Soft Computing (2019) 23:11141–11165
https://doi.org/10.1007/s00500-018-3666-z

METHODOLOGIES AND APPLICATION

Multi-objective search-based software modularization: structural and non-structural features

Nafiseh Sadat Jalali[1] · Habib Izadkhah[1] · Shahriar Lotfi[1]

Published online: 29 November 2018

Abstract
Software modularization techniques are employed to understand a software system. The purpose of modularization is to decompose a software system from a source code into meaningful and understandable subsystems (modules). Since modularization of a software system is an NP-hard problem, the modularization quality obtained using evolutionary algorithms is more reasonable than greedy algorithms. All evolutionary algorithms presented for software modularization only consider structural features that are dependent on the syntax of programming languages. For most programming languages, does not exist a tool to extract structural features, so it is not possible to modularize them. To overcome this problem, this paper presents a new multi-objective fitness function, named MOF, which exploits the structural (such as calling dependency and inheritance dependency) and non-structural features (such as semantic contained in the code comments and identifier names), aiming to automatically guide optimization algorithms to find a good decomposition of software systems. To evaluate the performance of this objective function, three optimization strategies, namely global-based search, combining global and local search, and Estimation of Distribution (EoD), are adapted to optimize it. The results on Mozilla Firefox indicate that the proposed algorithm which is based on EoD along with the new MOF function outperforms the algorithms that use structural-based objective functions in guiding the optimization process, in finding more understandable modules.

Keywords Evolutionary algorithms · Modularization · Clustering · Estimation of distribution algorithm · Reverse Engineering

1 Introduction

Today, all large organizations are dependent on software systems. Adapting to the new requirements is one of the basic principles of these organizations. Therefore, to handle these requirements, it is necessary to keep the related software system up-to-date. During the maintenance process, to rejuvenate the application for newer needs, the software architecture deviates from its original architecture, while these modifications are not well documented. Without up-to-date documentation, in the process of software maintenance, software engineers spend considerable time to understand the code of the software system, which indicates the importance of the program's understanding. For example, a study (see Pfleeger 2001; Pigoski 1996) reports that from 40 to 60% of the maintenance activity is spent on studying the program to understand it. Understanding a program is possible from design features, such as software architecture. The architecture of a software system represents the components and connections between them. Software modularization techniques are used to extract the software architecture from the source code, aiming to understand and maintenance of the software. Modularization has been used in two different areas in software engineering in the literature: software architecture recovery and refactoring. The purpose of the software architecture recovery is to use modularization

Communicated by V. Loia.

✉ Habib Izadkhah
izadkhah@tabrizu.ac.ir

Nafiseh Sadat Jalali
nafisejalali1989@yahoo.com

Shahriar Lotfi
shahriar_lotfi@tabrizu.ac.ir

[1] Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran

techniques to partition a software system from the source code into meaningful and understandable subsystems. This helps to understand the software in the process of reverse engineering software. Refactoring also utilizes modularization, but with different goals. Refactoring approaches support big-bang re-modularization (meaning those taking a system as an input and providing it as output, as a completely new system with a new modularization). Some papers that use modularization to refactor the source code are (Bavota et al. 2012; Candela et al. 2016; Mkaouer et al. 2015). This paper aims to present modularization approaches to support software architecture recovery, not software refactoring.

Artifact Dependency Graph (ADG) is the input of modularization algorithms. In a software, depending on the type of software, any component such as class, method, function, or file can be considered as an artifact. Also, calling dependency or similarity between comments and etc. can be used as a dependency between artifacts.

To be a reasonable structure of software systems, artifacts placed within the same module should have the maximum connection/similarity to each other and artifacts in different modules should have the minimum relationship. Hence, most modularization methods use cohesion and coupling criteria to modularize a software system. Cohesion indicates the coherence of a module and coupling is the degree of interaction between modules. Candela et al. (2016) in an excellent paper named "Using Cohesion and Coupling for Software re-modularization: Is It Enough?". It was concluded that these criteria alone are not enough and other criteria are needed. Note that, they did not argue about what criteria are needed. In this paper, we aim at answering the following research questions:

(1) Which features in the source code can lead to modularization which is closer to the modularization created by the domain expert?
(2) Does a multi-objective fitness function that takes into account structural and non-structural features can improve modularization quality?
(3) Is the estimation of distribution-based algorithm able to produce higher-quality modularization compared with other search-based modularization algorithms?

In the most modularization algorithms, structural features such as cohesion and coupling are exploited for modularization. In this paper, in addition to the cohesion and coupling, we will examine the impact of inheritance relationships and non-structural features, e.g., identifier names and comments, on the modularization quality of a software system. Then, considering these structural and non-structural features, we propose a number of objectives and present a relationship which considers these objectives for modularization. Due to the NP-hardness of software modularization problem, the use of search-based and evolutionary approaches such as genetic algorithm to be more efficient than other approaches (Isazadeh et al. 2017). The searching procedure in search-based algorithms can be performed in two ways: global search and local search. Global search-based methods usually consider the whole search space to find a good solution. These methods have an operator to explore new regions in search space. The genetic algorithm is one of the main search methods that uses the global search strategy. Local search algorithms utilize exploitation ability to improve the quality of the already existing solutions. These algorithms start from an initial solution and then iteratively move from the current solution to a neighbor solution (to be defined) in the search space by applying local changes. Hill climbing is one of the main search methods that uses the local strategy for search. To optimize the proposed multi-objective function, in this paper, we apply the genetic algorithm alone and also with two different combinations with the hill climbing algorithm, aiming to use the advantage of these two search algorithms.

Mutation and crossover operators and their probability have a great influence on the efficiency of the genetic algorithm. The main problem with a genetic algorithm is the lack of keep of building blocks and important patterns obtained during the evolutionary process. The building blocks may be destroyed after applying the crossover and mutation operators. This will slow down the process of the optimal solution finding. Estimation of Distribution (EoD)-based algorithms are used to maintain the building blocks. To solve a problem, the EoD-based algorithms generate an initial population and construct a probabilistic model from it. The next generation is created based on the constructed probabilistic model, aiming to maintain valuable building blocks. Determining the mutation and crossover rates is one of the problems of genetic algorithms. EoD-based algorithms do not use concepts like mutations and crossovers to create a new population. In this way, eliminating the good solutions in the answers will be avoided as far as possible.

This paper proposes a new multi-objective fitness function which considers structural features (such as calling dependency and inheritance dependency) and latent semantic contained in non-structural features (such as identifier names and comments) for software modularization. We used the machine learning techniques for extracting latent semantic contained in non-structural features. To evaluate the performance of this objective function, three optimization strategies, namely global-based search, combining global and local search, and estimation of distribution, are adapted to optimize it. Each of these algorithms has unique features. The results on Mozilla Firefox, as a large-scale application, demonstrate that

modularization which is based on structural features outperforms those which are based on non-structural features. Moreover, combining the structural and non-structural features (i.e., the proposed multi-objective function) outperforms the structural-based objective functions in guiding the optimization algorithms to find more understandable modules. Furthermore, the results indicate that the algorithm which is based on the estimation of distribution strategy is better than the rest of the strategies in software system modularization.

The contributions of this paper are summarized as below:

1. Proposing a new objective function which considers structural and non-structural features;
2. The proposed multi-objective function is the first objective function that can use the latent semantic contained in non-structural features for software modularization;
3. Identifying non-structural features in the source code that can replace structural features. Most modularization algorithms utilize structural features such as Call Dependency for modularizing a software system. In a multi-lingual program, due to the different syntactic structure used in programming languages, it is not possible to extract calling dependency among programming languages. In this case, it is not possible to modularize it. Furthermore, for most programming languages, does not exist a tool to extract structural features, so it is not possible to modularize them. Therefore, in this study, we investigate the effect of non-structural features on modularization quality. The results demonstrate that identifier names, as a non-structural feature, can be used instead of calling dependency with acceptable reliability.
4. Applying three search strategies for software modularization. These strategies are: (1) using global search, (2) combining global and local search, and (3) a proposed algorithm which is based on the estimation of distribution. These algorithms have not been studied by any researcher, to solve the software modularization problem, taking into account structural and non-structural features simultaneously.

The structure of this paper is organized as follows: Sect. 2 addresses and evaluates the related works. Section 3 presents the proposed multi-objective functions and a number of evolutionary algorithms to optimize it. Section 4 evaluates the proposed algorithms, and Sect. 5 concludes the paper.

2 Related works

In the literature, clustering techniques can be categorized into data clustering techniques and graph clustering techniques. Some examples of data clustering techniques are (Abualigah et al. 2017a, b, 2018; Alswaitti et al. 2018; Liu et al. 2017). Graph-based clustering techniques can be applied to different areas such as social network, protein complex identification, image segmentation, and software clustering. Some examples of graph clustering techniques are (Chang et al. 2017; Wen et al. 2017).

Software modularization (or software clustering) algorithms can be classified into hierarchical and non-hierarchical (including search-based methods and greedy algorithms) categories.

Most presented hierarchical methods for software modularization are agglomerative. The overall process of modularization in these methods is that, initially, all artifacts are individually placed in separate modules. During an iterative process, based on certain criterion, e.g., Jaccard similarity criterion or Euclidean distance criterion, the similarity or dissimilarity is calculated between all artifacts, and artifacts with the highest similarity or least dissimilarity are merged together. Hierarchical methods tend to have the following issues:

(1) A well-defined criterion does not exist to determine where the modularization operation should end;
(2) Arbitrary decisions are one of the main issues in the hierarchical modularization methods. These decisions have a great impact on the final modularization;
(3) It is not possible to go back and correct the wrong choices;
(4) These algorithms cannot search the problem space well.

In the following, a number of hierarchical modularization methods are discussed.

Saeed et al. (2003) proposed a modularization algorithm named combined algorithm. This algorithm uses a binary data table to show relationships between artifacts and features. The Jaccard similarity measure is used in this algorithm to calculate the similarity between artifacts, and OR operator is employed to calculate the new feature vector from two previous feature vectors. Two major drawbacks of this algorithm are: it does not handle the arbitrary decision problem and does not take into account the number of artifacts that have access to a feature. To overcome this drawback, Maqbool and Babri (2004) proposed a modularization algorithm named weighted combined algorithm. This algorithm uses a weighted version of Jaccard, named Ellenberg similarity measure, to calculate

the similarity between artifacts. Andritsos and Tzerpos (2005) presented an entropy-based hierarchical clustering algorithm which uses structural and non-structural features to modularize a software system. The non-structural features considered are developers, directory path, lines of code, and time of the last update. Their results showed that ownership information had a positive impact on modularization and the use of lines of code is not a good idea. Time can be appropriate for a modularization factor in a different environment. To overcome the drawbacks in individual similarity criteria, such as Jaccard, Naseem et al. (2013) used cooperative clustering, which is a consensus-based technique. In this method, two similarity criteria named Jaccard and Jaccard-NM are employed to calculate the similarity between artifacts. The arbitrary decisions are the main problem of this algorithm. In Srinivas et al. (2013a, b), a method is proposed that applies the XOR similarity function to the Boolean matrix after specifying duplicate items for modularization. This method is greedy, and the input must be in binary forms. Naseem et al. (2014) used two modalities for modularization: one is a similarity criterion and another is distance aiming to improve the quality of the modules by reducing the number of arbitrary decisions. Misra (2012) and Misra et al. (2012) presented a hierarchical modularization algorithm that considers structural and non-structural features.

