Research Article

A New Metaheuristic-Based Hierarchical Clustering Algorithm for Software Modularization

Masoud Aghdasifam, Habib Izadkhah, and Ayaz Isazadeh

Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran

Correspondence should be addressed to Habib Izadkhah; izadkhah@tabrizu.ac.ir

Received 7 May 2020; Revised 10 July 2020; Accepted 20 July 2020; Published 30 September 2020

Guest Editor: Narayanan Kumarappan

Copyright © 2020 Masoud Aghdasifam et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Software refactoring is a software maintenance action to improve the software internal quality without changing its external behavior. During the maintenance process, structural refactoring is performed by remodularizing the source code. Software clustering is a modularization technique to remodularize artifacts of source code aiming to improve readability and reusability. Due to the NP hardness of the clustering problem, evolutionary approaches such as the genetic algorithm have been used to solve this problem. In the structural refactoring literature, there exists no search-based algorithm that employs a hierarchical approach for modularization. Utilizing global and local search strategies, in this paper, a new search-based top-down hierarchical clustering approach, named TDHC, is proposed that can be used to modularize the system. The output of the algorithm is a tree in which each node is an artifact composed of all artifacts in its subtrees and is a candidate to be a software module (i.e., cluster). This tree helps a software maintainer to have better vision on source code structure to decide appropriate composition points of artifacts aiming to create modules (i.e., files, packages, and components). Experimental results on seven folders of Mozilla Firefox with different functionalities and five other software systems show that the TDHC produces modularization closer to the human expert’s decomposition (i.e., directory structure) than the other existing algorithms. The proposed algorithm is expected to help a software maintainer for better remodularization of a source code. The source codes and dataset related to this paper can be accessed at https://github.com/SoftwareMaintenanceLab.

1. Introduction

Software maintenance is the process of modifying a software product after releasing it to reduce faults, improve performance, or improve the design. Software maintenance tasks are important for future software development and consume approximately 90 percent of the total cost [1].

In software maintenance, some changes including add, delete, or modify code lead to the growth of code blocks and difficulty in code’s understandability in the future. Code smells (or bad code smells) are part of source code which do not cause faults on external behavior and do not have a significant problem in internal behavior at this moment but may cause issues in the future development process [2]. Software refactoring is modifying the source code to rectify code smells without any change in the external behavior of the system. It improves the quality of software source code by reducing the potential occurrence of bugs and keeping the code easier to maintain or extend in the future.

Fowler et al. reported some possible code smells in their book [3] for object-oriented programming-based systems and proposed possible refactoring scenarios for them. Since then, many studies have been done to propose new refactoring scenarios or validating effects of applying various scenarios in the source code to achieve better quality.

Refactoring techniques are classified into two major conceptual and structural groups. For example, rename method refactoring is a conceptual refactoring scenario that changes the name of a method for a better explanation of its responsibility. Some structural refactoring scenarios are about methods or functions composing. For example, long code blocks usually have multiple responsibilities or
duplicate blocks that should be refactored. Some other structural refactoring scenarios are to improve the functionality of code blocks. As an example, move method refactoring (MMR) is a refactoring scenario that is defined as the act of moving a method from one class to another class which has the most relation with that method. The relation between methods can be structural relations like calls or semantic relations. There are also some composite refactorings that are defined as a sequence of primitive refactorings that reflect complex transformations.

To illustrate a structural refactoring task, Figure 1 depicts an example modularization for a small software system. In this figure, each node is a class and edges represent a collaboration between the classes. These classes are separated into two modules according to their collaborations. Figure 2 shows several changes on this software after some maintenance actions. As shown, the relations between nodes are changed and also a new class "I" is added to the system. In Figure 2, relations of the node "G" with the nodes in the left module are more than relations in the right module. So it is necessary to relocate the position of this node (and node "I") by a remodularization. The result of remodularization is shown in Figure 3.

Manually analyzing the source code to refactoring is a costly and time-consuming process. Hence, many researches have been done about automatic refactoring. One approach for structural refactoring is remodularization, as shown in Figure 2, so that the remodularization is performed by clustering techniques. According to [4], "The aim of the software clustering process is to partition a software system into modules (subsystems or packages), where a module is made up of a set of software artifacts which collaborate with each other to implement a high-level attribute or provide a high-level service for the rest of the software system." The input of a clustering algorithm is artifact dependency graph (ADG), where the nodes of this graph indicate artifacts and the edges show the relationships between artifacts. An artifact can be an entity such as a function, a file, a software class, or even a collection of classes so-called package or files in a source code folder. The relation between artifacts can be created from structured features like calls or nonstructured features like semantic relations. Figure 4 shows an example of clustering in which artifacts of a small compiler are partitioned into four modules (clusters) according to their relations. These modules are expected to have maximum cohesion and minimum coupling with other modules [6, 7].

Current clustering strategies for obtaining proper modularization are based on two major hierarchical or nonhierarchical techniques. In hierarchical methods, a tree of relations is constructed from the artifacts at the leaf to the root. These techniques give developers a hierarchical view for decision-making about the number and appropriate cutpoint in a tree to construct modules. Most presented hierarchical methods for software clustering are agglomerative (bottom-up). In such algorithms, each artifact starts in its own cluster; based on certain criterion, e.g., Jaccard, the proximity is calculated between all clusters, and pairs of clusters with the highest proximity are merged as one moves up the hierarchy [8]. The main limitations of hierarchical algorithms are as follows [8]:

1. Due to the presence of zigzag, to identify modules, it is necessary to make the whole tree to the end.
2. There exists no well-defined criterion to decide where the clustering process should stop.
3. Arbitrary decisions are one of the main problems in hierarchical clustering methods. These decisions have a magnificent impact on the final clustering. When faced with arbitrary decisions and a wrong choice, there is no possibility of reversing and correcting wrong choices.
4. These algorithms are greedy and hence cannot explore the problem space well. Several previous studies [9–11] have shown these methods do not perform well in software clustering. On the contrary, there is no hierarchical clustering algorithm that proposes cut points from different levels of the dendrogram.

There are also nonhierarchical modularization methods based on search-based approaches which explore solution space by global search or local search algorithms. But these methods do not give the developer a vision about upper-level relationships between modules.

In the literature, because of the NP hardness of clustering problem, search-based methods (such as genetic algorithm) have been widely used [8, 12]. Because of their exploration and exploitation ability, they are an effective way to solve the clustering problem [13]. Currently, search-based works on
software refactoring with remodularization approaches are
in the flat mode (i.e., nonhierarchical methods) and do not
offer appropriate composing at higher levels.

1.1. The Problem. In this paper, we focus on a specific
restructuring problem in the context of object-oriented
and procedural programs: given an ADG constructed
from an existing code, decompose it into smaller and
meaningful modules that have a higher cohesion and
lower coupling. Cohesion is defined as "the degree to
which the internal contents of a module are related" [1].
Our method supports "big-bang" remodularization; i.e.,
all the artifacts of the software system are considered for
remodularization.

The main problem addressed in this paper is to suggest a
possible hierarchical remodularization for a source code,
while keeping that accurate in terms of proximity to (hu-
man) expert decomposition. In this paper, a hierarchical
top-down clustering algorithm is proposed to structurally
refactor the source code from its artifact dependency graph
(ADG) with a branch and bound approach. The aim is to
find the appropriate composition tree and recommend the
lowest appropriate levels to merge artifacts as a module. It,
therefore, will be easier for the developer to recognize the
position of the different levels, such as files, packages, or
components. In the proposed method, a genetic algorithm
(GA) along with a neighboring search algorithm is designed
to search in trees of the composition of artifacts. The pro-
posed algorithm is evaluated on seven folders of Mozilla
Firefox and five other open-source systems. The results
indicate that the method is able to propose an acceptable
refactoring by hierarchical remodularization of artifacts, by
giving a vision about high-level relation between modules
for developers.

1.2. Contribution. The contributions of this paper are
summarized as follows:

(1) Proposing a new software refactoring method with a
top-down hierarchical modularization technique. The
output of the algorithm is a tree generated from
source code which helps software maintainer to have
better vision on source code structure to decide
appropriate composition points of artifacts aiming to
create modules (i.e., files, packages, and compo-
nents). It is important to note that, in the literature,
there exists no search-based algorithm that employs
a hierarchical approach for modularization.

(2) Prüfer sequence is utilized in GA for encoding tree.
Existing encoding methods used in software mod-
ularization are real-based (e.g., BUNCH [5], ECA
[12], and SGA [14]) or permutation-based (e.g.,
DAGC [15] and E-CDGM [16]), in which these
methods show only a flat modularization.

(3) A new objective function is proposed to evaluate
hierarchical remodularization.

The rest of paper is as follows: in Section 2, some research
studies on software refactoring are discussed; Section 3
introduces the proposed algorithm, and in Section 4, ex-
perimental results are presented. The result of research and
threats to validity are discussed in Sections 4 and 5, re-
spectively. Finally, Section 6 is conclusions of this research
and future work.
2. Related Work

After publishing Fowler’s book [3] on software source code refactoring, many studies have been done to refine the concepts of this reference, as well as an automated solution for detecting and repairing code smells, e.g., [17–20].

Remodularization of source code artifacts is an approach for structural refactoring. Due to the large space of the solution space for modularization, many search-based research studies have been done. In Bunch algorithm [5, 7, 21], a GA, namely, Bunch-GA, and two hill-climbing algorithms, namely, Bunch-NAHC and Bunch-SAHC, are utilized to search in solution space. In this algorithm, the space size of solutions is \(n^m\) (\(n\) is the number of artifacts), in which most of them represent the same modularization. Parsa and Bush-eiran introduced DAGC coding [15] to solve this problem, which reduces the space of states to \(n!\). Tajgardan et al. [22] presented an algorithm based on estimation of distribution algorithm (EDA) which does not have the challenge of specifying the parameters of GA algorithms. Izadkhah et al. [16] presented E-CDGM method that at first converts the source code to an intermediate code called mCode from call dependency graph (CDG) and then proposes a modularization with a fitness function (using class-property, class-method, and method-method relations) and self-automata algorithm and DAGC encoding. Amarjeet et al. [23] presented the MaABC algorithm for software modularization which is a multiobjective optimization method using the bee population algorithm. They also presented PSOMC [24], a PSO-based module clustering, which partitions software system by optimizing intracluster dependency, intercluster dependency, number of clusters, and number of modules per cluster.

