Automatic Clustering Constraints Derivation from Object-Oriented Software Using Weighted Complex Network with Graph Theory Analysis

Article *in* Journal of Systems and Software · August 2017

DOI: 10.1016/j.jss.2017.08.017

Chun Yong Chong
Monash University (Malaysia)

Automatic Clustering Constraints Derivation from Object-Oriented Software Using Weighted Complex Network with Graph Theory Analysis

Chun Yong Chong[1], Sai Peck Lee[2]

[1] *School of Information Technology, Monash University Malaysia, Jalan Lagoon Selatan, 47500 Bandar Sunway, Selangor Darul Ehsan, Malaysia*

[2] *Department of Software Engineering, Faculty of Computer Science and IT, University of Malaya, 50603 Lembah Pantai, Kuala Lumpur, Malaysia*

Email: chong.chunyong@monash.edu, saipeck@um.edu.my

Abstract

Constrained clustering or semi-supervised clustering has received a lot of attention due to its flexibility of incorporating minimal supervision of domain experts or side information to help improve clustering results of classic unsupervised clustering techniques. In the domain of software remodularisation, classic unsupervised software clustering techniques have proven to be useful to aid in recovering a high-level abstraction of the software design of poorly documented or designed software systems. However, there is a lack of work that integrates constrained clustering for the same purpose to help improve the modularity of software systems. Nevertheless, due to time and budget constraints, it is laborious and unrealistic for domain experts who have prior knowledge about the software to review each and every software artifact and provide supervision on an on-demand basis. We aim to fill this research gap by proposing an automated approach to derive clustering constraints from the implicit structure of software system based on graph theory analysis of the analysed software. Evaluations conducted on 40 open-source object-oriented software systems show that the proposed approach can serve as an alternative solution to derive clustering constraints in situations where domain experts are non-existent, thus helping to improve the overall accuracy of clustering results.

Keyword: constrained clustering; software clustering; software remodularisation; graph theory; complex network

1. Introduction

Maintenance of existing software requires plenty of time in analysing and comprehending the available source code and software documentation. Successful accomplishment of software maintenance is highly dependent on how much information can be extracted by software maintainers. However, due to prolonged maintenance and software updates, the architectural design of software system tends to deviate away from the original design, causing further difficulties in software maintenance. Recovering the architecture of software is therefore an important step to aid in software maintenance. In general, software architecture recovery aims to extract a high-level representation of the architectural information from low-level software artifacts, such as source code, to ensure the fulfilment of requirements, identification of reusable software components, and estimation of cost and risks associated to any changes in requirements (Maqbool & Babri, 2007; Riva, 2000).

Software clustering has received a substantial attention in recent years due to its capability to help in recovering a semantic representation of the software design, which directly aid in software architecture recovery (Maqbool & Babri, 2006, 2007). 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. 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 classic unsupervised clustering approaches was proposed in the work by (Basu, Banerjee, & Mooney, 2004), commonly referred as semi-supervised clustering or constrained clustering, where side information is integrated to further improve the accuracy of clustering results. In the domain of software clustering, semi-supervised approaches typically use small samples of software modules with known cluster assignment which enhances the process of model training (clustering process) with software modules for which the cluster information is not available. The side information, which is commonly referred as clustering constraints that reveal the similarity between pairs of clustering entities or user preferences about how those entities should be grouped during clustering, can be originated explicitly from the domain expert or implicitly from the background knowledge of the problem domain. The clustering constraints may impose certain restrictions such as forcing a pair of clustering entities to be always grouped into the same cluster, or separated into disjoint clusters. These constraints are commonly referred as must-link (ML) and cannot-link (CL) constraints respectively. It has been proven in several fields of research that even some minimal supervision can improve the reliability and accuracy of clustering results (Davidson & Ravi, 2009).

To help reveal the structure and behaviour of software systems, domain experts can exert their opinions in the form of ML or CL constraints to alter the clustering process. However, manually supervising and providing clustering constraints is costly and time consuming (Wagstaff, 2007) for large and complex software systems. Identifying relationships between software components of a software that contains thousand or million lines of code would require a significant amount of time and effort to read all of them carefully. Besides that, most of the time, software maintainers are not directly involved in the early stage of software design especially if the maintenance tasks is outsourced to a third party company. The situation is even worse if the software is poorly designed or the software documentation is not up-to-date, which is common for systems developed in an ad-hoc manner. While most of the existing studies often assumed that feedbacks from domain experts are always readily available, the same assumption cannot be applied in software development especially when dealing with poorly designed or poorly documented software systems.

Most of the existing studies require access to domain experts or a small set of clustering constraints (supervised labelled data) as a pre-requisite. Although feedbacks or supervision by domain experts are useful, one important and non-trivial research question remains open i.e. how to retrieve clustering constraints if experts are not confident with the constraints given or ~~experts are non-existent at all~~such expertise is not available? While various studies have shown that a small number 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. In normal software development practice, the availability of clustering constraints is limited due to reasons such as high cost, time constraint, out-dated software documentations, or limited background knowledge on the software to be maintained (Harman, Mansouri, & Zhang, 2012). For instance, domain experts 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 might be ambiguous or contain erroneous information. In such cases, regular supervised or semi-supervised techniques discussed above cannot be used to effectively recover a high-level abstraction of software design. Hence, constrained clustering approaches that automatically generate or derive constraints from the implicit structure and behaviour of the dataset, and rely less on human effort, are more preferred in the domain of software.

To address this issue, this research proposes an approach to automatically derive ML and CL constraints from the implicit structure of the software itself based on graph theory analysis of the studied software, without feedbacks and supervision from domain experts. First, the software to be analysed is represented using a weighted complex network, followed by graph theory analysis to reveal some extra deterministic information about relationships among all the associated classes. The information is then used to support the subsequent constrained clustering approach to form cohesive clusters that are representative enough to show a high-level representation of the software design. 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. The contribution of this paper can be summarised as follows:

1. An approach to apply semi-supervised constraint clustering to aid in recovering a high-level abstraction of object-oriented software design.
2. An approach to derive clustering constraints from software systems without the feedback and supervision from domain experts.
3. An alternative solution to derive clustering constraints that helps in improving the accuracy of conventional software clustering approaches.

The paper is organized as follows: Section 2 discusses the background and related work in constrained clustering, including ways to generate and acquire constraints. Section 3 presents the proposed approach to automatically derive clustering constraints from an object-oriented software system. Section 4 presents the experimental design, along with the execution of the experiment. Section 5 gives an overall discussion based on the results obtained in the previous section. Section 6 discusses the threats to validity in this study. Finally, concluding remarks and potential future work are presented in Section 7.

2. Background and Related works

Semi-supervised clustering, or commonly referred as constrained clustering has proven to be a reliable alternative to classic unsupervised approaches where a small quantity of clustering constraints is introduced in the clustering process. Clustering constraints in the form of ML and CL constraints guide the clustering algorithm into an adequate partitioning of the data, and often, improves the clustering performance significantly. Existing methods for constrained clustering fall into three categories: distance based (Bilenko & Mooney, 2003; Klein, Kamvar, & Manning, 2002; Shental & Weinshall, 2003), constrained based (Davidson & Ravi, 2009; Kestler, Kraus, Palm, & Schwenker, 2006), and the hybrid of both.

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 development 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. Domain experts often act as oracles in constrained clustering (Basu et al., 2004), where in general, a pair of clustering entities are chosen at random and presented to the oracle to judge and decide if they should or should not be grouped into the same cluster.

On the other hand, implicit information refers to some extra deterministic information about the interrelationships between software components derived from the source code itself. In various fields of research, a limited degree of side information can be revealed when performing an exploratory data analysis (Greene & Cunningham, 2007). For instance, two classes associated with inheritance relationship in object-oriented (OO) paradigm typically have stronger tendency to be grouped into the same cluster. While the given example is a straightforward one, effectively deriving implicit information from the source code requires in-depth understanding on the structure and behaviour of software systems. Software maintainers would 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. The vast amount of information hidden in the source code are worthless unless there is a proper way to synthesise them.

Due to the numerous ways clustering constraints are acquired and derived in existing studies, in this research, a review on state-of-the-art constraints generation and acquisition methods was conducted to highlight the challenges faced in the current works.

2.1 Constraints Generation and Acquisition

Clustering constraints are generated by domain experts with some knowledge about the problem domain. Formally, it is assumed that one can pose a number of queries to an accurate and noiseless domain expert (sometimes known as oracle), who is capable of assigning a ML or CL constraint on a given pair of cluster entities (x_i, x_j). Existing works (Frigui & Hwang, 2008; Xiong, Azimi, & Fern, 2014) often assumed that supervision by domain experts is readily available, and exploited this assumption by iteratively choosing a random pair of cluster entities, and querying the experts if the selected pair should or should not be grouped into the same cluster. Clustering constraints in the form of ML and CL can then be formed based on the feedbacks, which is feasible but costly in terms of time and human effort.

A general problem in constrained clustering that solely relies on expert feedback is that a large number of queries may be required before any noticeable improvement in clustering accuracy is achieved. Hence, several studies have attempted to minimise the number of queries by identifying the most “informative” data. For instance, the work by Basu et al. (2004) focused on identifying and deriving constraints by performing an exploratory analysis on a given dataset with a two-stage Explore and Consolidate approach, based on k-mean clustering.

~~In the domain of software fault prediction, the work by Khoshgoftaar and Seliya (2003); Seliya and Khoshgoftaar (2007) proposed an approach that uses k-mean clustering algorithms to detect fault prone software components. The assumption made by the authors is that fault prone software modules will have higher tendency to be grouped into the same cluster if they share similar characteristics. However, the shortcoming of this approach is that the accuracy of the clustering results is dependent on the experience of software engineering experts. In addition, the approach cannot be automated, as it depends on experts for the prediction. It also does not scale well with large datasets because the experts will need to spend much more time labelling all the software modules.~~

2.2 Deriving Additional Constraints from Existing Supervised Labelled Data

Apart from solely depending on experts or oracles to provide clustering constraints, there are also several studies that attempt to deduce and expand additional clustering constraints from the given sets of supervised data.