In search-based modularization techniques, the problem of modularization is treated as a search problem. Since searching the entire state space, transforms the problem into an NP-hard problem, it is used evolutionary approaches such as the genetic method to find the solution. Most search-based algorithms are single-objective, and their main aim is to find a modularization with maximum cohesion and minimum coupling. These algorithms aim is to maximize the objective function, MQ, indicated in Eq. 2. This function is applicable to weighted graphs, and its computational complexity is low, and this is one of the advantages of this objective function. This function is obtained from the sum of several module factors. The module factor for module i is shown with CF_i, where μ_i represents the intra-connections and ε_i represents inter-connections between two modules.

$$CF_i = \frac{2 \times \mu_i}{2 \times \mu_i + \varepsilon_i} \quad (1)$$

$$MQ = \sum_{i=1}^{k} CF_i \quad (2)$$

Mitchell in his Ph.D. thesis (Mitchell and Mancoridis 2002, 2006, 2008) proposed a genetic-based algorithm, named Bunch, for software modularization aiming to maximize the MQ. The encoding used in the Bunch is a value-based encoding so that the search space in it is n^n. Parsa and Bushehrian (2005) proposed a modularization algorithm, named DAGC, which uses a permutation-based encoding against value-based encoding. This encoding is able to reduce the search space to n!. Both Bunch and DAGC are single-objective and consider CDG, as a structural feature, for modularization. Praditwong et al. (2011) proposed two multi-objective algorithms, named ECA and MCA, for modularization a software system. The objectives used in ECA are:

- The sum of edges within all modules should be maximized,
- The sum of edges within all modules should be minimized,
- The number of modules should be maximized,
- MQ should be maximized,
- The number of single-member modules should be minimized.

The objectives used in the MCA are similar to those used in ECA, with the difference that, instead of the last one, "the difference between the maximum and minimum number of artifacts in a module should be minimized," has been used. These two multi-objective algorithms consider only structural features and do not include the non-structural features. Tables 1, 2, 3, 4 and 5 show the algorithms available for hierarchical modularization methods, local search algorithms, global search algorithms, combining global and local search-based algorithm, and graph-based algorithms, respectively.

According to the literature, most modularization methods exploit search-based algorithms to modularize a software system. Also, according to Tables 2, 3, 4 and 5, most search-based methods utilize structural features such as calling relationship to perform modularization and do not consider the impact of inheritance relationship, as a structural feature, and non-structural features such as comments and identifier names. In this paper, in addition to structural features, we will also investigate the effect of non-structural features on modularization quality.

3 The proposed method

Due to the various factors affecting the result of modularization, this section, presents a method that resolves the limitations of the existing methods. Most existing methods only use call dependency graph as a structural feature for modularization, but for most programming languages, in the lack of the necessary tools, we will not be able to extract the call dependency graph. Hence, the proposed methods also consider semantic features (i.e., non-structural features) to extract software architecture. In this section, we will first present our multi-objective function and then five search-

Table 1 Hierarchical algorithms for modularization

Approach	Considering the non-structural features	Considering the structural features	Disadvantages
Single linkage (Maqbool and Babri 2007)	✗	✔	Arbitrary decision, greedy
Complete linkage (Maqbool and Babri 2007)	✗	✔	Arbitrary decision, greedy
Average linkage (Maqbool and Babri 2007)	✗	✔	Arbitrary decision, greedy
CA algorithm (Saeed et al. 2003)	✗	✔	Arbitrary decision, greedy
WCA algorithm (Maqbool and Babri 2007)	✗	✔	Arbitrary decision, not to consider the number of access to a feature, greedy
Information-theoretic software clustering (Andritsos and Tzerpos 2005)	✔	✔	Arbitrary decision, greedy
Cooperative clustering (Naseem et al. 2013)	✗	✔	Arbitrary decision, greedy
Clustering software using hybrid XOR similarity function (Srinivas et al. 2013a, b)	✗	✔	Arbitrary decision, greedy
A probabilistic based approach toward software system clustering (Corazza et al. 2010)	✔	✗	Need to determine the number of modules
Software re-modularization by estimating structural and conceptual relations among classes and using hierarchical clustering (Rathee and Chhabra 2017)	✗	✔	Arbitrary decision, greedy

Table 2 Local search-based algorithms for modularization

Approach	Considering the non-structural features	Considering the structural features	Single-objective/multi-objective	Disadvantages
Hill climbing algorithm (Mitchell and Mancoridis 2008)	✗	✔	Single-objective	Trapping in local optima, if there is no call graph it will not work
SA algorithm (Isazadeh et al. 2017)	✗	✔	Single-objective	Trapping in local optima, if there is no call graph it will not work
Multiple hill climbing approach (Mahdavi 2005)	✗	✔	Single-objective	Trapping in local optima, if there is no call graph it will not work
Learning Automata (Izadkhah et al. 2016)	✗	✔	Single-objective	Trapping in local optima, if there is no call graph it will not work
Large neighborhood search applied to the software module clustering problem (Monçores et al. 2018)	✗	✔	Single-objective	If there is no call graph it will not work
Unifying syntactic and semantic features (Misra 2012; Misra et al. 2012)	✔	✔	Multi-objective	Trapping in local optima
HC-SMCP (Huang and Liu 2016)	✗	✔	Single-objective	If there is no call graph it will not work

based algorithms from three different search-based strategies, are adapted to optimize this objective function.

3.1 New multi-objective fitness function

The new multi-objective function is able to consider both structural and non-structural features simultaneously and also in separation. The structural features considered are calling dependency and inheritance similarity between artifacts, and the non-structural features considered are similarity between identifier names and similarity between comments. Clearly, for having a good modularization the following should hold:

Table 3 Global search algorithms for modularization

Approach	Considering the non-structural features	Considering the structural features	Single-objective/multi-objective	Disadvantages
Bunch (Mitchell and Mancoridis 2002; Mitchell and Mancoridis 2006; Mitchell and Mancoridis 2008)	✗	✔	Single-objective	If there is no call graph it will not work
DAGC (Parsa and Bushehrian 2005)	✗	✔	Single-objective	If there is no call graph it will not work
Spectral and meta-heuristic algorithms (Shokoufandeh et al. 2005)	✗	✔	Single-objective	Encoding complexity compared to Bunch
Modified firefly algorithm (Mamaghani and Hajizadeh 2014)	✗	✔	Single-objective	If there is no call graph it will not work
A multi-agent evolutionary algorithm (Huang et al. 2017)	✗	✔	Single-objective	It does not work well for large graphs
Genetic algorithms with *K*-means (Cincotti et al. 2002)	✗	✔	Single-objective	If there is no call graph it will not work
Socio-evolutionary algorithms (Jeet and Dhir 2016a, b)	✗	✔	Single-objective	If there is no call graph it will not work
Harmony search (Chhabra 2017)	✗	✔	Single-objective	Need to determine the number of modules
GA-SMCP (Huang and Liu 2016)	✗	✔	Single-objective	If there is no call graph it will not work
Hyper-heuristic approach (Kumari and Srinivas 2016)	✗	✔	Multi-objective	If there is no call graph it will not work
Black Hole Algorithm (Jeet and Dhir 2016a, b)	✗	✔	Multi-objective	If there is no call graph it will not work
Estimation of distribution approach (Tajgardan et al. 2016)	✗	✔	Single-objective	If there is no call graph it will not work
Search-based object-oriented software re-structuring with structural coupling strength (Chhabra 2015)	✗	✔	Single-objective	If there is no call graph it will not work
Measuring the impact of code dependencies on software architecture recovery techniques (Lutellier et al. 2017)	✗	✔	Single-objective	If there is no call graph it will not work

Table 4 Combining global and local search-based algorithms for modularization

Approach	Consider the non-structural feature	Consider the structural feature	Type of algorithm	Disadvantages
Clustering of software systems using new hybrid algorithms (Mamaghani and Meybodi 2009)	✗	✔	Single-objective	If there is no call graph it will not work
Finding building blocks for software clustering (Mahdavi et al. 2003)	✗	✔	Single-objective	If there is no call graph it will not work

Table 5 Graph-based algorithms for modularization

Approach	Consider the non-structural feature	Consider the structural feature	Type of algorithm	Disadvantages
Neighborhood tree algorithm (Mohammadi and Izadkhah 2018)	✗	✔	Single-objective	If there is no call graph it will not work
Spectral methods to software clustering (Shokoufandeh et al. 2002)	✗	✔	Single-objective	If there is no call graph it will not work
WDCG (Qiu et al. 2015)	✗	✔	Single-objective	If there is no call graph it will not work

- The similarity between the comments should be maximized.
- The similarity between identifier names should be maximized.
- If there exist the inheritance relationships between artifacts, the similarity between them should be increased.
- Considering the calling dependency, the inter-connections within a module should be maximized and between modules should be minimized.

The overall process of the proposed method is shown in Fig. 1. To calculate these objectives (Fig. 1), we use the following relationships:

Textual similarity This criterion calculates the similarity between the two artifacts, considering the similarity between the comments in these two artifacts. Two artifacts are similar if they contain some of the same features. To calculate this similarity between two software's artifacts, we extract the comment strings from the source code and then tokenizes the strings into separate words. For example, consider the following comment.

Then, the stop words are eliminated from the lists generated, e.g., at, the, will, and so on. Finally, using Eq. 3, a co-occurrence matrix is constructed where the rows represent the artifacts and columns represent the number of repetitions of tokens.

$$\delta_{\text{textual}}[i,j] = \sum_{r=1}^{r=t} C[i,r]C[j,r] \Big/ \sqrt{\sum_{r=1}^{r=t} C[i,r]^2}\sqrt{\sum_{r=1}^{r=t} C[i,r]^2} \tag{3}$$

t, the total number of extracted tokens in the textual features; $C[i,r]$, the number of times of token r occurs at artifact i; $C[j,r]$, the number of times of token r occurs at artifact j; $\delta_{\text{textual}}[i, j]$, the textual similarity calculated between the two artifacts i and j.

Identifier names similarity The artifact names, such as class name, method name, function name, are extracted from the source code and then tokenizes the extracted artifact names into separate words. Equation 4 is used to create the co-occurrence matrix is. For example:

''Finance Control Manager''

includes the following tokens:

Finance, Control,Manager

$$\delta_{\text{identifier}}[i,j] = \sum_{r=1}^{r=Z_c} C[i,r]C[j,r] \Big/ \sqrt{\sum_{r=1}^{r=Z_c} C[i,r]^2}\sqrt{\sum_{r=1}^{r=Z_c} C[i,r]^2} \tag{4}$$

Z_c, the total number of extracted tokens from artifact names; $\delta_{\text{identifier}}[i, j]$, the identifier names similarity between the two artifacts i and j.

Inheritance-based similarity To calculate the similarity between classes considering inheritance dependency, we extract the list of class-names, which are being extended or implemented by the class. For example:

Class className implements A1, A2, A3

The inheritance list extracted for the className would be $\{A1, A2, A3\}$. Let $W_{\text{in}}[i]$ and $W_{\text{in}}[j]$ denote the inheritance list calculated for two classes i and j, respectively. This similarity is calculated by Eq. 5.

$$\delta_{\text{in}}[i,j] = \left|\frac{W_{\text{in}}[i] \cap W_{\text{in}}[j]}{W_{\text{in}}[i] \cup W_{\text{in}}[j]}\right| \tag{5}$$

$\delta_{\text{in}}[i, j]$ the inheritance similarity between the two classes i and j.