Recent research on multiobjective search methods has expanded. Praditwong et al. [12] presented two equal-size cluster approaches (ECA) and the maximizing cluster approach (MCA) for software modularization using a multiobjective genetic algorithm and Pareto optimality. Harman and Tratt [25] also had used Pareto optimality to combine two metrics: CBO [26] and a new metric called SDMPC. Seng et al. [27] proposed a GA-based approach to suggest refactorings by a fitness function composing of coupling, cohesion, complexity, and stability. Kebir et al. [28] presented a genetic algorithm-based approach, which consists of detecting component-relevant code smells and eliminating these code smells by searching for the best sequence of refactorings using a genetic algorithm. In [29], Kumari and Srinivas proposed MHypEA (multiobjective hyperheuristic evolutionary algorithm) to suggest software module clusters while maximizing cohesion and minimizing coupling of the software modules. It is based on different methods of selection, crossover, and mutation operations of evolutionary algorithms, and the selection mechanism to select a low-level heuristic is based on reinforcement learning with adaptive weights.

In [30], Huang and Liu introduced a new objective function called MS to automatically guide optimization algorithms to find a good partition of software systems which consider both global modules and edge directions. Then, three modularization algorithms named HC-SMCP, GA-SMCP, and MAEA-SMCP are proposed in this paper which are adopted to optimize MS for software systems.

Bavota et al. have some researches on refactoring. In [31], a new technique is proposed for automatic remodularization of packages, which use structural and semantic measures to decompose a package into smaller, more cohesive ones. The results showed that the decomposed packages have better cohesion without deterioration of coupling, and the remodularization proposed by the tool is also meaningful from a functional point of view. In [32], they introduced a tool called R3 that automatically analyzed the underlying latent topics inferred from identifiers, comments, and string literals in the source code classes as well as structural dependencies among these classes. They presented [33] a method for extract class refactoring based on three SSM [34], CDM [35], and CSM [36] structural and semantic factors which strongly increase the cohesion of the refactored classes without leading to significant increase in terms of coupling. In [37], they proposed a technique based on relational topic models to identify MMR opportunities.

Maletic and Marcus [38] proposed an algorithm which uses semantic and structural data to propose refactoring decisions. In [39], Palomba et al. presented a technique, called TACO (textual analysis for code smell detection), that exploits textual analysis to detect a family of smells of different natures and different levels of granularity.

Jalali et al. [8] proposed a new multiobjective fitness function for modularization, named MOF, which uses the structural and nonstructural features with EoD algorithm. In [40], a new deterministic clustering algorithm named neighborhood tree algorithm is presented which creates a neighborhood tree using available knowledge in an ADG. Mahouachi [41] proposed a method which used NSGA-II [42] to find the best sequence of refactorings that maximize structural quality, maximize semantic cohesiveness of packages, and minimize the refactoring effort that is able to produce a coherent and useful sequence of recommended refactorings both in terms of quality metrics and from the developer’s points of view. Ouni et al. [43] proposed a new refactoring recommendation, called MORE, to improve design quality and fix code smells using NSGA-III [42]. Dallal [44] introduced a measure to precisely predict whether a class includes methods in need of MMR. Me et al. [45] presented a new mathematical programming model for the software remodularization problem with a novel metric based on the principle of complexity balance and a hybrid genetic algorithm (HGA).

Kargar et. al have some research studies on the remodularization of multiprogramming language software systems. In [14], they have presented two dependency graphs called semantic dependency graph (SDG) and nominal similarity graph (NSG). Both of these graphs are constructed independently of programming languages syntax. The SDG is constructed based on all nouns of the source code, and the NSG is constructed based on the similarity between artifact names. Then, in [46], they proposed a genetic algorithm to modularize programs by combining the constructed dependency graphs (i.e., call complexity...
Complexity

dependency graph, semantic dependency graph, and nominal similarity graph).

In summary, search-based algorithms are described in three aspects. One aspect is the scope of the search (local strategy and global strategy). Some algorithms are based on local search strategy, and the result may not be the optimal solution. Global search techniques always aim to find good solutions. Single objective or multiobjective is another grouping for search algorithms. In multiobjective algorithms, there are multiple functions or metrics aiming to guide the search process. The last aspect is to use semantic features vs structured features for clustering. In semantic search optimizations, lexical analysis or latent semantic analysis (or both) is considered in search progress. In structural features, the function call between two artifacts, inheritance, etc. is considered for clustering. Some search-based clustering algorithms are shown in Table 1.

In the hierarchical methods, all the artifacts are initially considered as units of modularization, and during a repetitive process, the more similar modules are merged to create a new module. Single-linkage, complete-linkage, and average-linkage algorithms are most common hierarchical clustering algorithms which Maqbool et al. adapted to modularize source codes [59]. Kuhn et al. proposed a new algorithm using the average linkage that used nonstructural features for modularization [60]. The authors of this paper have used program code property attributes and variables’ naming for communication recognition, which makes the output of the algorithm dependent on the level of knowledge of developers in inserting descriptions and naming variables. Andritsos and Tzerpos introduced a method called LIMBO [61] as a hierarchical algorithm combining structural and nonstructural information. This algorithm is a hierarchical sampling algorithm based on minimizing the loss of information during the modularization of a software system. Rathee et al. [62] proposed a new hierarchical technique of software remodularization by estimating conceptual similarity among software artifacts that uses both structural and semantic coupling measurements together to get much more accurate coupling measures. They also presented a new weighted dependency measurement scheme in which combined structural, conceptual, and change history-based relations are among software elements together.

In addition to the search-based and hierarchical methods discussed above, there are a number of graph-based and pattern-based methods. Mohammadi and Izadkhah in [40] 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 less than evolutionary methods. Spectral values [63] use algebraic properties of the graph, such as eigenvalues and eigenvectors in the corresponding Laplacian matrix to perform clustering. Algorithm for comprehension-driven clustering (ACDC) [64] is a pattern-based algorithm that was introduced by Tzerpos and Holt. It uses several patterns to cluster code artifacts.

2.1. Gaps in the Literature. Using hierarchical property is not practically new and has been used for many years in the remodularization field, but there is no previous research using the hierarchical property with an evolutionary approach for remodularization. Due to the NP hardness of the modularization problem, most modularization methods utilize search-based clustering methods and evolutionary algorithms [8, 12]. These clustering algorithms show only a flat modularization of a program. Therefore, these algorithms cannot represent the hierarchy properties of a program, so there is no way to specify the encapsulation levels, e.g., module, package, and component, in it by the designer.

3. The Proposed Clustering Algorithm

Most of the work on remodularization is based on clustering techniques [31]. Hierarchical clustering algorithms proposed up to now are greedy algorithms and have arbitrary decisions that may lead to undesired results. On the contrary, these algorithms do not recommend an appropriate cutpoint in the dendrogram or modularization point from different levels of it. In this section, a new clustering algorithm with a hierarchical approach is proposed for source code remodularization which does not have these problems. To this end, we design a genetic algorithm with a new encoding and fitness function. The encoding presented is utilized to construct a tree from source code’s artifacts and the fitness function with a branch and bound approach is applied to determine appropriate levels in the constructed tree, which result can be a qualified modularization. To improve the quality of the resulting modularization, we also designed a hill-climbing algorithm. This local search algorithm will be applied on the outcome of the genetic algorithm for a neighboring search. The algorithm’s input is an ADG constructed from source code, and its output is a modularization suggested for software maintainer. Our method supports “big-bang” remodularization; i.e., all the artifacts of the software system are considered to perform modularization, and the current structure (modularization) will not be considered.

We consider classes and files are the smallest composing unit as an artifact to perform modularization in object-oriented and structured software systems, respectively. These parts are combined in larger modules such as packages or components in which members of each module are contributing to other parts of that module for a single responsibility. Hence, it is important to have proper upper-level compositions. We, also, consider call dependency to create a dependency between two artifacts, i.e., edges, in the ADG. Some artifacts that just are called by other artifacts are utility classes or files. So, they can be removed at the beginning and address them after completion of the algorithm. For each one, if all calls are from one module, this artifact will also be added to that module. But if it was used by multiple modules, it is considered as a utility.

To design a genetic-based algorithm, five features encoding (chromosomal representation), fitness function (evaluation), selection, crossover, and mutation must be described.

3.1. Encoding. A chromosome in GA is a parameter collection that represents a solution to the problem. The aim of GA is to find a chromosome with an optimal or near-optimal
solution. These parameters can be a binary string or any other data structure. In this paper, the Prufer sequence [65] is employed to encode the tree to a sequence of numbers as a chromosome. Prufer sequence is a one-to-one mapping between a sequence of numbers and a labeled tree. The steps of constructing Prufer numbers for a tree are shown as Algorithm 1. Let $p_1, p_2, p_3, \ldots, p_{n-2}$ denote Prufer sequence. The corresponding tree of a Prufer sequence is constructed as Algorithm 2.

For example, the Prufer sequence for the tree in Figure 5 is 2, 1, 3, 3, 1, and vice versa. To encode the tree to a Prufer sequence, the node with label 4 (as a leaf node with the smallest number) is removed and number 2 is added to the sequence. Then, the node labeled 2 is removed and number 1 is added to the sequence. In the two next steps, the nodes 5 and 6 are removed and number 3 is added to the sequence twice. In the final step, node 3 is removed and number 1 is added to the sequence as the last number of the sequence.

