A fast clustering algorithm for modularization of large-scale software systems

Navid Teymourian, Habib Izadkhah, Ayaz Isazadeh

Abstract—A software system evolves over time in order to meet the needs of users. Understanding a program is the most important step to apply new requirements. Clustering techniques through dividing a program into small and meaningful parts make it possible to understand the program. In general, clustering algorithms are classified into two categories: hierarchical and non-hierarchical algorithms (such as search-based approaches). While clustering problems generally tend to be NP-hard, search-based algorithms produce acceptable clustering and have time and space constraints and hence they are inefficient in large-scale software systems. Most algorithms which currently used in software clustering fields do not scale well when applied to large and very large applications. In this paper, we present a new and fast clustering algorithm, FCA, that can overcome space and time constraints of existing algorithms by performing operations on the dependency matrix and extracting other matrices based on a set of features. The experimental results on ten small-sized applications, ten folders with different functionalities from Mozilla Firefox, a large-sized application (namely ITK), and a very large-sized application (namely Chromium) demonstrate that the proposed algorithm achieves higher quality modularization compared with hierarchical algorithms. It can also compete with search-based algorithms and a clustering algorithm based on subsystem patterns. But the running time of the proposed algorithm is much shorter than that of the hierarchical and non-hierarchical algorithms. The source code of the proposed algorithm can be accessed at https://github.com/SoftwareMaintenanceLab.

Index Terms—Software clustering, Software modularization, Software maintenance, Software comprehension, Architecture recovery.

1 Introduction

Software plays a key role in government agencies and organizations and as an interface, it has an important role in communications. Over the time, the requirements of an organization will change according to the environmental conditions and software engineers need to make changes in the software system to meet the needs of the organization. To make changes to the software, developers require to understand the software structure (software architecture). During software maintenance, software engineers spend a considerable amount of time on program comprehension activities [1]. Because of the complex structure and relationships, understanding the structure of a large software system and applying new changes is not an easy task. Recovering software architecture to understand software systems is therefore particularly important because it facilitates the maintenance and evolution of software systems [2], [3].

The software architecture recovery aims to use techniques to partition a software system into meaningful subsystems (modules) [4], [5]. For this reason, numerous attempts have been done to develop techniques for extracting software architecture automatically. One of these techniques is clustering [5]. "The objective of software clustering is to reduce the complexity of a large software system by replacing a set of artifacts with a cluster, a representative abstraction of all artifacts grouped within it. Thus, the obtained decomposition is easier to understand" [6]. The purpose of clustering is to divide a software system into clusters (modules) so that the relationships between artifacts in a cluster are maximized (i.e., cohesion) and the relationships between clusters (i.e., coupling) are minimized. Figure 1 illustrates the clustering process of a software system. The input of a clustering algorithm is an Artifact Dependency Graph (ADG) which is constructed from source code [6]. The nodes of this graph indicate the artifacts (e.g., class, method, file, function, etc.), and the links indicate the relationships between artifacts (e.g., calling dependency, inheritance dependency, semantic dependency, etc.).

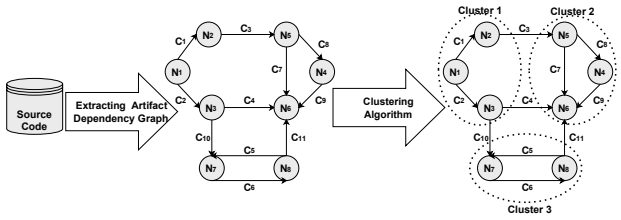

Fig. 1. Software Clustering Process

In general, clustering algorithms are classified into hierarchical and non-hierarchical categories (such as search-based and graph-based) [7]. In the hierarchical algorithms, artifacts are first considered as separate clusters and are then merged in a repeating process. These algorithms require a data table representing relationships between artifacts and a table that

• *N. Teymourian, H. Izadkhah, and A. Isazadeh are with the Department of Computer Science, University of Tabriz, Tabriz, Iran*

• *Corresponding author: E-mail: izadkhah@tabrizu.ac.ir*

Manuscript received -; revised August -.

shows similarities between artifacts. These tables should be updated at each step of the clustering process. In very large software systems, these tables will be very big and it will also be time consuming to calculate the similarities between the artifacts. For example, the LIMBO algorithm [6], at the first step of the clustering process, requires about 15×10^{11} operations for a graph with 10,000 artifacts. Given all the clustering steps, the number of operations will be much higher than this number. This algorithm requires a large amount of time for large graphs, which reduces its efficiency.

In the literature, because of the NP-Hardness of clustering problem, search-based methods (such as genetic algorithm) have been widely used [8], [9]. Search-based algorithms are an effective way to solve the clustering problem [9]. The search-based algorithms may take a long time to execute, if computing the fitness function in each iteration takes a long time to perform. Therefore, the main drawback of these methods is that they work very slowly when faced with large-sized graphs. Due to the problems of the existing algorithms, in this paper, we designed a new clustering algorithm that operates on artifact dependency matrix constructed from source code.

The main problem addressed in this paper is scalability in terms of running time. We aim to provide a deterministic clustering algorithm that its running time grows slowly as the input size increases. We claim that by performing a series of simple operations on the dependency matrix and extracting other matrices based on a set of features, a developer can quickly cluster a software system while the clustering quality is acceptable. The proposed algorithm was tested on ten small-sized software systems, ten folders of Mozilla Firefox, and two large and very large software systems. The results showed that the algorithm achieves acceptable clustering quality according to evaluation criteria (internal and external criteria), in less run time, compared to the tested algorithms.

This paper is structured as follows. In Section 2, we discuss related work on software clustering. Section 3 introduces the proposed algorithm, and we present the experimental setup in Section 4. The result of research and threats to validity are discussed in Sections 5 and 6, respectively. Finally, section 7 is the conclusions of this research and future work.

2 Related work

So far many algorithms have been developed for clustering, but given the NP-Hardness of clustering problem, designing a proper clustering algorithm is a difficult work. In this section, first, a description of the clustering algorithms classification is presented, and then some state-of-the-art clustering algorithms are described.

Most software clustering algorithms fall into two major categories: agglomerative hierarchical and search-based algorithms, and there also are a few algorithms that are graph-based [8] and pattern-based. Hierarchical algorithms are greedy and phased and at each stage, the most similar artifacts are merged. Single linkage, complete linkage, average linkage, weighted average link are some of the classical hierarchical algorithms [10]. Maqbool and Babri presented two hierarchical clustering algorithms for software architecture recovery, namely combined Algorithm (CA) and Weighted combined Algorithm (WCA) [11]. Weighted combined Algorithm is a popular hierarchical clustering technique. Considering the ways to compute the similarity between artifacts, Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM), WCA has two versions WCA-UE and WCA-UENM. Another popular hierarchical clustering algorithm is scaLable InforMation BOttleneck (LIMBO) [6]. This algorithm employs information theory and entropy concepts for software clustering. In [10], the cooperative clustering technique (CCT) as a consensus-based technique is utilized for the software clustering problem. In this technique more than one similarity measure cooperates during the hierarchical clustering process.