To get the inheritance list for each class, we perform the following:

(a) Find the names of the classes which are in the inheritance list of this class.
(b) Find the names of the classes that this class is in their inheritance list.

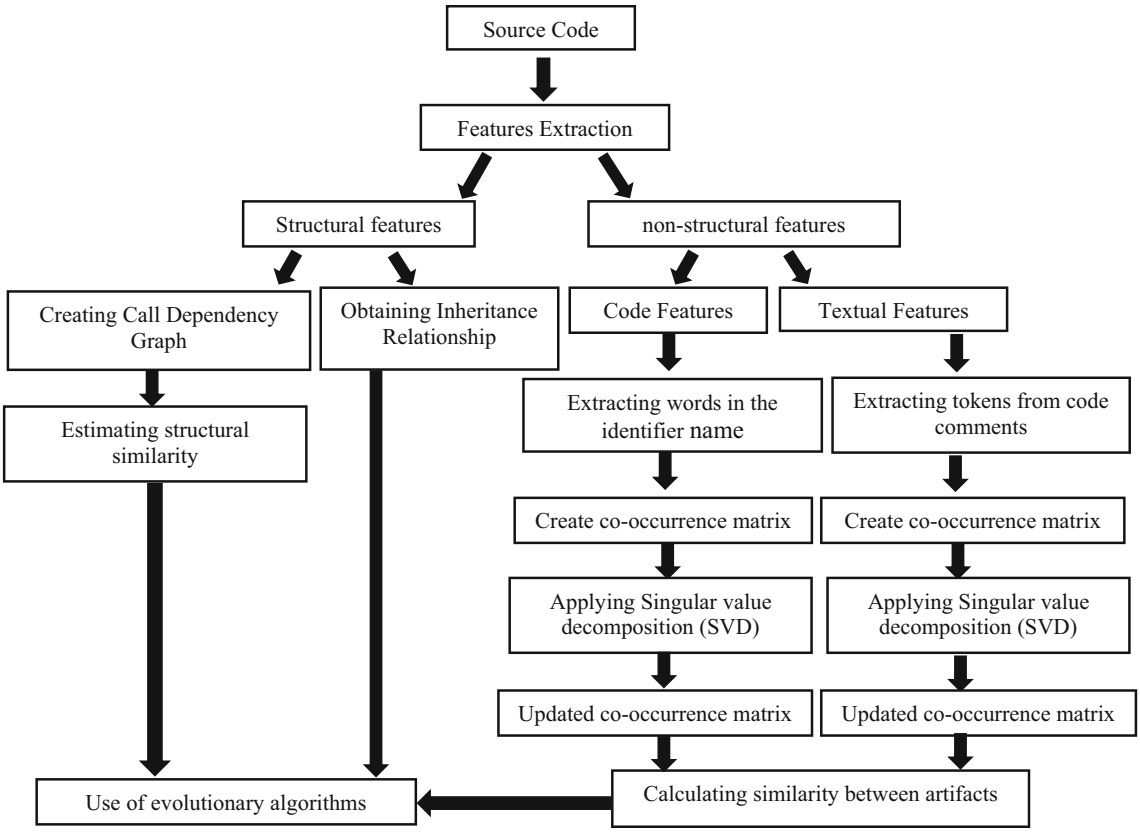


Fig. 1 The overall process of combining structural and non-structural features

Considering these three similarity functions, the combined similarity between artifacts i and j is calculated by Eq. 6.

$$\alpha_t + \alpha_c + \alpha_{\text{in}} = 1$$
$$\delta_{\text{combined}}[i,j] = \alpha_t \times \delta_{\text{textual}}[i,j] + \alpha_c \times \delta_{\text{class}}[i,j] + \alpha_{\text{in}} \times \delta_{\text{in}}[i,j] \quad (6)$$

Structural similarity To calculate the structural similarity, we consider the calling dependency between two artifacts. Let μ_i and $\varepsilon_{i,j}$ represent the intra-connections of module i and inter-connections between two modules i and j, respectively. The quality of module i and an obtained modularization calculated by Eqs. 7 and 8, respectively.

$$\text{CF}_i = \begin{cases} 0 & \mu_i = 0 \\ \mu_i + \frac{1}{2}\sum_{i=1, j\neq i}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i}) & \text{otherwise} \end{cases} \quad (7)$$

$$\delta_{\text{strutural}} = \sum_{i=1}^{k} \text{CF}_i \quad (8)$$

where k indicates the number of modules. Let m denote the number of artifacts, considering the combined similarity and structural similarity functions, the new multi-objective fitness function is calculated by Eq. 9.

$$\delta_n + \delta_s = 1$$
$$\text{MOF} = \delta_n \times \frac{\sum_{i,j=1}^{m} \delta_{\text{combined}}[i,j]}{\frac{m(m-1)}{2}} + \delta_s \times \frac{\delta_{\text{strutural}}}{k} \quad (9)$$

3.2 Discovering latent semantic between features

Latent Semantic Analysis (LSA) is a technique in natural language processing for discovering hidden concepts in document data and applied to a term-document matrix (Dumais, 2004). LSA represents the semantic similarity between documents that are utilized in the similar context. We use this technique for discovering latent semantic between non-structural features. An LSA uses a mathematical technique from linear algebra named Singular Value Decomposition (SVD) to reveal latent semantic. This technique applies on a term-document matrix and is represented by Eq. 10 and Fig. 2.

$$M = USV^{\text{T}} \quad (10)$$

S, (r: rank of the matrix); U_{mr}, (m: document, r: rank of the matrix); V^{T}_{nr}, (n: term, r: rank of the matrix).

Dimensionality reduction Let M indicate an artifact-feature matrix containing features counts per artifact (rows represent each artifact and columns represent unique

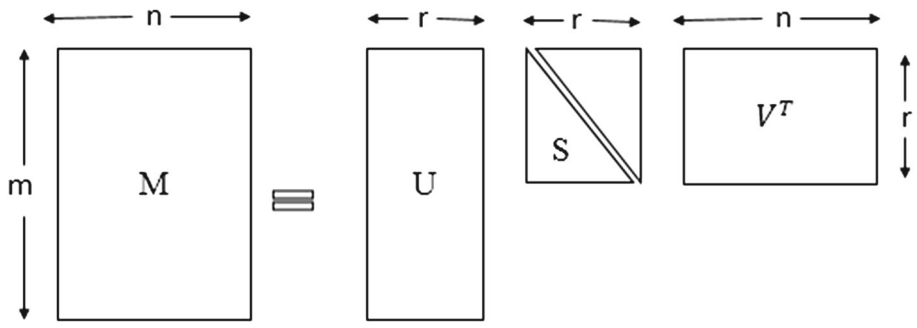


Fig. 2 The form of singular value decomposition after dimensionality reduction

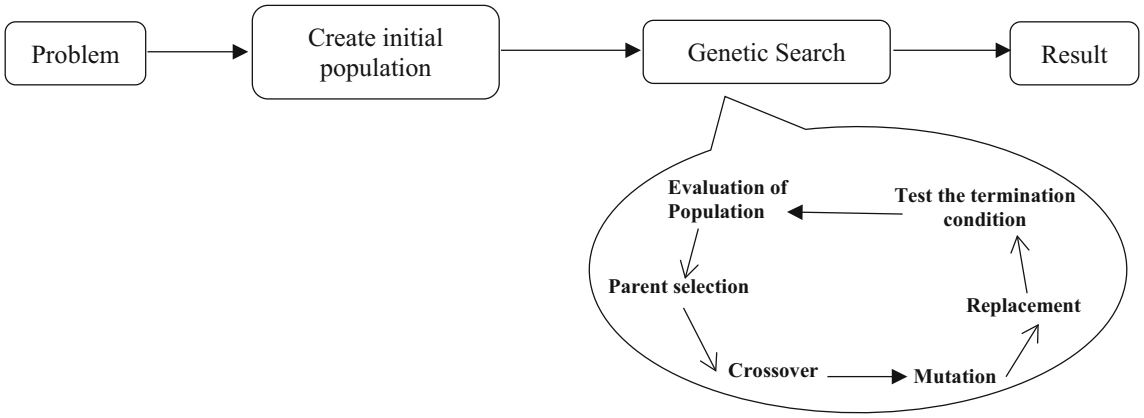


Fig. 3 The overall evolutionary process of genetic algorithm

features) is formed from the source code. We use tf-idf (Abualigah et al. 2017a, b) to compute the value that each entry in the matrix should take. Then, SVD is used to reduce the number of features (i.e., dimensionality reduction) while preserving the similarity structure among artifacts. After reducing this matrix dimensions, artifacts that share with together, at the start, become more similar to together, and unlike artifacts, become more dissimilar as well. Therefore, dimensionality reduction makes it possible two artifacts that the exact same token do not appear in all of them, but both are from a particular subject, become more similar.

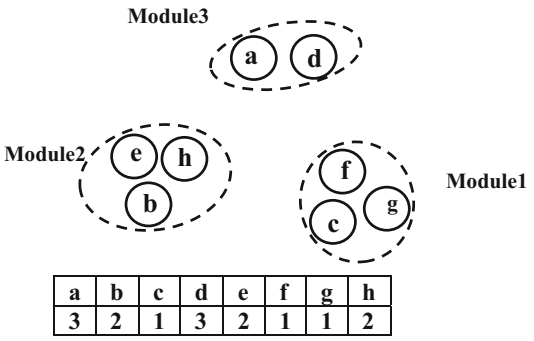
After extracting features and applying dimensionality reduction, we use genetic algorithm with two encodings, combining genetic algorithm and hill climbing in two different modes, and estimation of distribution to maximize the multi-objective fitness function. These algorithms are explained in the following subsections.

3.2.1 Modularization using genetic algorithm

A genetic algorithm is a search-based optimization technique, which mimics the principles of Genetics and Natural Selection to find a near optimal solution. This kind of algorithms use biological concepts such as selection, crossover, and mutation. This algorithm is an iterative optimization procedure that works with a population of possible solutions represented by a chromosome aiming to improve the quality of chromosomes in each iterative. The steps to solve a problem with the genetic algorithm are shown in Fig. 3.

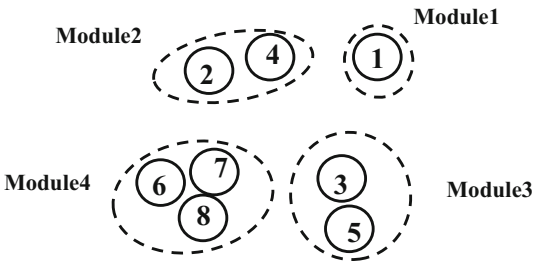
The main problem with a genetic algorithm is how to encode the solutions. Selecting the proper encoding scheme for solutions is a vital task. We use two types of encoding in the developed genetic algorithm named value-based encoding and permutation-based encoding. In both encodings, the number of genes of a chromosome is equal to the number of artifacts. Let N denote the number of artifacts in the source code. In the value-based encoding, the content of each gene indicates a module number $(1 \leq \text{module number} \leq N)$ that contains the corresponding artifact. In the permutation-based encoding, the content of gene 'm' of a chromosome includes an artifact number like 'p' $(1 \leq p \leq N)$, so that if 'p' is equal or greater than 'm', then 'm' is placed in a new module otherwise 'm' belongs to the same module that 'p' is located. Figures 4 and 5 give an example of these two encodings.