In the proposed method, trees are binary tree, and Prufer sequences follow the following rules:

1. The trees always have $n$ leaves numbered from 1 to $n$ for artifacts and $n-1$ inner nodes numbered from $n+1$ to $2n-1$.

2. All the artifacts are in the leaves of the tree in which degree is one. Hence, numbers 1 to $n$ do not appear in the corresponding Prufer sequence.

3. The root of the tree (node number $2n-1$) is in degree 2, and according to rules of creating the Prufer sequence, it appears only once in the sequence.

(4) All inner nodes except root are in degree 3 (attached to their parent nodes and have two children nodes) and appear twice in the sequence.

Hence, each sequence of numbers $n + 1$ to $2n - 1$ which contains $n + 1$ to $2n - 2$ times and number $2n - 1$ has appeared once and represents a hierarchical modularization tree in this algorithm. Figure 6 shows corresponding hierarchical modularization tree for 8, 6, 8, 7, 9, 7 Prufer sequence.

3.2. Evaluation. Each chromosome in the population of a GA should be evaluated to determine the quality of solutions. In the following, we propose a new quality function to evaluate the chromosomes. In the proposed quality function, the fitness of a chromosome is calculated by using the dependencies between modules extracted from the corresponding tree of the chromosome. Let $\mu_i, \lambda_i$, and $\epsilon_i$ represent the number of connections between the artifacts inside the node (module), the number of connections with the artifacts in the sibling node, and the number of connections with other artifacts, respectively. The fitness of node (i.e., a module) is calculated by $\text{exCF}$ in the following equation:

$$
\text{exCF}(\mu_i, \lambda_i, \epsilon_i) = \begin{cases} 
0, & \text{if } \mu_i + \lambda_i + \epsilon_i = 0, \\
-1, & \text{if } \lambda_i < \epsilon_i, \\
\frac{2\mu_i}{2\mu_i + \lambda_i + \epsilon_i}, & \text{otherwise.}
\end{cases}
$$

### Table 1: Some search-based software clustering algorithms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type of algorithm</th>
<th>Type of objective function</th>
<th>Structural/semantic features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunch-NAHC and Bunch-SAHC [7, 21]</td>
<td>LS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>Multiple hill-climbing approach [47]</td>
<td>LS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>E-CDGM [16]</td>
<td>LS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>Large neighborhood search [48]</td>
<td>LS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>HC-SMCP [30]</td>
<td>LS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>SHC [49]</td>
<td>LS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>Bunch-GA [5]</td>
<td>GS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>DAGC [15]</td>
<td>GS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>A multiagent evolutionary algorithm [50]</td>
<td>GS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>Harmony search [51]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>GA-SMCP [30]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>Hyperheuristic approach [52]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>ECA and MCA [12]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>Estimation of distribution approach [22]</td>
<td>GS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>EoD, CGH, CGoH [8]</td>
<td>GS</td>
<td>MO</td>
<td>Both S and S</td>
</tr>
<tr>
<td>Search-based multiobjective software remodularization [53]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>Multiple relationship factors [54]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>Interactive evolutionary optimization [55]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>GAKH [56]</td>
<td>GS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>MaABC [57]</td>
<td>GS</td>
<td>MO</td>
<td>S</td>
</tr>
<tr>
<td>HGA [45]</td>
<td>GS</td>
<td>SO</td>
<td>S</td>
</tr>
<tr>
<td>ILOF [58]</td>
<td>GS</td>
<td>Support SO and MO</td>
<td>Both S and S</td>
</tr>
</tbody>
</table>

LS: local search; GS: global search; SO: single objective; MO: multiobjective; S: structural; Se: semantic.
This relation aims to increase cohesion in a module and reduce coupling with other modules. But coupling is separated into two types of sibling coupling ($\lambda_i$) and external coupling ($\epsilon_i$). When external relation is more than relations to the sibling node in the tree, this module (regardless of cohesion) is not in proper position and should be scored with a negative value. When $\epsilon_i$ is greater than $\lambda_i$, connections with artifacts in the sibling node is less than connections.
with the other artifacts and this shows that the artifact is not in appropriate position and we should give a penalty to total score by assigning -1 this node. Algorithm 3 shows the pseudocode of the evaluation part of this customized genetic algorithm. To evaluate the tree and propose modularization according to the structure of it, the tree is traversed by the breadth-first search (BFS) algorithm from the root. In traversing, if the sum of exCF for two child nodes is greater than or equal to the exCF of that node, they will be added to the process queue. If not, this node is the lowest appropriate position to compose artifacts in leaves of that subtree, as a module. When a node is partitioned into two child nodes, if \( \lambda_i < \varepsilon \), for one of the child nodes, that node cannot be part of the tree because its outer relation is more than inner relations with its sibling node. In this case, exCF is equal to -1, and child nodes will not be added to the BFS process queue. The total fitness of the tree, exTMQ, is calculated by (2), where \( K \) is the set of all nodes in which their children (if exists) did not proceed:

\[
\text{exTMQ} = \sum_{i \in K} \text{exCF}_i, \tag{2}
\]

Figure 7 shows an example of a tree evaluating in this algorithm. This tree has 55 nodes (28 nodes for artifacts in the leaves and 27 inner nodes numbered from 28 to 53) in which numbers in parentheses are exCF for each node. When the evaluation starts, the nodes numbered 37 and 42 are the leaves and 27 inner nodes numbered from 28 to 53) in parents exCF. Child nodes did not add to the BFS queue because the sum of position proposed by the algorithm to create modules. Sffl_his tree is traversed until the nodes in the set \( \{29, 42, 47, 52\} \) (colored in grey). Each of these nodes contains all artifacts in the leaves of its subtree and is the first position proposed by the algorithm to create modules. Their child nodes did not add to the BFS queue because the sum of exCF of sibling nodes is not greater than or equal to the parents exCF.

The three operations of GA for this algorithm are described as follows:

(1) Selection: to select the next generation of the population in GA generations, the classic roulette wheel selection operator is used in the proposed algorithm.

(2) Crossover: cycle crossover operation (CX) [66] is selected for this algorithm, which finds a genes cycle between two parents and swaps other genes. If \( C_1, C_2, C_3, \ldots, C_n \) and \( C_2, C_1, C_3, \ldots, C_n \) represents two parents, at first, one random position \( i \) is selected. If \( C_1 \) and \( C_2 \) are different, one of the locations of value \( C_2 \), in the first parent is selected, and this new position is added to the selected position lists. These selections continue until selecting a position like \( j \) in which \( C_2_j = C_1_j \). When finished, the values of the selected positions in the first parent are a permutation of values in the same positions of the second parent. Finally, the values of all other positions (unselected) swapped between two parents. Figure 8 shows an example of the crossover operation. In this example, the first position is selected randomly and then third and fourth positions are added to the selection list, respectively, to create a cycle. Values 6, 9, and 8 in the first parent are a permutation of 8, 6, and 9 in the second. In the last step, values in the other positions are swapped with the corresponding position in the other chromosome.

The output of CX is a permutation of the input. Hence, it does not disrupt the rules mentioned in Encoding section. However the structure of the tree (relationships between nodes) will be changed.

(3) Mutation: single swap operation is used for mutation of a chromosome in which the value of two random positions in the sequence is swapped. Figure 9 shows an example of single swap operation on a Prufer sequence. This change creates a new binary tree.

3.3. Neighboring Search. A genetic algorithm is a global search. To improve the resulting modularization quality at the last step of the GA, we design a hill-climbing local search strategy. The designed local search algorithm tries to produce a neighboring modularization for the resulting modularization with better quality. This operation is continued until no better modularization can be found. We used the steepest ascent strategy for searching neighboring modularizations. In this strategy, all neighboring modularizations for a specific modularization are generated, and then among them, the highest quality modularization is selected as the neighbor of the current modularization and replaces it. This operation for the new modularization is continued until no better modularization can be found. How to define a neighborhood is very important in the climbing algorithm. Depending on the type of problem, it is necessary to define the appropriate neighborhood with it.

3.4. Definition: Neighbor of a Modularization. Let \( M \) and \( M' \) be two modularizations from an ADG. Modularization \( M' \) is called a neighbor of modularization \( M \) if an artifact into module \( i \) in modularization \( M \) is moved to module \( j \). In fact, two modularizations are called neighbors if they differ only in the position of a node. Let \( G = (V, E) \) be a dependency graph, where \( V = \{v_1, v_2, \ldots, v_n\} \) represents artifacts and \( E \subseteq V \times V \) is a dependency between artifacts. For example, Figure 10 depicts a sample modularization and Figure 11 shows a neighbor modularization for that. The formal definition of this concept is as follows.

Let \( M = \{m_1, m_2, \ldots, m_k\} \) represents the \( k \) modules obtained for graph \( G \) such that \( \forall i, j \in \{1, \ldots, k\}, m_i \in V, m_i \cap m_j = \emptyset, m_i \neq \emptyset \). In \( M' = \{m_1', m_2', \ldots, m_k'\} \), let us take a node \( v \) such that \( v \in m_i \). The neighbor \( M' = \{m_1', \ldots, m_i', m_j, \ldots, m_k'\} \) is created such that \( m_i' = m_i \setminus \{v\} \) and \( m_j = m_j \cup \{v\} \), where \( m_j \) (\( j \neq i \)) is a module with at least one relation to \( v \). Now, \( M' \) is better than \( M \) if \( \text{exTMQ}(M') > \text{exTMQ}(M) \).

