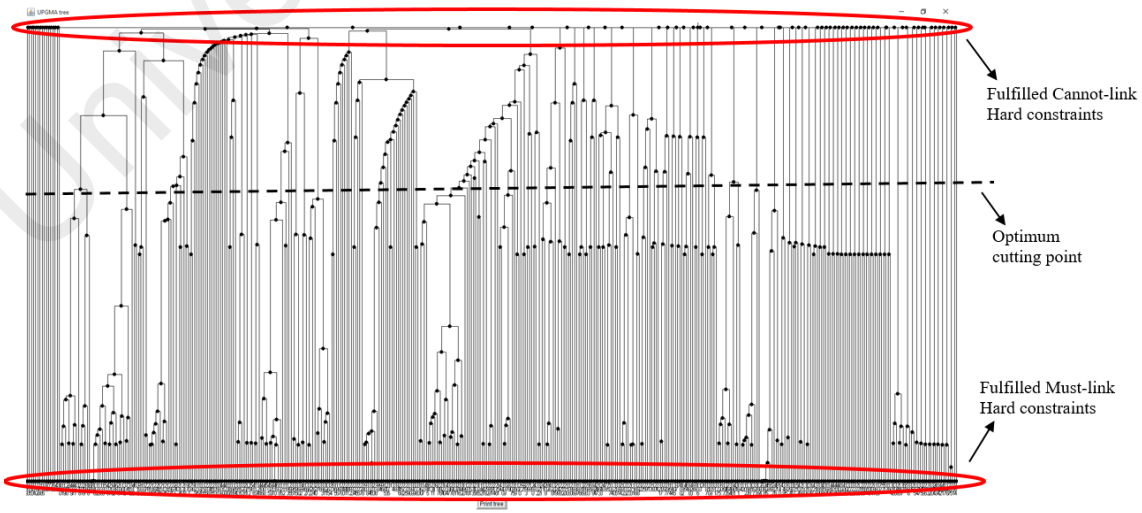


Figure 7.11: Dendrogram generated from Apache JSPWiki project

The circle at the bottom of the dendrogram shows the pairs of MLH constraints that form the base of the dendrogram. On the other hand, the circle at top of the dendrogram shows the CLH constraints. Since it is impossible to cut the dendrogram at the top of the dendrogram, one can be assured that CLH constraints are fulfilled regardless of any condition.

As for the soft constraints, the fulfilment of MLS and CLS constraints are evaluated after the dendrogram is partitioned by a cutting point. For instance, if the classes EncryptedKeyToken.java and Token.java of Apache Rampart project are grouped into the same cluster using a cutting point *x*, then it is deemed that this MLS constraint is fulfilled, hence no penalty score is enforced onto the cutting point *x*. However if the cutting point *x* failed to group the two classes EncryptedKeyToken.java and Token.java into the same cluster, then the penalty score of 0.29, as illustrated in Table 7.14, is enforced to discourage software maintainers from choosing the cutting point *x* as the optimum cutting point. The detailed steps involved in fulfilling soft constraints are described earlier in Section 5.3, Equation (4).

After generating the dendrogram, it is cut using the proposed dendrogram cutting method discussed in Section 5.4. With least-squares polynomial regression analysis, the optimum cutting point for JSPWiki project is found at the distance level of 3.712, which is illustrated as the dotted line in Figure 7.11. The resulting clustering results are shown in Figure 7.12.

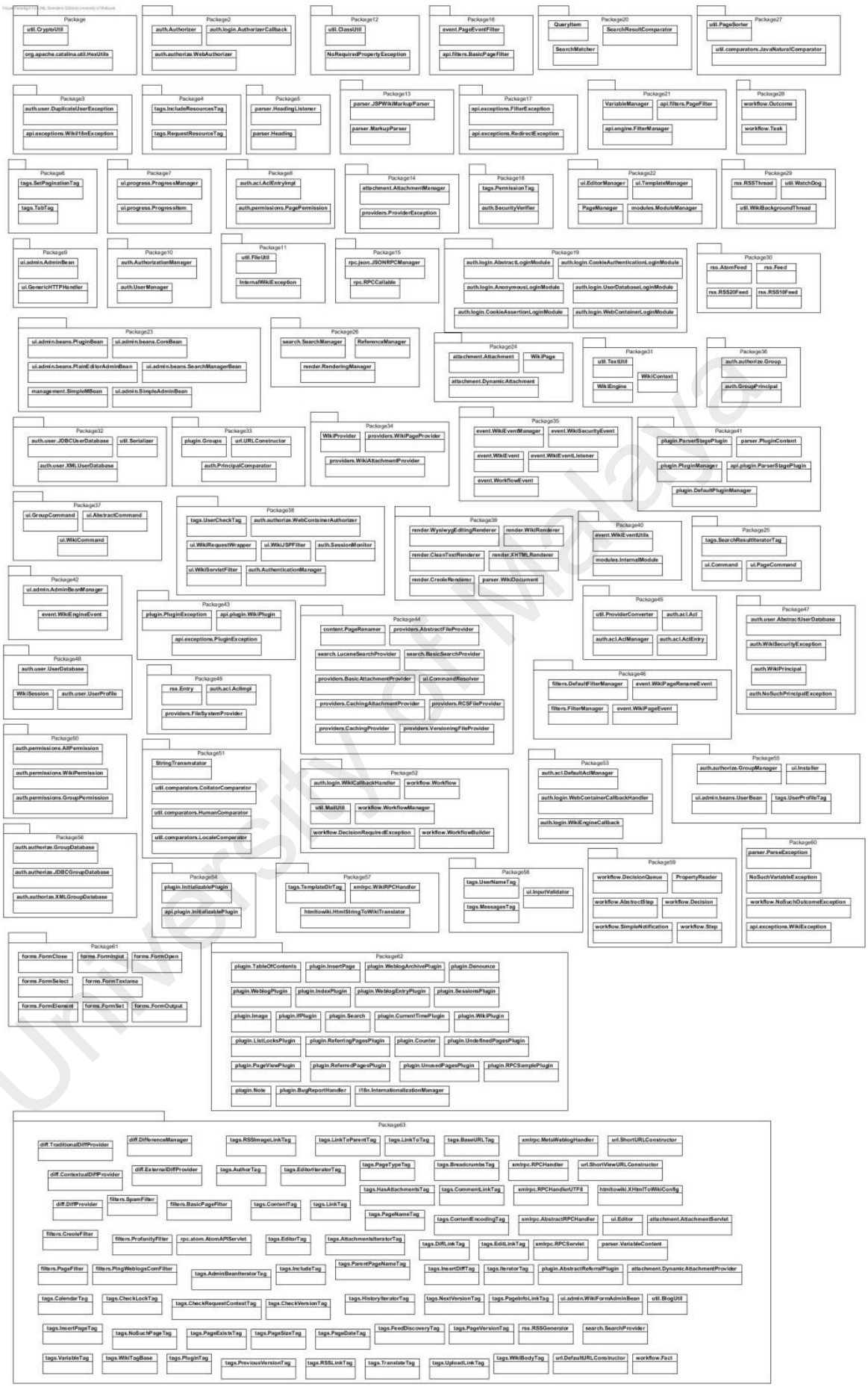


Figure 7.12: Clustering results for JSPWiki for cutting the dendrogram at 3.712

Cutting the dendrogram at 3.712 yields the lowest average $intra$ value and the highest average $inter$ value. As stated earlier, low $intra$ value signifies higher cohesion among classes inside the same cluster while high $inter$ value indicates that the clusters formed are well separated. This indicates that cutting at 3.712 produces the most cohesive clusters although it contains a lot of small clusters. Besides that, this particular cutting point managed to fulfil all the soft constraints generated from the previous steps. Next, the clustering results are compared against the original package diagram using the MoJoFM metrics to validate the second hypothesis, which is to verify if the proposed constrained clustering approach can produce better clustering results compared to the unconstrained approach.

7.3.4 Using MoJoFM to Compare Clustering Results

A MoJo tool written by (Wen & Tzerpos, 2004) is used to calculate the MoJoFM metric. Figure 7.13 shows the screenshot of the software tool.