For instance, the work by Xiong et al. (2014) introduced a way to deduce additional clustering constraints from a limited amount of supervised data. The authors proposed an active learning method to find the most significant pair of cluster entities, and iteratively query the domain experts to derive clustering constraints. The approach starts by assuming that for a given set of data, a small amount of labelled instances, herein referred as clustering constraints, is provided. Then, an active learning framework is created based on the neighbourhood structure of the clusters formed by the provided constraints. The active learning framework aims to expand the neighbourhood formed by the initial set of clustering constraints, by iteratively seeking the best pair of clustering entities to be included into the existing neighbourhood. Once the candidates are found, they are presented to the domain experts to ascertain the decision. However, since the framework requires repeated re-clustering of the data with incrementally growing constraints set, the approach can be computationally demanding for large data sets.

The work presented by Klein et al. (2002) also introduced a way to deduce additional clustering constraints from a given small amount of supervised data. Based on the given instance-level constraints (ML and CL), the authors proposed a method to deduce space-level constraints by means of constraints propagation. The authors argued that for a given ML constraint involving clustering entities (x_i, x_j), if entity x_i and x_j are very close together, then entities that are very close to x_i are close to x_j as well, resulting in a propagation of clustering constraints. The same concept is applied to CL constraints. The proposed approach is experimented using a constrained k-mean clustering algorithm.

<u>2.3 Automatically Derive Implicit Clustering Constraints</u>

The work by Greene and Cunningham (2007) attempted to solve this problem by identifying "obvious" or "easy" clustering constraints by examining the relationships between pairs of data over a large collection of clustering results. The fundamental assumption underlying this approach is that clustering entities belonging to the same natural cluster will frequently be co-assigned during repeated executions of a clustering algorithm. The authors used k-mean clustering algorithm with varying *k* value (*k* = number of clusters) to generate a large collection of baseline clustering results. Next, based on the generated results, clustering entities that have been repeatedly grouped into the same cluster are identified. These sets of clustering entities are then labelled as ML constraints since they are frequently co-assigned during repeated executions of the k-mean clustering algorithm.

On the other hand, a way to automatically generate clustering constraints based on a two-phase k-mean clustering and hierarchical clustering algorithm was proposed in the work by Diaz-Valenzuela, Loia, Martin-Bautista, Senatore, and Vila (2016). First, the authors performed multiple iterations of k-mean clustering algorithm with varying *k* values to get an initial set of clustering results. Then, clustering entities that are repeatedly co-assigned into the same clusters over multiple clustering iterations are labelled as ML or CL constraints, similar to the work by Greene and Cunningham (2007). Subsequently, the generated constraints are applied using a hierarchical clustering algorithm. The authors argued that the hierarchical clustering algorithm often produces more accurate clustering results in the domain of document clustering because the hierarchical structure is often more informative than the unstructured set of clusters produced by flat clustering, i.e. k-mean clustering. Besides that, in hierarchical clustering, one does not need to specify the number of clusters because it is capable of finding the natural number of partitions on the dataset.

Table 1 provides a summary of all the discussed papers. In summary, most of the existing work use k-mean to perform constrained clustering, partly because the fulfilment of ML and CL constraints can be achieved easier by manipulating the clustering assignment, i.e. initial seeding of clustering entities involved in ML or CL constraints. However, it is more difficult to achieve the same results for hierarchical clustering because all clustering entities are linked together at some level of the cluster hierarchy (Bair, 2013). One needs to ensure that all the ML and CL constraints are fulfilled at every level of cluster hierarchy.

Table 1: Summary of related works on constrained clustering

Author	Problem domain	Clustering technique	Source of clustering constraints	Objective	Evaluation method
(Frigui & Hwang, 2008)	Generic	Fuzzy c-mean	Experts	Perform fuzzy clustering and aggregation of relational data with limited supervision.	Evaluated the proposed approach in two different applications; one using 23 mushroom species, and another using a collection of 500 color images.
(Xiong et al., 2014)	Generic	Constrained k-mean	Experts, active learning	Create a generic framework that actively learns the most important information exist in the dataset, and forms queries to experts accordingly to retrieve ML and CL constraints.	Evaluated the proposed approach on eight benchmark datasets from UC Irvine Machine Learning (UCI ML) repository.
(Basu et al., 2004)	Generic	Constrained k-mean	Experts, active learning	Minimise the queries to experts by actively seeking pairs of entities that are informative enough to be ML or CL candidates.	Evaluated the proposed approach using textual data and UCI ML repository.
~~(Khoshgoftaar & Seliya, 2003; Seliya & Khoshgoftaar, 2007)~~	~~Software fault prediction~~	~~k-mean~~	~~Experts~~	~~Predict fault prone and non-fault prone software modules with the absence of defect data.~~	~~Evaluated the proposed approach using software measurement and defect data from a previously developed NASA software project JM1.~~
(Klein et al., 2002)	Generic	Agglomerative hierarchical clustering	Small amount of supervised data, active learning	Deduce additional clustering constraints from a given small amount of supervised data.	Evaluated the proposed approach using synthetic datasets and real-world datasets from UCL ML repository.
(Greene & Cunningham, 2007)	Generic	k-mean	Derived automatically from the datasets using a co-association method	Provide a way to identify clustering constraints without supervision from domain experts.	Evaluated the proposed approach using six textual datasets.
(Diaz-Valenzuela et al., 2016)	Document classification	K-mean and hierarchical clustering	Derived automatically from the datasets using a co-association method	A way to automatically generate clustering constraints in the absence of domain experts.	Evaluated the proposed approach using two real-world textual datasets.

As shown in Table 1, existing studies in constrained clustering mainly focus on the domain of data mining and machine learning to cluster or classify text documents, images, and to perform biological classifications. While there are several studies that apply a classic unsupervised software clustering technique to aid in remodularisation of poorly designed or poorly documented software systems (Chong et al., 2013; Cui & Chae, 2011; Fokaefs, Tsantalis, Chatzigeorgiou, & Sander, 2009; Maqbool & Babri, 2007) there is a lack of work that integrates domain knowledge or side information for the same purpose.

Strong understanding of software systems is generally 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. 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).

In spite of their wide usage, both 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; Subramanyam & Krishnan, 2003).

~~As software systems become larger and more complex, software maintainers need to gain a better understanding of the macroscopic properties of these systems for making 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 have 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 network enables 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. This research aims to fill in the research gap in constrained clustering, where we found that there is a lack of automated approach to derive clustering constraints from the implicit structure of a software system, in the case where there are no reliable resources such as experts or documentation can be referred to. The next section provides an in-depth discussion on the existing studies that represent software systems using complex network.

2.4 Representing Software Systems using Complex Network

In recent years, research in software engineering in the aspect of representing software systems using complex network has started to emerge with the aim to gain a high-level abstraction view of the analysed software systems (Giulio Concas, Marchesi, Murgia, Tonelli, & Turnu, 2011; Ma, He, Li, Liu, & Zhou, 2010; Tempero et al., 2010). Representing software systems using complex network 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).

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.

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 (Chong & Lee, 2015a; G. Concas, Marchesi, Pinna, & Serra, 2007; Louridas, Spinellis, & Vlachos, 2008; Pang & Maslov, 2013; Potanin, Noble, Frean, & Biddle, 2005).

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 (G. 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 minimum 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 degrees, $P(k)$, 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 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 complex 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.

2.5 Summary

As discussed earlier, it is unrealistic to assume that feedbacks from domain experts or experienced software developers are always available in typical software development practices due to its ad-hoc nature.

Less attention is focused on how to automatically derive clustering constraints from the software itself if experts' opinions are not available. We argued that other options to automatically extract constraints from the implicit features and behaviours of the software systems are needed. However, identifying and analysing constraints from the implicit structure of software systems remains as a challenging research problem.

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.

In our previous work, an approach to represent an OO software system using a weighted complex network was proposed in order to capture its structural characteristics, with respect to maintainability and reliability (Chong & Lee, 2015a). Nodes and edges are modeled based on the complexities of classes and their dependencies. We had applied several graph theory metrics onto the transformed weighted complex network with the purpose to evaluate the maintainability of a software system. Experimental results showed that representing a software system using a weighted complex network is capable of revealing some extra-deterministic information on the studied software, and offering additional insights toward understanding its structure and behavior through the application of well-established graph theory metrics.

Hence, guided by our previous work to represent software systems using weighted complex network, we aim to propose an automated approach to derive clustering constraints from the implicit structure of a software system with the aid of graph theory analysis.

3. Proposed Approach

Based on the summary of issues highlighted in Section 2.5, an approach to automatically derive clustering constraints from the implicit structure and behaviour of software system is proposed. Figure 1 depicts the steps involved.