In the following, we compute the time complexity of the algorithm. Let \( N, P, \) and \( G \) represent the number of artifacts, population size, and the number of generations, respectively. We have the following:
To initiate the population, a chromosome with length $2N - 1$ is generated in which all numbers between 1 and $N - 1$ are repeated twice and one $N$.

Sffl_hen, for each chromosome, a shuffle (replacing each genome with a random one) is applied on it to

**Input**: a chromosome as ch

**Output**: update ch.fitness

ch.fitness = 0

tree ← decode the chromosome ch to tree

$q ← Queue()$

tree.root.cf = 1

q.push(tree.root)

**while** q is not empty **do**

parent ← q.pop()

child$_1$, child$_2$ ← t.children()

child$_1$.cf ← exCF(child$_1$, parent)

child$_2$.cf ← exCF(child$_2$, parent)

if child$_1$.cf + child$_2$.cf ≥ parent.cf **then**

q.push(child$_1$)

q.push(child$_2$)

else

ch.fitness ← ch.fitness + parent.cf

**end if**

**end while**

**Algorithm 3**: Pseudocode of chromosome evaluation.

![Figure 7](image7.png)

**Figure 7**: An example of applying branch and bound technique on a modularization tree (see Algorithm 2).

**3.3. Genetic Operations.**

![Figure 8](image8.png)

**Figure 8**: An example for crossover operation of TDHC.

![Figure 9](image9.png)

**Figure 9**: An example for mutation operation of TDHC.

(1) To initiate the population, a chromosome with length $2N - 1$ is generated in which all numbers between 1 and $N - 1$ are repeated twice and one $N$. Then, for each chromosome, a shuffle (replacing each genome with a random one) is applied on it to

![Figure 10](image10.png)

**Figure 10**: An obtained modularization.
generate a new random chromosome. So, the order of this step is $O((2N - 1) \times P) = O(NP)$.

(2) To evaluate the chromosome, the data are converted to a tree in $O(N \log N)$, and then, the tree is explored in $O(N)$. Hence, the order of evaluation is $O(N \log N + N) = O(N \log N)$.

(3) Selection step with roulette wheel is in order $O(N)$.

(4) The crossover for each pair will be in $O(N)$, and the mutation is a simple swap in order $O(1)$. So, this step for whole population will be in order $O(P(N + 1)) = O(PN)$.

Steps 2–4 will be repeated $G$ times. Hence, the total order is

$$O(NP + GN \log N + GN + GPN) = O(GN^2).$$

In the last step, a NAHC algorithm is applied to search in neighbors for better solution. Each solution will have at most $(N/2)$ clusters, and each iteration of NAHC will cost $O(N \times (n/2)) = O(N^2)$. So, for $M$ iteration, it will be $O(MN^2)$.

According to the paragraphs above, the total order is $O(GN^2 + MN^2)$, but, in practice, $M$ is a small number and the total order can be explained by $O(GN^2)$.

4. Experimental Setup

In this section, we outline in detail the experimental setup we carried out to empirically assess the proposed clustering algorithm.

4.1. Case Study. Mozilla Firefox, a web browser, is a large-scale and open-source application developed by the Mozilla Foundation and its subsidiary Mozilla Corporation. Based on open hub (http://www.openhub.net) report, this application is the most popular project among other open-source applications, and this application has the largest development teams in the world, more than 13000 developers. We select the Mozilla Firefox 3.7, a developer preview version, for the experiments (https://ftp.mozilla.org/pub/). This version is stable and has approximately five million lines of code. Seven folders with different sizes and functionalities are chosen from this software system. Details of these folders are listed in Table 2. Besides, five medium-size open-source software has been chosen whose details are given in Table 3. In all experiments, the file is considered as an artifact.

The authoritative decomposition (domain expert decomposition or ground-truth structure) is utilized to evaluate the soundness of a remodularization algorithm [67]. The proximity of the remodularization generated by an algorithm to the decomposition given by a domain expert shows the acceptable achievement of the remodularization algorithm [67]. Like in [14, 67], we use the directory structure to prepare an expert decomposition from source code. In this paper, we used Mozilla Firefox and five other software systems, whose authoritative decomposition (i.e., directory structure) is there to assess the proposed algorithm. For example, the “extensions” folder has 179 files that have been assigned by Mozilla Firefox developers to 13 subfolders (package). Using a designed tool, we merged the files in the different folders in a single folder $F$, aiming to consider these 179 as flat. After modularizing the flatted files $F$, the aims to measure how much modularization achieved by the proposed algorithm will be similar to the directory structure implemented by Mozilla Firefox developers. In other words, the proposed algorithm is applied to the $F$ in order to reconstruct (or improve) the original structure.

4.2. Research Questions. To evaluate the effectiveness of TDHC, we answer the following research questions:

RQ1. Does the proposed clustering approach produce modularization having a better precision, recall, F-measure, MoJo, and MoJoFM compared to existing approaches?

RQ2. Is TDHC a stable algorithm?

RQ3. By using TDHC, can we give better view of hierarchical modularization?

To answer these research questions, five software systems and the seven folders of Mozilla Firefox are remodularized by the proposed clustering algorithm and some other available clustering algorithms.

4.3. Algorithmic Parameters. The setting of parameters is necessary for search-based algorithms. We obtained the implementations of five of the selected clustering techniques—ACDC (https://wiki.eecs.yorku.ca/project/cluster/protected/acdc), Bunch (https://www.cs.drexel.edu/spiros/bunch/) (SAHC and GA), SGA and SNDGA (https://github.com/Masoud-Kargar-QIAU), and EoD—from their original authors or official web sites. On the contrary, we got the working implementations of DAGC, ECA, and MCA from https://github.com/Masoud-Kargar-QIAU.

Values of crossover and mutation rates affect exploration and exploitation of solution space during the evolutionary process. Adding one extra artifact to the input of this problem will add two genes to chromosomes. Hence, the problem space grows exponentially. So the crossover and mutation rates are set dynamically based on population to cover the solution space better. Crossover rate usually is

![Figure 11: A neighbor for modularization Figure 10.](image)
selected as a number more than 0.7, and the mutation rate is usually very low. In this research, the numbers 0.7 and 0.9 are selected as boundaries to crossover with linear steps. Because mutation steps are with log, it should not increase much. Table 4 shows the parameters setting for TDHC, in which \( N \) is the number of artifacts after the preprocessing operation. For the TDHC, we followed the algorithmic parameters setting used in [12, 30]. Algorithmic parameters are dependent on the number of artifacts (\( N \)).

As in [8, 12, 14], to reduce randomness in the results of our experiments, we collect the average and best of 30 independent runs. To perform a fair comparison, the average of runs is used, and to determine the performance of an algorithm, the best value of runs is utilized.

### 4.4. Assessment of Results

The comparison has been performed by comparing modules in the leaves of solution tree by modules in the source code (which is developed by the expert team) using precision/recall [4] and MoJoFM [68] and F-measure [4] metrics. The precision/recall metric is used to compare the modularization obtained by the
proposed algorithm against expert modularization by (3) in which TP (true positive) is the number of comodules that are relevant (appeared in the original modularization) and were retrieved correctly by the algorithm, FP (false positive) is the number of comodules that are irrelevant but were retrieved, and FN (false negative) is the number of comodules that are relevant but were not retrieved. F-Measure is defined as the harmonic mean of the precision and recall (4). A high value for precision/recall and F-measure shows more similarity between two modularizations:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Let mno denotes the number of move or join operations in which one modularization can be transformed to another. The MoJoFM between extracted modularization $\hat{F}$ and original modularization $F$ is calculated with the relationship shown in (5). A high value for MoJoFM shows more similarity between two modularizations:

$$\text{MoJoFM}(\hat{F}, F) = \left(1 - \frac{\text{mno}(\hat{F}, F)}{\max(\text{mno}(V\hat{F}, F))}\right) \times 100\%.$$  

To compare the overall results of TDHC against other tested algorithms in terms of precision/recall, F-measure, and MoJoFM, we utilized a nonparametric effect size statistic, namely, Cliff’s $\delta$ which is used to quantify the amount of difference between two algorithms.

With having different results of algorithms on different criteria, and considering all criteria, deciding which algorithm performs well is not easy. In such circumstances, multicriteria decision-making (MCDM) can be utilized [69]. This technique measures the performance of various algorithms and assigns to each algorithm a value between zero and one, where zero indicates the weakest performance and one indicates the best performance. To this end, let $n$ and $m$ denote the number of algorithms and the number of criteria, respectively. A matrix, called $X_{nm}$, is created, and then based on entropy, the efficiency of each algorithm is calculated. Algorithm 4 shows these steps.

### 5. Empirical Study Results

To compare and evaluate the proposed algorithm, five software systems with different domains and sizes have been selected. Also, seven folders with different functionalities have been selected from the Mozilla Firefox application.

To answer the research question RQ1, for comparison, in this paper, nine search-based algorithms with different characteristics including single objective, multiobjective, global search, local search, structured-based methods, and semantic-based approaches are chosen. The algorithms selected are Bunch-GA, DAGC, ECA, MCA, Bunch-SAHC, SGA, GA-SMCP, EoD, and SNDGA. The characteristics of these algorithms are described in Table 5. We, also, selected ACDC as a pattern-based algorithm for comparison. Several previous studies [9–11] have shown that ACDC routinely outperformed the others. Because ACDC is a pattern-based method, it produces the same clustering each time it is repeated, so the best and average results are always the same.

The best and average results of TDHC on seven folders of Firefox folders and five other software systems are compared with the results of selected state-of-the-art algorithms with different features in terms of precision, recall, F-measure, and MoJoFM. The details are reported in Tables 6–9.

In Table 6, the TDHC has better performance in most cases, and the "dom" and "Intl" folders the ACDC algorithm has better results in best and average, respectively. Table 7 shows that, in terms of precision, MCA and ACDC have the best performance against other algorithms. In Table 8, the algorithms are compared in terms of recall in which TDHC has better performance in most cases. In Table 9, for F-measure, the TDHC and SNDGA perform almost the same.