In search-based algorithms, the clustering process is considered as an optimization problem [5]. Then, heuristic or meta-heuristic search methods are used to find the near-optimal solution. The search process is guided and evaluated by a quality function (objective function) that shows how appropriate the solution is. The searching process in these algorithms is performed in two global and local ways. In the global search-based systems algorithm, the entire search space is considered the solution space. In these algorithms, an operator is intended to discover new areas in the search space. Local search algorithms start from a selective solution and then gradually move from the current solution to the neighboring solution using search changes. However, these algorithms inherently have the problem of being trapped in local optima solution [8]. It is also possible to combine global and local search algorithms. Graph-based clustering algorithms can be applied to various areas such as social networking, image segmentation, and software clustering. Mohammadi and Izadkhah in [7] use a neighboring tree generated from the ADG to cluster a software system. The clustering quality obtained by this algorithm is better than hierarchical methods and worse than evolutionary methods. Spectral methods [12] use algebraic properties of the graph, such as eigenvalues and eigenvectors in the corresponding Laplacian matrix to perform clustering. ACDC [13] is a pattern-based algorithm that was introduced by Tzerpos and Holt. It uses several patterns to cluster code artifacts. Several previous studies, such as [14], [15], [16], have shown that ACDC performed well on the tested applications.

Depending on the features used for clustering, clustering techniques can be categorized in terms of structural and non-structural (or semantic). Some algorithms only use structural properties and some others use non-structural properties such as comments and name of artifacts. Some algorithms combine both of them. Table 1 shows some clustering algorithms with different categories. Here, we briefly describe some clustering algorithms.

Bunch: the Bunch toolset was proposed by Mitchell for the software clustering problem [5], [17]. Two search-based algorithms with different searching strategy namely genetic algorithm (GA) and Hill-climbing were employed in this

TABLE 1
Some search-based software clustering algorithms

Name	Type of algorithm	Type of objective function	Structural/Semantic Features
Bunch-NAHC and Bunch-SAHC [17]	Local Search	Single Objective	Structural
E-CDGM [18]	Local Search	Single Objective	Structural
Large neighborhood search [19]	Local Search	Single Objective	Structural
HC-SMCP [20]	Local Search	Single Objective	Structural
SHC [21]	Local Search	Single Objective	Semantic
Bunch-GA [5]	Global Search	Single Objective	Structural
DAGC [22]	Global Search	Single Objective	Structural
A multi-agent evolutionary algorithm [23]	Global Search	Single Objective	Structural
Harmony search [24]	Global Search	Multi-Objective	Structural
GA-SMCP [20]	Global Search	Single Objective	Structural
Hyper-heuristic approach [25]	Global Search	Multi-Objective	Structural
ECA and MCA [9]	Global Search	Multi-Objective	Structural
Estimation of distribution approach [26]	Global Search	Single Objective	Structural
EoD, CGH, CGoH [8]	Global Search	Multi-Objective	Structural and Semantic
Search based multiobjective software remodularization [27]	Global Search	Multi-Objective	Structural
Multiple relationship factors [28]	Global Search	Multi-Objective	Structural
Interactive evolutionary optimization [29]	Global Search	Multi-Objective	Structural
GAKH [30]	Global Search	Single Objective	Structural
MaABC [31]	Global Search	Multi-objective	Structural
ILOF [32]	Global Search	Support multi-objective	Structural

toolset for the software clustering problem. Next Ascent Hill climbing (NAHC) and Steepest Ascent Hill climbing (SAHC) are two versions of the Hill-climbing algorithm which are presented in the Bunch. The algorithms presented in the Bunch use a real-valued encoding to represent the solutions. The search space produced by this encoding equals n^n, where n is the number of artifacts. Bunch input is a call dependency graph (CDG) made from source code. The output of the Bunch is clustering with minimum coupling and maximum cohesion among clusters.

DAGC: similar to Bunch, this algorithm [22] employs genetic algorithm to perform clustering. But, each chromosome is encoded by a permutation of graph's nodes. The search space in this algorithm equals n!. It can be said that it has less search space than Bunch, in contrast to it uses a sophisticated encoding method.

Maximizing Cluster Approach (MCA) and Equal-size Cluster Approach (ECA) [9]: both algorithms are multi-objective and use two-archive Pareto optimal genetic algorithm to optimize the objectives. The objectives used for clustering in MCA are "maximize the number of edges inside the clusters", "minimize the number of edges between all the clusters", "maximize the number of clusters", "maximize the TurboMQ", and "minimize the number of single-member clusters"; and the objectives used in the ECA are similar to the MCA with one difference. ECA replaced the fifth objective of the MCA with "minimizing the difference between the maximum and minimum number of modules in a cluster."

In semantic-based algorithms, how words are selected and which semantic analysis method employed is the main reason for the differences between these algorithms. Garcia et al. [33] proposed a hierarchical clustering algorithm- named Architecture Recovery using Concerns (ARC)- that uses concerns to perform an architecture recovery. ARC considers textual information (identifiers and comments) extracted from source code and extracts concerns based on Latent Dirichlet Allocation (LDA) model. Some studies, e.g., [16], have shown that this algorithm has acceptable accuracy.

Corazza et al. [34] proposed a natural language processing based clustering approach that partitions textual information (identifiers and comments) into different zones. The zones are weighted based on the Expectation-Maximization algorithm and then clustered by hierarchical agglomerative technique. Using the Hill-climbing algorithm, in [21], a semantic-based clustering algorithm, namely SHC, was presented which uses artifact names and comments for semantic analysis. Taking into account syntactic features such as call dependency and inheritance dependency and semantic features such as code comment and identifier name, Misra et al. [35] proposed an algorithm for clustering.

Most software clustering algorithms use static dependencies between artifacts. Xiao and Tzerpos [36] presented an approach for investigating the dynamic dependencies. The results of their experiments on some applications showed that dynamic clusters have significant competencies.

To sum up, the main limitations of the existing algorithms are:

- Search-based algorithms, such as genetic algorithms, are usually used to cluster software systems [8], [9]. Because of their exploration and exploitation ability, they can produce good quality answers. But on large applications, they are very time-consuming. Normally it takes more than a month for graphs with more than 10,000 nodes without parallelization to find a proper clustering. It should be noted, however, that parallelization cannot greatly reduce time. This is because the software systems selected for clustering in these methods are small- or medium-sized, e.g., see [8], [9], [17], [20], [24], [37]. Also, because evolutionary algorithms are stochastic, they need to be executed

many times, which for large graphs is practically impossible because they are time-consuming.

- Hierarchical algorithms require a data table that represents relationships between artifacts and a table that shows similarities between artifacts. In very large software systems, these tables will be very big and it will also be very time consuming to calculate similarities between artifacts. For example, the LIMBO algorithm for a software system with 10,000 artifacts requires about 15×10^{11} computations, at the first step of the clustering process.

3 Proposed clustering algorithm

The dependency graph is a mathematical way to model the relationships between artifacts. Let x and y denote two artifacts (two nodes in the graph) so that an edge between the two artifacts x and y indicate the existing dependency between them. In the dependency matrix corresponding to the dependency graph, the intersection of two nodes is placed one if there exists an edge between them, and zero if the two nodes are not connected.