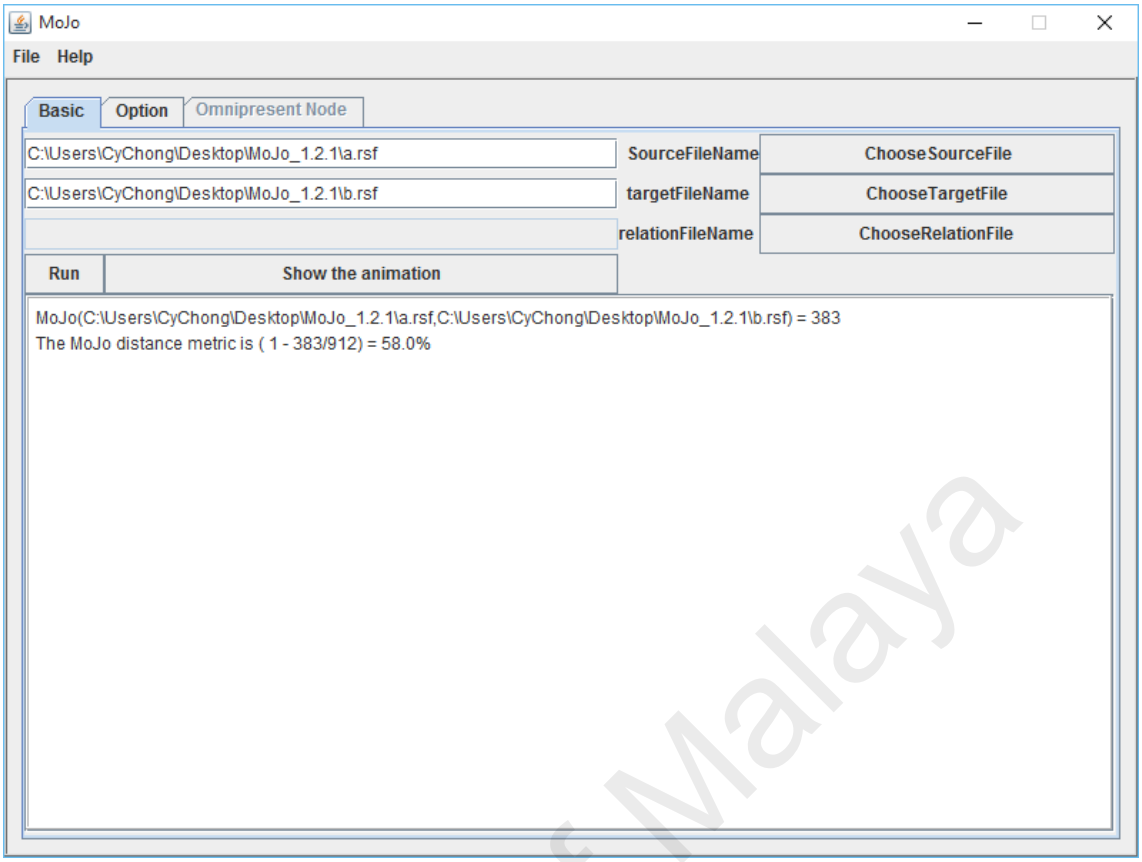


Figure 7.13: Screenshot of the MoJo distance software tool

The software is capable of automatically comparing and calculating the MoJo metric of two clustering results. As shown in Figure 7.13, users need to specify the source file and the target file before running the tool. The target file in this context refers to the ground truth or the golden result. The source file refers to the clustering result to be compared with the golden clustering result. The tool will calculate the number of Move and Join operations needed to transform the source file into the target file. Minimisation of Move and Join operations are preferred. In Figure 7.13, an example source file named a.rsf is chosen, which contains the unconstrained clustering results of jOOQ project. The target file, which is the ground truth, is represented by b.rsf. Note that the tool only accepts Rigi Standard Format extension (.rsf), which is a textual format to represent binary relations between groups of entities. Next, the tool calculates the number of Move and Join operations needed to convert a.rsf to b.rsf. Based on the result shown in Figure 7.13, 383

operations are needed to accomplish the conversion task. The value 912 indicates the number of classes found in the input file. Hence, the final MoJoFM metric value is calculated, where it is concluded that the source file (a.rsf) is 58.0% similar to the target file (b.rsf). MoJoFM value of 100% indicates that the source file is identical to the target file. Therefore, maximisation of the MoJoFM metric value is preferred.

In order to address the second hypothesis, a comparison was made between the proposed constrained clustering approach and the conventional unconstrained clustering approach. Each test subject undergoes two clustering processes, one using the proposed constrained clustering approach, and another one without making use of any clustering constraint. In order to provide a fair comparison and reduce the biasness of the result, the proposed dendrogram cutting method and the enhanced Davies-Bouldin index are also implemented on the unconstrained clustering approach. This allows one to observe the effect of clustering constraints on the accuracy of clustering results. Table 7.15 shows the MoJoFM metric value for all the 40 test subjects. The third column shows the MoJoFM values of the unconstrained clustering approach when compared against the original package diagram. The fourth column shows the MoJoFM values of the proposed constrained clustering approach when compared to the original package diagram.

Table 7.15: MoJoFM values for constrained and unconstrained clustering results when compared to the original package diagram

Project	Number of hard constraints	Unconstrained clustering approach (MoJoFM value)	Proposed constrained clustering approach (MoJoFM value)	Differences (MoJoFM)
Apache Maven Wagon	13	75.8%	85.6%	9.8%
IWebMvc	4	80.5%	92.3%	11.8%
JEuclid	15	72.3%	85.2%	12.9%
openFAST	17	61.5%	75.3%	13.8%

Apache Commons VFS	18	63.2%	76.5%	13.3%
Apache XBean	10	50.8%	73.5%	22.7%
Apache Tika	24	56.2%	76.2%	20%
Jajuk	27	53.1%	78.5%	25.4%
Fitnesse	30	49.8%	72.4%	22.6%
Apache Tobago	30	55.4%	80.2%	24.8%
Apache Shindig	3	58.8%	65.2%	6.4%
Apache Mahout	36	52.8%	77.9%	25.1%
Apache Synapse	54	44.5%	77.4%	32.9%
Apache Gora	3	72.3%	86.2%	13.9%
Jackcess	11	78.5%	88.4%	9.9%
Apache Sirona	9	80.4%	86.3%	5.9%
Apache Pluto	12	75.3%	80.5%	5.2%
Apache Commons BCEL	23	72.4%	85.6%	13.2%
JSPWiki	22	68.3%	82.8%	14.5%
Apache Commons Collections	10	78.5%	83.6%	5.1%
Apache EmpireDB	41	75.3%	88.5%	13.2%
Apache Roller	14	79.2%	84.5%	5.3%
Titan	10	80.4%	87.3%	6.9%
Apache Log4J	41	68.6%	90.2%	21.6%
Eclipse SWTBot	32	62.8%	83.5%	20.7%
Apache Wink	11	70.5%	78.9%	8.4%
Apache Karaf	69	55.8%	89.3%	33.5%
Apache Deltaspike	53	64.2%	92.8%	28.6%
JFreeChart	6	52.5%	55.1%	2.6%
jOOQ	39	58.0%	82.8%	24.5%
Apache Hudson	23	60.8%	71.1%	10.3%
Apache Rampart	7	72.5%	83.9%	11.4%
Apache Falcon	3	70.5%	71.8%	1.3%
Kyro	12	77.5%	82.7%	5.2%
Apache Archiva	15	65.8%	73.2%	7.4%
Apache Mina	11	70.6%	80.5%	9.9%
Apache Abdera	5	70.5%	72.6%	2.1%
Apache Helix	7	65.3%	69.2%	3.9%
Struts	39	67.2%	87.6%	20.4%
ApacheDS	14	65.3%	78.1%	12.8%
AVERAGE	**20.58**	**66.4%**	**80.1%**	**13.8%**

Based on Table 7.15, it can be summarised that the proposed constrained clustering approach achieves an aggregated average of 80.33% accuracy when compared against the original package diagrams of the forty software systems, and performs better than the

unconstrained clustering approach. It has to be noted that the original package diagram is by no means the optimum or best clustering result since there is no way to verify that it is the best clustering result to represent the software design. However, it can be treated as a guideline to evaluate and compare between the results produced by the proposed constrained clustering approach and the unconstrained one.