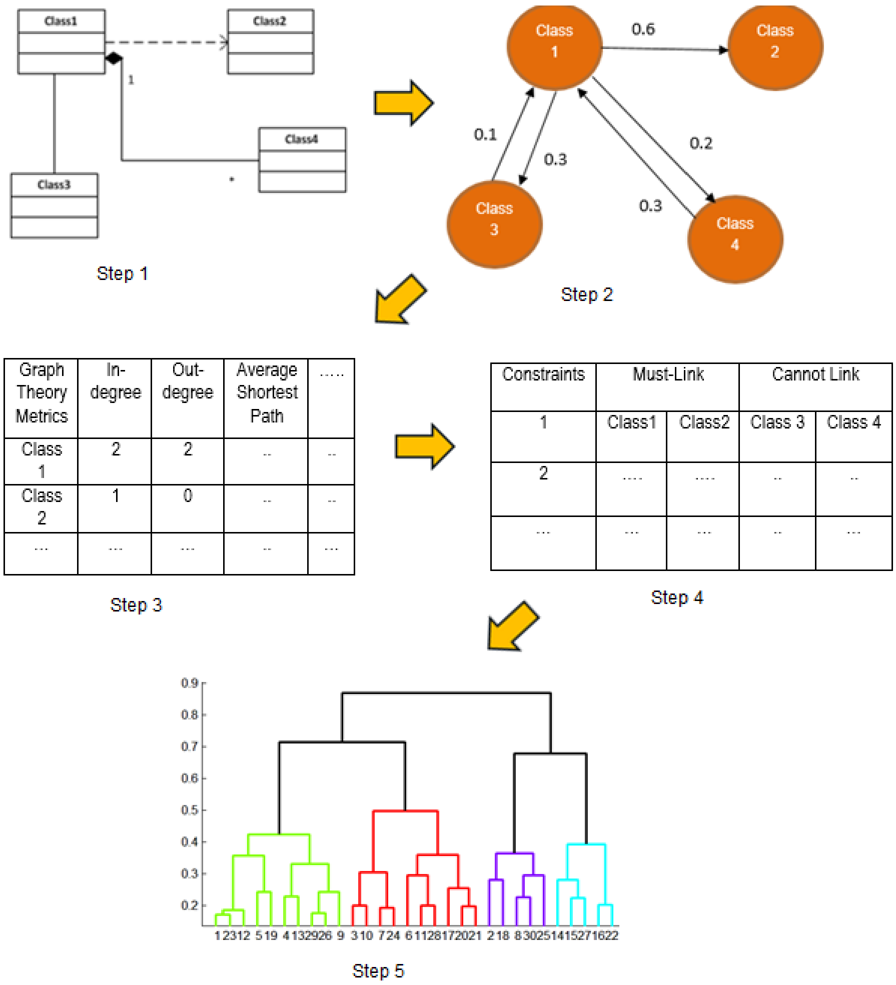

Figure 1: Steps involved to derive clustering constraints

Step 1 and Step 2 in Figure 1 show the steps involved to convert UML classes into a weighted complex network adopted from our earlier work (Chong & Lee, 2015a). An OO software system in the form of source code is first reverse engineered into UML class diagrams using an off-the-shelf round trip engineering tool. When representing software systems using complex network, UML class diagram is a better choice 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 modeling of UML class diagrams are standardised, it is easier to construct complex networks based on class diagrams. Based on the reverses engineered UML class diagrams, classes are converted into nodes while relationships between classes are converted into edges of a complex network. The nodes and edges are weighted using our formulated software metrics in Chong and Lee (2015a) that focus on the complexity of classes and their relationships. In this study, the nodes and edges are weighted based on two parameters, the complexity of classes and the complexity of relationships. Relationships (dependency, realization, 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, UML class relationships are ordered in ordinal scale, as shown in Table 2. In the work by (Dazhou, Baowen, Jianjiang, & Chu, 2004), the authors argue that since the complexities of different relationships are relative ~~with~~ to each other, arbitrary values of 1–10 can be assigned to H1-H10 respectively. Based on this ordinal scale, one 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 2.

Table 2: Ordering of relationships in UML class diagram 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	Generalization (concrete parent)	H7
8	Binding	H8
9	Generalization (abstract parent)	H9
10	Realize	H10

The way to rank UML relationships using an ordinal scale is also observed in the work by (Hu, Fang, Lu, Zhao, & Qin, 2012). The UML relationships are ranked in order to differentiate the importance of the associated classes based on their inherent characteristics and relationships with other classes. However, the work by Hu et al. only tackled 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 object-oriented 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 (generalization and realization), 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, we argue that the notion of ordering UML class relationship in an ordinal scale, and subsequently identifying the importance or complexity of classes, is feasible. Since the UML relationships are ranked according to an ordinal scale, it is more significant to identify the ranking of a relationship, *R*, in terms of its complexity, rather than quantifying weightage value.

In this paper, the way to calculate the weights of edges for a software-based weighted complex network is as follows. 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 2, one 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).

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

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,* which will be discussed later. H_R indicates the relative complexity of relationship R (by referring to Table 2).

In this paper, the complexity of relationship is calculated by assigning a relative weight in the range of [0, 1] to each relationship, *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.

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 to 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.

The complexity of the class, on the other hand, is measured using Weighted Methods per Class (WMC) and Lack of Cohesion of Methods (LCOM) to measure its complexity at the code level and the class level respectively (Chong & Lee, 2015a). The choice of software metrics used in this study is not random, as it is based on previous research (Basili, Briand, & Melo, 1996; Olague, Etzkorn, Gholston, & Quattlebaum, 2007; Subramanyam & Krishnan, 2003) that WMC and LCOM4 are complementary with each other when used to analyse the complexity of object-oriented software systems. 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 $W\left(D_j\right)$ respectively. The following equation is used to quantify the complexity of D_j.

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

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. In depth discussion on how to calculate the weight of a particular edge can be found in (Chong & Lee, 2015a).

In this research, given raw source code of a particular project, the steps to convert the software system into its respective weighted complex network are as follows:

1. Analyse the raw source code using WMC and LCOM4 metrics to measure the complexity of each of the classes. SonarQube (SonarQube, 2014) is used to perform the static code analysis.
2. Transform the raw source code into its respective UML class diagram using an off-the-shelf round trip engineering tool. In this research, Visual Paradigm is chosen to perform the transformation because it is capable of preserving the directionality of method calls using a build-in function called the Impact Analysis.
3. Analyse the complexity of UML relationships based on the ranking shown in Table 2.

4. Convert the UML class diagram into its respective weighted complex network using the method proposed in (Chong & Lee, 2015a).

Next, in Step 3, several graph theory metrics are applied onto the weighted complex network in order to perform graph theory analysis of the analysed software. Based on the results of the analysis, constraints are derived consisting of ML and CL constraints as shown in Step 4. Finally, based on the derived constraints, a dendrogram is generated and partitioned to form sets of cohesive clusters. The constraints derivation method in Step 3 and Step 4, which is the major contribution in this paper, is designed to be generic and applicable to any software-based complex network. In order to perform a comprehensive and insightful graph theory analysis of a software system, selection of appropriate graph theory metrics is important in understanding and analysing the structural and behavioural characteristics of the software system, ultimately deriving clustering constraints.

3.1 Selection of Graph Theory Metrics

In this research, 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 (G. Concas et al., 2007; Jenkins & Kirk, 2007; Valverde & Solé, 2003). The details of the metrics have been discussed in Section 2.4. Table 3 presents a summary of the selected graph theory metrics.

Table 3: Selected graph theory metrics and implication toward the analysed software systems

Graph theory metrics	**Software engineering point of view**
In-degree	Represent the usage of a particular class by other classes in the software. Demonstrate the level of reusability of a class.
Out-degree	Represent 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. Help 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.

3.2 Translating Graph Theory Analysis into Implicit Clustering Constraints

Apart from using the graph-level metrics to analyse and evaluate the software quality aspect of software systems, the ultimate goal of this study is to translate the results of graph theory analysis into implicit clustering constraints.

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 research, several community detection techniques that are commonly used in the field of brain network research are adopted to discover the community structure of software systems. Next, the findings are converted to clustering constraints in the form of ML or CL constraints to improve the results of software clustering.

3.2.1 Identifying Network Hubs

Figure 2 shows a snippet of weighted complex network constructed using our approach proposed in Chong and Lee (2015a), on an open-source software written in Java called the Apache Gora.

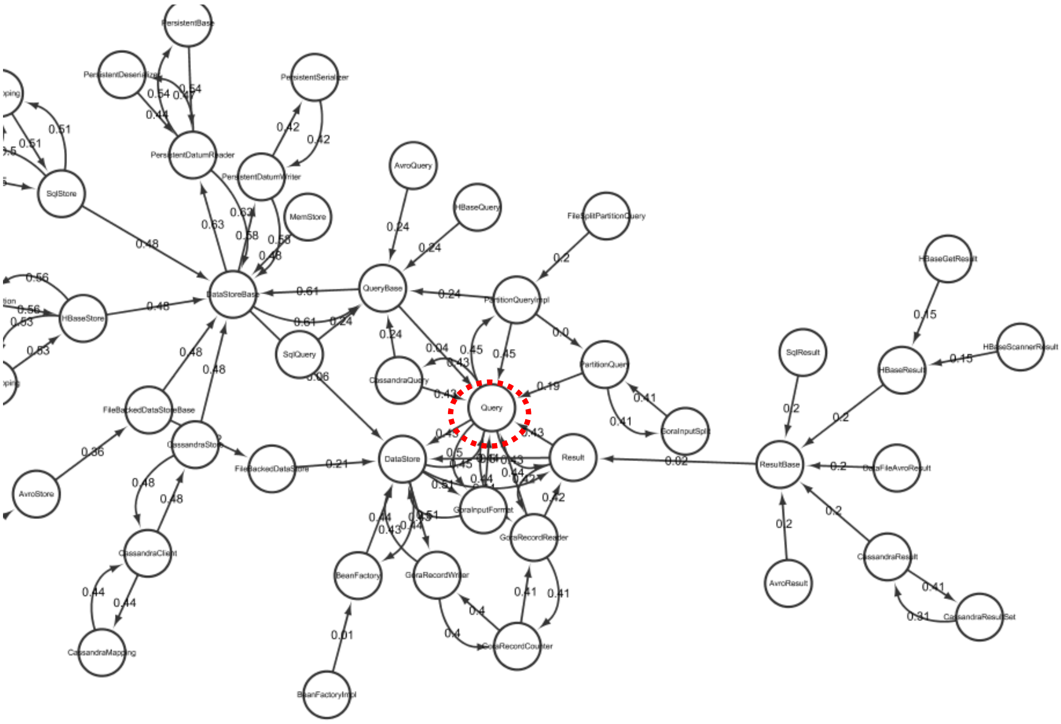

Figure 2: Snippet of Apache Gora project represented in weighted complex network using our approach proposed in Chong and Lee (2015a)

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 in Figure 2 possesses high in-degree because a lot of other nodes are converging and directed toward this particular node. In graph theory, 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 a 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 classes 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 3 shows an example of the in-degree distribution of Apache Gora 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 3, 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.