A fitness function is employed to evaluate the quality of each chromosome. In our algorithm, the fitness function for each chromosome is calculated as follows:



a	b	c	d	e	f	g	h
3	2	1	3	2	1	1	2

Fig. 4 Value-based encoding



m	1	2	3	4	5	6	7	8
p	1	4	5	2	3	8	6	7

Fig. 5 Permutation-based encoding

1. From the updated co-occurrence matrix, the artifact-to-artifact similarity is calculated using Eq. 6;
2. The structural similarity is calculated for obtained modularization using Eq. 8;
3. The multi-objective fitness function is calculated using Eq. 9.

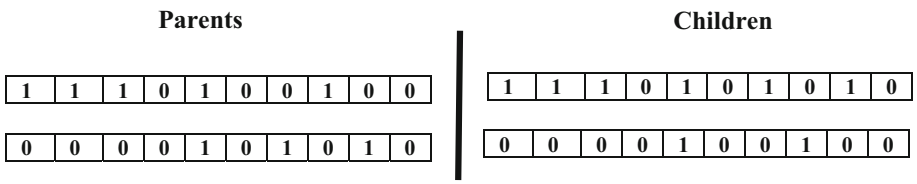
The roulette wheel method has been used to select parents. The main task of the crossover operator is to improve the fitness of a population and the most important task of the mutation operator is to avoid convergence to the local optimum. To apply a crossover operator, for each chromosome, a random number is chosen between [0, 1], if this number is less than or equal to the probability of crossover, the crossover is applied to that chromosome. The one-point crossover that has been applied is shown in Fig. 6.

To apply the mutation operator, the total number of genes that exist in the population is computed. The resulting value in the mutation probability is multiplied to determine the number of genes that can be mutated. To determine the location of the gene, a random gene is selected between genes; then, a random number is chosen between [0, 1], that if it is smaller or equal to the mutation probability, the selective gene with a random value among the number of genes of the chromosome is replaced. Replacing is a selection method that forces an evolutionary algorithm to keep a number of the best individuals in each generation. After producing a new population and replacing it with the previous population, the chromosome that has the highest fitness is replaced by the chromosome that has the worst fitness. As with any other algorithm, this algorithm there must be a termination condition which, in this work, is to achieve a predetermined number of generations, then the best chromosome is given as the best solution.

3.2.2 Modularization using distribution estimation algorithm

Estimation of Distribution algorithm (EoD) is a stochastic optimization that uses a probabilistic model to represent the distribution over candidate solutions and utilize this model to guide the search for finding an optimum. In EoD, a new population is generated without the use of crossover and mutations operators. Instead, the new population is sampled from a probability model. To protect the building blocks found during the evolutionary process of the algorithm is the main aim of these algorithms. In this section, we present a new population-based EoD algorithm with a new probabilistic model for software modularization. Algorithm 1 describes the proposed algorithm.

Fig. 6 One-point crossover



Algorithm 1: Estimation of Distribution Algorithm

1) Creating a probability Matrix
- Create a square matrix n * n (n denotes the number of artifacts)
- Initially, the values of all elements except the main diagonal are 1/n.

2) Generate initial Population
3) Evaluating the chromosomes by Eq. 9
4) Selecting and Updating:
- Find the highest quality and lowest quality chromosomes
- Update the probability model from the highest quality and lowest quality chromosomes as follows:
  - If two artifacts i and j are located in the same module in the highest quality chromosome but are not located in the same module in the lowest quality chromosome, their probability values are added in the probability matrix by 1/t, where t is the number of iterations.
  - If two artifacts i and j are located in the same module in the lowest quality chromosome but are not located in the same module in the highest quality chromosome, their probability in the probability matrix are reduced by 1/t, where t is the number of iterations.

5) Sampling (creating a new population using new probability matrix):
- Creating a new population using the new probability model
  - Use the upper triangular matrix to produce each chromosome
  - Select, randomly, a row from probability matrix
  - Generate a random number for each selected row element between [0,1]
    - If this value is smaller or equal to the probability of the element, the related artifacts are located in the same module.

6) Elitism
7) Test the termination condition

We illustrate with an example how to create and update a probabilistic model. Suppose a software system consists of five artifacts denoted by F1–F5. Figure 7 shows the initial probabilistic model matrix for this software system so that the values of all elements except the main diagonal are $1/n$ (n is the number of artifacts). Also, assume that a, b and $t = 10$ indicate the chromosome with the highest quality in the initial population, the chromosome with the lowest quality in the initial population, and the algorithm iterations, respectively (Fig. 8). To update the probabilistic model matrix, since two artifacts F1 and F2 are located in the same module in the highest quality chromosome but are not located in the same module in the lowest quality chromosome, hence, their probability values are added in the probability matrix by 1/10. In contrast, because two artifacts F3 and F4 are located in the same module in the lowest quality chromosome but are not located in the same module in the highest quality chromosome; hence, their probabilities in the probability matrix are reduced by 1/10. Figure 9 shows the updated probabilistic model matrix.

In the EoD, the new population is sampled as follows: from the probabilistic model, a row is randomly selected. Given the upper triangular part of the probabilistic matrix,

	F1	F2	F3	F4	F5
F1	0	0.2	0.2	0.2	0.2
F2	0.2	0	0.2	0.2	0.2
F3	0.2	0.2	0	0.2	0.2
F4	0.2	0.2	0.2	0	0.2
F5	0.2	0.2	0.2	0.2	0

Fig. 7 A sample initial probability matrix for a software system with five artifacts

	F1	F2	F3	F4	F5
a	2	2	1	3	1
b	2	1	3	3	3

a: the highest quality chromosome

b: the lowest quality chromosome

Fig. 8 Selecting the highest quality and lowest quality chromosome

	F1	F2	F3	F4	F5
F1	0	0.3	0.2	0.2	0.2
F2	0.3	0	0.2	0.2	0.2
F3	0.2	0.2	0	0.1	0.2
F4	0.2	0.2	0.1	0	0.1
F5	0.2	0.2	0.2	0.1	0

Fig. 9 Updated probability matrix

for each element of the selected row, a random number is generated between 0 and 1; if this random number is equal to or less than the probability of the element in the row, these two artifacts will be placed in the same module in the new chromosome. This is repeated for all elements in the selected row. If all elements of the selected row in the probability matrix were examined, but the chromosome was not completely modularized, then a new row from the probability model matrix would be randomly selected and, until the chromosome is completed, the random number generation operation is repeated.

For example, consider Fig. 10. In this figure, R1 shows the selected row and F1–F5 represents artifacts. For each element of the selected row, a random number is generated

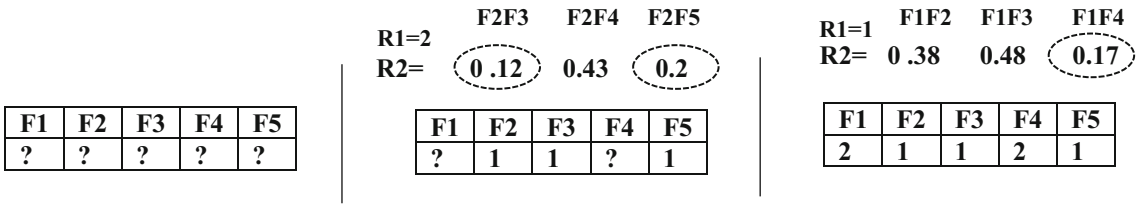


Fig. 10 Create a new population using a new probability matrix

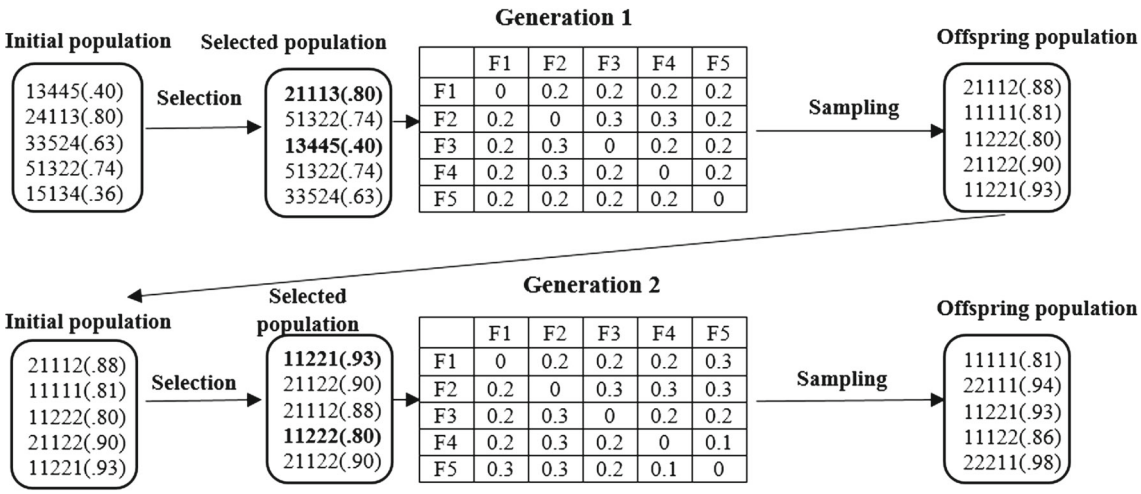


Fig. 11 Two first generations of distribution estimation algorithm

between 0 and 1, which is represented by $R2$. Suppose $R1 = 2$ and the random number generated is equal to 0.12; since this random number is smaller than the probability value of the element in F2F3 (Fig. 9), these two artifacts in the chromosome will be in the same module (Fig. 10). A random number is generated for all elements of row 2; as shown in Fig. 10, the random number 0.43 is larger than the probable value in F2F4. So this entry will remain currently empty. The random number 0.2 is generated for the F2F5 element, which is equal to its probability value; so F5 with F2 and F3 will be placed in the same module. Since all the elements of the selected row in the probability matrix were examined, but the chromosome was not completely modularized, the new row is randomly selected again, as shown in Fig. 10, $R1 = 1$ is randomly selected, and the steps above are repeated until the chromosome is completed.

Figure 11 shows first two generations of our proposed algorithm, and Fig. 12 shows the construction process of probability model for distribution estimation algorithm.

3.2.3 Modularization using the genetic and hill climbing algorithm

The hill climbing algorithm is one of the most popular local search algorithms to modularize the source code. It is an iterative algorithm that starts with a random initial solution from the search space and then, a new solution is produced by modifying the initial one (Isazadeh et al. 2017). Two major drawbacks of the hill climbing algorithm are trapping in local optima and its low stability. In the hill climbing, several percent of the chromosomes are selected from a population and each chromosome selected from this population is regularly improved by this algorithm. The concept of neighbor modularization is illustrated in Fig. 13. In this figure, A shows the current modularization and B–D are three neighbor modularizations for it. For example consider the node a, so that in B, it moved into module 4, in C, it moved into module 2, and in D, it moved into module 3.

We define two versions of hill climbing namely Next Ascent Hill Climbing (NAHC) and Steepest Ascent Hill Climbing (SAHC). In NAHC, the algorithm stops when the first neighboring modularization found with a higher fitness value. In SAHC, the algorithm stops after examining the percentage of neighboring modularizations and choosing the one with the largest fitness value. The total number of neighbors for the current modularization is calculated by Eq. 11. Algorithm 2 shows the hill climbing algorithm.