In general, test subjects with more clustering constraints achieve better improvement in terms of MoJoFM metric when compared against the unconstrained approach. There are a few exceptions, such as the Apache Pluto, Apache Roller, and Apache Archiva project, which record less than 10% improvements. This is mainly because several pairs of classes involved in the must-link or cannot-link constraints had already been placed in the intended clusters prior to the implementation. Furthermore, it can be observed that improvement (in terms of MoJoFM) is more significant on larger projects with low level of maintenance efforts such as the Jajuk (543 classes), Apache Tobago (873 classes), Apache Synapse (1276 classes), Apache Karaf (773 classes), and Apache Deltaspike (1002 classes). One of the contributing factors is because it is relatively easier to identify clustering constraints such as hubs in larger projects with low maintenance efforts. Although ApacheDS contains 2408 classes, only 14 constraints can be derived due to its inherent complexity and complex structure.

In summary, the results presented in this subsection are capable of providing a concrete answer toward addressing the second research hypothesis, such that the proposed constrained clustering approach is able to produce highly cohesive clusters when measured using the MoJoFM metric. It is when applied on larger projects, the improvements are relatively more significant.

7.4 Chapter Summary

This chapter presented the analysis of experimental results using the proposed constrained clustering approach. Experiments were conducted using 40 open-source OO software systems with different sizes, complexity, and maintenance efforts. The experiment results were analysed extensively using several statistical analyses. Finally, a discussion on the analysed results was conducted to address the research hypotheses and objectives.

CHAPTER 8: CONCLUSION AND FUTURE WORK

This chapter summarises the research that has been conducted. Then, a discussion on the research contribution is presented. Finally, some suggestions on the potential enhancement to the proposed constrained clustering approach are provided for future research work.

8.1 Thesis Summary

A thorough literature review had been conducted to search for methods to reverse engineer poorly documented software systems. The review has successfully identified and analysed the state-of-the-art constrained and unconstrained clustering approaches. During the analysis, issues related to constrained clustering were identified and discussed. As a result, a constrained clustering approach with the aid of weighted complex network is proposed in this thesis to support remodularisation of poorly designed or poorly documented software systems.

The main objective of the proposed constrained clustering approach is to create a high-level abstraction of the software design with highly cohesive clusters based on the clustering constraints derived explicitly from domain experts and also from the implicit structure of software systems. Using a round-trip engineering tool, raw source code are first converted into UML class diagrams. Then, a method is proposed to analyse information extracted from the class diagrams such as complexity of classes, relationships between classes, and the complexity of relationships, in order to represent the analysed software with a weighted complex network. Classes are represented as nodes while relationships such as association, generalisation and realisation are represented as edges

that connect pairs of nodes. The complex network is further extended by assigning weights to the edges of the network using a unique weighting mechanism based on two parameters, namely the complexity of classes and the complexity of relationships.

Based on the constructed weighted complex network, statistical analysis using best fit distribution and boxplot is conducted to identify the common statistical behaviour of software systems, and to address the first research hypothesis - *Given any number and size of test subjects, the constructed weighted complex network based on the proposed approach should be able to demonstrate common statistical patterns of real-world OO software systems. When the test subjects are grouped and compared based on their levels of maintenance efforts, their statistical patterns are more distinguishable.*

Based on the experiments conducted on 40 open-source OO software systems, it is found that all the chosen test subjects demonstrate the power law behaviour, such that most of the classes possess low in-degree and out-degree except for a few classes that are frequently used by other classes. Next, with the aid of SQALE rating, the maintenance efforts of all the chosen test subjects are rated and grouped into three categories, namely A-rated, B-rated, and C-rated software. Based on the boxplot analysis, it is found that graph-level metrics for C-rated software deviate away from the statistical patterns found in A-rated and B-rated software, signifying the structural weakness in software with high level of maintenance efforts. Then, a method to automatically derive must-link and cannot-link constraints is introduced based on the graph theoretical analysis performed in the previous step. The proposed method is capable of revealing some extra deterministic information regarding the software, which is otherwise hidden using conventional software metrics.

Constrained clustering is performed on all the 40 test subjects in order to provide a way to address the second hypothesis - *The proposed constrained agglomerative hierarchical software clustering approach is able to form relatively more cohesive clusters as compared to the unconstrained clustering approach*. Clustering constraints are derived from two sources of information, i.e. explicitly from domain experts and implicitly from the results of graph theoretical analysis. The proposed method offers extra flexibility in the way clustering constraints are supplied to software maintainers. Clustering constraints are fulfilled by altering the resemblance matrix to ensure that classes involved in must-link constraints are always grouped into the same clusters, while those involved in cannot-link constraints must always be separated.

A dendrogram is then formed based on the altered resemblance matrix to illustrate the arrangement of the clusters produced by hierarchical clustering. Then, a dendrogram cutting technique is introduced to minimise redundant effort in finding optimum cutting points while maintaining the integrity of results. This issue had not been tackled explicitly in the current literature. The number of cutting points needed to find the optimum set of clusters can be minimised due to the adaptive nature of the proposed dendrogram cutting method.

Besides, an enhanced version of Davies-Bouldin index is introduced to measure the quality of clustering results. A penalty mechanism is introduced in the enhanced Davies-Bouldin index. This penalty mechanism works by penalising the formation of singleton clusters and it is always a good practice to prevent them from happening. The penalty mechanism is also adaptive in such a way that it penalises according to the highest gap between a pair of cluster entities.

Then, a least-squares polynomial regression analysis technique is introduced to find the cutting points that produce the best validity index. This technique uses the output produced by the proposed dendrogram cutting method to perform the analysis. The cutting point that produces the best result in terms of intra-cluster cohesion, intra-cluster separation, and fulfilment of clustering constraints, is recommended as the optimum cutting point.

From the software maintainers' perspective, several implications can be drawn from the experimental results. First, it is shown that UML class diagram can be an effective input to aid in the modelling of a software-based weighted complex network. Majority of the existing studies only focus on using raw source code as the sole input, which limits the applicability of their approaches because the findings cannot be applied on software systems written in other programming languages. With the use of UML class diagram, software maintainers can recover a high-level abstraction of the OO software design using the proposed approach as long as the class diagrams can be retrieved or reverse-engineered using an off-the-shelf round-trip engineering tool. The recovered high-level software design can then be used to raise questions about the structure and behaviour of the software, even for very well-maintained software, to aid in software remodularisation. Therefore when the proposed approach suggests a clustering result which should produce a noticeable improvement in cohesion and coupling of classes, this will serve as a guide to the software maintainers for consideration and further investigation.

In Chapter 4, a method to represent software systems using weighted complex networks is proposed. The method is based on a unique weighting mechanism to weight the edges and nodes of a software-based complex network. The proposed method has shown to be able to successfully measure the complexity of classes and their relationships, and provide

an alternative to the conventional techniques that only count the frequencies of method interactions. With the aid of graph theoretical analysis, software maintainers are able to identify common statistical behaviour found in software with different levels of maintenance efforts. The method to derive clustering constraints, on the other hand, provides a means to automatically identify and derive constraints in situations where software documentation are not up-to-date or domain experts are non-existent. Existing studies in constrained clustering often assumed that user feedbacks are always reliable and accessible prior to the clustering process, which is unrealistic in software development especially when dealing with poorly designed or poorly documented software systems.

Finally, the proposed dendrogram cutting method can be used as a complementary mechanism to improve the effectiveness of other clustering algorithms. The adaptive regression analysis, for instance, can help to reduce the computational cost of existing clustering algorithms while the penalty mechanism can help to prevent the formation of singleton clusters.

8.2 Contributions

The following summarises the contributions of the thesis:

- **A constrained clustering approach supported by weighted complex network to help in recovering a high-level software design of poorly designed or poorly documented OO software systems**. This contribution is aligned to RO1 - "*To propose a constrained clustering approach with the aim to recover a high-level abstraction of OO software design that is coherent and consistent with the actual code*

structure", where the proposed approach is capable of analysing the structure, behaviour, and complexity of OO software systems, and ultimately recovers a high-level abstraction of the software design. In addition, it is capable of utilising both explicit and implicit constraints to help in recovering a high-level software design that is coherent and consistent with the actual code structure. The proposed approach can not only help in minimising the cost of software maintenance, but also ensure that the maintained software can adapt to future changes.