From Tables 6–9, we conclude that DAGC, ECA, Bunch-SAHC, GHA, and GA-SMCP, compared to the other algorithms, systematically provide an extremely low precision/recall, F-measure, and MoJoFM. On the contrary, if we ignore the precision criterion, TDHC clearly seems to be among the best algorithms, always at the top. It often competes with ACDC, EoD, and SNDGA, which sometimes clearly outperform TDHC.

To exact and direct compare the results of the TDHC against other algorithms Cliff’s $\delta$ is calculated for them which results are represented in Table 10. Cliff’s $\delta$ is a

### Table 4: GA parameters.

<table>
<thead>
<tr>
<th>Population size (# population)</th>
<th>300N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum generation count</td>
<td>0.7</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>0.9</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>0.7 + ((#population - 1000/9000) × 0.2) if #population ≤ 10000</td>
</tr>
<tr>
<td>Crossover operation</td>
<td>0.04 log, N Cycle crossover</td>
</tr>
<tr>
<td>Mutation operation</td>
<td>Single swap</td>
</tr>
</tbody>
</table>
nonparametric effect size metric that quantifies the difference among two groups of observations (here TDHC against other tested algorithms). The result of this metric is in range 0 to 1, and higher value shows that results of the first group (here, TDHC) generally is better than the second group (other algorithms). To interpret, as in [10], the following magnitudes are used: negligible ($\delta < 0.147$), small ($0.147 < \delta < 0.33$), medium ($0.33 < \delta < 0.474$), and large ($\delta > 0.474$). The results indicate that the values for MoJoFM, precision, recall, and F-measure of TDHC output are better than the other algorithms in general.

In addition to the above experiments, we use MCDM to compare the performance of the tested algorithms considering all criteria employed for experiments. Table 11 shows the modularization quality in TDHC is better than other tested algorithms in most cases with an acceptable difference. The numbers in Table 11 show the superiority of the algorithms. The proximity of the produced numbers to one indicates that the algorithm, in that case, performed better than the rest in most experiments and most criteria.

To answer the research question RQ2, the genetic algorithm is a stochastic optimizer, and the results achieved may be different in each run. The results achieved by the algorithm for several independent runs are expected to be close enough to each other. Therefore, to answer RQ2, the proposed algorithm is executed 30 times for each case and the stability of the results is analyzed by the t-test statistical technique. To apply the t-test, the results are grouped into two groups with the same size, named G1 and G2, and then some descriptive and inferential statistics are extracted from them. According to [70], having 30 rows of data is enough to suppose that the distribution is normal. This is a critical condition to use the t-test for analyzing. But we also utilized the Wilcoxon-signed rank test [71] as a nonparametric statistical hypothesis test to check stability of the results without considering being in normal distribution.

The results are represented in Table 12. The three first columns show the average, the standard deviation, and the standard error between mean of the two groups, respectively, as descriptive statistics. The two last columns of the table
Table 6: Algorithms comparison in terms of MoJoFM.

<table>
<thead>
<tr>
<th>Folder name</th>
<th>Bunch-GA Best (%)</th>
<th>Average (%)</th>
<th>ECA Best (%)</th>
<th>Average (%)</th>
<th>MCA Best (%)</th>
<th>Average (%)</th>
<th>DAGC Best (%)</th>
<th>Average (%)</th>
<th>Bunch-SAHC Best (%)</th>
<th>Average (%)</th>
<th>SGA Best (%)</th>
<th>Average (%)</th>
<th>GA-SMCP Best (%)</th>
<th>Average (%)</th>
<th>EoD Best (%)</th>
<th>Average (%)</th>
<th>ACDC Best (%)</th>
<th>Average (%)</th>
<th>SNDGA Best (%)</th>
<th>Average (%)</th>
<th>TDHC Best (%)</th>
<th>Average (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>52</td>
<td>72</td>
<td>68</td>
<td>45</td>
<td>36</td>
<td>52</td>
<td>48</td>
<td>55</td>
<td>50</td>
<td>70</td>
<td>65</td>
<td>52</td>
<td>49</td>
<td>66</td>
<td>66</td>
<td>68</td>
<td>64</td>
<td>80</td>
<td>78</td>
</tr>
<tr>
<td>Dom</td>
<td>58</td>
<td>55</td>
<td>53</td>
<td>48</td>
<td>54</td>
<td>46</td>
<td>26</td>
<td>25</td>
<td>39</td>
<td>36</td>
<td>45</td>
<td>42</td>
<td>58</td>
<td>55</td>
<td>58</td>
<td>55</td>
<td>83</td>
<td>83</td>
<td>77</td>
<td>75</td>
<td>74</td>
<td>75</td>
</tr>
<tr>
<td>Accessible</td>
<td>42</td>
<td>39</td>
<td>37</td>
<td>36</td>
<td>40</td>
<td>36</td>
<td>27</td>
<td>24</td>
<td>27</td>
<td>26</td>
<td>55</td>
<td>53</td>
<td>38</td>
<td>36</td>
<td>42</td>
<td>39</td>
<td>42</td>
<td>42</td>
<td>65</td>
<td>61</td>
<td>78</td>
<td>54</td>
</tr>
<tr>
<td>Extensions</td>
<td>50</td>
<td>46</td>
<td>46</td>
<td>53</td>
<td>44</td>
<td>22</td>
<td>21</td>
<td>28</td>
<td>25</td>
<td>35</td>
<td>33</td>
<td>48</td>
<td>46</td>
<td>51</td>
<td>46</td>
<td>77</td>
<td>77</td>
<td>65</td>
<td>62</td>
<td>84</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>Gfx</td>
<td>54</td>
<td>50</td>
<td>61</td>
<td>54</td>
<td>67</td>
<td>57</td>
<td>29</td>
<td>29</td>
<td>42</td>
<td>41</td>
<td>57</td>
<td>55</td>
<td>54</td>
<td>50</td>
<td>60</td>
<td>57</td>
<td>73</td>
<td>73</td>
<td>84</td>
<td>81</td>
<td>86</td>
<td>75</td>
</tr>
<tr>
<td>IpC</td>
<td>81</td>
<td>81</td>
<td>81</td>
<td>81</td>
<td>81</td>
<td>81</td>
<td>40</td>
<td>38</td>
<td>81</td>
<td>81</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>79</td>
<td>80</td>
<td>79</td>
<td>61</td>
<td>61</td>
<td>61</td>
<td>91</td>
<td>90</td>
<td>67</td>
</tr>
<tr>
<td>Intl</td>
<td>80</td>
<td>76</td>
<td>75</td>
<td>75</td>
<td>84</td>
<td>81</td>
<td>41</td>
<td>39</td>
<td>75</td>
<td>75</td>
<td>78</td>
<td>76</td>
<td>71</td>
<td>68</td>
<td>75</td>
<td>71</td>
<td>92</td>
<td>92</td>
<td>86</td>
<td>80</td>
<td>89</td>
<td>79</td>
</tr>
<tr>
<td>Mini-Tunis</td>
<td>80</td>
<td>78</td>
<td>80</td>
<td>76</td>
<td>80</td>
<td>73</td>
<td>71</td>
<td>69</td>
<td>80</td>
<td>71</td>
<td>80</td>
<td>78</td>
<td>59</td>
<td>51</td>
<td>80</td>
<td>73</td>
<td>46</td>
<td>46</td>
<td>80</td>
<td>75</td>
<td>84</td>
<td>67</td>
</tr>
<tr>
<td>JUnit 4</td>
<td>60</td>
<td>58</td>
<td>60</td>
<td>58</td>
<td>60</td>
<td>56</td>
<td>59</td>
<td>51</td>
<td>60</td>
<td>58</td>
<td>60</td>
<td>59</td>
<td>60</td>
<td>57</td>
<td>67</td>
<td>65</td>
<td>63</td>
<td>63</td>
<td>65</td>
<td>60</td>
<td>60</td>
<td>51</td>
</tr>
<tr>
<td>Servlet-API</td>
<td>90</td>
<td>88</td>
<td>91</td>
<td>90</td>
<td>88</td>
<td>85</td>
<td>73</td>
<td>68</td>
<td>78</td>
<td>75</td>
<td>71</td>
<td>61</td>
<td>83</td>
<td>80</td>
<td>90</td>
<td>84</td>
<td>89</td>
<td>89</td>
<td>80</td>
<td>71</td>
<td>93</td>
<td>90</td>
</tr>
<tr>
<td>Easy mock</td>
<td>78</td>
<td>73</td>
<td>75</td>
<td>69</td>
<td>71</td>
<td>60</td>
<td>61</td>
<td>53</td>
<td>63</td>
<td>51</td>
<td>65</td>
<td>54</td>
<td>78</td>
<td>69</td>
<td>78</td>
<td>73</td>
<td>67</td>
<td>67</td>
<td>60</td>
<td>49</td>
<td>87</td>
<td>91</td>
</tr>
<tr>
<td>Calculator</td>
<td>40</td>
<td>37</td>
<td>38</td>
<td>35</td>
<td>33</td>
<td>31</td>
<td>30</td>
<td>31</td>
<td>38</td>
<td>31</td>
<td>38</td>
<td>38</td>
<td>40</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>44</td>
<td>44</td>
<td>62</td>
<td>54</td>
<td>85</td>
<td>79</td>
</tr>
</tbody>
</table>
### Table 7: Algorithms comparison in terms of precision.