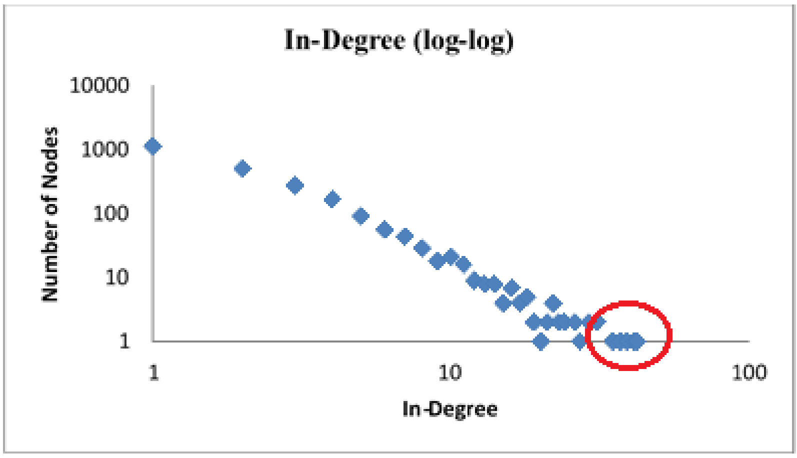

Figure 3: Identify hubs by observing the degree distribution of in-degree

However, it is possible that the identified nodes (classes) with high in-degree might actually be god classes or utility classes. God classes are associated by a very large 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 theory's point of view (G. 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 study, 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?

3.2.2 Identifying 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 study are considered core functional classes that contain the methods and information of a particular software feature. It is common for other classes to invoke methods or parse parameters to and from 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 study, 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 classes responsible for a particular software functionality. The enforcement and fulfilment of clustering constraints are discussed in Section 3.3.

3.2.3 Identifying Must-Link between Hubs and Direct Neighbours

In graph theory, the clustering coefficient of a node is the average tendency of pairs of neighbours of the 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 hub and clustering coefficient, software maintainers can identify the neighbouring classes that are closely related to the hub. Neighbouring classes that form a complete clique with the 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.

3.2.4 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 3.2.3. 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 the 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.

Table 4: 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

3.3 Fulfilment of Clustering Constraints

As mentioned earlier, fulfilment of ML and CL constraints by a hierarchical clustering algorithm is not straightforward because we need to ensure that all the constraints are fulfilled at each level of cluster hierarchy. Hence, an approach to fulfil clustering constraints derived from the graph theory analysis in Section 3.2 is discussed.

In this study, we adopt the technique proposed by Miyamoto (2012) to fulfil ML constraints. All ML constraints are fulfilled by forcing the associated cluster entities 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 ML constraint to zero.

> Given a set of entities $T = \{x_1, x_2, \cdots, x_n\}$ with entities $x_1, x_2, \cdots, x_n$.
>
> For every $(x_i, x_j) \in \{ML\}$, the distance between x_i and x_j, $d(x_i, x_j)$, 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 ML 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. On the other hand, the CL constraints are fulfilled by changing the distance between pair of entities, commonly referred as distance based approach (Malliaros & Vazirgiannis, 2013). Distance based approach modify the distance between a pair of entities linked by a CL constraint to be a value high enough to prevent them from merging.

> Given a set of entities $T = \{x_1, x_2, \cdots, x_n\}$ with entities $x_1, x_2, \cdots, x_n$.
>
> For every $(x_i, x_j) \in \{CL\}$, $d(x_i, x_j) = d(x_i, x_j) + Const$
>
> 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 CL 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 CL constraints will then be merged at the top of the hierarchy to form the complete dendrogram. An example is illustrated in Figure 4, where the circle at the top of dendrogram indicates the merging of classes linked by CL constraints. By observing Figure 4 from another perspective, some CL constraints are actually violated at the top of the hierarchy since without violating them, "dead-end" situation will occur. However, violating CL constraints at the top of the hierarchy are negligible because it is almost impossible to cut the dendrogram at that location (Chong & Lee, 2015b; Lung, Zaman, & Nandi, 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 study to enforce the CL constraints.

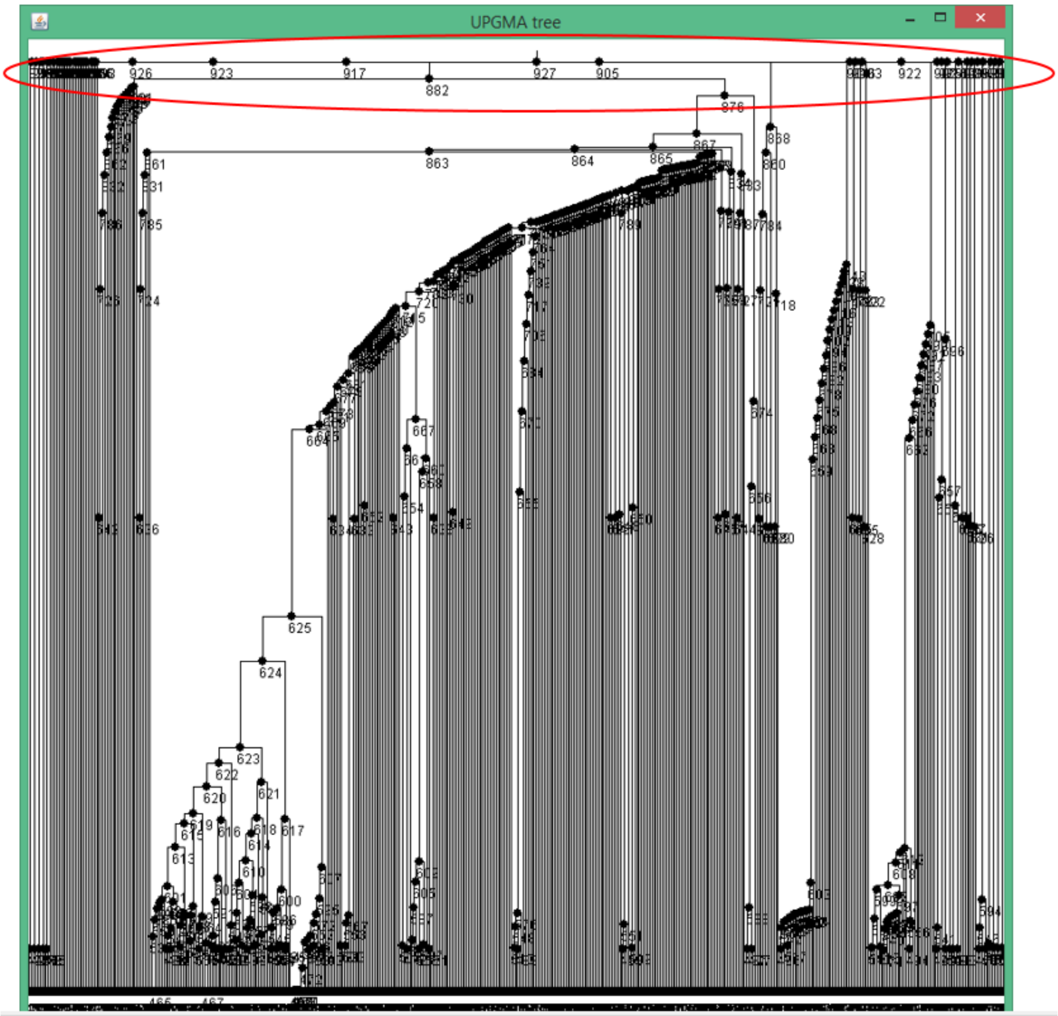

Figure 4: Example of imposing CL constraints by modifying the distance measure between pairs of entities

However, changing the distance measure of entities involved in ML and CL 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. (Klein et al., 2002). Violating the triangle inequality of resemblance matrix means that for some classes $(x_s, x_t) \notin \{ML\}, (x_s, x_t) \notin \{CL\}$ with distance $d(x_s, x_t)$ apart before imposing ML or CL constraints, may now be $d'(x_s, x_t) < d(x_s, x_t)$ along some path which skips through the ML or CL 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 ML or CL constraints using all-pairs-shortest-path algorithm. The algorithm will search for the shortest path between all pairs of classes after the enforcement of ML and CL 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 5a 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.

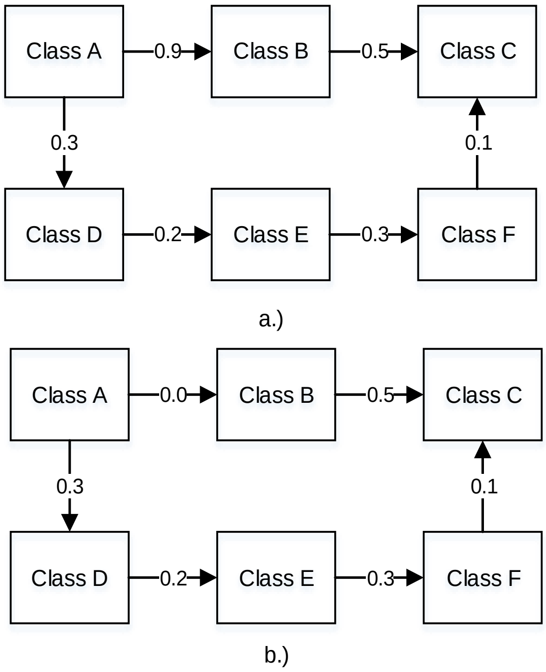

Figure 5: Potential triangle inequality problem when imposing ML and CL constraints

Given that a new ML constraint involving Class A and Class B was derived based on the graph theory analysis. Thus, the distance between A and B is now change to 0.0 in order to reflect the ML constraint, as illustrated in Figure 5b. In this case, the shortest path between Class A and Class C after the imposition of the ML 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 constrained clustering method addresses this violation in fulfilling both ML and CL constraints using the following algorithm:

Input: A set of entities S = $\{x_1, x_2, \cdots, x_n\}$, a set of ML (must-link constraints) and a set of CL (cannot-link 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 ¬ $[(\forall(x_p, x_q) \in \{ML\}) \cap (\forall(x_r, x_s) \in \{CL\}) > 0]$
   i. for every $(x_i, x_j) \in \{ML\}$, 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 $(x_i, x_j) \in \{CL\}$, 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.

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 (Chong & Lee, 2015a), the software is represented in a weighted complex network.

iii. Graph theory metrics are applied onto the transformed weighted complex network. ML and CL constraints are then derived based on the proposed approach with the aid of graph theory analysis.

iv. The derived ML and CL constraints are then supplied as the side information to perform the constrained hierarchical clustering to recover a high-level abstraction of the software design. If there is a pair of classes (x, y), such that (x, y) belongs to both ML and CL, 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.

v. The dendrogram is cut based on the available clustering constraints. The cutting point that can fulfil the most clustering constraints and produce the best clusters with strong intra-cluster cohesion and inter-cluster separation is preferred and chosen as the optimum cutting point.