In this paper, we cluster software systems by defining a series of operations on the dependence matrix and extracting several features from it. We derive additional matrices from the dependence matrix, based on a set of features, as well as applying mathematical operations on the matrices to perform the clustering. In this algorithm, the dependency graph is the input of the algorithm and a modularized ADG is the output of the algorithm. Because the input of the proposed algorithm is the dependency graph, so the algorithm is independent of the programming language used. Tools such as Understand (https://scitools.com/) or NDepend (https://ndepend.com) can be used to extract the dependency graph from the source code of most programming languages. The proposed algorithm aims to maximize intra-connectivity within the clusters (i.e., cohesion) and minimize inter-dependencies between the clusters (i.e., coupling).

Algorithm steps (FCA)

1) Input: dependency matrix constructed from the artifact dependency graph,
2) The neighborhood degree matrix is created from the dependency matrix. This matrix shows degree information. Each node has several neighbors, and this matrix shows the degree of neighbor nodes for each node. The steps to build this matrix are as follows:
   a) The number of rows and columns in this matrix is equal to the number of nodes in the artifact dependency graph.
   b) Let x and y denote the row number (node number) and column number in the matrix, respectively. For each node x in the dependency graph connected to node y, the degree of node y is placed at the intersection from x to y in the neighborhood degree matrix.

   We use this matrix to select nodes for clustering. High-degree nodes and nodes that are connected to high-degree nodes are not good options to start the clustering process. To perform clustering, the algorithm starts with nodes that have a small degree and are not connected to high-degree nodes. We call these nodes "border" nodes. Larger-degree nodes tend to absorb the rest of the nodes and create larger clusters. In steps 3 and 4, the nodes are ranked and used as primary nodes in the clustering process.
3) To rank the nodes, the numbers in each row of the neighborhood matrix are summed up and placed in an $n \times 1$ matrix named *Sum* matrix, where *n* is the number of nodes. The entries of this matrix are S_1, S_2, ..., S_n.
   The *Sum* matrix is not enough to prioritize nodes. Because, in large graphs, the number of nodes in which the sum of their rows is equal is large. Therefore, in order to prioritize the nodes, it is necessary to normalize this matrix. Step 4 is used for this purpose.
4) To normalize the *Sum* matrix, for each node x (row x), the following equation is calculated and the results are saved in an $n \times 1$ matrix named *Effect* matrix. Let NDM denotes the neighborhood degree matrix, the entries of *Effect* matrix, $E_1, E_2, \ldots, E_n$, are calculated as follows:

$$E_x = \frac{S_x}{\sum_{i=1}^{n} (k_i \times S_i)} \tag{1}$$

   where

$$k_i = \begin{cases} 0, & NDM_{x,i} = 0; \\ 1, & NDM_{x,i} \neq 0. \end{cases}$$

   The numbers in the *Effect* matrix are sorted in descending order. Node x having the largest E_x is used as the first node for clustering. It is important to note that the largest E_x is for a border node and the clustering process starts with this node.
5) The clustering steps start from the first node in the *Effect* matrix as follows:
   a) For node x, the algorithm finds node y which has the lowest numeric value in the row associated with x in the neighborhood matrix. The reverse of this rule must hold: for the node y found by the algorithm, the node x must also have the lowest numeric value in the row associated with y in the neighborhood matrix. In such a case, the node x is co-clustered with the node y. Otherwise, the node x is added to an array named *Incompatible*.
   b) Repeat step 5(a) for all nodes in *Effect* matrix. If the addressed node has already been clustered, it will be ignored.
6) Clustering all nodes in array *Incompatible*. For node x, the algorithm finds node y that has the first or second smallest numerical value in the row associated with

x in the neighborhood matrix. The reverse of this rule must hold: for the node y found by the algorithm, the node x must also have the first or second smallest numerical value in the row associated with y in the neighborhood matrix. In such a case, the node x is co-clustered with the node y. Otherwise, the node x is added to an array named *Closed*.
The intuition behind steps 5 and 6 is to cluster nodes that are related to each other and also have a lower degree. This will reduce coupling. These steps prevent the formation of very large clusters.

7) The clusters obtained in steps 5 and 6 are merged if they have at least one node in common. The number of clusters obtained in steps 5 and 6 is high. This step reduces the number of clusters by merging some of them and increases cohesion.
8) Clustering all nodes in array *Closed*. Up to this step, there may be nodes that have not been clustered. All these nodes are in the array *Closed*. Using one of the following steps, we determine the cluster of these nodes.
    a) There are nodes in array *Closed* that are only connected to one cluster. These nodes merge with the related clusters. The next condition is considered for the remaining nodes.
    b) For node x in array *Closed*, all nodes, y, which have an edge to node x are extracted in the dependency matrix. The node x goes to the cluster that has the least amount of *inter-connectivity*. Intuitively, *inter-connectivity* measures a cluster coupling. The lower the value of this relationship, the lower the coupling of a cluster, which is desirable.
    Let X, Y, and Z denote the sum of the outer edges of the cluster containing the node y, the sum of E_x values of nodes in the cluster containing the node y, and the number of nodes in the cluster containing the node y, respectively. We have:

$$inter - connectivity = \frac{X + Y}{Z} \qquad (2)$$

    The value of $\frac{X}{Z}$ may be the same for clusters with different sizes and may not show the value of the coupling well. While one is superior to the other. That's why Y is used. If the *inter-connectivity* value for the clusters is equal, the next condition is investigated.
    c) The nodes go to a cluster that has the most relationship with the members of that cluster. If the number of relationships is the same, the node go to a cluster that has fewer members.

Using an example, we illustrate the steps of the proposed algorithm. Artifact dependency graph (ADG) of a sample software is shown in Fig. 2.
Step 1- Table 2 shows the dependency matrix constructed from the ADG shown in Fig. 2.

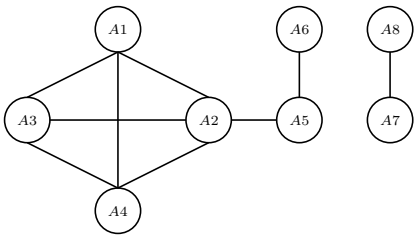

Fig. 2. A sample of an artifact dependency graph

TABLE 2
Dependency matrix constructed from Fig. 2

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	1	1	1	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	1	0	0	0	0
4	1	1	1	0	0	0	0	0
5	0	1	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	1	0

Step 2- The neighborhood degree matrix is constructed for all nodes, as shown in Table 3. This matrix shows the degree of neighbor nodes for each node. For example, row 1 shows the neighbors of node 1, while row 3 shows the neighbors of node 3.
Step 3- The *Sum* matrix is shown in Table 4. This matrix represents the sum of the rows in Table 3.
Step 4- According to Eq. 1, the *Effect* matrix is created, as shown in Table 5. For example:

$$E_1 = S_1/(S_2 + S_3 + S_4) = 10/(11 + 11 + 10) = 0.32$$

TABLE 3
The neighborhood degree matrix

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	4	3	3	0	0	0	0
2	3	0	3	3	2	0	0	0
3	3	4	0	3	0	0	0	0
4	3	4	3	0	0	0	0	0
5	0	4	0	0	0	1	0	0
6	0	0	0	0	2	0	0	0
7	3	0	0	0	0	0	0	1
8	3	0	0	0	0	0	1	0

TABLE 4
The Sum matrix

Sum	
S_1	10
S_2	11
S_3	10
S_4	10
S_5	5
S_6	2
S_7	1
S_8	1

TABLE 5
The Effect matrix

Effect	
E_1	0.32
E_2	0.31
E_3	0.32
E_4	0.32
E_5	0.38
E_6	0.40
E_7	1
E_8	1

The calculated Effect matrix is sorted in descending order as:

$$\text{Effect name} = [E_7, E_8, E_6, E_5, E_1, E_3, E_4, E_2]$$

Step 5- The clustering process starts from the first node in the *Effect* matrix.