- **Proposed method to represent software systems using weighted and directed complex networks**. This contribution is aligned to RO1.1- *"To develop a method for representing OO software systems using weighted complex network"*, and RO1.2 - *"To identify appropriate measure constructs that are capable of quantifying maintainability and reliability of software systems represented in weighted complex networks"*, where a unique weighting mechanism is proposed to weight the constructed software-based complex network based on the complexity of classes and the complexity of relationships. Based on the in-depth literature review done in Chapter 2, it is concluded that weighted and directed complex network is more suitable to capture the structure and behaviour of software systems. The proposed weighting mechanism is capable of capturing the maintainability and reliability aspects of software systems.

- **Using statistical analysis technique to investigate the statistical pattern of weighted complex network constructed based on the proposed approach**. This contribution is aligned to RO1.3 - *"To investigate the correlation between the statistical patterns of real-world OO software systems and their level of maintenance efforts"*, where all the weighted complex networks constructed from the 40 test

subjects are evaluated using the best fit probability distribution. Based on the evaluation, it is found that several graph theory metrics such as in-degree, out-degree, average weighted degree, and betweenness centrality of the constructed weighted complex networks corroborate with observations commonly found in existing studies, where these metrics follow a power law behaviour. Besides that, the constructed weighted complex networks also obey the small world behaviour based on the statistical pattern found in average shortest path length and average clustering coefficient. Furthermore, a comparative analysis is performed using the boxplot analysis to identify if the statistical patterns are more distinguishable when the test subjects are grouped and compared based on their levels of maintenance efforts. Based on the analysis, it is discovered that the statistical pattern of C-rated software systems tend to deviate away from the patterns found in A-rated and B-rated software systems. This finding shows that the proposed approach can be a useful alternative to identify potential design faults and assess the maintainability of software systems if the software documentations are not available.

- **Proposed method to automatically derive implicit clustering constraints from the implicit structure of OO software systems**. This contribution is aligned to RO2 - *"To propose a method that helps in deriving implicit clustering constraints from the implicit structure of OO software systems with the aid of weighted complex network and graph theoretical analysis"*, where clustering constraints such as must-link and cannot-link constraints are automatically derived from the software systems itself without human intervention. Existing studies in constrained clustering do not explicitly address the problem of deriving clustering constraints when domain experts are not available. Most studies assumed that constraints are provided prior to the clustering process, which is unrealistic in software development especially when

dealing with poorly designed or poorly documented software systems. The proposed method utilises well known graph theory metrics such as in-degree, out-degree, and average shortest path, to automatically identify important classes that contribute toward a particular software functionality. Based on the analysis, the results are translated into clustering constraints such as must-link and cannot-link constraints to help improve the accuracy of software clustering. The proposed method is beneficial in situations where domain experts are non-existent.

- **Proposed method to rank and prioritise explicit clustering constraints in order to reach a consensus among all the domain experts and software developers**. This contribution is aligned to RO3 – "*To propose a method that is capable of deriving explicit clustering constraints from domain experts or software developers who have prior knowledge regarding the software systems*", where the explicit constraints are ranked using fuzzy AHP method. Explicit constraints are further categorised into soft constraints due to their fuzzy and ambiguous nature. Furthermore, an objective function is proposed in order to maximise the fulfilment of all the derived explicit constraints.

- **Proposed method to maximise the fulfilment of clustering constraints, while penalising the violation of constraints**. This contribution is aligned to RO4 - *"To formulate an appropriate objective function that maximises the fulfilment of explicit and implicit constraints, while penalising violation of the constraints"*, such that the proposed method offers extra flexibility in the way clustering constraints are supplied to the software maintainers. Implicit constraints are automatically derived using complex network and graph theoretical analysis, while the explicit constraints are provided by the domain experts. The proposed method has the capability to accept

either one or both sources of the input, and translate them into clustering constraints. Constraints are categorised into hard and soft constraints in this research. Hard constraints are constraints derived from a reliable source of information, such as those derived from the implicit structure of the software itself. They are absolute and must be fulfilled regardless of any condition. Soft constraints, on the other hand, are derived from domain experts who have prior knowledge regarding the software to be maintained. They are good to have but not compulsory. An objective function has been proposed in this research with the aim to maximise the fulfilment of soft constraints. The clustering result that maximises the fulfilment of both hard and soft constraints is preferred as the optimum result.

- **Evaluate the proposed approach using 40 open-source OO software systems**. This contribution is aligned to RO5 - *"To evaluate the accuracy and scalability of the proposed approach using open-source OO software systems"*. In order to improve the generalisation of the research findings, the proposed constrained clustering approach is evaluated using 40 open-source OO software systems that vary according to number of classes, application domains, lines of code, and levels of maintenance efforts. The accuracy of the clustering results is evaluated using the MoJoFM metric. Based on the MoJoFM metric, the proposed constrained clustering approach achieves an aggregated average of 80.11% accuracy when compared against the original package diagrams of the 40 software systems. In terms of scalability, the enhanced Davies-Bouldin index with an adaptive penalty mechanism has been proposed to detect and penalise singleton clusters which has not been dealt with in existing literature. The proposed dendrogram cutting method is also able to scale properly with large datasets such that it minimises the number of cutting points needed to perform constrained clustering.

8.3 Limitations

Although the research managed to achieve its goals, there are some unavoidable limitations in this research. First, because source code is considered very low-level software artifact, there is a limited amount of information that can be reverse-engineered to provide an exact representation of the software design. Hence, when measuring the complexity of class relationships in order to weight the edges of software-based complex network, it is impossible to consider all types of relationships, as shown in Table 4.1, in the research. For instance, it is impossible to clearly differentiate between common association, qualified association, and association class by inspecting the source code alone. Existing round-trip engineering tools available in the market do not have the capability to differentiate between the three types of relationships.

SQALE rating is used in this research in order to estimate the maintainability of the chosen test subjects by grouping them according to their level of maintenance efforts. In general, C-rated projects are deemed to be designed in a less ideal manner (poorly designed) when compared to A-rated and B-rated projects. Thus, the usage of the SQALE rating also provides a means to compare and contrast the accuracy of the proposed constrained clustering approach when applied on relatively well-designed and poorly designed software systems. However, it is difficult to evaluate the proposed approach in extreme cases, such as evaluating on software systems that are severely and very badly designed, i.e. highly coupled software modules and do not follow any kind of software design principles. There are two main reasons why this research did not attempt to address this issue. First, for a software that is severely and badly designed, it is impossible to retrieve a ground truth to validate the accuracy of the clustering results. Unless experts'

opinions are available to validate the clustering results, it is hard to justify the accuracy of the clustering results. Secondly, if a software is severely and very badly designed, it is often abandoned or scrapped before entering the production stage, making it almost impossible to obtain the source code for this type of projects to be evaluated in this research.

Besides that, two types of software quality attributes, namely maintainability and reliability, are considered in this research. This is mainly because maintainability and reliability are directly related to the modularity of software systems, which is the goal of this research – to help in remodularisation of poorly designed or poorly documented software systems. Thus, the scope of the research is limited to these two software quality attributes.

Finally, the proposed constrained clustering approach can only be applied on software systems written in OO programming languages. This is because there is no way to reverse-engineer software systems written in structured programming languages into UML class diagrams. Thus, the proposed approach is applicable to any software systems other than those written in non-OO programming languages.

8.4 Future Work

There are several directions in which the outcome of this research can be extended and improved. Future work can be considered by including more software quality attributes when converting the UML class diagrams into weighted complex networks. Furthermore, when converting source code into UML class diagram, a simple approach is used to identify aggregation and composition relationships. Additional work can be considered

by looking into a formal way of converting UML class diagram notations. Besides that, further work to correlate the graph theory metrics with a more direct measurement of maintenance effort, for instance, by measuring changes and issues of software in multiple releases can be considered. Measuring the frequency of changes between different releases of software systems can be a reliable way to measure the maintainability and reliability of software systems, such that the more changes that are required to address a bug, the greater the maintenance effort.