Algorithm 2: Hill-Climbing Algorithm

Input: getting a current modularization
Output: the best neighbor of a modularization
Step1- Compute the total number of neighbors for current modularization using Eq. 11, and determine the threshold t. This threshold determines how many percents of the current modularization neighbors must be assessed to find the next modularization.
Step 2- Searching for the best neighbor for current modularization:

- In the NAHC: algorithm generates a random neighbor modularization and examine it. If the quality of the neighbor modularization, i.e. MOF, is higher than the current modularization, another neighbor modularization is produced to the new modularization, and so on until no further improvements can be found.
- In the SAHC: algorithm generates all neighbors for a modularization and examine them, then, the best neighbor modularization is returned.

Total number of neighbors
= total number of modules in modularization
× the total number of nodes in the modularization
× t (11)

The genetic algorithm (GA) is a global search algorithm and the hill climbing algorithm is a local search algorithm. To use the advantage of these two search algorithms, we combine them. This approach is applied in two ways: (1) the combining genetic algorithm with hill climbing algorithm named CGH, and (2) applying genetic algorithm and

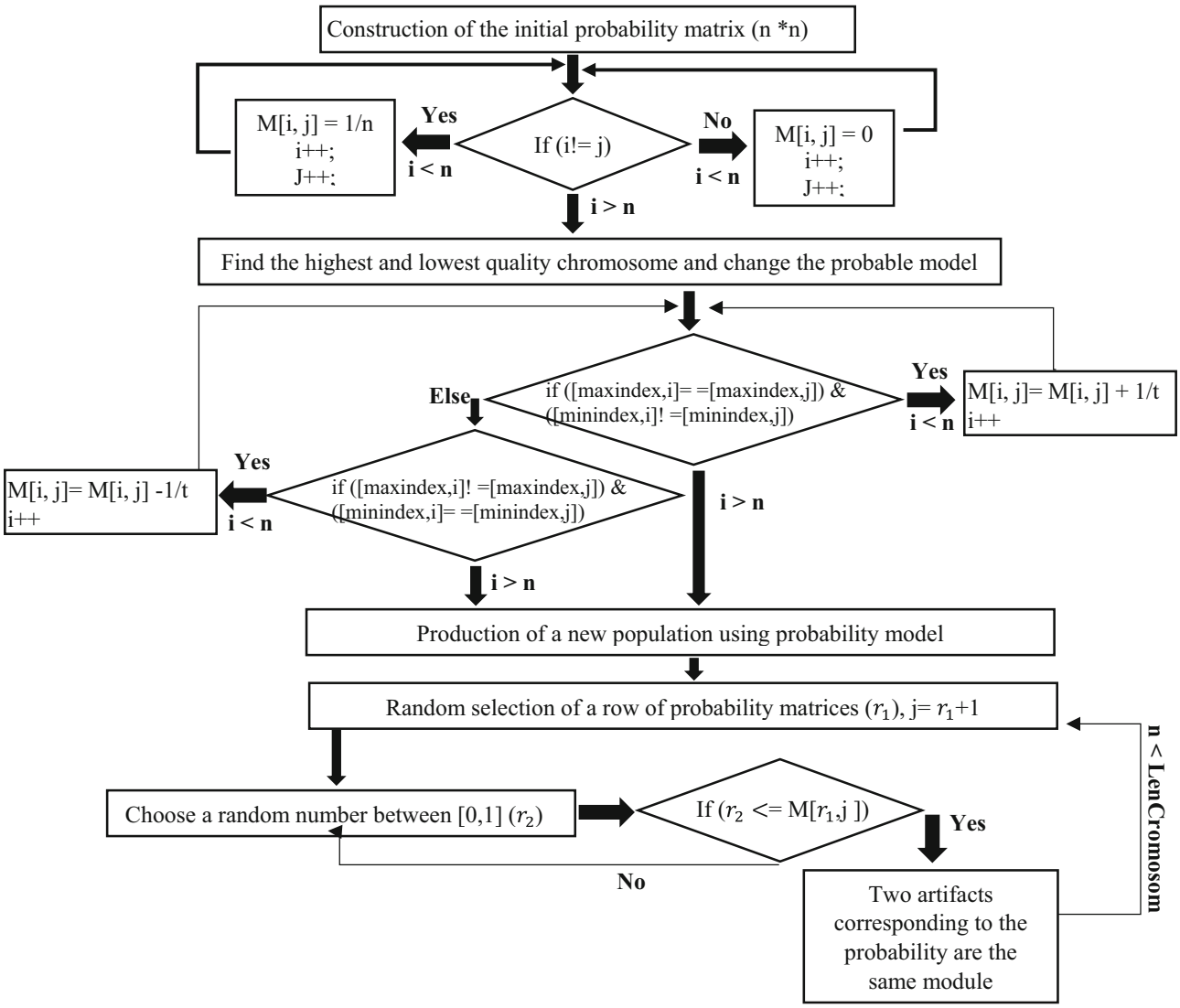


Fig. 12 Probability model of distribution estimation algorithm

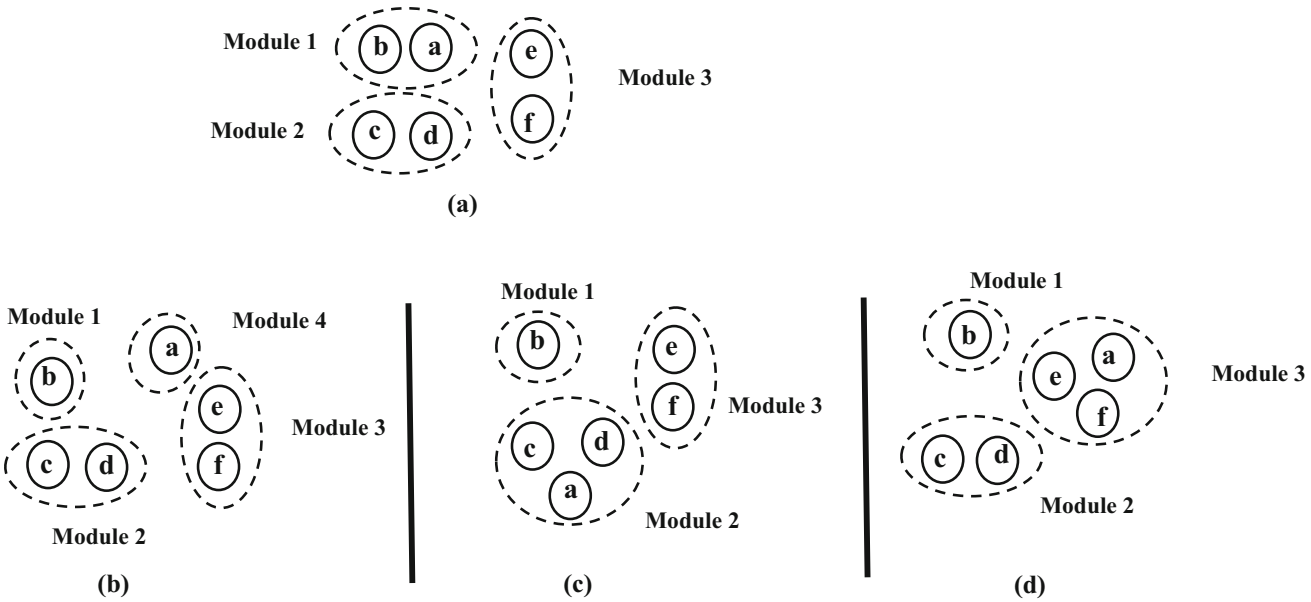


Fig. 13 Neighborhood for node a

then running hill climbing algorithm once at the end of process named CGoH. In the combination of these two algorithms, the process of the genetic algorithm is similar to Sec. 3.2.1, with the difference that in the proposed approach after the elitism stage, on the percentage of the selected population, the hill climbing algorithm is applied. Among the selected chromosomes, a chromosome has the highest fitness value, and the rest are randomly selected. In the CGH, in each generation of the genetic algorithm, a number of solutions are given to the hill climbing algorithm aiming to improve the quality of these solutions. Figure 14 shows this process. In the CGoH, after the run of the genetic algorithm is completed, for improving the quality of the modules obtained, the solutions are given to the hill climbing algorithm. Figure 15 shows this process.

4 Experimental results

In this section, the performance of MOF is evaluated using the developed algorithms. Also, the effect of discussed features (structural and non-structural) is addressed on the modularization quality of a software system. The ground-truth architectures are used to evaluate the reliability of the modularization algorithms (Garcia et al. 2013). In this paper, the MoJo (Tzerpos and Holt 1999) and MoJoFM (Wen and Tzerpos 2004) criteria are exploited to validate and compare the modularization obtained by the algorithms on Mozilla Firefox, which have a ground-truth structure. We used Build, Browser, Db, which are the folders from Mozilla Firefox, for this aim. The MoJo criterion calculates the number of "move" and "join" operations needed to reach from one modularization to another and is defined as a distance criterion. The less number of moves and join operations, the more similar the two clusters are. The MoJoFM measure is specified in Eq. 12.

$$\text{MoJoFM} = 1 - \frac{\text{mno}(A,B)}{\max(\text{mno}(\forall A,B))} \times 100\% \tag{12}$$

where $\text{mno}(A,B)$ indicates the minimum number of move or join operations required to transform modularization A to B and $\max(\text{mno}(\forall A,B))$ indicates the maximum of the minimum number of possible operations of move or join needed to transform A to B. This measure allows us to compare the modularization obtained by various techniques according to their similarity with the expert modularization, i.e., the higher value of MoJoFM, the more similar of modularization obtained by an algorithm to expert modularization. A value of 100% confirms that the modularization obtained is the same with an expert's modularization.

In brief, a small value of MoJo and large value of MoJoFM show that the resultant modularization is close to the domain expert modularization (ground-truth modularization). Let n denote the number of artifacts, Table 6 gives the parameter setting for the experiments. In genetic algorithms, the number of generations is usually greater than the population size. Therefore, we have followed this principle in the proposed algorithms, and we also considered this value linearly and a coefficient of the number of artifacts. To set the probability of mutation and crossover operators, twenty different pairs of mutation and crossover probabilities have been tested on the Db folder using the proposed algorithms. For example, Fig. 16 shows one of