1) Node 7 can be co-clustered with a node that has the lowest numerical value in the row of Table 3, i.e., node 8. The important point is that node 8 must also have the lowest number in its corresponding row with node 7. This condition is true.
   a) First cluster $\{7, 8\}$
   b) Since node 8 is in the first cluster, then it is not checked in *Effect* matrix.
2) Node 6 can be co-clustered with a node that has the lowest numerical value in the row of Table 3, i.e., node 5. Node 5 also has the lowest number in its corresponding row with node 6. Thus
   a) Second cluster $\{5, 6\}$
   b) Since node 5 is in the first cluster, then it is not checked in *Effect* matrix.

   After addressing the condition above for node 1, third cluster is $\{1, 3, 4\}$
3) Node 2 can be co-clustered with a node that has the lowest numerical value in its corresponding row of Table 3, which is node 5.
   a) since the lowest number in row 5 is equal to 1, which corresponds to node 6, so node 2 doesn't cluster with node 3.
   b) the condition is not fulfilled and node 2 is added to array *Incompatible*.

Step 6- Clustering array *Incompatible* starts from its first node i.e., node 2. In the neighborhood matrix, the first and second small numbers in row 2 are related with nodes 1, 3, 4 and 5. That is, node 2 can be clustered with these nodes if the inverse of these relationships are also true.

1) The first and second small numbers in row 1 (i.e., node 1) are related to nodes 2, 3 and 4; so the condition is true, and node 2 can be clustered with node 1.
2) This step is also applied on nodes 3, 4 and 5.
3) Fourth cluster $\{2, 1, 3, 4, 5\}$

Step 7- The second, third and fourth clusters are merged due to having common nodes, and the final clustering is as follows.

$$\text{First cluster} = \{1, 2, 3, 4, 5, 6\}, \quad \text{Second cluster} = \{7, 8\}$$

Step 8- Due to the small size of the dependency graph used, array *Closed* is empty, and thus step 8 is not checked.

4 Experimental Setup

To evaluate the proposed algorithm, it is necessary to mention the following.

TABLE 6
The description of tested software systems

Name	Description	#Files	#Links
compiler	A small compiler developed at the University of Toronto	13	32
nos	A file system	16	52
boxer	Graph drawing tool	18	29
spdb	A tool to analyze several proteins at the same time	21	33
ispell	Spelling and typographical error correction software	24	103
ciald	Program dependency analysis tool	26	64
rcs	System used to manages multiple revisions of files	29	163
star	A program understanding tool	36	89
bison	Parser generator	37	179
cia	Program dependency graph generator for C programs	38	87

4.1 Software system

Software systems play an important role in the evaluation and comparison of clustering algorithms. We selected ten real-world small-sized applications, as shown in Table 6. In this table, the number of links indicates the number of relationships between the artifacts within the dependency graph, as described in the Introduction. Mozilla Firefox[1] is an open-source software system. Based on the Open hub site report, more than 13000 developers work on this application. This application has 55 folders (clusters), so we selected ten of them with different functionalities and sizes. The names and specifications of these folders are presented in Table 7.

In addition to the above applications, two large and very large applications, namely ITK, Chromium, are selected. In place of those large-sized projects, we included ITK (including 7,310 files). We also included a very large project, Chromium (including 18,698 files). Detailed information about these projects can be found in Table 8.

4.2 Expert decompositions

Expert decomposition (other names: ground-truth architecture or authoritative decomposition) is employed to evaluate the accuracy of the clustering algorithms [7], [10], [14], [15], [38]. An algorithm is reliable if its modularization result is close to decomposition provided by an expert [10]. In large projects, the directory structure of the project originally usually reflects the architecture of the project [39].

In this paper, the developer preview version of the Mozilla Firefox has been selected, because there is a credible (human) expert decomposition (the directory structure) of that. It is important to note that this version is a stable version. For example, folder DB has 97 files organized by the developers of this software in four sub-folders. Our method considers all of 97 files as flat and the aim is to determine how much the algorithm can achieve a clustering similar to the decomposition of Mozilla Firefox's developers.

We also used the ground-truth architectures created by Lutellier et al. [15] for ITK and Chromium applications to evaluate the accuracy of the clustering algorithms[2].

[1] https://ftp.mozilla.org/pub/firefox/releases/devpreview/1.9.3a4/source/

[2] The ground-truth architecture for ITK and Chromium are available at http://asset.uwaterloo.ca/ArchRecovery

TABLE 7
Properties of the selected folders

Folder Name	#Files	#Links	#Modules	Some Folder Functionalities
Accessible	179	293	8	enabling as many people as possible to use Web sites, even when those people's abilities are limited in some way. Files for accessibility (i.e., MSAA (Microsoft Active Accessibility), ATK (Accessibility Toolkit, used by GTK+ 2) support files).
Browser	45	45	4	Contains the front end code (in XUL, Javascript, XBL, and C++) for the Firefox browser Contains the front end code for the DevTools (Scratchpad, Style Editor, etc.) . Contains images and CSS files to skin the browser for each OS (Linux, Mac and Windows)
Build	21	4	2	Miscellaneous files used by the build process.
Content	881	2948	13	The data structures that represent the structure of Web pages (HTML, SVG, XML documents, elements, text nodes, etc.) containing the implementation of many DOM interfaces and also implement some behaviors associated with those objects, such as link handling, form control behavior, and form submission. This directory also contains the code for XUL, XBL, XTF as well as the code implementing XSLT and event handling.
Db	97	494	4	Container for database-accessing modules.
Dom	163	324	5	IDL definitions of the interfaces defined by the DOM specifications The parts of the connection between JavaScript and the implementations of DOM objects Implementations of a few of the core "DOM Level 0" objects, such as window, window.navigator, window.location, etc.
Extensions	179	206	13	Contains several extensions to mozilla, which can be enabled at compile-time Implementation of the negotiate auth method for HTTP and other protocols. Has code for SSPI, GSSAPI, etc. Permissions backend for cookies, images, etc., as well as the user interface to these permissions and other cookie features. Support for the datetime protocol; Support for the finger protocol. A two-way bridge between the CLR/.NET/Mono/C#/etc. world and XPCOM Implementation of W3C's Platform for Privacy Preferences standard. Support for implementing XPCOM components in python. Support for accessing SQL databases from XUL applications; Support for Webservices.
Gfx	342	644	7	Contains interfaces that abstract the capabilities of platform specific graphics toolkits, along with implementations on various platforms These interfaces provide methods for things like drawing images, text, and basic shapes It also contains basic data structures such as points and rectangles used here and in other parts of Mozilla.
Intl	573	957	7	Internationalization and localization support; Code for "sniffing" the character encoding of Web pages Code for dealing with Complex Text Layout, related to shaping of south Asian languages Code related to determination of locale information from the operating environment Code that converts (both ways: encoders and decoders) between UTF-16 and many other character encodings Code related to implementation of various algorithms for Unicode text, such as case conversion.
Ipc	391	59	4	Container for implementations of IPC (Inter-Process Communication).