4. Experimental Design and Execution

This research follows an empirical research methodology where the proposed approach is validated using real-world OO software systems. 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 classic unconstrained approaches. The details of the implementation are discussed in the following subsections.

4.1 Selection of Subjects

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.

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 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.

In this work, the Software Quality Assessment Based on Life-cycle Expectations (SQALE) rating is used to measure the quality of open-source software (Izurieta, Griffith, Reimanis, & Luhr, 2013) (Letouzey & Ilkiewicz, 2012). The SQALE rating provides a systematic model to estimate technical debt, and subsequently ranks the severity of debts using five scales, ranging from A to E. Hence, the inclusion of SQALE method as a basis of measuring the maintenance costs of the selected software systems will allow for better comparative analysis.

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 tool (SonarQube, 2014), 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 extracted from Table 5.

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 5: 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

4.2 Evaluation of Experiments

According to Anquetil and Lethbridge (1999), 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 or reference decomposition (Zhihua & Tzerpos, 2004). Gold standard or reference decomposition 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 constrained clustering approach are compared against prior studies related to software clustering and also the gold standard. Generally, a gold standard or reference decomposition can be created manually by domain experts, or by using the current, factual architecture of the system created by the developers (e.g., the package structure of an OO system) (Bavota et al., 2013; F. Beck, Melcher, & Weiskopf, 2016; Fabian Beck & Diehl, 2013). However, engaging domain experts to provide a reference decomposition is relatively more expensive, in terms of time and effort, than retrieving the factual architecture from the source code because the latter can be automated. Hence, in this research, the gold standard is retrieved automatically from the package structure of all the selected test subjects.

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

1. Perform the classic software clustering approach that does not make use of any clustering constraints, or commonly referred as unconstrained clustering approach.

2. Perform constrained clustering using the proposed approach by incorporating ML and CL constraints derived automatically from the implicit structure of the software system.

3. Retrieve the package structure of the analysed open-source software from the project website or repository. The package structure 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 approach and the documented artifact.

4. Use MoJoFM to calculate the similarities between all three decompositions (clustering results from the unconstrained clustering approach, clustering results from the proposed constrained clustering approach, and the package structure of the test subject).

4.3 Experiment Execution

Experiments were carried out based on the design and setup discussed in the previous subsections. First, the resemblance matrix of each project is constructed based on Dijkstra's shortest path algorithm (Dijkstra, 1976). One resemblance matrix is created for each project. Shortest path algorithm 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 paper, 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. By using Dijkstra's shortest path algorithm, software maintainers can identify how closely related two classes are, and provide a means to indicate whether they belong to the same functional group. Next, based on the resemblance matrix produced, merging of entities will then take place. Un-weighted Pair-Group Method using Arithmetic Average (UPGMA) will be used to merge clusters and form a dendrogram (Gronau & Moran, 2007; Lung & Zhou, 2008). UPGMA defines the similarity measure between two clusters as the arithmetic average of similarity strengths among all pairs of entities in the two clusters. UPGMA is more suitable in this work because it is less sensitive toward the effect of outlier as compared to other clustering algorithms such as Single-Linkage Algorithm (SLINK) and Complete-Linkage Algorithm (CLINK) (Gronau & Moran, 2007; Lung & Zhou, 2008).

In general, conventional software clustering involves the following five 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

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 configurations (validity index used, clustering algorithm used, similarity measure used, etc.) used by the proposed constrained clustering approach and the baseline unconstrained clustering approach must be identical. In this research, the configurations of the constrained clustering approach and unconstrained clustering approach are as follows:

1. Identification of entities or components - Classes
2. Identification of features – UML relationships (associations, composition, aggregation, etc)
3. Calculation of similarity measure - complexity of UML relationships (edges) and the complexity of classes (nodes)
4. Application of clustering algorithm – UPGMA
5. Evaluation of clustering results - MoJoFM

The only difference between the proposed constrained clustering approach and the unconstrained clustering approach is that the latter does not make use of any clustering constraints.

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 paper. 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/

4.3.1 Derivation of Clustering Constraints

Due to size and page constraints, all the clustering constraints derived from the 40 test subjects are presented in Table A1 in Appendix. Some examples of Table A1 are illustrated in Table 6, which shows the clustering constraints derived from Apache Gora, openFAST, and Apache Tika.

The second column in Table 6 and Table A1 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 with the class with high BC.

For hubs and high BC classes in C-rated software, fewer cliques can be identified, resulting in a less number of constraints derived from these test subjects. The main reason behind this observation is due to the existence of god classes in software with higher level of maintenance efforts. God classes in the context of software engineering refer to classes that are associated by a huge number of simple data container classes and contain many instance variables, which 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.

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 A1, 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 their overall maintainability.

The results in Table 6 and Table A1 show that graph theory analysis is able to automatically derive clustering constraints from the implicit structure of 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 lists down the number of clustering constraints derived from each test subject, sorted according to SQALE rating.

Table 6: 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		

Table 7: 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 project. 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 the 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 ML and CL constraints using the distance based approach discussed in Section 3.3.

4.3.2 Applying the Derived Constraints to Agglomerative Hierarchical Clustering

Now that implicit clustering constraints are derived, the next step is to generate a dendrogram for each of the associated test subjects. Since all the ML constraints are unconditionally fulfilled at the bottom of the dendrogram, software maintainers do not have to worry about the fulfilment of these constraints. Due to the size and scale of the experiment, one example is chosen and shown in Figure 6, where it depicts the dendrogram generated from Apache JSPWiki project.

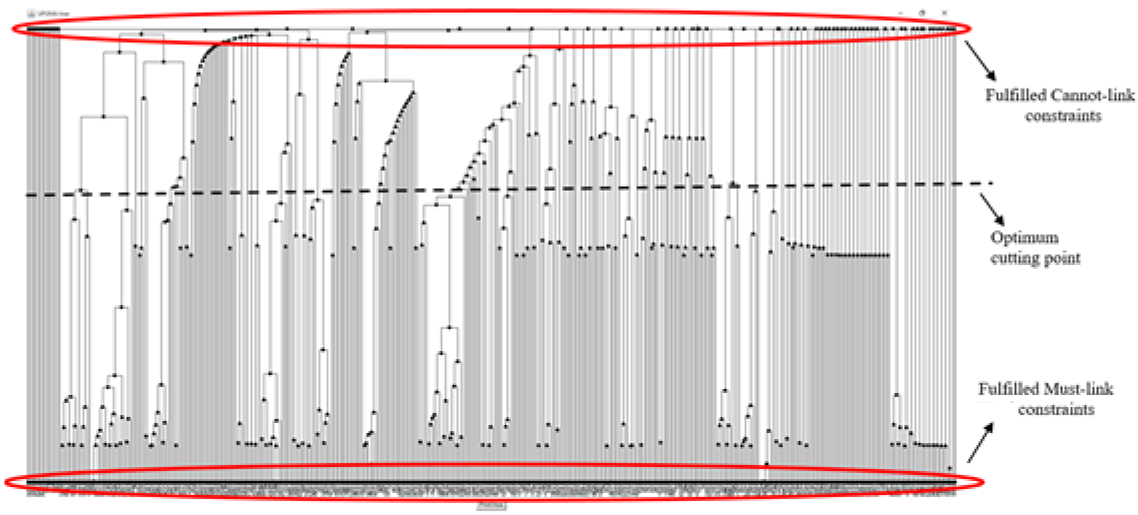

Figure 6: Dendrogram generated from Apache JSPWiki project

The circle at the bottom of the dendrogram shows the pairs of ML constraints that form the base of the dendrogram. On the other hand, the circle at top of the dendrogram shows the CL constraints. Since it is impossible to cut the dendrogram at the top of the dendrogram, one can be assured that CL constraints are fulfilled regardless of any condition. After generating the dendrogram, it is cut at a particular point to form groups of cohesive clusters. In this case, we have adopted the dendrogram cutting point technique proposed in (Chong et al., 2013).

4.3.3 Evaluating Clustering Results using MoJoFM

Next, the clustering results are compared against the original package structure of the test subjects using the MoJoFM metrics to verify if incorporating software clustering with ML and CL constraints derived from the proposed approach can help produce better clustering results compared to the classic unconstrained approach. Comparison was made between the proposed constrained clustering approach and the classic 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.

Table 8 shows the MoJoFM metric value for all the 40 test subjects. The third column shows the MoJoFM values of the classic unconstrained clustering approach when compared against the original package structure of the test subjects. The fourth column shows the MoJoFM values of the proposed constrained clustering approach when compared against the package structure of the test subjects.

Table 8: MoJoFM values for constrained and unconstrained clustering results when compared to the original package structure of the test subjects

Project	Number of derived constraints	Unconstrained clustering approach (MoJoFM value)	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 8, it can be summarised that by integrating clustering constraints derived from the proposed approach, the clustering results achieve an aggregated average of 80.1% accuracy when compared against the original package structure of the forty software systems, and perform better than the classic unconstrained clustering approach. It has to be noted that the original package structure 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 constrained clustering approach and the unconstrained one.

5. Discussion

In general, test subjects with more clustering constraints extracted using the proposed approach 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 projects, which record less than 10% improvement. This is mainly because several pairs of classes involved in the must-link or cannot-link constraints have 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.

The proposed method to automatically derive clustering constraints from software systems involve several automated steps which will definitely occur some overhead during the process. When running on a PC with Intel i7-4770 CPU and 8GB of RAM, the constraint derivation process took about 2-3 minutes per project, which is negligible from our perspective. As discussed earlier, the input of the proposed constrained clustering approach is in the form of raw source code of object-oriented software systems. In order to perform either constrained (supervised) or unconstrained (unsupervised) clustering methods, the raw data need to be pre-processed depending on the clustering algorithm used, similarity measures, level of granularity, etc. Based on our experiment findings, the average time taken to finish the whole clustering process is about 30 minutes per project. This mean that the constraint derivation process adds less than 10% overhead, which is negligible when compared to the manual effort of acquiring clustering constraints from a domain expert, under the assumption that such expertise is available. Hence, in our opinion, the overhead is negligible when evaluated against the degree of improvement in terms of MoJoFM metric.