REFERENCES

Abreu, F. B., & Carapuça, R. (1994). *Object-oriented software engineering: Measuring and controlling the development process.* Paper presented at the proceedings of the 4th International Conference on Software Quality.

Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology, 58*(0), 231-249. doi: 10.1016/j.infsof.2014.08.002

Anquetil, N., & Laval, J. (2011). *Legacy Software Restructuring: Analyzing a Concrete Case.* Paper presented at the 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011.

Anquetil, N., & Lethbridge, T. C. (1999a). *Experiments with clustering as a software remodularization method.* Paper presented at the Sixth Working Conference on Reverse Engineering, 1999. Proceedings.

Anquetil, N., & Lethbridge, T. C. (1999b). Recovering software architecture from the names of source files. *Journal of Software Maintenance-Research and Practice, 11*(3), 201-221.

Anquetil, N., & Lethbridge, T. C. (2003). Comparative study of clustering algorithms and abstract representations for software remodularisation. *IEEE Software Proceedings, 150*(3), 185-201.

Ares, M. E., Parapar, J., & Barreiro, A. (2012). An experimental study of constrained clustering effectiveness in presence of erroneous constraints. *Information Processing & Management, 48*(3), 537-551. doi: 10.1016/j.ipm.2011.08.006

Arisholm, E., Briand, L. C., & Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering, 30*(8), 491-506. doi: Doi 10.1109/Tse.2004.41

Bagheri, E., Di Noia, T., Ragone, A., & Gasevic, D. (2010). Configuring Software Product Line Feature Models Based on Stakeholders' Soft and Hard Requirements. In J. Bosch & J. Lee (Eds.), *Software Product Lines: Going Beyond*: Springer Berlin Heidelberg.

Bair, E. (2013). Semi-supervised clustering methods. *Wiley Interdiscip Rev Comput Stat, 5*(5), 349-361. doi: 10.1002/wics.1270

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4-17. doi: 10.1109/32.979986

Barabasi, A. L., & Albert, R. (1999). Emergence of Scaling in Random Networks. *Science, 286*(5439), 509-512. doi: DOI 10.1126/science.286.5439.509

Barabasi, A. L., Albert, R., & Jeong, H. (2000). Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics*

and its Applications, 281(1–4), 69-77. doi: http://dx.doi.org/10.1016/S0378-4371(00)00018-2

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751-761. doi: 10.1109/32.544352

Basu, S., Banjeree, A., Mooney, E. R., Banerjee, A., & Mooney, R. J. (2004). Active Semi-Supervision for Pairwise Constrained Clustering. In Proceedings of the 2004 SIAM International Conference on Data Mining SDM-04.

Beck, F., & Diehl, S. (2013). On the impact of software evolution on software clustering. *Empirical Software Engineering, 18*(5), 970-1004. doi: 10.1007/s10664-012-9225-9

Bilenko, M., & Mooney, R. J. (2003). *Adaptive duplicate detection using learnable string similarity measures*. Paper presented at the Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, D.C.

Binkley, A. B., & Schach, S. R. (1998). *Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures*. Paper presented at the Proceedings of the 20th international conference on Software engineering, Kyoto, Japan.

Bollen, K. A. (1990). Overall fit in covariance structure models: Two types of sample size effects. *Psychological bulletin, 107*(2), 256.

Briand, L. C., Labiche, Y., & Yihong, W. (2001). *Revisiting strategies for ordering class integration testing in the presence of dependency cycles.* Paper presented at the 12th International Symposium on Software Reliability Engineering, 2001. ISSRE 2001. Proceedings.

Briand, L. C., Labiche, Y., & Yihong, W. (2003). An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering, 29*(7), 594-607. doi: 10.1109/TSE.2003.1214324

Briand, L. C., Wüst, J., Ikonomovski, S. V., & Lounis, H. (1999). *Investigating quality factors in object-oriented designs: an industrial case study*. Paper presented at the Proceedings of the 21st international conference on Software engineering, Los Angeles, California, USA.

Bullmore, E., & Sporns, O. (2009). Complex brain networks: graph theoretical analysis of structural and functional systems. *Nat Rev Neurosci, 10*(3), 186-198. doi: 10.1038/nrn2575

Canfora, G., Cimitile, A., De Lucia, A., & Di Lucca, G. A. (2001). Decomposing legacy systems into objects: an eclectic approach. *Information and Software Technology, 43*(6), 401-412. doi: http://dx.doi.org/10.1016/S0950-5849(01)00149-5

Canfora, G., Czeranski, J., & Koschke, R. (2000). *Revisiting the Delta IC Approach to Component Recovery*. Paper presented at the Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00).

Canfora, G., Di Penta, M., & Cerulo, L. (2011). Achievements and Challenges in Software Reverse Engineering. *Communications of the ACM, 54*(4), 142-151. doi: 10.1145/1924421.1924451

Chatzigeorgiou, A., & Melas, G. (2012). *Trends in object-oriented software evolution: Investigating network properties.* Paper presented at the 34th International Conference on Software Engineering (ICSE), 2012.

Chidamber, S. R., Darcy, D. P., & Kemerer, C. F. (1998). Managerial use of metrics for object-oriented software: an exploratory analysis, *IEEE Transactions on Software Engineering, 24*(8), 629-639. doi: 10.1109/32.707698

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493. doi: Doi 10.1109/32.295895

Chong, C. Y., & Lee, S. P. (2015a). Analyzing maintainability and reliability of object-oriented software using weighted complex network. *Journal of Systems and Software, 110*, 28-53. doi: http://dx.doi.org/10.1016/j.jss.2015.08.014

Chong, C. Y., & Lee, S. P. (2015b). *Constrained Agglomerative Hierarchical Software Clustering with Hard and Soft Constraints*. Paper presented at the ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain. http://dx.doi.org/10.5220/0005344001770188

Chong, C. Y., Lee, S. P., & Ling, T. C. (2013). Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. *Information and Software Technology, 55*(11), 1994-2012.

Chong, C. Y., Lee, S. P., & Ling, T. C. (2014). Prioritizing and Fulfilling Quality Attributes For Virtual Lab Development Through Application of Fuzzy Analytic Hierarchy Process and Software Development Guidelines. *Malaysian Journal of Computer Science, 27*(1).

Cilibrasi, R. L., & Vitanyi, P. M. B. (2007). The Google Similarity Distance. *IEEE Transactions on Knowledge and Data Engineering, 19*(3), 370-383. doi: 10.1109/TKDE.2007.48

Concas, G., Marchesi, M., Murgia, A., Tonelli, R., & Turnu, I. (2011). On the Distribution of Bugs in the Eclipse System. *IEEE Transactions on Software Engineering, 37*(6), 872-877. doi: 10.1109/Tse.2011.54

Concas, G., Marchesi, M., Pinna, S., & Serra, N. (2007). Power-Laws in a Large Object-Oriented Software System. *IEEE Transactions on Software Engineering, 33*(10), 687-708. doi: 10.1109/tse.2007.1019

Cui, J. F., & Chae, H. S. (2011). Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems. *Information and Software Technology, 53*(6), 601-614. doi: 10.1016/j.infsof.2011.01.006

Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the Principal of an Application's Technical Debt. *IEEE Software, 29*(6), 34-42. doi: 10.1109/MS.2012.156

Danielsson, P. E. (1980). Euclidean Distance Mapping. *Computer Graphics and Image Processing, 14*(3), 227-248. doi: Doi 10.1016/0146-664x(80)90054-4

Davey, J., & Burd, E. (2000). *Evaluating the suitability of data clustering for software remodularisation.* Paper presented at the Seventh Working Conference on Reverse Engineering, 2000. Proceedings.

Davidson, I., & Ravi, S. S. (2009). Using instance-level constraints in agglomerative hierarchical clustering: theoretical and empirical results. *Data Mining and Knowledge Discovery, 18*(2), 257-282. doi: 10.1007/s10618-008-0103-4

Davies, D. L., & Bouldin, D. W. (1979). A cluster separation measure. *IEEE Trans Pattern Anal Mach Intell, 1*(2), 224-227.