TABLE 8
Data sets specifications for two large and very large software systems

Project	Version	Description	SLOC	#File
ITK	4.5.2	Image Segmentation Toolkit	1M	7,310
Chromium	svn-171054	Web Browser	10M	18,698

4.3 Assessment of results

In the literature, there are many software clustering algorithms and thus to determine the appropriate algorithm, some metrics were presented for determining the quality of clustering. There exist generally two types of metrics for determining the quality of clustering: internal metrics, and external metrics. Internal metrics are independent of any ground-truth architecture, which calculate the quality of the recovered architectures.

TurboMQ presented in [5] is one of the internal metrics which is used in many research papers to evaluate the quality of the recovered architectures e.g., [9], [15], [40]. This metric is defined as follows:

$$TurboMQ = \sum_{i=1}^{k} \frac{2A_i}{2A_i + \sum_{j=1}^{k} (E_{i,j} + E_{j,i})} \quad (3)$$

where A_i represents the internal communication into cluster i, E_{ij} represents the external communication between two clusters i and j. The higher TurboMQ value indicates better clustering.

The MoJoFM metric is also used to evaluate clustering techniques [41]. This metric is a well-known and widely used external assessment (e.g., see [7], [8], [10], [15], [37]). In external assessment, the automatically prepared clustering, A, is compared with the decompositions prepared by human experts, B [10]. The value of MoJoFM is between 0 and 100, and the higher value means the more proximity between clustering generated by an algorithm and decomposition created by an expert and hence better results [10]. The MoJoFM is calculated by Eq. 4:

$$\text{MoJoFM(A, B)}(\%) = 1 - \frac{mno(A,B)}{max(mno\,(\forall A,B))} \quad (4)$$

where $mno(A,B)$ denotes the minimum operations required for converting clustering A to clustering B.

Cluster-to-cluster coverage (C2C) [14], [15], [16] is an external metric to assess component-level accuracy that measures the degree of overlap between the two architecture's clusters. Before calculating C2C between two recovered architecture, it is necessary to calculate the following equation.

$$c2c(c_i, c_j) = \frac{|entities(c_i) \cap entities(c_j)|}{\max(|entities(c_i)|, |entities(c_j)|)}$$

where c_i is an automatically prepared clustering; c_j is a ground-truth cluster; and entities(c) is the set of entities in cluster c. $C2C$ is computed as following.

$$C2C(A_1, A_2)(\%) = \frac{|simC(A_1, A_2)|}{|A_1.C|} \quad (5)$$

$$simC(A_1, A_2) = \{c_i | (c_i \in A_1, \exists c_j \in A_2) \wedge c2c(c_i, c_j) > th_{cvg})\}$$

A_1 is the recovered architecture; A_2 is a ground-truth architecture; and $A_1.C$ are the clusters of A_1. $simC(A_1, A_2)$

TABLE 9
The parameter setting for experiments

Parameter	Value
Population size	10N
Maximum number of generations	200N
Crossover Rate	0.8
Mutation Rate	$0.004\log_2(N)$
Termination condition	There has been no improvement in the population for 50 iterations

returns $A_1's$ clusters for which the c2c value is above a threshold th_{cvg}.

To compare the overall results of FCA against other tested algorithms in terms of TurboMQ, MoJoFM, C2C, and running time, we utilized a non-parametric effect size statistic namely Cliff's δ which is used to quantify the amount of difference between two algorithms.

4.4 Algorithmic Parameters

The setting of parameters is necessary for search-based algorithms. For genetic-based algorithms, for our comparisons, we followed the algorithmic parameters setting used in [9], [20]. Algorithmic parameters are dependent on the number of artifacts (N). For the rest of the algorithms (i.e., Hill-climbing algorithm and Estimation of Distribution algorithm), we used the same parameters as those used by the authors of these algorithms. We obtained the implementation of the ACDC from its official web sites.

As references [8], [9], [37], to reduce randomness the results of the search-based algorithms used in comparisons, we collect the best of 30 independent runs. For MoJoFM, TurboMQ, and C2C, we report the best values rounded to the closest integer. Let N denote the number of artifacts, the parameter setting for experiments is shown in Table 9.

4.5 Research questions

The following questions are answered to evaluate the effectiveness of the proposed algorithm.

RQ1. Does the proposed algorithm perform better than the hierarchical and non-hierarchical clustering algorithms in terms of TurboMQ, MoJoFM, and C2C?

RQ2. Is the proposed algorithm scalable?

RQ3. Is there a statistically significant improvement between the FCA and the algorithms compared?

We ran the algorithms on a Laptop with Intel core i7 processor 2.60GHz and 16GB of memory.

5 Experimental results

This section presents the results of the empirical study. The aim is to compare the proposed algorithm, FCA, against some hierarchical and non-hierarchical algorithms in terms of TurboMQ, MoJoFM, C2C, and running time. To this end, eight search-based algorithms with different characteristics are chosen. The algorithms selected vary from each other in some different ways including single-objective, multi-objective, global search, local search, structured-based methods, and semantic-based methods. The software clustering approaches to which we compared FCA are Bunch-GA, DAGC, ECA, MCA, Bunch-NAHC, SHC, GA-SMCP, and EoD. The characteristics of these algorithms are described in Table 10. K-means, a basic machine learning algorithm, and ACDC, a clustering algorithm based on subsystem patterns are also used for comparison. Several previous studies [14], [15], [16] have shown that ACDC performed well on the tested applications. Besides, we selected agglomerative clustering algorithms such as Complete, Single, Average (Weighted), WCA-UE, WCA-UENM, and LIMBO for comparison.

To compare and evaluate the proposed algorithm, several software systems with different domains and sizes have been selected. Tables 6-8 show the specifications of these software systems. Note that the dependency used in the software systems shown in Table 6 are call and include dependencies. The dependency used in the software systems shown in Table 7 are call and include dependencies, that we are obtained from their source code using Understand toolset (https://scitools.com/).

Lutellier et al. [15] have extracted various dependencies for ITK and Chromium applications such as include, symbol, Function call, etc. These dependencies alone do not cover the entire program. For example, in Chromium, the function call dependency only covers 12,627 artifacts of 18,698 artifacts. So, to cover the whole program, we merged these dependencies and removed duplicate dependencies.

The size of projects used for the experiments are 13 to 18,698 source files. Table 11 shows the size of projects used for experiments in some existing clustering algorithms.