<table>
<thead>
<tr>
<th>Folder name</th>
<th>Bunch-GA</th>
<th>ECA</th>
<th>MCA</th>
<th>DAGC</th>
<th>Bunch-SAHC</th>
<th>SGA</th>
<th>GA-SMCP</th>
<th>EoD</th>
<th>ACDC</th>
<th>SNDGA</th>
<th>TDHC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best (%)</td>
<td>Average (%)</td>
<td>Best (%)</td>
<td>Average (%)</td>
<td>Best (%)</td>
<td>Average (%)</td>
<td>Best (%)</td>
<td>Average (%)</td>
<td>Best (%)</td>
<td>Average (%)</td>
<td>Best (%)</td>
</tr>
<tr>
<td>Browser</td>
<td>83</td>
<td>62</td>
<td>33</td>
<td>321</td>
<td>100</td>
<td>96</td>
<td>7</td>
<td>5</td>
<td>49</td>
<td>28</td>
<td>55</td>
</tr>
<tr>
<td>Dom</td>
<td>53</td>
<td>41</td>
<td>27</td>
<td>24</td>
<td>44</td>
<td>43</td>
<td>2</td>
<td>2</td>
<td>33</td>
<td>21</td>
<td>38</td>
</tr>
<tr>
<td>Accessible</td>
<td>27</td>
<td>2</td>
<td>23</td>
<td>14</td>
<td>85</td>
<td>62</td>
<td>3</td>
<td>2</td>
<td>17</td>
<td>13</td>
<td>45</td>
</tr>
<tr>
<td>Extensions</td>
<td>26</td>
<td>26</td>
<td>19</td>
<td>14</td>
<td>35</td>
<td>28</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>Gfx</td>
<td>25</td>
<td>23</td>
<td>26</td>
<td>21</td>
<td>65</td>
<td>61</td>
<td>0.95</td>
<td>0.88</td>
<td>20</td>
<td>15</td>
<td>33</td>
</tr>
<tr>
<td>Ipcc</td>
<td>50</td>
<td>29</td>
<td>25</td>
<td>25</td>
<td>26</td>
<td>26</td>
<td>0.43</td>
<td>0.42</td>
<td>50</td>
<td>28</td>
<td>57</td>
</tr>
<tr>
<td>Intl</td>
<td>45</td>
<td>32</td>
<td>15</td>
<td>15</td>
<td>49</td>
<td>47</td>
<td>0.51</td>
<td>0.44</td>
<td>17</td>
<td>15</td>
<td>72</td>
</tr>
<tr>
<td>Mini-Tunis</td>
<td>81</td>
<td>78</td>
<td>81</td>
<td>79</td>
<td>81</td>
<td>78</td>
<td>69</td>
<td>63</td>
<td>81</td>
<td>72</td>
<td>68</td>
</tr>
<tr>
<td>JUnit 4</td>
<td>63</td>
<td>61</td>
<td>60</td>
<td>60</td>
<td>61</td>
<td>60</td>
<td>57</td>
<td>51</td>
<td>59</td>
<td>53</td>
<td>48</td>
</tr>
<tr>
<td>Servlet-API</td>
<td>50</td>
<td>47</td>
<td>51</td>
<td>49</td>
<td>83</td>
<td>80</td>
<td>48</td>
<td>45</td>
<td>48</td>
<td>42</td>
<td>50</td>
</tr>
<tr>
<td>Easy mock</td>
<td>41</td>
<td>36</td>
<td>65</td>
<td>61</td>
<td>60</td>
<td>53</td>
<td>38</td>
<td>30</td>
<td>35</td>
<td>28</td>
<td>41</td>
</tr>
<tr>
<td>Calculator</td>
<td>35</td>
<td>31</td>
<td>37</td>
<td>35</td>
<td>37</td>
<td>32</td>
<td>14</td>
<td>10</td>
<td>12</td>
<td>8</td>
<td>29</td>
</tr>
</tbody>
</table>
Table 8: Algorithms comparison in terms of recall.

<table>
<thead>
<tr>
<th>Folder name</th>
<th>Bunch-GA Best (%)</th>
<th>Average (%)</th>
<th>ECA Best (%)</th>
<th>Average (%)</th>
<th>MCA Best (%)</th>
<th>Average (%)</th>
<th>DAGC Best (%)</th>
<th>Average (%)</th>
<th>Bunch-SAHC Best (%)</th>
<th>Average (%)</th>
<th>SGA Best (%)</th>
<th>Average (%)</th>
<th>GA-SMCP Best (%)</th>
<th>Average (%)</th>
<th>EoD Best (%)</th>
<th>Average (%)</th>
<th>ACDC Best (%)</th>
<th>Average (%)</th>
<th>SNDGA Best (%)</th>
<th>Average (%)</th>
<th>TDHC Best (%)</th>
<th>Average (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser</td>
<td>65</td>
<td>55</td>
<td>51</td>
<td>46</td>
<td>56</td>
<td>51</td>
<td>70</td>
<td>54</td>
<td>42</td>
<td>36</td>
<td>32</td>
<td>51</td>
<td>61</td>
<td>57</td>
<td>10</td>
<td>10</td>
<td>56</td>
<td>52</td>
<td>61</td>
<td>51</td>
<td>60</td>
<td>49</td>
</tr>
<tr>
<td>Dom</td>
<td>43</td>
<td>40</td>
<td>40</td>
<td>37</td>
<td>38</td>
<td>35</td>
<td>48</td>
<td>44</td>
<td>29</td>
<td>29</td>
<td>31</td>
<td>30</td>
<td>41</td>
<td>33</td>
<td>10</td>
<td>10</td>
<td>56</td>
<td>51</td>
<td>30</td>
<td>10</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>Accessible</td>
<td>31</td>
<td>23</td>
<td>27</td>
<td>25</td>
<td>29</td>
<td>24</td>
<td>50</td>
<td>37</td>
<td>21</td>
<td>19</td>
<td>26</td>
<td>26</td>
<td>31</td>
<td>25</td>
<td>10</td>
<td>10</td>
<td>40</td>
<td>40</td>
<td>30</td>
<td>10</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>Extensions</td>
<td>35</td>
<td>30</td>
<td>51</td>
<td>37</td>
<td>39</td>
<td>27</td>
<td>36</td>
<td>31</td>
<td>20</td>
<td>17</td>
<td>18</td>
<td>17</td>
<td>30</td>
<td>24</td>
<td>28</td>
<td>22</td>
<td>98</td>
<td>98</td>
<td>48</td>
<td>45</td>
<td>50</td>
<td>49</td>
</tr>
<tr>
<td>Gfx</td>
<td>48</td>
<td>39</td>
<td>68</td>
<td>48</td>
<td>62</td>
<td>43</td>
<td>42</td>
<td>40</td>
<td>31</td>
<td>30</td>
<td>42</td>
<td>39</td>
<td>33</td>
<td>28</td>
<td>46</td>
<td>42</td>
<td>25</td>
<td>25</td>
<td>80</td>
<td>72</td>
<td>75</td>
<td>58</td>
</tr>
<tr>
<td>JUnit</td>
<td>70</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>68</td>
<td>63</td>
<td>62</td>
<td>62</td>
<td>68</td>
<td>68</td>
<td>72</td>
<td>72</td>
<td>70</td>
<td>65</td>
<td>69</td>
<td>63</td>
<td>6</td>
<td>6</td>
<td>84</td>
<td>81</td>
<td>73</td>
<td>85</td>
</tr>
<tr>
<td>Servlet-API</td>
<td>74</td>
<td>72</td>
<td>62</td>
<td>61</td>
<td>75</td>
<td>72</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>59</td>
<td>71</td>
<td>70</td>
<td>58</td>
<td>44</td>
<td>76</td>
<td>71</td>
<td>94</td>
<td>94</td>
<td>82</td>
<td>74</td>
<td>84</td>
<td>63</td>
</tr>
<tr>
<td>Mini-Tunis</td>
<td>71</td>
<td>63</td>
<td>73</td>
<td>70</td>
<td>71</td>
<td>70</td>
<td>48</td>
<td>39</td>
<td>68</td>
<td>61</td>
<td>60</td>
<td>51</td>
<td>60</td>
<td>52</td>
<td>71</td>
<td>69</td>
<td>92</td>
<td>92</td>
<td>69</td>
<td>52</td>
<td>78</td>
<td>67</td>
</tr>
<tr>
<td>JUnit 4</td>
<td>48</td>
<td>41</td>
<td>48</td>
<td>45</td>
<td>48</td>
<td>41</td>
<td>28</td>
<td>25</td>
<td>31</td>
<td>22</td>
<td>45</td>
<td>39</td>
<td>38</td>
<td>32</td>
<td>53</td>
<td>50</td>
<td>49</td>
<td>49</td>
<td>50</td>
<td>43</td>
<td>34</td>
<td>31</td>
</tr>
<tr>
<td>Servlet-API</td>
<td>73</td>
<td>65</td>
<td>78</td>
<td>74</td>
<td>75</td>
<td>68</td>
<td>51</td>
<td>42</td>
<td>41</td>
<td>30</td>
<td>59</td>
<td>48</td>
<td>48</td>
<td>41</td>
<td>78</td>
<td>72</td>
<td>41</td>
<td>41</td>
<td>41</td>
<td>75</td>
<td>69</td>
<td>100</td>
</tr>
<tr>
<td>Easy mock</td>
<td>79</td>
<td>72</td>
<td>81</td>
<td>74</td>
<td>81</td>
<td>76</td>
<td>52</td>
<td>45</td>
<td>38</td>
<td>31</td>
<td>80</td>
<td>73</td>
<td>68</td>
<td>61</td>
<td>83</td>
<td>78</td>
<td>32</td>
<td>32</td>
<td>78</td>
<td>70</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>Calculator</td>
<td>48</td>
<td>40</td>
<td>55</td>
<td>43</td>
<td>61</td>
<td>50</td>
<td>28</td>
<td>19</td>
<td>21</td>
<td>12</td>
<td>50</td>
<td>42</td>
<td>39</td>
<td>35</td>
<td>55</td>
<td>44</td>
<td>52</td>
<td>52</td>
<td>55</td>
<td>49</td>
<td>89</td>
<td>80</td>
</tr>
</tbody>
</table>

Complexity
Table 9: Algorithms comparison in terms of $F$-measure.