In summary, the results presented in Table 8 are capable of providing a concrete answer toward addressing the research goal, such that integrating constraints derived from the proposed approach is able to produce highly cohesive clusters when measured using the MoJoFM metric. 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 to software maintainers in situations where they do not have a reliable point of reference and domain experts are non-existent.

6. Threats to Validity

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 ~~the selection of subjects~~confounding variables.

The selection of graph theory metrics used in this study to derive clustering constraints used might impose the threat of confounding variables. The chosen metrics are in-degree, out-degree, average weighted degree, average shortest path of nodes, average clustering coefficient, and betweenness centrality. The chosen graph theory metrics are selected based on their interpretation toward the behavior of object-oriented software systems. Several existing studies have shown that the chosen metrics they are correlated to software qualities, and can be effective to the maintainability and reliability of software systems (G. Concas et al., 2007; Jenkins & Kirk, 2007; Valverde & Solé, 2003). The details of the metrics have been discussed in Section 2.4, supported with a summary presented in Table 3.

The external validity threats are examined with respect to sampling bias and pre-test assumption. With respect to the threat from ~~subject selection~~sampling bias, 40 open-source software systems that vary according to their size, class-count and maintenance effort are selected in this paper to reflect some representative distribution on the population of open-source OO software systems. 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, SQALE rating is used to provide a means to estimate the maintainability of the selected test subjects.

~~The external validity threats are concerned with the~~The external validity threat to pre-test assumption is concern with the~~of~~ ~~removing~~ removal of~~the~~ isolated and utility classes before performing software clustering, which might result in a biased outcome. In this paper, utility classes and isolated classes (classes that do not have any relationships with other classes) are removed prior to the clustering process to avoid biasness in clustering results. 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 can result in ambiguity in the organization of a software system. The work by Patel et al. (2009) also makes a pre-test assumption by removing all of the utility classes before the initiation of a clustering process.

7. Conclusion and Future Work

This paper presents a method to integrate the concept of graph theory analysis to automatically derive clustering constraints from the implicit structure of software systems. Existing studies in constrained clustering often assumed that feedback from domain experts are always reliable and accessible prior to the clustering process, which is unrealistic in software development especially when dealing with poorly designed and poorly documented software systems. The proposed method has been successfully implemented on 40 open-source object-oriented software systems. We managed to derive a number of clustering constraints without feedback from domain experts to help facilitate in the subsequent clustering process. When compared against the classic unconstrained clustering approach, our proposed method managed to achieve better results measured using MoJoFM metric. It is to be noted that the proposed constraints derivation approach can be applied to both partitional and hierarchical clustering algorithms, as long as the software can be analysed using graph theory metrics in conjunction with complex network.

We believe that there are several directions in which the outcome of this research can be extended and improved. For instance, further work to correlate the graph theory metrics with a more direct measurement of structural behaviour, for instance, by measuring changes and issues of software in multiple releases can be considered when deriving clustering constraints from the source code. Measuring the frequency of changes between different releases of software systems can be a reliable way to measure the structural stability of software systems, such that the more changes that are required to address a bug, the greater the maintenance effort, and higher likelihood that the associated software component should be grouped into the same cluster to avoid propagation of bugs.

8. Acknowledgment

This work is carried out within the framework of the research project supported by High Impact Research Grant with reference UM.C/625/1/HIR/MOHE/FCSIT/13, Fundamental Research Grant Scheme with reference FP001-2016, funded by the Ministry of Higher Education, Malaysia, and School of IT Collaborative Research Grant with reference 5140871-112-00, funded by Monash University Malaysia.

Appendix

Table A1: Summary of clustering constraints derived from all the 40 test subjects

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	-
Apache Maven Wagon	AbstractWagon	ScmWagon ProxyInfo ProxyInfoProvider AuthenticationInfo	SshServerEmbedded	-
	AbstractJschWagon	SftpWagon ScpWagon	Resource	-
	SshServerEmbedded	TestPasswordAuthenticator TestPublickeyAuthenticator AbstractEmbeddedScpWagonTest AbstractEmbeddedScpWagonWithKeyTest		
Apache Synapse	SynapseXPath	DefaultInMemorySubscriptionManager	FIXSessionFactory	FIXTransportSender FIXOutgoingMessageHandler
	SynapseConfiguration	SynapseArtifactDeploymentStore	Protocol	PipeEndpoint
	AbstractMediatorFactory	AggregateMediatorFactory AnnotatedCommandMediatorFactory BeanMediatorFactory CacheMediatorFactory CalloutMediatorFactory ClassMediatorFactory CloneMediatorFactory	PipeEndpoint	Protocol
	Endpoint	IndirectEndpoint TemplateEndpoint FailoverEndpoint SessionInformation	AMQPTransportReconnectHandler	AMQPTransportPollingTask AMQPTransportConnectionFactoryManager AMQPTransportListener AMQPTransportEndpoint AMQPTransportHAEntry
	MessageContext	GetPropertyFunction SynapseXPathVariableContext SynapseXPathFunctionContext AsyncCallback	SynapseCallbackStoreView	-
	Value	-	SecretCallback	-
	ServerHandler	-	ClassVisitor	-
	SynapseEnvironment	-		
IWebMvc	DAO	-	Filter	-
	I18nText	-	DAO	-
	SecurityDAO	SecurityDAOImpl		
JEuclid	AbstractJEuclidElement	ForeignElement Mfrac Mspace	MathVariant	FontFamily CodePointAndVariant
	JEuclidView	Graphics2DImagePainterMathML LayoutView	TypeWrapper	-

	JMathComponent	ParametersDialog		
	TypeWrapper	Parameter		
Jajuk	JajukAction	-	ObservationManager	-
	File	RefactorAction DeleteSelectionAction PhysicalItem Playlist Directory CoverView	JajukEvent	-
	SelectionAction	-	AbstractAnimation	-
	Observer	-		
	Const	-		
	ViewAdapter	-		
	JajukFileFilter	-		
Apache Mahout	Vector	CachingCVB0PerplexityMapper TopicModel	ResultAnalyzer	-
	Matrix	-	PathType	-
	AbstractJDBCDataModel	-	ARFFModel	-
	AbstractFactorizer	GenericDataModel ALSWRFactorizer ParallelSGDFactorizer	RegexTransformer	-
	DistanceMeasureCluster	InteractionValueEncoder PathParameter	Vector	CachingCVB0PerplexityMapper TopicModel
	DataModel	RatingSGDFactorizer		
	VectorWritable	-		
	RefreshHelper	-		
Fitnesse	WikiPage	SymbolicPage MovedPageReferenceRenamer PageReferenceRenamer TestPageWithSuiteSetUpAndTearDown	Game	-
	FitNesseContext	JunitReFormatter SuiteXmlReformatter	CommandLine	-
	SymbolType	HeaderLine HorizontalRule LastModified PlainTextTable Preformat Image See Literal	PluginsLoader	ComponentFactory PluginFeatureFactory PluginFeatureFactoryBase PropertyBasedPluginFeatureFactory
	Fixture	Counts FitServer		
	Symbol	-		
Apache Shindig	FeatureRegistry	FeatureResourceLoader FeatureFileSystem	GlobalId	-
	JsonDbOpensocialService	-	GadgetAdminData	-
Apache XBean	LogFacade	-	XBeanNamespaceHandler	-
	Filter	-	LogFacade	-
	AbstractConverter	-	Repository	-
	Option	-	BundleClassFinder	-
	Archive	-		

Apache Commons VFS	Capability	WebdavFileProvider JarFileProvider	Cryptor	-
	FileObject	AbstractFileObject LocalFile UrlFileObject		
	DefaultFileSystemManager	-		
	AbstractFileSystem	FileSystemKey LocalFileSystem UrlFileSystem		
	FileType	-		
Apache Tobago	LabelExtensionTag	TobagoExtensionBodyTagSupport OutExtensionTag SelectManyShuttleExtensionTag SelectManyListboxExtensionTag DateExtensionTag FileExtensionTag InExtensionTag	AutoSuggestItem	-
	HasMarkup	-	LabelExtensionTag	TobagoExtensionBodyTagSupport OutExtensionTag SelectManyShuttleExtensionTag SelectManyListboxExtensionTag DateExtensionTag FileExtensionTag InExtensionTag
	HasIdBindingAndRendered	-	TobagoContext	-
	Measure	IntervalList BankHead		
	ClientPropertiesKey	-		
	TobagoConfigImpl	-		
	IsGridLayoutComponent	-		
Apache Karaf	FeaturesService	FeaturesPlugin FeatureDeploymentListener	FilterType	-
	AbstractAction	TacAction CatAction SourceAction SortAction SleepAction PrintfAction NewAction MoreAction	Col	-
	Completer	MyCompleter FileCompleter CommandsCompleter CommandNamesCompleter ActionMetaData CompleterAsCompletor NullCompleter	ServerInfoImpl	-
	OsgiCommandSupport	SshAction Wait Install BundleContextAware MyCommand	RegionsPersistence	-

		ListServices SshServerAction		
	JaasCommandSupport	RoleDeleteCommand RoleAddCommand UserDeleteCommand UserAddCommand ListRealmsCommand ListPendingCommand ManageRealmCommand ListUsersCommand	BundleInfoImpl	-
	EncryptionSupport	-	WikiVisitor	-
	Feature	FeatureEvent State	DefaultActionPreparator	-
	InstanceCommandSupport	StartCommand RenameCommand StopCommand ChangeOptsCommand ChangeRmiRegistryPortCommand ChangeRmiServerPortCommand ChangeSshPortCommand		
Apache EmpireDB	ErrorType	InvalidKeyException InvalidArgumentException ItemExistsException InvalidPropertyException InternalException StatementFailedException QueryFailedException UnexpectedReturnValueException BeanPropertySetException BeanPropertyGetException	DefaultHtmlTagDictionary	-
	DBColumnExpr	-	Column	-
	DBDatabase	DBView OracleSYSDatabase	ErrorType	H2DDLGenerator HSqlDDLGenerator MSSqlDDLGenerator MySQLDDLGenerator OracleDDLGenerator PostgreDDLGenerator SQLiteDDLGenerator DerbyDDLGenerator
	DBDDLGenerator	H2DDLGenerator HSqlDDLGenerator MSSqlDDLGenerator MySQLDDLGenerator OracleDDLGenerator PostgreDDLGenerator SQLiteDDLGenerator DerbyDDLGenerator	FieldValueException	-
	TagEncodingHelper	ControlTag InputTag TitleTag LabelTag RecordTag		