To answer the research question RQ1, we compared the proposed algorithm against some hierarchical algorithms, k-means and ACDC in terms of TurboMQ on ten small-sized applications shown in Table 6. Table 12 shows the comparison results. The results demonstrate that in all cases the proposed algorithm has been able to obtain higher quality clustering than the algorithms tested.

We have also selected the Mozilla Firefox application. The reason for this choice is that there is an expert decomposition for it. Ten folders with different functionalities have been selected from this application. We have clustered these folders with eight evolutionary algorithms with different features, k-means and ACDC. Because the clustering problem is an NP-hard problem, evolutionary algorithms usually produce plausible solutions [8], [9]. In terms of MoJoFM, Table 13 shows that the proposed algorithm performs better in six out of ten cases. In the *Build* folder, there is a significant difference between the FCA and the other algorithms in the value of MoJoFM, and the FCA did not work well. The reason for this improper performance is that the dependency graph of this folder is disconnected and also has several isolated vertices. Note that, the FCA works better in large folders than other algorithms. In terms of C2C ($th_{cvg} > 33\%$), the FCA can compete with other algorithms. In terms of

TABLE 10
Features of selected search-based algorithms for comparison with the proposed algorithm

Algorithm	Algorithm type	#objective used	Search type	Structural based/ Semantic-based	Encoding type
Bunch-GA [5]	Genetic algorithm	Single-objective	Global	Structural	real-valued
DAGC [22]	Genetic algorithm	Single-objective	Global	Structural	permutation-based
ECA [9]	two-Archive genetic algorithm	Multi-objective	Global	Structural	real-valued
MCA [9]	two-Archive genetic algorithm	Multi-objective	Global	Structural	real-valued
Bunch-SAHC [17]	Hill-climbing algorithm	Single-objective	Local	Structural	real-valued
SHC [21]	Hill-climbing algorithm	Single-objective	Local	Semantic	real-valued
GA-SMCP [20]	Genetic algorithm	Single-objective	Global	Structural	real-valued
EoD [8]	Estimation of Distribution algorithm	Multi-objective	Global	Semantic & Structural	real-valued

TABLE 11
Projects size used for experiments in some clustering algorithms

Reference	Projects size (#modules or #files)
[9], [20]	20 to 198
[5], [17]	13 to 413
[31]	13 to 124
[8]	21 to 97
[32]	63 to 401
[24]	4 to 93
[21]	21 to 881
[20]	20 to 198
[40]	41 to 97
in this paper	13 to 18698

TABLE 12
Comparison of proposed algorithm with some Hierarchical algorithms, k-means and ACDC in terms of TurboMQ

Software systems	WCA-UE	Average Linkage	Complete Linkage	Single Linkage	ACDC	k-means	FCA
Compiler	0.836	0.527	0.527	0.933	1	0.85	**1.22**
Boxer	1.343	0.964	0.983	0.964	2.82	0.79	**3.020**
Ispel	1.489	1.739	1.639	0.995	1.75	1.2	**1.97**
Bison	0.994	0.994	0.994	0.994	1	1	**2.25**
Cia	0.997	0.997	0.997	0.997	1.86	1.78	**2.049**
Ciald	0.984	0.487	1.093	0.984	1.70	0.78	**1.72**
Nos	0.969	0.990	0.990	0.990	1	0.98	**1.08**
Rcs	0.977	0.990	1.018	1.018	1	1.26	**1.81**
Spdb	0.933	0.933	0.933	0.933	**5**	1.15	**5**
Star	1.388	0.989	0.805	0.989	2.09	0.81	**3.048**

TurboQ, Table 14 shows that the FCA has comparable results.

The important point is that the running time of the algorithm is much shorter than the algorithms compared. Table 15 shows the running times of the algorithms. In the folders where there are many artifacts, the difference in running time is considerable. In terms of time, the FCA has significant superiority over evolutionary algorithms. For example, the well-known MCA algorithm took about 260 hours to cluster the *content* folder, while the proposed algorithm took about 3 seconds. With the increase in the size of software systems, the performance of evolutionary algorithms slows down due to the size of their solutions, the time-consuming operators, and in some cases, memory problems. But the proposed solution only works on the matrix, which makes it less time consuming than the others.

To answer the research question RQ2, to further investigate the performance of the proposed algorithm, we have selected two large (ITK including 7,310 files) and very large software systems (Chromium including 18,698 files).

For future comparison, we selected four state-of-the-art algorithms, which all are published in IEEE Transactions on Software Engineering Journal. The algorithms selected are Bunch-SAHC, WCA-UE, WCA-UENM, and LIMBO. The Bunch-SAHC is a search-based algorithm and others are hierarchical algorithms. We also selected two famous algorithms k-means and ACDC. Due to time and memory problems, we were unable to select other algorithms from search-based methods. Table 16 shows the results in terms of TurboMQ, MoJoFM, C2C, and run time. The results show that the proposed algorithm performs better than hierarchical algorithms and can also compete with the Bunch-SAHC algorithm and ACDC. But the running time of the proposed algorithm is much shorter, which is discussed below.

For ITK and Chromium, the techniques compared take several hours to days to run. Considering the existing tested algorithms, running all experiments for Chromium would take more than a week of CPU time on a single machine or time out (TO), and the ACDC takes 10 hours for clustering. Bunch-SAHC, and LIMBO timed out after 24 and 8 days, respectively. For Bunch-SAHC, we report here the intermediate architecture recovered at that time. K-means can take varying numbers of clusters as input. For k-means, the algorithm has been executed with different values of k in steps of 5 increment, up to a specified runtime. It is important to note that it is not possible to terminate hierarchical methods in the clustering process and these algorithms must be executed until the last step. But it is possible to terminate search-based algorithms in the clustering process and report the intermediate results. The results obtained from this research question demonstrate that the proposed algorithm is scalable, and can cluster large applications in less time.

To answer the research question RQ3, Cliff's δ effect size metric is utilized. This test is a non-parametric effect size metric that quantifies the difference among two groups of observations (here FCA against other tested algorithms). The result of this metric is in range -1 to 1 and higher value shows that results of the first group (here, FCA) generally are better than the second group (other algorithms). To interpret, as [15], the following magnitudes are used: negligible ($|\delta| < 0.147$), small ($|\delta| < 0.33$), medium ($|\delta| < 0.474$), and large ($0.474 \geq |\delta|$). The results (Table 17) indicate that, in

TABLE 13
Comparison of proposed algorithm with some state-of-the-art search-based algorithms in terms of MoJoFM (M)(%) and C2C (C)(%)