Dazhou, K., Baowen, X., Jianjiang, L., & Chu, W. C. (2004). *A complexity measure for ontology based on UML*. Paper presented at the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004. FTDCS 2004. Proceedings.

DeMarco, T. (1979). *Structured analysis and system specification*: Yourdon Press.

Deursen, A. v., & Kuipers, T. (1999). *Identifying objects using cluster and concept analysis*. Paper presented at the Proceedings of the 21st international conference on Software engineering, Los Angeles, California, USA.

Dhillon, I. S., Mallela, S., & Kumar, R. (2003). A divisive information theoretic feature clustering algorithm for text classification. *The Journal of Machine Learning Research, 3*, 1265-1287.

Dijkstra, E. W. (1976). *A discipline of programming*: Prentice-Hall Englewood Cliffs.

Ducasse, S., & Pollet, D. (2009). Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering, 35*(4), 573-591. doi: 10.1109/Tse.2009.19

Dunn, J. C. (1973). A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics, 3*(3), 32-57. doi: citeulike-article-id:6774992

Durbin, R. (1998). *Biological sequence analysis : probabalistic models of proteins and nucleic acids*. Cambridge, UK Cambridge University Press.

Eisenblätter, A., Geerdes, H.-F., & Siomina, L. (2007). *Integrated Access Point Placement and Channel Assignment for Wireless LANs in an Indoor Office*

Environment. Paper presented at the IEEE International Symposium on World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007.

El Emam, K., Benlarbi, S., Goel, N., & Rai, S. N. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering, 27*(7), 630-650.

Ferreira, K. A. M., Bigonha, M. A. S., Bigonha, R. S., Mendes, L. F. O., & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software, 85*(2), 244-257. doi: 10.1016/j.jss.2011.05.044

Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., & Sander, J. (2009). *Decomposing object-oriented class modules using an agglomerative clustering technique.* Paper presented at the IEEE International Conference on Software Maintenance, 2009. ICSM 2009.

Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2012). Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software, 85*(10), 2241-2260. doi: 10.1016/j.jss.2012.04.013

Genero, M., Piattini, M., Manso, E., & Cantone, G. (2003). *Building UML class diagram maintainability prediction models based on early metrics.* Paper presented at the Ninth International Software Metrics Symposium, 2003.

Grand, M. (2003). *Patterns in Java: a catalog of reusable design patterns illustrated with UML* (Vol. 1): John Wiley & Sons.

Greenacre, M. (2012). A simple permutation test for clusteredness. Retrieved August, 2014, from http://hdl.handle.net/10230/19856.

Gronau, I., & Moran, S. (2007). Optimal implementations of UPGMA and other common clustering algorithms. *Information Processing Letters, 104*(6), 205-210. doi: 10.1016/j.ipl.2007.07.002

Guoai, X., Yang, G., Fanfan, L., Aiguo, C., & Miao, Z. (2008). *Statistical Analysis of Software Coupling Measurement Based on Complex Networks.* Paper presented at the International Seminar on Future Information Technology and Management Engineering, 2008. FITME '08.

Gurrutxaga, I., Albisua, I., Arbelaitz, O., Martin, J. I., Muguerza, J., Perez, J. M., & Perona, I. (2010). SEP/COP: An efficient method to find the best partition in hierarchical clustering based on a new cluster validity index. *Pattern Recognition, 43*(10), 3364-3373. doi: 10.1016/j.patcog.2010.04.021

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897-910. doi: Doi 10.1109/Tse.2005.112

Hamilton, J., & Danicic, S. (2012). *Dependence communities in source code.* Paper presented at the 28th IEEE International Conference on Software Maintenance (ICSM), 2012

Harman, M., Mansouri, S. A., & Zhang, Y. Y. (2012). Search-Based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys, 45*(1), 11.

Heitlager, I., Kuipers, T., & Visser, J. (2007). *A Practical Model for Measuring Maintainability*. Paper presented at the 6th International Conference on the Quality of Information and Communications Technology, 2007. QUATIC 2007.

Henderson-Sellers, B., Constantine, L. L., & Graham, I. M. (1996). Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems, 3*(3), 143-158.

Hitz, M., & Montazeri, B. (1995). *Measuring coupling and cohesion in object-oriented systems*. Paper presented at the Proceedings of the International Symposium on Applied Corporate Computing.

Hong, Z., & Yiu-Ming, C. (2012). Semi-Supervised Maximum Margin Clustering with Pairwise Constraints. *IEEE Transactions on Knowledge and Data Engineering, 24*(5), 926-939. doi: 10.1109/tkde.2011.68

Hosking, J. R. M., & Wallis, J. R. (1987). Parameter and Quantile Estimation for the Generalized Pareto Distribution. *Technometrics, 29*(3), 339-349. doi: 10.1080/00401706.1987.10488243

Hu, H., Fang, J., Lu, Z., Zhao, F., & Qin, Z. (2012). *Rank-directed layout of UML class diagrams*. Paper presented at the Proceedings of the First International Workshop on Software Mining, Beijing, China.

Huang, J.-H., Wang, L.-C., & Chang, C.-J. (2005). *Deployment strategies of access points for outdoor wireless local area networks*. Paper presented at the Vehicular Technology Conference, 2005. VTC 2005-Spring. 2005 IEEE 61st.

Hughes, M. M. (1979). Exploration and Play Re-Visited: A Hierarchical Analysis. *International Journal of Behavioral Development, 2*(3), 215-224. doi: 10.1177/016502547900200301

Hyland-Wood, D., Carrington, D., & Kaplan, S. (2006). *Scale-free nature of java software package, class and method collaboration graphs*. Paper presented at the Proceedings of the 5th International Symposium on Empirical Software Engineering, Rio de Janeiro, Brasil.

Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. *The American Statistician, 50*(4), 361-365. doi: 10.2307/2684934

Ichii, M., Matsushita, M., & Inoue, K. (2008). *An Exploration of Power-Law in Use-Relation of Java Software Systems*. Paper presented at the 19th Australian Conference on Software Engineering, 2008. ASWEC 2008.

Inman, H. F. (1994). Pearson,Karl and Fisher,R.A. On Statistical Tests - a 1935 Exchange from Nature. *American Statistician, 48*(1), 2-11. doi: 10.1080/00031305.1994.10476010

Izurieta, C., Griffith, I., Reimanis, D., & Luhr, R. (2013). *On the Uncertainty of Technical Debt Measurements.* Paper presented at the International Conference on Information Science and Applications (ICISA), 2013.

Jain, A. K., & Dubes, R. C. (1988). *Algorithms for clustering data*. Englewood Cliffs, N.J.: Prentice Hall.

Jenkins, S., & Kirk, S. R. (2007). Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences, 177*(12), 2587-2601. doi: 10.1016/j.ins.2007.01.021

Karsai, G., Maroti, M., Ledeczi, A., Gray, J., & Sztipanovits, J. (2004). Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology, 12*(2), 263-278. doi: Doi 10.1109/Tcst.2004.824311

Kemerer, C. F. (1995). Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering, 1*(1), 1-22. doi: 10.1007/bf02249043

Kestler, H. A., Kraus, J. M., Palm, G., & Schwenker, F. (2006). On the Effects of Constraints in Semi-supervised Hierarchical Clustering. In F. Schwenker & S. Marinai (Eds.), *Artificial Neural Networks in Pattern Recognition* (pp. 57-66): Springer Berlin Heidelberg.

Kim, M., & Ramakrishna, R. (2005). New indices for cluster validity assessment. *Pattern Recognition Letters, 26*(15), 2353-2363.

Klein, D., Kamvar, S. D., & Manning, C. D. (2002). *From Instance-level Constraints to Space-Level Constraints: Making the Most of Prior Knowledge in Data Clustering*. Paper presented at the Proceedings of the Nineteenth International Conference on Machine Learning.

Kollmann, R., Selonen, P., Stroulia, E., Systa, T., & Zundorf, A. (2002). *A study on the current state of the art in tool-supported UML-based static reverse engineering.* Paper presented at the Ninth Working Conference on Reverse Engineering, 2002. Proceedings.

Kumar, S., & Phrommathed, P. (2005). *Research methodology*: Springer.

LaBelle, N., & Wallingford, E. (2004). Inter-package dependency networks in open-source software. *arXiv preprint cs/0411096*.