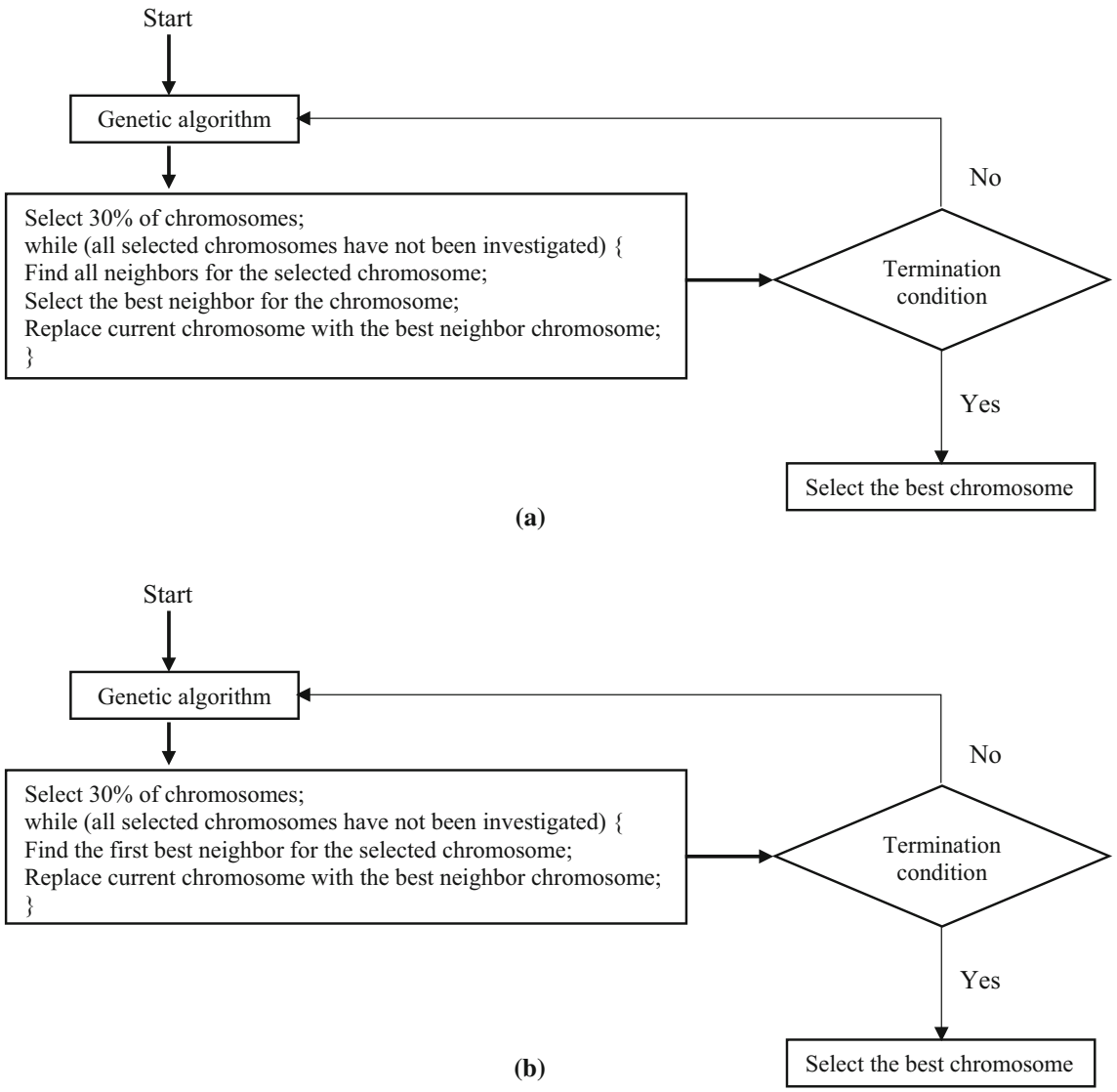


Fig. 14 The combination of genetic and hill climbing algorithms. **a** Combining genetic algorithm and Steepest Ascent Hill climbing algorithm (SAHC), **b** combining genetic algorithm and Next Ascent Hill climbing algorithm (NAHC)

these experiments using the genetic with value-based encoding.

The MoJoFM values computed for the algorithms, considering each extracted feature and combination of these features, on three folders of Mozilla Firefox are shown in Tables 7, 8, 9 and 10. NA (Next Ascent) and SA (Steepest Ascent) indicate the type of hill climbing, and "Best" and "Average" show the best and mean results of the used criterion among 10 runs of the algorithms.The best results in Tables 7, 8, 9 and 10 are highlighted in bold. As shown in Tables 7, 8, 9 and 10, in 75% of the cases, the distribution estimation algorithm (EoD) gives better results compared to the best MoJoFM, and in 67% of the cases, this algorithm gives better results compared with the average. After that, the combination of genetic algorithm and hill climbing, i.e., CGH, leads to better results.

In the following, the convergence and stability of the algorithms and features are evaluated; and moreover, for more evaluation, *T*-test is used to assess the stability.

Convergence To evaluate more precisely, we investigate the results of the algorithms from the convergence diagrams. Figures 17, 18, 19, 20, 21, 22 and 23 indicate the convergence of algorithms on features. The algorithms for Build, Browser, and Db folders are, respectively, executed in three generations of 400, 800 and 1000. The best individual of each generation in these figures is represented by a blue line. As can be seen in these figures, the best individual of every generation is above the average of that generation's fitness. In comparison between algorithms, the CGH and EoD algorithms are convergent in a smaller number of iteration. Compared to the features, the

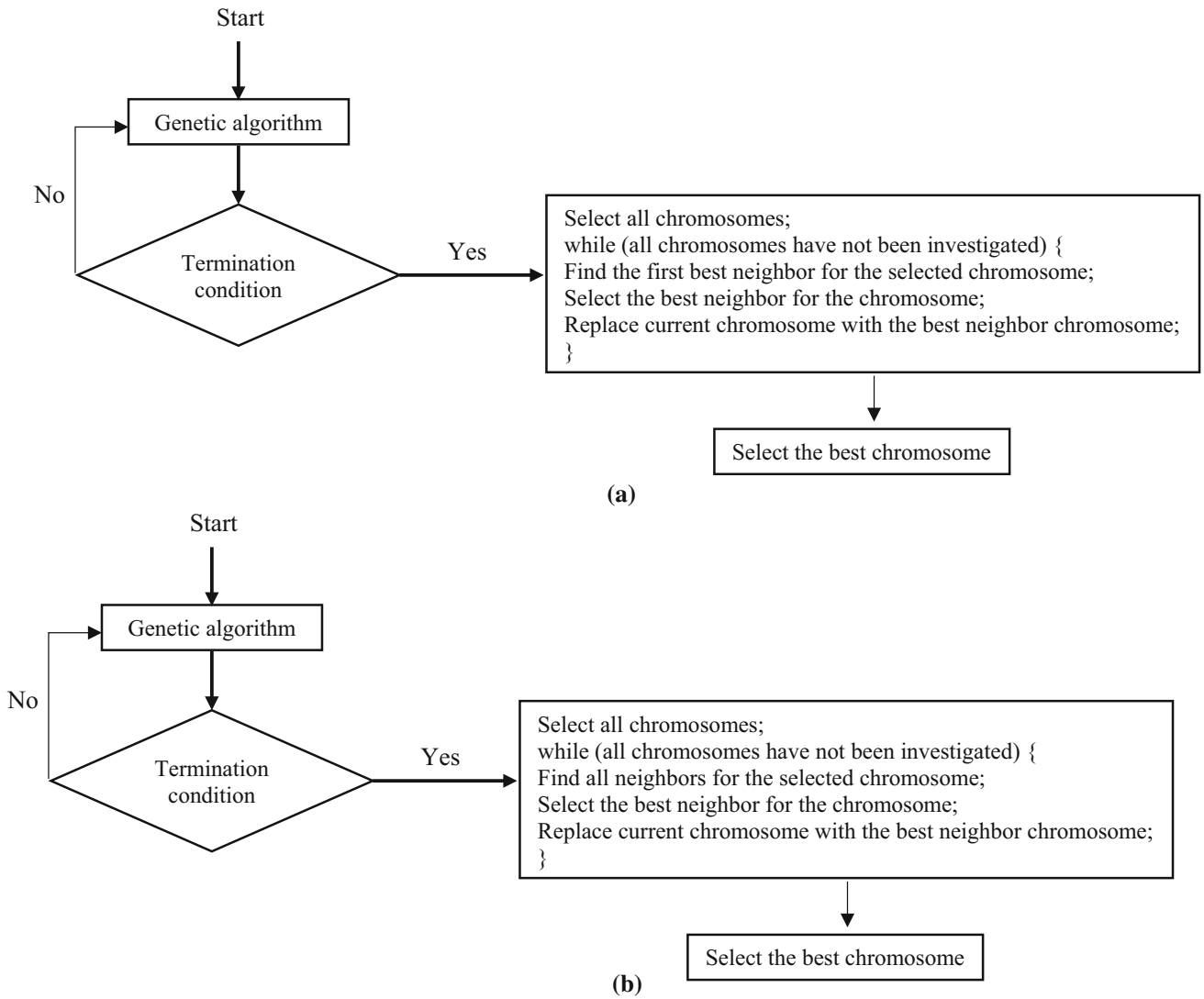


Fig. 15 Applying genetic algorithm and then running hill climbing algorithm once at the end of process. **a** Combining genetic algorithm and Next Ascent Hill climbing algorithm (NAHC), **b** combining genetic algorithm and Steepest Ascent Hill climbing algorithm (SAHC)

Table 6 The parameter setting for experiments

Parameters	Value
Popsize	10n
Gen (number of generations)	20n
P_c (crossover probability)	0.8
P_m (mutation probability)	0.05
Selection	Roulette wheel selection
Crossover operation	One-point
Mutation operation	randomly changed a gene
Stop criteria	When convergence is reached

structural feature converges in a smaller number of iterations than the rest, but other features also have no significant difference with the structural feature.

Stability The meta-heuristic algorithms such as genetic algorithms are stochastic optimizers. Hence, to compare the results, they need to be run multiple time and some statistical techniques such t-test should be used to compare the results. The stability of evolutionary algorithms in different runs is one of the problems with these algorithms. An evolutionary algorithm is stable if the results are close to each other, in different runs. Figures 24, 25, 26, 27, 28, 29, 30, 31 and 32 depict stability of algorithms on features. In these figures, the width axis represents the fitness value obtained from different runs and the length axis represents the run number. To calculate the stability, the algorithms are executed ten times. The comparisons between the algorithms in the stability diagrams show that in the CGH and EoD algorithms, the results are close together in different runs. The comparisons between the features in

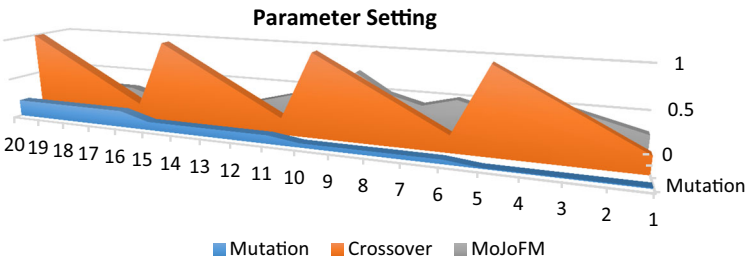


Fig. 16 Parameter setting for crossover and mutation

Table 7 Comparing the algorithms on text features in terms of MoJoFM

	Build		Browser		Db	
	Best (%)	Average (%)	Best (%)	Average (%)	Best (%)	Average (%)
Genetic with value-based encoding	73.68	62.10	40	33.83	51.12	48.17
Genetic with permutation-based encoding	52.63	35.26	23.08	19.23	43.62	35.32
EoD	**78.95**	**78.95**	**50**	**34.71**	**73.4**	**72.87**
CGoH (NA)	78.95	78.95	40	33.75	67.02	66.27
CGoH (SA)	78.95	78.95	50	33.75	68.09	66.56
CGH (NA)	77.78	77.78	33.33	30.51	67.35	67.1
CGH (SA)	77.78	77.78	33.33	26.92	68.09	67.57

Table 8 Comparing the algorithms on identifier names features in terms of MoJoFM

	Build		Browser		Db	
	Best (%)	Average (%)	Best (%)	Average (%)	Best (%)	Average (%)
Genetic with value-based encoding	78.95	70.52	**55**	**47.25**	53.26	50.84
Genetic with permutation-based encoding	52.63	42.63	30.77	23.84	38.3	33.23
EoD	**94.74**	**94.213**	47.5	39.25	**89.36**	**85.32**
CGoH (NA)	73.68	73.16	47.5	42.25	69.15	64.25
CGoH (SA)	68.42	66.84	50	45	69.15	66.48
CGH (NA)	77.22	68.78	53.85	41.79	70.21	68.83
CGH (SA)	88.89	68.11	48.72	40.25	70.21	69.15

Table 9 Comparing the algorithms on structural features in terms of MoJoFM

	Build		Browser		Db	
	Best (%)	Average (%)	Best (%)	Average (%)	Best (%)	Average (%)
Genetic with value-based encoding	68.42	59.47	60	48.75	56.5	53.09
Genetic with permutation-based encoding	55	36.03	46.15	31.72	51.06	45.1
EoD	**84.28**	79.47	52.5	47.25	92.55	90
CGoH (NA)	84.21	**84.21**	65	64.25	96.81	96.17
CGoH (SA)	84.21	82.63	65	64.75	**96.81**	**96.81**
CGH (NA)	83.33	83.33	69.23	64.87	**96.81**	**96.81**
CGH (SA)	83.33	83.33	**76.92**	**71.27**	96.81	96.59

Figs. 29, 30, 31 and 32 demonstrate that using non-structural features for modularization, the stability is also close to the algorithms that use structural features.