Folder name	Bunch-GA		DAGC		ECA		MCA		Bunch-NAHC		SHC		GA-SMCP		EoD		ACDC		k-means		FCA	
	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C
Accessible	42	40	27	0	37	45	39	49	28	20	27	0	38	28	42	45	40	31	35	10	42	40
Browser	70	55	45	12	60	49	72	55	50	38	52	19	70	58	48	58	65	33	57	60	87	60
Build	84	69	47	29	78	61	78	61	84	61	84	61	84	61	94	77	66	33	83	50	67	55
Content	31	12	10	2	27	15	35	18	21	11	22	12	18	9	31	25	57	43	48	10	60	30
Db	94	60	50	43	96	72	96	72	94	71	94	65	91	69	91	81	96	72	95	50	95	50
Dom	58	33	25	0	52	21	53	23	40	19	38	20	58	31	58	42	82	67	47	33	83	67
Extensions	49	30	22	2	58	35	53	31	24	12	28	19	48	25	51	40	76	71	45	26	79	71
Gfx	54	38	29	11	60	49	67	53	42	28	42	22	54	21	60	31	71	36	65	20	64	26
Intl	79	69	40	15	74	65	83	68	75	68	75	61	71	65	75	62	91	79	79	59	81	73
Ipc	81	65	39	15	80	65	80	65	81	65	81	68	80	63	80	58	60	30	68	50	82	70

TABLE 14
Comparison of proposed algorithm with some state-of-the-art search-based algorithms, k-means and ACDC in terms of TurboMQ

Folder name	Bunch-GA	DAGC	ECA	MCA	Bunch-NAHC	SHC	GA-SMCP	EoD	ACDC	k-means	FCA
Accessible	6.26	0.93	**22.17**	28.98	4.80	10.01	14	29.21	12.26	0.82	17.92
Browser	3.72	0.92	4	**28.5**	5.85	9	6.5	9	11.45	0.98	7.57
Build	**3**	0.5	**3**	**3**	1	0.92	1.23	2.9	**3**	0.85	**3**
Content	6.76	0.19	**46.09**	39.40	5.41	16.97	12.01	10.11	28.05	2.69	36.66
Db	2.34	0.86	5.5	**6.9**	2.60	2.34	1.90	2.51	3.12	0.70	2.7
Dom	6.16	0.92	23.51	**79.38**	4.30	12.87	5.11	8	6.93	0.67	7.44
Extensions	11.80	0.91	24.92	**32.85**	6.66	8.90	10.99	13	7	1.8	26.62
Gfx	6.50	0.82	29.14	**70.43**	4.32	3.01	6.86	12.22	10.58	1.63	19.83
Intl	5.46	0.90	60.04	**116.63**	2.54	7.90	4.11	12.90	12.16	0.86	31.19
Ipc	5.64	0.84	17.7	18.29	3.92	4.50	5.64	10.90	**19.68**	1.63	11.41

TABLE 15
Comparison of proposed algorithm with some state-of-the-art search-based algorithms in terms of time (second)

Folder name	Bunch-GA	DAGC	ECA	MCA	Bunch-NAHC	SHC	GA-SMCP	EoD	ACDC	k-means	FCA
Accessible	4535	7471	4521	4437	101.5	869	5921	3990	0.86	4.03	**0.30**
Browser	708	609.5	733.5	796.5	4.95	12.25	901	541	0.86	0.72	**0.18**
Build	512	383.805	421.5	425	3	3.2	540	431	0.26	0.05	**0.03**
Content	950337.5	4794247.5	943040	943021	698441.5	6531479	5224315	890021	3.31	2007.9	**3.30**
Db	494.5	1834.495	1363.5	1470.5	47.4	1350.5	2301	481	1.25	0.32	**0.26**
Dom	3110	6139	3082.5	3153	110.5	101.4	6341	2208	0.79	0.95	**0.25**
Extensions	6264	7653	6461	6730	266.5	4032	6421	6259	0.36	3.24	**0.28**
Gfx	15563	28173	14781	14966	1131	2036	21540	13238	0.70	19.05	**0.51**
Intl	222888	1333765.5	223645	222884	238100.5	419513.5	1034921	22198	1.24	40	**0.82**
Ipc	62770.5	424153.5	62642.5	62683.5	1196	899.5	99101	61211	0.15	0.48	**0.11**

TABLE 16
Comparison of the proposed algorithm with other algorithms on ITK and Chromium in terms of TurboMQ (T)(%), MoJoFM (M)(%), C2C (C)(%), Time (d: day, h: hour, s: second). † Scores denote results for intermediate architectures recovered at that time.

	ITK				Chromium			
Algorithm	T	M	C	Time	T	M	C	Time
Bunch-SAHC	14	42	1	24† d	14	53	10	24† d
WCA-UE	1	33	0	16 h	1	21	0	31 h
WCA-UENM	2	31	0	18 h	1	23	0	38 h
LIMBO	10	28	0	8 d	TO	TO	TO	TO
ACDC	15	55	0	565 s	18	58	41	10 h
k-means	14	26	0	24† h	6	33	7	36† h
FCA	28	44	6	272 s	16	36	45	8 h

TABLE 17
Cliff's δ Effect Size Test. A positive value indicates that the effect size favor of the FCA. The interpretation of the effect size is indicated in parenthesis. neg. stands for negligible and med. for medium.

Algorithm	TurboMQ	MoJoFM	C2C	Time
Bunch-GA	.59 (large)	.31 (small)	.24 (small)	1 (large)
ECA	-.19 (small)	.5 (large)	.23 (small)	1 (large)
MCA	-.48 (large)	.28 (small)	.15 (small)	1 (large)
Bunch-NAHC	.87 (large)	.44 (med.)	.39 (med.)	1 (large)
Bunch-SAHC	1 (large)	-.5 (large)	.5 (large)	1 (large)
SHC	.68 (large)	.44 (med.)	.48 (large)	1 (large)
GA-SMCP	.7 (large)	.34 (med.)	.31 (small)	1 (large)
EoD	.24(small)	.36 (med.)	.05 (neg.)	1 (large)
ACDC	.19 (small)	.05 (neg.)	.1 (neg.)	.37 (med.)
k-means	.80 (large)	.26 (small)	.45 (large)	.48 (large)
WCA-UE	.91 (large)	1 (large)	1 (large)	1 (large)
WCA-UENM	1 (large)	1 (large)	1 (large)	1 (large)
LIMBO	1 (large)	1 (large)	1 (large)	1 (large)

terms of time, the FCA is better than the other algorithms. Also, the values of MoJoFM, TurboMQ, and C2C in FCA are better than the other algorithms in general.

From the short review above, the main achievements,

including contributions to the field can be summarized as follows:

1) Compared with hierarchical algorithms, the FCA results in a modularization of higher quality and is also comparable with search-based algorithms and ACDC, a state-of-the-art algorithm, in terms of the internal and external metrics.
2) Because the FCA has fewer and simpler operations than the other algorithms, it can cluster large graphs in less time and therefore it is scalable. Compared to the state-of-the-art algorithms, the proposed algorithm is the fastest software clustering algorithm.

6 Threats to validity

In this section, we discuss the threats that could affect the validity of the results obtained from the evaluation. Despite our efforts to avoid/reduce as many threats to validity as possible, some are inevitable. In the following, we address the threats to validity from two aspects of external and internal validity.

Threats to External Validity. Several factors may restrict the generality and limit the interpretation of our results. The main external threat arises from the possibility that the selected application is not representative of software systems in general, with the result that the findings of the experiments do not apply to 'typical' software systems. To address these concerns, a variety of applications with different functionalities and sizes are considered. There is, therefore, reasonable cause for confidence in the results obtained and the conclusions drawn from them.