Lan, W., Zhou, K., Feng, J., & Chi, Z. (2010). *Research on Software Cascading Failures.* Paper presented at the International Conference on Multimedia Information Networking and Security (MINES), 2010.

Leg, C., & Babos, A. (2006). *Cluster validity measurement techniques*. Paper presented at the Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, Madrid, Spain.

Letouzey, J., & Ilkiewicz, M. (2012). Managing Technical Debt with the SQALE Method. *IEEE Software, 29*(6), 44-51. doi: 10.1109/MS.2012.129

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*(2), 111-122. doi: http://dx.doi.org/10.1016/0164-1212(93)90077-B

Lian, W., Kirk, D., & Dromey, R. G. (2007). *Software Systems as Complex Networks.* Paper presented at the 6th IEEE International Conference on Cognitive Informatics.

Lim, E., Taksande, N., & Seaman, C. (2012). A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software, 29*(6), 22-27. doi: 10.1109/MS.2012.130

Linoff, G. S., & Berry, M. J. A. (2011). *Data mining techniques : for marketing, sales, and customer relationship management* (3rd ed.). Indianapolis, IN: Wiley Pub.

Liou, T. S., & Wang, M. J. J. (1992). Ranking Fuzzy Numbers with Integral Value. *Fuzzy Sets and Systems, 50*(3), 247-255. doi: Doi 10.1016/0165-0114(92)90223-Q

Liu, Y. Y., Slotine, J. J., & Barabasi, A. L. (2011). Controllability of complex networks. *Nature, 473*(7346), 167-173. doi: 10.1038/nature10011

Louridas, P., Spinellis, D., & Vlachos, V. (2008). Power Laws in Software. *ACM Transactions on Software Engineering and Methodology, 18*(1), 1-26.

Lung, C., & Zhou, C. (2008). *Using Hierarchical Agglomerative Clustering in Wireless Sensor Networks: An Energy-Efficient and Flexible Approach.* Paper presented at the Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE.

Lung, C., Zaman, M., & Nandi, A. (2004). Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software, 73*(2), 227-244. doi: 10.1016/s0164-1212(03)00234-6

Ma, Y. T., He, K. Q., Li, B., Liu, J., & Zhou, X. Y. (2010). A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems. *Journal of Computer Science and Technology, 25*(6), 1184-1201. doi: 10.1007/s11390-010-1094-3

Malliaros, F. D., & Vazirgiannis, M. (2013). Clustering and community detection in directed networks: A survey. *Physics Reports-Review Section of Physics Letters, 533*(4), 95-142. doi: 10.1016/j.physrep.2013.08.002

Maqbool, O., & Babri, H. A. (2006). Automated software clustering: An insight using cluster labels. *Journal of Systems and Software, 79*(11), 1632-1648. doi: 10.1016/j.jss.2006.03.013

Maqbool, O., & Babri, H. A. (2007). Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering, 33*(11), 759-780. doi: 10.1109/Tse.2007.70732

Martin, R. (1994). OO design quality metrics. *An analysis of dependencies*, 151-170.

MathArc - Ensuring Access to Mathematics Over Time. (2009) Retrieved 1st April, 2014, from http://www.library.cornell.edu/dlit/MathArc/web/index.html.

MathWave. (2014). EasyFit. Retrieved 1st April, 2014, from http://www.mathwave.com/easyfit-distribution-fitting.html

Maulik, U., & Bandyopadhyay, S. (2002). Performance evaluation of some clustering algorithms and validity indices. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 24*(12), 1650-1654.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308-320. doi: 10.1109/TSE.1976.233837

Mcsweeney, P. J. (2008). Random Network Plugin. Retrieved August, 2014, from https://sites.google.com/site/randomnetworkplugin/Home

Mei-Huei, T., Ming-Hung, K., & Mei-Hwa, C. (1999). *An empirical study on object-oriented metrics.* Paper presented at the Sixth International Software Metrics Symposium, 1999. Proceedings.

Milanova, A. (2005). *Precise identification of composition relationships for UML class diagrams*. Paper presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA.

Milanova, A. (2007). Composition inference for UML class diagrams. *Automated Software Engineering, 14*(2), 179-213. doi: 10.1007/s10515-007-0010-8

Mirkin, B. (2004). Cluster analysis for researchers. *Journal of Classification, 21*(2), 279-283. doi: DOI 10.1007/s00357-004-0020-2

Mitchell, B. S., & Mancoridis, S. (2001). *Comparing the decompositions produced by software clustering algorithms using similarity measurements.* Paper presented at the IEEE International Conference onSoftware Maintenance, 2001. Proceedings.

Miyamoto, S. (2012). An Overview of Hierarchical and Non-hierarchical Algorithms of Clustering for Semi-supervised Classification. In V. Torra, Y. Narukawa, B. López, & M. Villaret (Eds.), *Modeling Decisions for Artificial Intelligence* (Vol. 7647, pp. 1-10): Springer Berlin Heidelberg.

Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E, 68*(4), 046116.

Newman, M. E. J. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences, 103*(23), 8577-8582. doi: 10.1073/pnas.0601602103

Olague, H. M., Etzkorn, L. H., Gholston, S., & Quattlebaum, S. (2007a). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering, 33*(6), 402-419. doi: 10.1109/Tse.2007.1015

Olague, H. M., Etzkorn, L. H., Gholston, S., & Quattlebaum, S. (2007b). Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering, 33*(6), 402-419. doi: 10.1109/TSE.2007.1015

Oracle. Java Platform SE 7 Documentation.

Ovatman, T., Weigert, T., & Buzluca, F. (2011). Exploring implicit parallelism in class diagrams. *Journal of Systems and Software, 84*(5), 821-834. doi: http://dx.doi.org/10.1016/j.jss.2011.01.005

Pang, T. Y., & Maslov, S. (2013). Universal distribution of component frequencies in biological and technological systems. *Proceedings of the National Academy of Sciences, 110*(15), 6235-6239. doi: 10.1073/pnas.1217795110

Passos, L., Terra, R., Valente, M. T., Diniz, R., & Mendonca, N. (2010). Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software, 27*(5), 82-89. doi: 10.1109/MS.2009.117

Patel, C., Hamou-Lhadj, A., & Rilling, J. (2009). *Software Clustering Using Dynamic Analysis and Static Dependencies.* Paper presented at the 13th European Conference on Software Maintenance and Reengineering, 2009. CSMR '09.

Perez-Castillo, R., & Piattini, M. (2014). Analyzing the Harmful Effect of God Class Refactoring on Power Consumption. *IEEE Software, 31*(3), 48-54.

Pirzadeh, H., Alawneh, L., & Hamou-Lhadj, A. (2009). *Quality of the Source Code for Design and Architecture Recovery Techniques: Utilities are the Problem.* Paper presented at the 9th International Conference on Quality Software, 2009. QSIC '09.

Porres, I., & Alanen, M. (2003). *A generic deep copy algorithm for MOF-based models.* Paper presented at the Model Driven Architecture: Foundations and Applications.

Potanin, A., Noble, J., Frean, M., & Biddle, R. (2005). Scale-free geometry in OO programs. *Communications of the ACM, 48*(5), 99-103. doi: Doi 10.1145/1060710.1060716

Praditwong, K., Harman, M., & Yao, X. (2011). Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering, 37*(2), 264-282. doi: 10.1109/Tse.2010.26

Ragab, S. R., & Hany, H. A. (2010). *Object oriented design metrics and tools a survey.* Paper presented at the 7th International Conference on Informatics and Systems (INFOS), 2010

Ravasz, E., & Barabasi, A. L. (2003). Hierarchical organization in complex networks. *Phys Rev E Stat Nonlin Soft Matter Phys, 67*(2 Pt 2), 026112. doi: 10.1103/PhysRevE.67.026112

Saaty, T. L. (1980). *The analytic hierarchy process : planning, priority setting, resource allocation*. New York ; London: McGraw-Hill International Book Co.

Saha, S., & Bandyopadhyay, S. (2009). Performance Evaluation of Some Symmetry-Based Cluster Validity Indexes. *IEEE Transactions on Systems Man and Cybernetics Part C-Applications and Reviews, 39*(4), 420-425. doi: 10.1109/Tsmcc.2009.2013335

Santos, G., Valente, M. T., & Anquetil, N. (2014). *Remodularization analysis using semantic clustering.* Paper presented at the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week.