T-test Independent samples test has been used to assess the stability of an algorithm by computing the difference between the mean of two populations with each other.

Table 10 Comparing the algorithms considering the multi-objective fitness function in terms of MoJoFM

	Build		Browser		Db	
	Best (%)	Average (%)	Best (%)	Average (%)	Best (%)	Average (%)
Genetic with value-based encoding	73.68	66.31	52.5	40	54.3	52.23
Genetic with permutation-based encoding	52.63	42.10	35.9	26.84	43.62	40.42
EoD	**94.74**	**94.74**	**52.5**	**48.75**	**96.91**	**91.53**
CGoH (NA)	89.47	89.47	50	46	69.15	67.98
CGoH (SA)	89.47	89.47	50	46	69.15	67.98
CGH (NA)	88.89	83.33	48.72	45.12	70.21	68.62
CGH (SA)	94.44	84.44	48.72	45.64	70.21	69.15

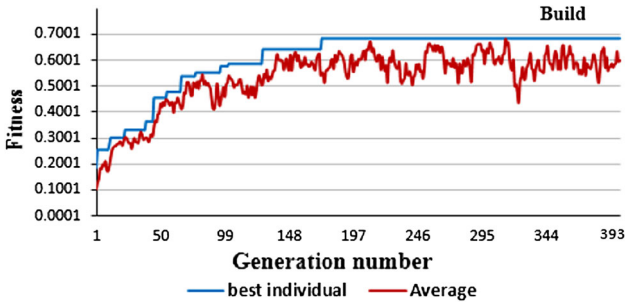


Fig. 17 Genetic with value-based encoding on textual features

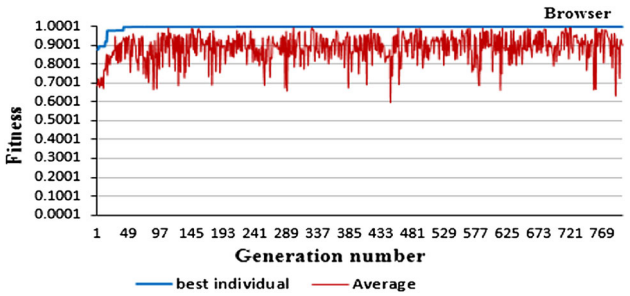


Fig. 18 Genetic with permutation-based encoding on textual features

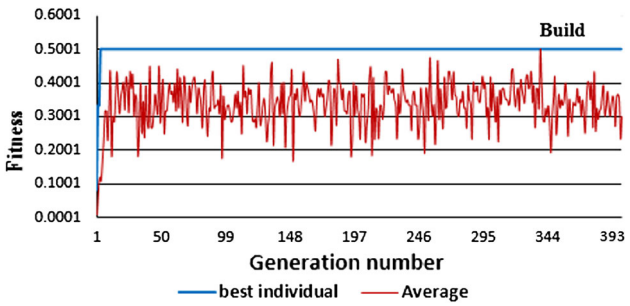


Fig. 19 EoD algorithm on structural features

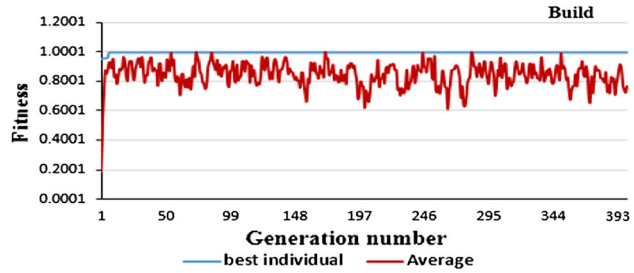


Fig. 20 CGH on text features

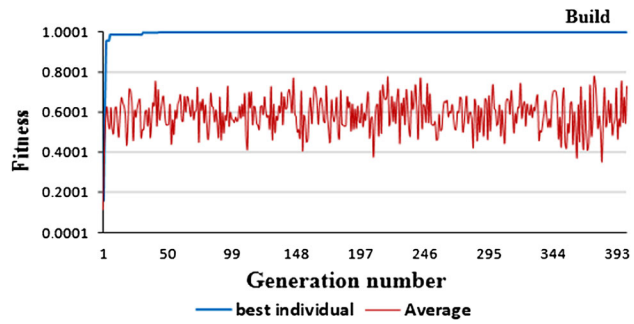


Fig. 21 EoD algorithm on textual features

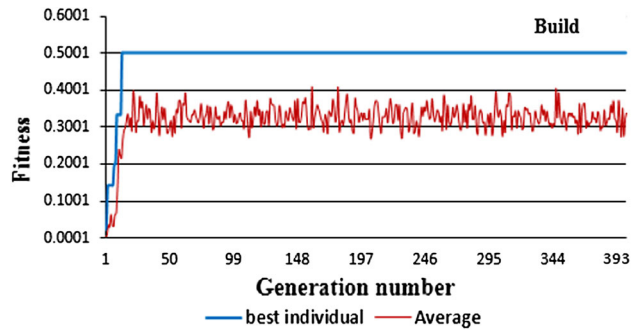


Fig. 22 EoD algorithm on identifier names features

Tables 11, 12, 13 and 14 show *T*-test results for the algorithms. To apply this test, the modularization results of the 10 runs of the algorithms are grouped into two groups of 5, named *N*1 and *N*2. We apply this test in two ways: descriptive statistics (including the first three columns) and the inferential statistics (including last two columns). In the descriptive statistics, Mean, SD and SE Mean denote the

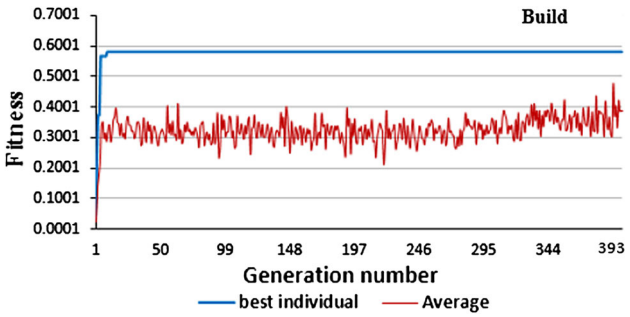


Fig. 23 EoD algorithm on MOF

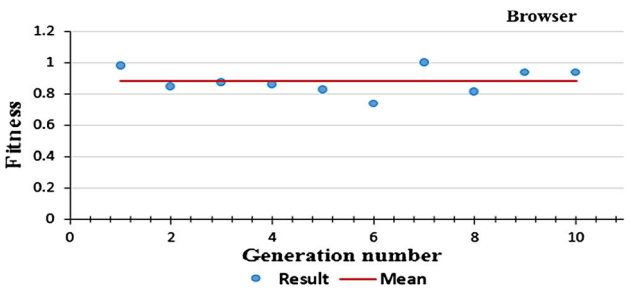


Fig. 24 Genetic with permutation-based encoding on structural features

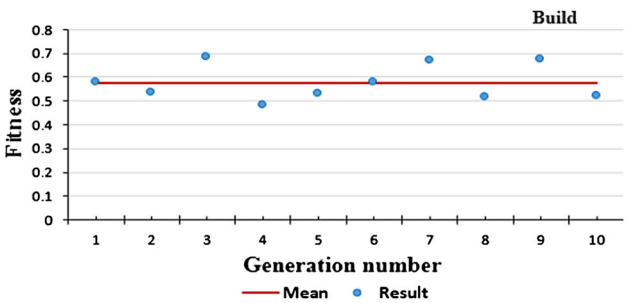


Fig. 25 Genetic with value-based encoding on textual features

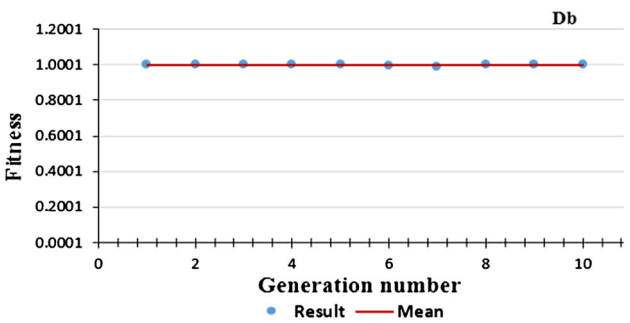


Fig. 26 EoD algorithm on structural features

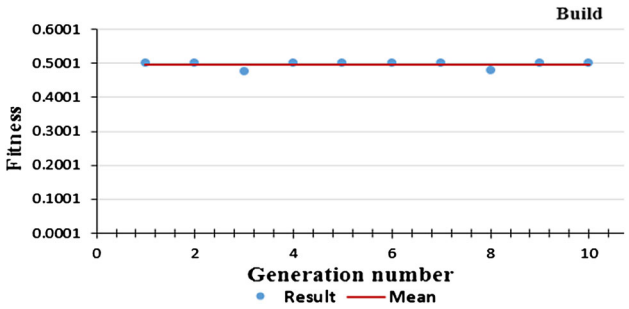


Fig. 27 CGH on structural features

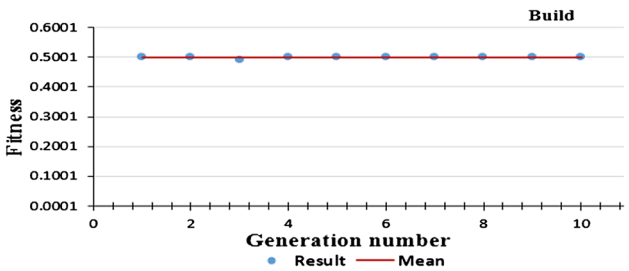


Fig. 28 CGoH on structural features

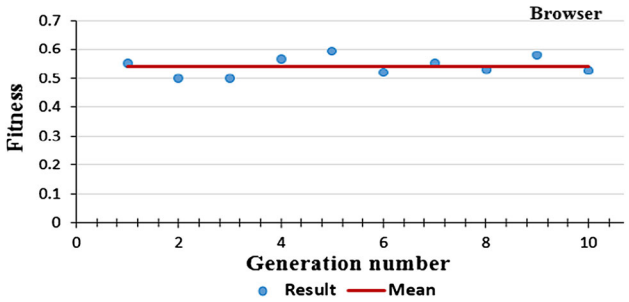


Fig. 29 EoD on textual features

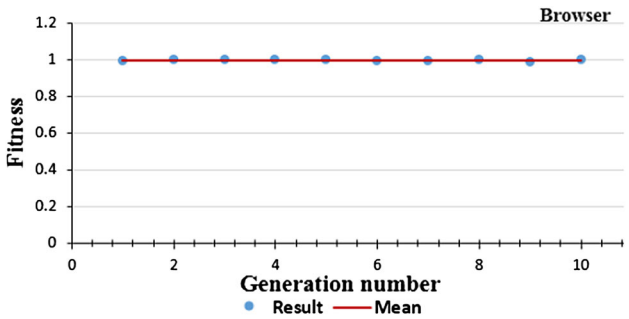


Fig. 30 EoD on identifier names features

average of both groups, the standard deviation of the two groups, and the standard error between mean the two groups, respectively. Inferential statistics also include two parts. In the first part of the fourth column, the output of the inferential statistics, if Sig. be greater than 0.05, the assumption of the equality of variance is not rejected, and the information in the second part of this column is used to assess the average. if Sin. be greater than 0.05, the assumption of the equality of the meanings is determined but if the equality of variance is rejected. The information of the second part of the fifth column is examined for the comparison of the mean. Comparison of stability and