Threats to Internal Validity. The external metrics used to the evaluation can affect the validity of the results. As [10], [15], we utilized two well-known and widely used metrics, namely MoJoFM, C2C, for the evaluation. Different metrics, such as architecture-to-architecture [15], and EdgeSim [4], may produce different results for the same software system.

Another important factor that affects the experiment results is the accuracy of the authoritative decomposition achieved from a software system. We used the package structure (directory structure) of the Mozilla Firefox as an authoritative decomposition. The expert decompositions that we selected have been used earlier in software modularization experiments in [7], [8]. We know that there is a big threat as the directory structure of a project is often different from the actual "ground-truth decomposition." In well-structured projects, the directory structure of the project originally usually reflects the architecture of the project [39]. To handle this threat, we selected the developer preview version of the Mozilla Firefox, because there is a credible (human) expert decomposition (the directory structure) of that. It is worth mentioning that the selected version is a stable version, as small changes have been made to it and its directory structure has not changed.

Isolated vertices (single vertices). These nodes have no connection to other nodes. Thus, it is not possible to assign them into specific clusters. These single vertices are one of the reasons for the discrepancy between the results of the algorithms and the expert clustering.

7 Conclusion

Given the importance of clustering in understanding and maintaining software as well as its importance for extracting software architecture, in this paper, we proposed a new style of software system clustering based on the artifact dependency graph. To this end, we proposed a clustering algorithm that works on the dependency matrix. Comparative results indicated that it performs better than hierarchical algorithms and competes with search-based algorithms in terms of TurboMQ (an internal metric) and two external metrics namely MoJoFM and C2C. The main feature of the FCA is its scalability. It can cluster very large software systems within a reasonable amount of time.

Future research should be devoted to the development of:

1) **Using the non-structural features.** The algorithm will be developed to consider some nonstructural features such as artifacts name and comments, along with structural features, in the process of software clustering.
2) **Preprocessing for determining libraries and utilities.** Some algorithms try to delete libraries and utilities before the clustering process.

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.
[2] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.
[3] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Advances in Software Engineering*, vol. 2012, 2012.
[4] A. Isazadeh, H. Izadkhah, and I. Elgedawy, *Source code modularization: theory and techniques*. Springer, 2017.
[5] B. S. Mitchell and S. Mancoridis, *A heuristic search approach to solving the software clustering problem*. Drexel University Philadelphia, PA, USA, 2002.
[6] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.
[7] S. Mohammadi and H. Izadkhah, "A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code," *Information and Software Technology*, vol. 105, pp. 252–256, 2019.
[8] N. S. Jalali, H. Izadkhah, and S. Lotfi, "Multi-objective search-based software modularization: structural and non-structural features," *Soft Computing*, pp. 1–25, 2018.
[9] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.
[10] R. Naseem, O. Maqbool, and S. Muhammad, "Cooperative clustering for software modularization," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2045–2062, 2013.
[11] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.
[12] A. Shokoufandeh, S. Mancoridis, and M. Maycock, "Applying spectral methods to software clustering," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 3–10.

[13] V. Tzerpos and R. C. Holt, "Acdc: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 2000, pp. 258–267.

[14] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 69–78.

[15] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2017.

[16] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 486–496.

[17] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[18] H. Izadkhah, I. Elgedawy, and A. Isazadeh, "E-cdgm: An evolutionary call-dependency graph modularization approach for software systems," *Cybernetics and Information Technologies*, vol. 16, no. 3, pp. 70–90, 2016.

[19] M. C. Monçores, A. C. Alvim, and M. O. Barros, "Large neighborhood search applied to the software module clustering problem," *Computers & Operations Research*, vol. 91, pp. 92–111, 2018.

[20] J. Huang and J. Liu, "A similarity-based modularization quality measure for software module clustering problems," *Information Sciences*, vol. 342, pp. 96–110, 2016.

[21] M. Kargar, A. Isazadeh, and H. Izadkhah, "Semantic-based software clustering using hill climbing," in *2017 International Symposium on Computer Science and Software Engineering Conference (CSSE)*. IEEE, 2017, pp. 55–60.

[22] S. Parsa and O. Bushehrian, "A new encoding scheme and a framework to investigate genetic clustering algorithms," *Journal of Research and Practice in Information Technology*, vol. 37, no. 1, p. 127, 2005.

[23] J. Huang, J. Liu, and X. Yao, "A multi-agent evolutionary algorithm for software module clustering problems," *Soft Computing*, vol. 21, no. 12, pp. 3415–3428, 2017.

[24] J. K. Chhabra *et al.*, "Harmony search based remodularization for object-oriented software systems," *Computer Languages, Systems & Structures*, vol. 47, pp. 153–169, 2017.

[25] A. C. Kumari and K. Srinivas, "Hyper-heuristic approach for multi-objective software module clustering," *Journal of Systems and Software*, vol. 117, pp. 384–401, 2016.

[26] M. Tajgardan, H. Izadkhah, and S. Lotfi, "Software systems clustering using estimation of distribution approach," *Journal of Applied Computer Science Methods*, vol. 8, no. 2, pp. 99–113, 2016.

[27] A. Prajapati and J. K. Chhabra, "An efficient scheme for candidate solutions of search-based multi-objective software remodularization," in *International Conference on Human Interface and the Management of Information*. Springer, 2016, pp. 296–307.

[28] J. Hwa, S. Yoo, Y.-S. Seo, and D.-H. Bae, "Search-based approaches for software module clustering based on multiple relationship factors," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 07, pp. 1033–1062, 2017.

[29] A. Ramírez, J. R. Romero, and S. Ventura, "Interactive multi-objective evolutionary optimization of software architectures," *Information Sciences*, vol. 463, pp. 92–109, 2018.

[30] M. Akbari and H. Izadkhah, "Hybrid of genetic algorithm and krill herd for software clustering problem," in *2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI)*. IEEE, pp. 565–570.

[31] H. Izadkhah and M. Tajgardan, "Information theoretic objective function for genetic software clustering," in *5th International Electronic Conference on Entropy and Its Applications*. MDPI, 2019, pp. 1–9.

[32] J. K. Chhabra *et al.*, "Many-objective artificial bee colony algorithm for large-scale software module clustering problem," *Soft Computing*, vol. 22, no. 19, pp. 6341–6361, 2018.

[33] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 552–555.

[34] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 35–44.

[35] J. Misra, K. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus, "Software clustering: Unifying syntactic and semantic features," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 113–122.

[36] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Ninth European Conference on Software Maintenance and Reengineering*. IEEE, 2005, pp. 124–133.

[37] M. Kargar, A. Isazadeh, and H. Izadkhah, "Multi-programming language software systems modularization," *Computers & Electrical Engineering*, vol. 80, p. 106500, 2019.

[38] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 901–910.

[39] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 525–535.

[40] I. Hussain, A. Khanum, A. Q. Abbasi, M. Y. Javed *et al.*, "A novel approach for software architecture recovery using particle swarm optimization." *Int. Arab J. Inf. Technol.*, vol. 12, no. 1, pp. 32–41, 2015.

[41] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.