<table>
<thead>
<tr>
<th>Folder name</th>
<th>Bunch-GA Best (%)</th>
<th>Average (%)</th>
<th>ECA Best (%)</th>
<th>Average (%)</th>
<th>MCA Best (%)</th>
<th>Average (%)</th>
<th>DAGC Best (%)</th>
<th>Average (%)</th>
<th>Bunch-SAHC Best (%)</th>
<th>Average (%)</th>
<th>SGA Best (%)</th>
<th>Average (%)</th>
<th>GA-SMCP Best (%)</th>
<th>Average (%)</th>
<th>EoD Best (%)</th>
<th>Average (%)</th>
<th>ACDC Best (%)</th>
<th>Average (%)</th>
<th>SNDGA Best (%)</th>
<th>Average (%)</th>
<th>TDHC Best (%)</th>
<th>Average (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser</td>
<td>73</td>
<td>57</td>
<td>40</td>
<td>37</td>
<td>71</td>
<td>67</td>
<td>13</td>
<td>9</td>
<td>41</td>
<td>30</td>
<td>41</td>
<td>39</td>
<td>65</td>
<td>59</td>
<td>71</td>
<td>65</td>
<td>18</td>
<td>18</td>
<td>55</td>
<td>44</td>
<td>66</td>
<td>52</td>
</tr>
<tr>
<td>Dom</td>
<td>45</td>
<td>40</td>
<td>32</td>
<td>30</td>
<td>40</td>
<td>38</td>
<td>3</td>
<td>3</td>
<td>31</td>
<td>24</td>
<td>34</td>
<td>31</td>
<td>45</td>
<td>36</td>
<td>48</td>
<td>41</td>
<td>56</td>
<td>56</td>
<td>69</td>
<td>65</td>
<td>54</td>
<td>38</td>
</tr>
<tr>
<td>Accessible</td>
<td>27</td>
<td>21</td>
<td>24</td>
<td>20</td>
<td>36</td>
<td>32</td>
<td>6</td>
<td>4</td>
<td>18</td>
<td>15</td>
<td>41</td>
<td>39</td>
<td>32</td>
<td>22</td>
<td>32</td>
<td>28</td>
<td>26</td>
<td>26</td>
<td>45</td>
<td>40</td>
<td>51</td>
<td>41</td>
</tr>
<tr>
<td>Extensions</td>
<td>30</td>
<td>28</td>
<td>34</td>
<td>25</td>
<td>35</td>
<td>28</td>
<td>5</td>
<td>4</td>
<td>15</td>
<td>12</td>
<td>24</td>
<td>23</td>
<td>23</td>
<td>19</td>
<td>28</td>
<td>24</td>
<td>81</td>
<td>81</td>
<td>41</td>
<td>38</td>
<td>46</td>
<td>35</td>
</tr>
<tr>
<td>Gfx</td>
<td>33</td>
<td>29</td>
<td>37</td>
<td>29</td>
<td>63</td>
<td>41</td>
<td>18</td>
<td>2</td>
<td>24</td>
<td>20</td>
<td>35</td>
<td>30</td>
<td>26</td>
<td>23</td>
<td>33</td>
<td>27</td>
<td>35</td>
<td>35</td>
<td>60</td>
<td>51</td>
<td>62</td>
<td>51</td>
</tr>
<tr>
<td>Ipc</td>
<td>51</td>
<td>39</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>0.83</td>
<td>0.84</td>
<td>58</td>
<td>39</td>
<td>60</td>
<td>45</td>
<td>55</td>
<td>48</td>
<td>61</td>
<td>55</td>
<td>12</td>
<td>12</td>
<td>64</td>
<td>54</td>
<td>65</td>
<td>61</td>
</tr>
<tr>
<td>Int1</td>
<td>56</td>
<td>44</td>
<td>24</td>
<td>24</td>
<td>59</td>
<td>57</td>
<td>1</td>
<td>0.87</td>
<td>30</td>
<td>24</td>
<td>73</td>
<td>66</td>
<td>51</td>
<td>43</td>
<td>52</td>
<td>44</td>
<td>96</td>
<td>96</td>
<td>37</td>
<td>33</td>
<td>77</td>
<td>59</td>
</tr>
<tr>
<td>Mini-Tunis</td>
<td>76</td>
<td>70</td>
<td>77</td>
<td>74</td>
<td>76</td>
<td>75</td>
<td>57</td>
<td>48</td>
<td>74</td>
<td>66</td>
<td>64</td>
<td>57</td>
<td>69</td>
<td>62</td>
<td>76</td>
<td>74</td>
<td>51</td>
<td>51</td>
<td>70</td>
<td>53</td>
<td>81</td>
<td>60</td>
</tr>
<tr>
<td>JUnit 4</td>
<td>54</td>
<td>49</td>
<td>54</td>
<td>51</td>
<td>54</td>
<td>49</td>
<td>38</td>
<td>34</td>
<td>41</td>
<td>31</td>
<td>46</td>
<td>40</td>
<td>47</td>
<td>41</td>
<td>57</td>
<td>55</td>
<td>51</td>
<td>51</td>
<td>58</td>
<td>50</td>
<td>39</td>
<td>36</td>
</tr>
<tr>
<td>Servlet-API</td>
<td>59</td>
<td>55</td>
<td>62</td>
<td>59</td>
<td>79</td>
<td>74</td>
<td>49</td>
<td>43</td>
<td>44</td>
<td>35</td>
<td>53</td>
<td>45</td>
<td>49</td>
<td>44</td>
<td>61</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>59</td>
<td>54</td>
<td>71</td>
<td>83</td>
</tr>
<tr>
<td>Easy mock</td>
<td>54</td>
<td>48</td>
<td>72</td>
<td>67</td>
<td>69</td>
<td>62</td>
<td>44</td>
<td>36</td>
<td>36</td>
<td>29</td>
<td>54</td>
<td>47</td>
<td>51</td>
<td>43</td>
<td>58</td>
<td>51</td>
<td>41</td>
<td>41</td>
<td>51</td>
<td>51</td>
<td>56</td>
<td>53</td>
</tr>
<tr>
<td>Calculator</td>
<td>40</td>
<td>35</td>
<td>44</td>
<td>39</td>
<td>46</td>
<td>40</td>
<td>19</td>
<td>13</td>
<td>15</td>
<td>10</td>
<td>37</td>
<td>31</td>
<td>26</td>
<td>23</td>
<td>47</td>
<td>38</td>
<td>29</td>
<td>29</td>
<td>46</td>
<td>40</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>Metric name</td>
<td>Bunch</td>
<td>ECA</td>
<td>MCA</td>
<td>DAGC</td>
<td>Bunch-SAHC</td>
<td>SGA</td>
<td>GA-SMCP</td>
<td>EoD</td>
<td>ACDC</td>
<td>SNDGA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>------------</td>
<td>-----</td>
<td>---------</td>
<td>-----</td>
<td>------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MoJoFM</td>
<td>0.57</td>
<td>0.37</td>
<td>0.60</td>
<td>0.43</td>
<td>0.56</td>
<td>0.39</td>
<td>0.93</td>
<td>0.90</td>
<td>0.76</td>
<td>0.60</td>
<td>0.78</td>
<td>0.76</td>
<td>0.56</td>
<td>0.60</td>
<td>0.45</td>
<td>0.50</td>
<td>0.27</td>
<td>0.39</td>
<td>0.22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>0.22</td>
<td>0.26</td>
<td>0.42</td>
<td>0.31</td>
<td>-0.01</td>
<td>-0.24</td>
<td>0.51</td>
<td>0.53</td>
<td>0.44</td>
<td>0.41</td>
<td>0.51</td>
<td>0.29</td>
<td>0.24</td>
<td>0.20</td>
<td>0.07</td>
<td>-0.02</td>
<td>-0.14</td>
<td>-0.26</td>
<td>0.19</td>
<td>0.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recall</td>
<td>0.28</td>
<td>0.41</td>
<td>0.22</td>
<td>0.43</td>
<td>0.21</td>
<td>0.36</td>
<td>0.51</td>
<td>0.72</td>
<td>0.60</td>
<td>0.74</td>
<td>0.49</td>
<td>0.69</td>
<td>0.51</td>
<td>0.69</td>
<td>0.20</td>
<td>0.35</td>
<td>0.27</td>
<td>0.35</td>
<td>0.17</td>
<td>0.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F-measure</td>
<td>0.29</td>
<td>0.25</td>
<td>0.44</td>
<td>0.31</td>
<td>0.10</td>
<td>-0.03</td>
<td>0.69</td>
<td>0.52</td>
<td>0.65</td>
<td>0.70</td>
<td>0.67</td>
<td>0.42</td>
<td>0.45</td>
<td>0.31</td>
<td>0.22</td>
<td>0.06</td>
<td>0.35</td>
<td>0.19</td>
<td>0.18</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm</td>
<td>Browser</td>
<td>Dom</td>
<td>Accessible</td>
<td>Extensions</td>
<td>Gfx</td>
<td>Ipc</td>
<td>Int Int</td>
<td>Min-Tunis</td>
<td>JUnit 4</td>
<td>Servlet-API</td>
<td>Easy mock</td>
<td>Calculator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>-----</td>
<td>------------</td>
<td>------------</td>
<td>-----</td>
<td>-----</td>
<td>---------</td>
<td>-----------</td>
<td>---------</td>
<td>-------------</td>
<td>-----------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bunch-GA</td>
<td>0.0334</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.2521</td>
<td>0</td>
<td>0</td>
<td>0.438</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.723</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCA</td>
<td>0.8641</td>
<td>0</td>
<td>0.1953</td>
<td>0</td>
<td>0.294</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
<td>0</td>
<td>0.0113</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAGC</td>
<td>0.0481</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bunch-SAHC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GA-SMCP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EoD</td>
<td>0.0334</td>
<td>0</td>
<td>0</td>
<td>0.331</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.50</td>
<td>0</td>
<td>0</td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDC</td>
<td>0</td>
<td>0.826</td>
<td>0</td>
<td>0.661</td>
<td>0</td>
<td>0.010</td>
<td>1</td>
<td>0.327</td>
<td>0</td>
<td>0.190</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNDGA</td>
<td>0</td>
<td>0.731</td>
<td>0.15</td>
<td>0</td>
<td>0.498</td>
<td>0.81</td>
<td>0</td>
<td>0.041</td>
<td>0.552</td>
<td>0</td>
<td>0</td>
<td>0.127</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDHC</td>
<td>0.5210</td>
<td>0.125</td>
<td>1</td>
<td>0.431</td>
<td>0.391</td>
<td>0.81</td>
<td>0</td>
<td>0.985</td>
<td>0</td>
<td>0.691</td>
<td>0.821</td>
<td>0.802</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 12: *t*-test for the experiments.