Satuluri, V., & Parthasarathy, S. (2011). *Symmetrizations for clustering directed graphs.* Paper presented at the Proceedings of the 14th International Conference on Extending Database Technology.

Sestoft, P. (1999). Programs for biosequence analysis. Retrieved August, 2014, from http://www.itu.dk/people/sestoft/bsa.html

Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., . . . Ideker, T. (2003). Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research, 13*(11), 2498-2504.

Shental, N., & Weinshall, D. (2003). *Learning Distance Functions using Equivalence Relations.* Paper presented at the In Proceedings of the Twentieth International Conference on Machine Learning.

Simon, H. A. (1991). The Architecture of Complexity *Facets of Systems Science* (Vol. 7, pp. 457-476): Springer US.

Singh, G. (2013). Metrics for measuring the quality of object-oriented software. *ACM SIGSOFT Software Engineering Notes, 38*(5), 1. doi: 10.1145/2507288.2507311

Smirnov, N. (1948). Table for Estimating the Goodness of Fit of Empirical Distributions. *Annals of Mathematical Statistics, 19*(2), 279-279. doi: DOI 10.1214/aoms/1177730256

SonarQube. (2014). SonarQube. Retrieved 1st April, 2014, from http://www.sonarqube.org/.

Sørensen, T. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr., 5*, 1-34. doi: citeulike-article-id:7654646

Stein, C. M. (1981). Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, 1135-1151.

Sterling, C. (2010). *Managing software debt: building for inevitable change*: Addison-Wesley Professional.

Stevens, W., Myers, G., & Constantine, L. (1979). Structured design. In Y. Edward Nash (Ed.), *Classics in software engineering* (pp. 205-232): Yourdon Press.

Stumpf, M. P. H., & Porter, M. A. (2012). Critical Truths About Power Laws. *Science, 335*(6069), 665-666. doi: 10.1126/science.1216142

Subramanyam, R., & Krishnan, M. S. (2003a). Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering, 29*(4), 297-310. doi: 10.1109/TSE.2003.1191795

Subramanyam, R., & Krishnan, M. S. (2003b). Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering, 29*(4), 297-310. doi: Doi 10.1109/Tse.2003.1191795

Sun, S., Xia, C., Chen, Z., Sun, J., & Wang, L. (2009). *On Structural Properties of Large-Scale Software Systems: From the Perspective of Complex Networks.* Paper presented at the Sixth International Conference on Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09.

Tanaka, J. S. (1987). "How Big Is Big Enough?": Sample Size and Goodness of Fit in Structural Equation Models with Latent Variables. *Child Development, 58*(1), 134. doi: 10.2307/1130296

Taube-Schock, C., Walker, R., & Witten, I. (2011). Can We Avoid High Coupling? In M. Mezini (Ed.), *ECOOP 2011 – Object-Oriented Programming* (Vol. 6813, pp. 204-228): Springer Berlin Heidelberg.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., . . . Noble, J. (2010). *The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies.* Paper presented at the Software Engineering Conference (APSEC), 2010 17th Asia Pacific.

Tonella, P. (2001). Concept Analysis for Module Restructuring. *IEEE Trans. Softw. Eng., 27*(4), 351-363. doi: 10.1109/32.917524

Turnu, I., Concas, G., Marchesi, M., & Tonelli, R. (2013). The fractal dimension of software networks as a global quality metric. *Information Sciences, 245*(0), 290-303. doi: 10.1016/j.ins.2013.05.014

Turnu, I., Marchesi, M., & Tonelli, R. (2012). *Entropy of the degree distribution and object-oriented software quality.* Paper presented at the 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM), 2012.

Valverde, S., Cancho, R. F., & Sole, R. V. (2002). Scale-free networks from optimal design. *EPL (Europhysics Letters), 60*(4), 512.

Valverde, S., & Solé, R. V. (2003). Hierarchical small worlds in software architecture. *arXiv preprint cond-mat/0307278*.

Van Solingen, R., Basili, V., Caldiera, G., & Rombach, H. D. (2002). Goal question metric (gqm) approach. *Encyclopedia of Software Engineering*.

Wagstaff, K., & Cardie, C. (2000). *Clustering with Instance-level Constraints*. Paper presented at the Proceedings of the Seventeenth International Conference on Machine Learning.

Wang, B., & Lu, J. (2012, 6-8 July 2012). *Modelling complex software systems via weighted networks.* Paper presented at the 10th World Congress on Intelligent Control and Automation (WCICA), 2012

Warrens, M. J. (2009). k-Adic Similarity Coefficients for Binary (Presence/Absence) Data. *Journal of Classification, 26*(2), 227-245. doi: 10.1007/s00357-009-9032-1

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature, 393*(6684), 440-442. doi: Doi 10.1038/30918

Wen, Z., & Tzerpos, V. (2004). *An effectiveness measure for software clustering algorithms.* Paper presented at the 12th IEEE International Workshop on Program Comprehension, 2004. Proceedings.

Wen, Z., & Tzerpos, V. (2005). *Software clustering based on omnipresent object detection.* Paper presented at the 13th International Workshop on Program Comprehension, 2005. IWPC 2005.

Wiggerts, T. A. (1997). *Using clustering algorithms in legacy systems remodularization.* Paper presented at the Proceedings of the Fourth Working Conference on Reverse Engineering, 1997.

Williamson, D. F., Parker, R. A., & Kendrick, J. S. (1989). The Box Plot: A Simple Visual Method to Interpret Data. *Annals of Internal Medicine, 110*(11), 916-921. doi: 10.7326/0003-4819-110-11-916

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*: Springer Science & Business Media.

Wolberg, J. R. (2006). *Data analysis using the method of least squares : extracting the most information from experiments*. Berlin ; New York: Springer.

Wu, J., Hassan, A. E., & Holt, R. C. (2005, September). Comparison of clustering algorithms in the context of software evolution. In Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on (pp. 525-535). IEEE.

Yang, G., Guoai, X., Yixian, Y., Xinxin, N., & Shize, G. (2010). *Empirical analysis of software coupling networks in object-oriented software systems.* Paper presented at the IEEE International Conference on Software Engineering and Service Sciences (ICSESS), 2010.

Yang, G., Jia, L., Shuai, S., Guoai, X., & Gong, C. (2013). Weighted Networks of Object-Oriented Software Systems: The Distribution of Vertex Strength and Correlation. In G. Yang (Ed.), *Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering* (Vol. 181, pp. 1185-1190): Springer Berlin Heidelberg.

Yann-Gaël, G. (2004). A reverse engineering tool for precise class diagrams. In Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (pp. 28-41). IBM Press.

Yoon, J., Blumer, A., & Lee, K. (2006). An algorithm for modularity analysis of directed and weighted biological networks based on edge-betweenness centrality. *Bioinformatics*, *22*(24), 3106-3108. doi: 10.1093/bioinformatics/btl533

Zimmermann, T., & Nagappan, N. (2008). *Predicting defects using network analysis on dependency graphs.* Paper presented at the Proceedings of the 30th international conference on Software engineering.

LIST OF PUBLICATIONS AND PAPERS PRESENTED

Chong, C. Y., & Lee, S. P. (2015a). Analyzing maintainability and reliability of object-oriented software using weighted complex network. *Journal of Systems and Software, 110*, 28-53. doi: http://dx.doi.org/10.1016/j.jss.2015.08.014

Chong, C. Y., & Lee, S. P. (2015b). *Constrained Agglomerative Hierarchical Software Clustering with Hard and Soft Constraints*. Paper presented at the ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain. http://dx.doi.org/10.5220/0005344001770188

Chong, C. Y., Lee, S. P., & Ling, T. C. (2013). Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. *Information and Software Technology, 55*(11), 1994-2012.

Chong, C. Y., Lee, S. P., & Ling, T. C. (2014). Prioritizing and Fulfilling Quality Attributes For Virtual Lab Development Through Application of Fuzzy Analytic Hierarchy Process and Software Development Guidelines. *Malaysian Journal of Computer Science, 27*(1).