		ValueTag		
	DBRowSet	-		
Apache Log4J	StatusLogger	ClockFactory TagUtils Configurator MessageAttributeConverter	Category	-
	BaseConfiguration	NullConfiguration DefaultConfiguration XMLConfiguration JSONConfiguration PluginManager	Action	-
	Marker	MarkerWrapper	AuditEvent	-
	PatternFormatter	RegexReplacementConverter StyleConverter HighlightConverter AbstractStyleNameConverter	StatusLogger	ClockFactory TagUtils Configurator MessageAttributeConverter
	RollingFileManager	SizeBasedTriggeringPolicy OnStartupTriggeringPolicy TimeBasedTriggeringPolicy FileManager RollingRandomAccessFileManager	StrSubstitutor	-
	Message	ObjectMessage MessageFormatMessage SimpleMessage ParameterizedMessage ThreadDumpMessage MultiformatMessage StringFormattedMessage		
Eclipse SWTBot	SWTWorkbenchBot	SWTBotViewTest ProjectCreationWizardTest SWTBotProject CommandFinderTest SWTBotPerspective WorkbenchContentsFinder	BidiMap	-
	GraphicsBackground	-	KeyboardLayout	-
	SWTBot	SWTBotDemo	GraphicsBackground	-
	Tab	CanvasTab CComboTab ComboTab CoolBarTab CTabFolderTab DateTimeTab DialogTab ExpandBarTab FillLayoutTab FormLayoutTab	PaintSurface	-
	ToolSettings	-		
	SWTBotText	-		
Apache Deltaspike	Deactivatable	ViewScopedExtension PartialBeanBindingExtension SecurityAwareViewHandler TransactionContextExtension	Car	-

		ViewControllerActionListener ViewConfigPathValidator DeltaSpikeNavigationHandler DeltaSpikeLifecycleFactoryWrapper		
	CdiQueryInvocationContext	-	EntityDescriptor	-
	ClientWindow	DisableClientWindowHtmlRenderer ClientWindowAdapter	ExecutableCallbackDescriptor	-
	MessageContext	-	AbstractBeanStorage	-
	ViewConfigResolver	DefaultViewConfigResolver ViewRootAccessHandler	TestStatementDecoratorFactory	
	SingleValueBuilder	Eq LessThanOrEqual Between Like LessThan NotLike GreaterThan NotEq GreaterThanOrEqual	ContextControl	WeldContainerControl OpenWebBeansContainerControl OpenEjbContainerControl
	WindowContext	-		
	ContextControl	OpenWebBeansContainerControl WeldContainerControl OpenEjbContainerControl OpenWebBeansContextControl		
JFreeChart	XYItemRenderer	-	CrosshairOverlay	-
	Plot	-	CrosshairLabelGenerator	-
	XYDataset	-	ChartEditorFactory	-
	Range	-		
Titan	StandardTitanTx	-	KeyIterator	-
	KeyColumnValueStore	-	RecordIterator	-
	StandardTitanGraph	RelationQueryCache VertexIterable	StandardTitanTx	-
	StaticBuffer	KeyRangeQuery KeySliceQuery	AbstractGenerator	-
Jackcess	TableImpl	ErrorHandler FKEnforcer CursorBuilder PropertyMaps PropertyMap TempPageHolder	JackcessException	-
	DatabaseImpl	JetFormat	TableImpl	ErrorHandler FKEnforcer CursorBuilder PropertyMaps PropertyMap TempPageHolder
	ColumnImpl	LongValueColumnImpl	ColumnImpl	LongValueColumnImpl
Apache Pluto	PortletAppType	ContainerRuntimeOptionType PortletApplicationDefinition	FileSystemInstaller	-
	PortalDriverServicesImpl	-	RequestDispatcherService	-
	PortletType	-	PortletContextImpl	-
	DescriptionType	-		

	DefaultOptionalContainerServices	-		
Apache Roller	Weblog	FileContent CommentSearchCriteria Setup	MenuTab	-
	WeblogEntry	EntriesPager	ParsedTab	-
	Weblogger	-	LiteDevice	-
	URLStrategy	-	RendererFactory	-
	JPAPersistenceStrategy	-	RequestMapper	-
			DeviceResolver	-
jOOQ	Field	TimestampDiff DateDiff Right Round Trunc Left Nvi Space Ln DateOrTime	JSONParser	-
	Clause	CustomField QualifiedTable CustomCondition FalseCondition TrueCondition	UNumber	-
	TableField	-		
	Table	TableOnConditionStep		
	JooqLogger	PostgresDatabase		
	AbstractDatabase	-		
	DataType	ArrayDataType		
Apache Sirona	Role	Unit	Role	Unit
	Cube	AsyncHttpClientCubeBuilder HttpClientCubeBuilder	BoomerangServlet	-
	Collector	-		
	CassandraSirona	-		
Apache Hudson	ExtensionPoint	-	VersionSupport	-
	Descriptor	-	DelegatingOutputStream	-
	Describable	-	RequestRootPathProvider	-
	Hudson	LocalPluginManager	CliEntryPoint	-
	Saveable	PersistedListEqualsHashCodeTest	ClassResult	-
	Action	-		
	AbstractDescribableImpl	-		
JSPWiki	WikiEngine	SecurityVerifier Installer PageSorter InputValidator TemplateManager PluginBean CoreBean AclManager	WikiEngine	SecurityVerifier Installer PageSorter InputValidator TemplateManager PluginBean CoreBean AclManager
	WikiContext	-	WikiContext	InputValidator
	WikiTagBase	-	XHtmlElementToWikiTranslator	-

	Command	PageCommand GroupCommand WikiCommand RedirectCommand		
	WikiPage	-		
Apache Wink	Prop	-	Parser	-
	DeploymentConfiguration	-	Propertybehavior	-
	ResourceRegistry	-	Prop	-
	ProvidersRegistry	-	RssChannel	RssCategory
	RssChannel	RssCategory	JavaType	-
Apache Commons Collections	Predicate	-	ReplacementsFinder	-
	Transformer	-	EditCommand	-
	Closure	-		
	IteratorUtils	-		
	Factory	-		
Apache Commons BCEL	Attribute	-		
	ConstantPool	-	Subroutines	-
	ArithmeticInstruction	-	ControlFlowGraph	-
	JavaClass	-	VerifierFactoryObserver	-
	InstructionHandle	InstructionFinder BranchHandle		
	Instruction	-		
	MethodGen	-		
Apache Rampart	SupportingToken	AlgorithmWrapper	RahasData	-
	STSClient	-		
	Binding	-		
	AbstractSecurityAssertion	-		
Kyro	Serializer	-	CachedFieldFactory	-
	FieldSerializer	FieldSerializerUnsafeUtilImpl Generics FieldSerializerGenericsUtil FieldSerializerUnsafeUtil	Serializer	-
	Kryo	ClassResolver ListReferenceResolver	Kryo	ClassResolver ListReferenceResolver
	UnsafeCachedField	-		
Apache Abdera	FOMFactory	-	FOMFactory	-
	ServerConfiguration	Configuration	ElementSerializer	-
	DefaultProvider	RouteManager		
ApacheDS	DirectoryService	ReplicaEventLogJanitor	AvlNode	-
	LdapServer	ExtendedRequestHandler ExtendedResponseHandler	Marshaller	-
	DefaultDirectoryService	-	PasswordPolicyConfiguration	-
	IndexEntry	-	KeyIntegrityChecker	-
	Store	CursorBuilder	NtpService	-
Apache Archiva	ArchivaConfiguration		TreeEntry	-
	ArchivaDavResourceFactory	-	AbstractTransactionEvent	-
	ManagedRepositoryAdmin	-	Artifact	-
	DefaultRepositoriesService	-		
	RepositoryContentFactory	-		
	FileTypes	-		

Apache Helix	HelixManager	-	ConstraintItemBuilder	-
	ZkClient	TaskCluster	WorkflowConfig	-
	ZkHelixParticipant	-		
	ZkHelixController	-		
Struts	Logger	JSONPopulator SecurityMemberAccess FreemarkerDecoratorServlet JSONCleaner StrutsConfigRetriever IteratorGenerator DateTimePicker RestActionMapper Restful2ActionMapper	PageContextImpl	-
	StrutsModels	-	ELParser	SimpleCharStream ELParserConstants
	ObjectFactory	CdiObjectFactory DefaultActionFactory ActionFactory InterceptorFactory ValidatorFactory	ReferenceMap	-
	Container	XWorkBasicConverter	ExpressionBuilder	-
	ActionSupport	-	Logger	JSONPopulator SecurityMemberAccess FreemarkerDecoratorServlet JSONCleaner StrutsConfigRetriever IteratorGenerator DateTimePicker RestActionMapper Restful2ActionMapper
	Configuration	-		
	JspCompilationContext	ServletWriter		
Apache Falcon	ConfigurationStore	-	ServiceInitializer	-
	AbstractEntityManager	-	ChainableMonitoringPlugin	-
	AbstractWorkflowEngine	-	RerunEvent	-
Apache Mina	AttributeKey	-	AbstractMessageEncoder	-
	IoSession	-	DefaultHttpResponse	-
	IoBuffer	-	HttpRequestImpl	-
	AbstractIoSession	-		
	ProxyIoSession	ProxyLogicHandler		
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		

References

Abreu, Fernando Brito, & Carapuça, Rogério. (1994). *Object-oriented software engineering: Measuring and controlling the development process.* Paper presented at the proceedings of the 4th International Conference on Software Quality.

Anquetil, Nicolas, & Lethbridge, Timothy C. (1999). Recovering software architecture from the names of source files. *Journal of Software Maintenance, 11*(3), 201-221.