<table>
<thead>
<tr>
<th>Case study</th>
<th>Mean</th>
<th>Standard deviation</th>
<th>Standard error between mean</th>
<th>Levene’s test</th>
<th><em>t</em>-test</th>
<th>Wilcoxon-signed rank test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G1</td>
<td>G2</td>
<td>G1</td>
<td>G2</td>
<td>G1</td>
<td>G2</td>
</tr>
<tr>
<td>Firefox-browser</td>
<td>1.4714</td>
<td>1.5572</td>
<td>0.337</td>
<td>0.246</td>
<td>0.1472</td>
<td>0.1003</td>
</tr>
<tr>
<td>Firefox-dom</td>
<td>0.573</td>
<td>0.614</td>
<td>0.247</td>
<td>0.306</td>
<td>0.1175</td>
<td>0.1370</td>
</tr>
<tr>
<td>Firefox-accessible</td>
<td>0.5228</td>
<td>0.5942</td>
<td>0.1275</td>
<td>0.1285</td>
<td>0.5702</td>
<td>0.0574</td>
</tr>
<tr>
<td>Firefox-extensions</td>
<td>0.553</td>
<td>0.636</td>
<td>0.212</td>
<td>0.2391</td>
<td>0.0.096</td>
<td>0.103</td>
</tr>
<tr>
<td>Firefox-gfx</td>
<td>0.3344</td>
<td>0.2688</td>
<td>0.1468</td>
<td>0.1254</td>
<td>0.06456</td>
<td>0.05611</td>
</tr>
<tr>
<td>Firefox-ipc</td>
<td>1.306</td>
<td>1.462</td>
<td>0.4081</td>
<td>0.436</td>
<td>0.179</td>
<td>0.195</td>
</tr>
<tr>
<td>Firefox-intl</td>
<td>0.347</td>
<td>0.319</td>
<td>0.364</td>
<td>0.1272</td>
<td>0.1628</td>
<td>0.0422</td>
</tr>
<tr>
<td>Mini-Tunis</td>
<td>0.9726</td>
<td>1.0157</td>
<td>0.14256</td>
<td>0.14043</td>
<td>0.06376</td>
<td>0.06280</td>
</tr>
<tr>
<td>JUnit 4</td>
<td>0.7950</td>
<td>0.7923</td>
<td>0.00759</td>
<td>0.01831</td>
<td>0.00339</td>
<td>0.00819</td>
</tr>
<tr>
<td>Servlet-API</td>
<td>2.5234</td>
<td>2.5134</td>
<td>0.05345</td>
<td>0.5871</td>
<td>0.02390</td>
<td>0.02626</td>
</tr>
<tr>
<td>Easy mock</td>
<td>1.9572</td>
<td>2.0360</td>
<td>0.29716</td>
<td>0.24379</td>
<td>0.13289</td>
<td>0.10903</td>
</tr>
<tr>
<td>Calculator</td>
<td>2.3094</td>
<td>2.3210</td>
<td>0.16658</td>
<td>0.29373</td>
<td>0.07449</td>
<td>0.13136</td>
</tr>
</tbody>
</table>
show the output of the inferential statistics. Levene’s test is an inferential statistic for assessing the equality of variances for a variable calculated for two groups. If the $p$ value (sig. column in the table) is greater than some significance level (0.05 in our tests), the null hypothesis of equal variances cannot be rejected. This is also true for the Wilcoxon-signed rank test. Two columns of Table 12 refer to the results of independent two-sample t-test with equal sample sizes and equal variances (according to results of Levene’s test) on two randomly separated groups of TDHC results, and the last two lines are for the results of Wilcoxon-signed rank test if the data are not in normal distribution. All the $p$ values are greater than 0.05, which shows we cannot reject the null hypothesis of equal means. Hence, the results of the different tests are converging to an acceptable range.

To answer the research question RQ3, the output of TDHC for Mini-Tunis is investigated. Mini-Tunis (mtunis) is an academic operating system with 20 artifacts numbered from 1 to 20 in Figure 12. Each artifact is a file, and all the artifacts with the same number in the parenthesis are in the same module [5]. According to the figure, artifact numbers 15 and 17 just are called by other artifacts and can be discarded in the preprocessing. Figure 13 shows the tree produced by the proposed algorithm for 18 remaining artifacts, and Figure 14 represents its corresponding hierarchical modularization in a flat view. The numbers between parentheses for each artifact are its module number in the expert modularization. As is shown in this figure, a new arrangement is proposed to the artifacts of module numbers 3 and 5 in the expert modularization, but they joined together in the upper level. The artifact numbers 15 and 17 have a relation with several of these modules. So they are identified as a new module in which the artifacts are utility libraries. These two artifacts are in a separate module in the expert modularization too.

The most important advantages of this method are that it can perform very well in to specify the encapsulation levels, e.g., module, package, and component by the designer.

### 6. Threats to Validity

In this section, to clarify the validity of TDHC, the limitations that can affect the results of the algorithm are discussed. Several factors may bias the validity of the study. These are typically divided into two categories: external and internal validity. External validity is about the ability to generalize the results to other than used case studies or indifferent settings for them:

1. The input of the algorithm is an ADG extracted from source code, and cohesion and coupling are considered as an indicator for refactoring. Candela et al. in [1] discussed that cohesion and coupling are not enough to remodularization of source code, and more indicators are probably needed. However, they did not discuss in their work what other indicators could improve the quality of the modularization.

2. In search-based techniques for source code remodularization, generalizing a technique to any software is an important threat to the validity of results. So, in this paper, Mozilla Firefox as a large-scale software system is selected alongside five medium-size other open-source systems. It is important to note that there are just some software systems that have more than Mozilla Firefox artifacts (here files) in a folder.

Internal validity is concerned with experimental treatments that affect the algorithm results, leading to poor results:

1. In this paper, precision, recall, F-measure, and MojoFM metrics have used to compare study results with current modularization algorithms. These
metrics are not necessarily in line with the developer expert’s opinion. Also, these metrics do not evaluate the structure of the tree, and none of them consider edges between artifacts in calculating similarity.

(2) In the preprocessing step of TDHC, some artifacts may be selected to set aside from input of the tree generation step. In the end, it is important to suggest an appropriate position in modules for them or aggregate them as a new module.

(3) The related rate of crossover and mutation operators used in GA is achieved from several experiments on the Mini-Tunis, JUnit, and Servlet-API software systems and applied to other case studies. However, these numbers may not work well on other software systems.

(4) In the proposed algorithm, labels of inner nodes are not important and the Prufer sequence generates the same modularization for different codes. For example, 5, 5, 6, 6, 7 and 6, 6, 5, 5, 7 both represent the same modularization. On the contrary, the concept of neighborhood in this encoding is not transparent, and small changes in number positions make a great change in the structure of the output tree.

7. Conclusion and Future Work

During software maintenance and evolution, the structure of the software deviates from its original structure. Thus, source code refactoring is an essential role in the software maintenance process. In this paper, a new clustering algorithm based on cohesion and coupling between artifacts is proposed. In this method, a top-down hierarchical approach has been used with a metaheuristic algorithm (combining genetic algorithm and hill-climbing). In the proposed algorithm, a suitable point to start modularization of artifacts is suggested for developers. The input of the algorithm is ADG, which is independent of the source code programming language. However, its prepossessing operations may depend on the programming language of the source code or the type of input artifacts (class, file, function, or low-level module). Because the proposed refactoring method is automatic, it is supposed to serve as an assistant to the developer. Design decisions are often more complex and subtle than just trying to maximize cohesion and minimize coupling in the modularization process. In outcome, the derived modularization is analyzed by the software developer who can accept the proposed remodularization as is or change it by moving artifacts from one module (package) to another. The following is suggested for future works:

(1) Increasing number of artifacts affect the quality of the optimal solution proposed by the algorithm. It is because of the exponential growth of the search space by increasing the input size. So it is important to improve the algorithm factors.

(2) According to the size of search space in the software source code, a new preprocessing method can be offered to reduce search space. For example, in software source code refactoring, there is a modularization as a current developer suggests, and artifacts in a module are usually closely in contact with each other on that module and only some of them are in relation to other modules. Therefore, they can be ignored in calculating the relationship between modules.

(3) Many research studies use structured or nonstructured features for refactoring of source code that can be used in this top-down search-based algorithm too.
(4) Other heuristic or metaheuristic algorithms can be used instead of GA.

Data Availability

The data used to support the findings of this study are available at https://github.com/SoftwareMaintenanceLab.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References