Bair, Eric. (2013). Semi-supervised clustering methods. *Wiley Interdisciplinary Reviews: Computational Statistics, 5*(5), 349-361. doi: 10.1002/wics.1270

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on, 22*(10), 751-761. doi: 10.1109/32.544352

Basu, Sugato, Banerjee, Arindam, & Mooney, Raymond J. (2004). *Active Semi-Supervision for Pairwise Constrained Clustering.* Paper presented at the SDM.

Bavota, Gabriele, Dit, Bogdan, Oliveto, Rocco, Penta, Massimiliano Di, Poshyvanyk, Denys, & Lucia, Andrea De. (2013). *An empirical study on the developers' perception of software coupling*. Paper presented at the Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA.

Beck, F., Melcher, J., & Weiskopf, D. (2016). *Identifying modularization patterns by visual comparison of multiple hierarchies.* Paper presented at the 2016 IEEE 24th International Conference on Program Comprehension (ICPC).

Beck, Fabian, & Diehl, Stephan. (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, Mikhail, & Mooney, Raymond 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.

Briand, L. C., Labiche, Y., & Yihong, Wang. (2001). *Revisiting strategies for ordering class integration testing in the presence of dependency cycles.* Paper presented at the Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on.

Briand, L. C., Labiche, Y., & Yihong, Wang. (2003). An investigation of graph-based class integration test order strategies. *Software Engineering, IEEE Transactions on, 29*(7), 594-607. doi: 10.1109/TSE.2003.1214324

Bullmore, Ed, & Sporns, Olaf. (2009). Complex brain networks: graph theoretical analysis of structural and functional systems. *Nat Rev Neurosci, 10*(3), 186-198.

Chatzigeorgiou, A., & Melas, G. (2012). *Trends in object-oriented software evolution: Investigating network properties.* Paper presented at the Software Engineering (ICSE), 2012 34th International Conference on.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on, 20*(6), 476-493. doi: 10.1109/32.295895

Chong, Chun Yong, & Lee, Sai Peck. (2015a). Analyzing maintainability and reliability of object-oriented software using weighted complex network. *Journal of Systems and Software*. doi: http://dx.doi.org/10.1016/j.jss.2015.08.014

Chong, Chun Yong, & Lee, Sai Peck. (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, Chun Yong, Lee, Sai Peck, & Ling, Teck Chaw. (2013). Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. *Information and Software Technology, 55*(11), 1994-2012.

Concas, G., Marchesi, M., Pinna, S., & Serra, N. (2007). Power-Laws in a Large Object-Oriented Software System. *Software Engineering, IEEE Transactions on, 33*(10), 687-708. doi: 10.1109/TSE.2007.1019

Concas, Giulio, Marchesi, Michele, Murgia, Alessandro, Tonelli, Roberto, & Turnu, Ivana. (2011). On the distribution of bugs in the eclipse system. *Software Engineering, IEEE Transactions on, 37*(6), 872-877.

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: DOI 10.1016/j.infsof.2011.01.006

Davidson, Ian, & 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

Dazhou, Kang, Baowen, Xu, Jianjiang, Lu, & Chu, W. C. (2004). *A complexity measure for ontology based on UML.* Paper presented at the Distributed Computing Systems, 2004. FTDCS 2004. Proceedings. 10th IEEE International Workshop on Future Trends of.

Diaz-Valenzuela, Irene, Loia, Vincenzo, Martin-Bautista, Maria J., Senatore, Sabrina, & Vila, M. Amparo. (2016). Automatic constraints generation for semisupervised clustering: experiences with documents classification. *Soft Computing, 20*(6), 2329-2339. doi: 10.1007/s00500-015-1643-3

Dijkstra, Edsger Wybe. (1976). *A discipline of programming* (Vol. 1): Prentice-Hall Englewood Cliffs.

El Emam, Khaled, Benlarbi, Saïda, Goel, Nishith, & Rai, Shesh N. (2001). The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on, 27*(7), 630-650.

Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., & Sander, J. (2009). *Decomposing object-oriented class modules using an agglomerative clustering technique.* Paper presented at the Software Maintenance, 2009. ICSM 2009. IEEE International Conference on.

Frigui, H., & Hwang, C. (2008). Fuzzy Clustering and Aggregation of Relational Data With Instance-Level Constraints. *IEEE Transactions on Fuzzy Systems, 16*(6), 1565-1581. doi: 10.1109/TFUZZ.2008.2005692

Greene, Derek, & Cunningham, Pádraig. (2007). Constraint Selection by Committee: An Ensemble Approach to Identifying Informative Constraints for Semi-supervised Clustering. In *Machine Learning: ECML 2007: 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007. Proceedings* (pp. 140-151). Berlin, Heidelberg: Springer Berlin Heidelberg.

Gronau, Ilan, & Moran, Shlomo. (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

Gyimothy, Tibor, Ferenc, Rudolf, & Siket, Istvan. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on, 31*(10), 897-910.

Harman, M., Mansouri, S. A., & Zhang, Y. Y. (2012). Search-Based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys, 45*(1), 11. doi: 10.1145/2379776.2379787

Hu, Hao, Fang, Jun, Lu, Zhengcai, Zhao, Fengfei, & Qin, Zheng. (2012). *Rank-directed layout of UML class diagrams*. Paper presented at the Proceedings of the First International Workshop on Software Mining, Beijing, China.

Izurieta, C., Griffith, I., Reimanis, D., & Luhr, R. (2013). *On the Uncertainty of Technical Debt Measurements.* Paper presented at the Information Science and Applications (ICISA), 2013 International Conference on.

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: http://dx.doi.org/10.1016/j.ins.2007.01.021

Kestler, HansA, Kraus, JohannM, Palm, Günther, & Schwenker, Friedhelm. (2006). On the Effects of Constraints in Semi-supervised Hierarchical Clustering. In F. Schwenker & S. Marinai (Eds.), *Artificial Neural Networks in Pattern Recognition* (Vol. 4087, pp. 57-66): Springer Berlin Heidelberg.

Klein, Dan, Kamvar, Sepandar D., & Manning, Christopher 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.

Letouzey, J., & Ilkiewicz, M. (2012). Managing Technical Debt with the SQALE Method. *Software, IEEE, 29*(6), 44-51. doi: 10.1109/MS.2012.129

Liu, Y. Y., Slotine, J. J., & Barabasi, A. L. (2011). Controllability of complex networks. *Nature, 473*(7346), 167-173. doi: 10.1038/nature10011

Louridas, Panagiotis, Spinellis, Diomidis, & Vlachos, Vasileios. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol., 18*(1), 1-26. doi: 10.1145/1391984.1391986

Lung, Chung-Horng, Zaman, Marzia, & Nandi, Amit. (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

Lung, Chung-Horng, & Zhou, Chenjuan. (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.

Ma, Yu-Tao, He, Ke-Qing, Li, Bing, Liu, Jing, & Zhou, Xiao-Yan. (2010). A hybrid set of complexity metrics for large-scale object-oriented software systems. *Journal of Computer Science and Technology, 25*(6), 1184-1201.

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: 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: Doi 10.1109/Tse.2007.70732

Miyamoto, Sadaaki. (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.

Olague, H. M., Etzkorn, L. H., Gholston, S., & Quattlebaum, S. (2007). Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *Software Engineering, IEEE Transactions on, 33*(6), 402-419. doi: 10.1109/TSE.2007.1015

Pang, Tin Yau, & Maslov, Sergei. (2013). Universal distribution of component frequencies in biological and technological systems. *Proceedings of the National Academy of Sciences*. doi: 10.1073/pnas.1217795110

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.

Potanin, Alex, Noble, James, Frean, Marcus, & Biddle, Robert. (2005). Scale-free geometry in OO programs. *Commun. ACM, 48*(5), 99-103. doi: 10.1145/1060710.1060716

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

Riva, C. (2000). *Reverse architecting: an industrial experience report.* Paper presented at the Reverse Engineering, 2000. Proceedings. Seventh Working Conference on.

Shental, Noam, & Weinshall, Daphna. (2003). *Learning Distance Functions using Equivalence Relations.* Paper presented at the In Proceedings of the Twentieth International Conference on Machine Learning.

Singh, Gagandeep. (2013). Metrics for measuring the quality of object-oriented software. *ACM SIGSOFT Software Engineering Notes, 38*(5), 1. doi: 10.1145/2507288.2507311

SonarQube, SonarQube, in, http://www.sonarqube.org/, 2014.

Subramanyam, R., & Krishnan, M. S. (2003). Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on, 29*(4), 297-310. doi: 10.1109/TSE.2003.1191795

Tempero, Ewan, Anslow, Craig, Dietrich, Jens, Han, Ted, Li, Jing, Lumpe, Markus, . . . Noble, James. (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.
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: http://dx.doi.org/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 Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on.
Valverde, Sergi, & Solé, Ricard V. (2003). Hierarchical small worlds in software architecture. *arXiv preprint cond-mat/0307278*.
Wagstaff, Kiri L. (2007). Value, Cost, and Sharing: Open Issues in Constrained Clustering. In S. Džeroski & J. Struyf (Eds.), *Knowledge Discovery in Inductive Databases: 5th International Workshop, KDID 2006 Berlin, Germany, September 18, 2006 Revised Selected and Invited Papers* (pp. 1-10). Berlin, Heidelberg: Springer Berlin Heidelberg.
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. (2005). *Software clustering based on omnipresent object detection.* Paper presented at the 13th International Workshop on Program Comprehension, 2005. IWPC 2005.
Xiong, S., Azimi, J., & Fern, X. Z. (2014). Active Learning of Constraints for Semi-Supervised Clustering. *IEEE Transactions on Knowledge and Data Engineering, 26*(1), 43-54. doi: 10.1109/TKDE.2013.22
Yoon, Jeongah, Blumer, Anselm, & Lee, Kyongbum. (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
Zhihua, Wen, & Tzerpos, V. (2004). *An effectiveness measure for software clustering algorithms.* Paper presented at the Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on.
Zimmermann, Thomas, & Nagappan, Nachiappan. (2008). *Predicting defects using network analysis on dependency graphs.* Paper presented at the Proceedings of the 30th international conference on Software engineering.