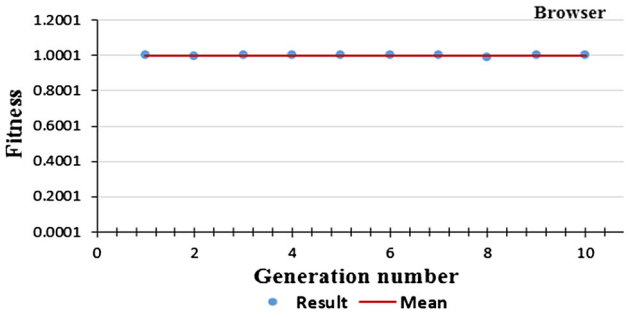


Fig. 31 EoD on structure features

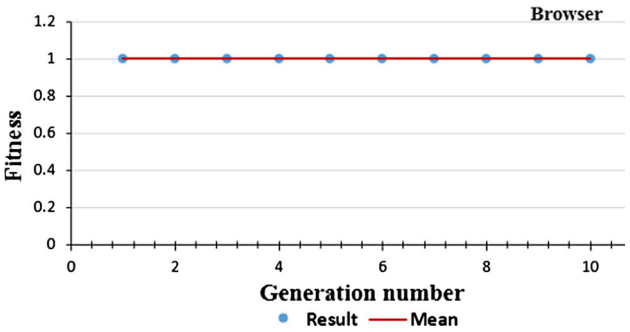


Fig. 32 EoD on MOF

convergence between features showed that in the absence of structural features, non-structural features can also be used.

Figures 33, 34 and 35 compare the number of modules generated from the proposed algorithms and created by the expert, considering structural and identifier names features. As these figures indicate, the number of modules generated by EoD and CGH algorithms is close to the number of modules created by the expert. As the figures show, in these two algorithms, the number of modules generated from the identifier names features is close to the number of modules generated from the structural features. This means that in the absence of tools for extracting structural features, identifier names features can be used.

Figures 36, 37 and 38 depict the best MoJo obtained by the proposed algorithms for the three folders of Mozilla Firefox. The blue color gives mojo for the structural features and orange color for the code features. In EoD and CGH algorithms, in most cases, the results are better, as can be seen in these figures. Given that MoJo has an inverse relation with quality, the smaller MoJo denotes the more quality. The comparison between the features demonstrates that structure feature, identifier names features, and then textual features, respectively, have the best MoJo (smaller MoJo are perfected). Moreover, it can be concluded that, in most cases, the modularization quality obtained using structural features is greater than those obtained using non-structural features.

Table 15 compares the EoD algorithm with a number of state-of-the-art modularization algorithms in terms of

Table 11 T-test code feature for the Build, Browser, and Db folders

Method	Mean		SD		SE mean		Equal variances assumed		Equal variances not assumed	
	*N*1	*N*2	*N*1	*N*2	*N*1	*N*2	Sig.	Sig.2tailed	Sig.	Sig.2tailed
Genetic with value-based encoding.Build	0.1538	0.1556	0.01916	0.02896	0.00857	0.01295	0.403	0.911	–	0.911
Genetic with permutation-based encoding.Build	0.5000	0.4667	0.00000	0.07454	0.00000	0.03333	0.029	0.347	–	0.374
EOD.Build	0.4334	0.5000	0.14892	0.00000	0.06660	0.00000	0.029	0.347	–	0.374
CGH.Build	0.1822	0.1802	0.08561	0.01807	0.03829	0.00808	0.097	0.960	–	0.961
CGoH.Build	0.1358	0.1394	0.00986	0.00805	0.00441	0.00360	0.252	0.545	–	0.545
Genetic with value-based encoding.Browser	0.9401	0.9618	0.07196	0.04191	0.03218	0.01874	0.385	0.576	–	0.580
Genetic with permutation-based encoding.Browser	0.9998	0.9999	0.00027	0.00003	0.00012	0.00002	0.041	0.196	–	0.229
EOD.Browser	0.9938	0.9958	0.00576	0.00342	0.00258	0.00153	0.180	0.523	–	0.527
CGH.Browser	0.9880	0.9920	0.02683	0.01789	0.01200	0.00800	0.481	0.789	–	0.790
CGoH.Browser	0.9998	0.9990	0.00045	0.00224	0.00020	0.00100	0.070	0.455	–	0.474
Genetic with value-based encoding.Db	0.9362	0.9409	0.04793	0.04106	0.02144	0.01836	0.754	0.874	–	0.874
Genetic with permutation-based encoding.Db	1.0000	1.0000	0.00002	0.00001	0.00001	0.00001	0.149	0.872	–	0.873
EOD.Db	0.4998	0.4998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000
CGH.Db	0.9998	0.9998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000
CGoH.Db	0.9998	0.9998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000

Table 12 T-test textual feature for the Build, Browser, and Db folders

Method	Mean		SD		SE mean		Equal variances assumed		Equal variances not assumed	
	*N*1	*N*2	*N*1	*N*2	*N*1	*N*2	Sig.	Sig.2tailed	Sig.	Sig.2tailed
Genetic with value-based encoding.Build	0.5948	0.5584	0.07487	0.07782	0.03348	0.03480	0.863	0.473	–	0.473
Genetic with permutation-based encoding.Build	0.9801	0.9749	0.02094	0.02023	0.00936	0.00905	0.915	0.705	–	0.705
EOD.Build	0.9995	0.9997	0.00058	0.00082	0.00029	0.00033	0.846	0.735	–	0.715
CGH.Build	0.9956	0.9942	0.00089	0.00402	0.00040	0.00180	0.077	0.469	–	0.486
CGoH.Build	0.9946	0.9956	0.00313	0.00089	0.00140	0.00040	0.104	0.512	–	0.525
Genetic with value-based encoding.Browser	0.9998	1.000	0.00045	0.00000	0.00020	0.00000	0.029	0.347	–	0.347
Genetic with permutation-based encoding.Browser	1.0000	1.0000	0.00005	0.00002	0.00002	0.00001	0.038	0.256	–	0.278
EOD.Browser	0.5326	0.5514	0.04020	0.02228	0.01798	0.00996	0.191	0.387	–	0.394
CGH.Browser	0.9998	0.9998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000
CGoH.Browser	1.0000	0.9998	0.00000	0.00045	0.00000	0.00020	0.029	0.347	–	0.347
Genetic with value-based encoding.Db	0.7871	0.8052	0.03832	0.02103	0.01714	0.00941	0.378	0.382	–	0.389
Genetic with permutation-based encoding.Db	0.9998	0.9998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000
EOD.Db	0.9882	0.9800	0.01178	0.02764	0.00527	0.01236	0.242	0.559	–	0.566
CGH.Db	1.0000	0.9998	0.00000	0.00045	0.00000	0.00020	0.029	0.347	–	0.374
CGoH.Db	1.0000	0.9998	0.00000	0.00045	0.00000	0.00020	0.029	0.347	–	0.374

Table 13 T-test combining structural and non-structural features for the Build, Browser, and Db folders

Method	Mean		SD		SE mean		Equal variances assumed		Equal variances not assumed	
	*N*1	*N*2	*N*1	*N*2	*N*1	*N*2	Sig.	Sig.2tailed	Sig.	Sig.2tailed
Genetic with value-based encoding.Build	0.2056	0.2096	0.01184	0.02715	0.00530	0.01214	0.239	0.770	–	0.774
Genetic with permutation-based encoding. Build	0.5119	0.5394	0.07657	0.08009	0.03424	0.03582	0.832	0.594	–	0.594
EOD.Build	0.5780	0.5796	0.00447	0.00089	0.00200	0.00040	0.070	0.455	–	0.474
CGH.Build	0.4892	0.4848	0.05012	0.05747	0.02241	0.02570	0.032	0.901	–	0.901
CGoH.Build	0.3826	0.3804	0.00089	0.00581	0.00040	0.00260	0.056	0.427	–	0.448
Genetic with value-based encoding. Browser	0.5680	0.5668	0.08579	0.05231	0.03837	0.02339	0.260	0.980	–	0.980
Genetic with permutation-based encoding. Browser	1.0000	0.9998	0.00000	0.00045	0.00000	0.00020	0.029	0.347	–	0.374
EOD.Browser	0.9996	0.9998	0.00089	0.00050	0.00040	0.00025	0.397	0.775	–	0.761
CGH.Browser	0.9998	1.0000	0.00045	0.00000	0.00020	0.00000	0.029	0.347	–	0.374
CGoH.Browser	1.0000	0.9996	0.00000	0.00451	0.00000	0.00201	0.051	0.130	–	0.167
Genetic with value-based encoding.Db	0.4229	0.4153	0.02961	0.01271	0.01324	0.00568	0.334	0.610	–	0.617
Genetic with permutation-based encoding. Db	1.0000	0.9998	0.00000	0.00045	0.00000	0.00020	0.029	0.347	–	0.374
EOD.Db	0.9932	0.9984	0.01057	0.00114	0.00473	0.00051	0.061	0.306	–	0.334
CGH.Db	0.9998	0.9998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000
CGoH.Db	0.9998	0.9998	0.00045	0.00045	0.00020	0.00020	1.000	1.000	–	1.000

Table 14 T-test Structure feature for the Build, Browser, and Db folders

Method	Mean		SD		SE mean		Equal variances assumed		Equal variances not assumed	
	*N*1	*N*2	*N*1	*N*2	*N*1	*N*2	Sig.	Sig.2tailed	Sig.	Sig.2tailed
Genetic with value-based encoding.Build	0.1670	0.1688	0.03361	0.02030	0.01503	0.00908	0.191	0.921	–	0.921
Genetic with permutation-based encoding. Build	0.5000	0.4998	0.00000	0.00045	0.00000	0.00020	0.029	0.347	–	0.374
EOD.Build	0.4940	0.4960	0.1342	0.00894	0.00600	0.00400	0.481	0.789	–	0.790
CGH.Build	0.4982	0.5000	0.00402	0.00000	0.00180	0.00000	0.029	0.347	–	0.374
CGoH.Build	0.4980	0.4990	0.00447	0.00224	0.00200	0.00100	0.267	0.667	–	0.671
Genetic with value-based encoding. Browser	0.5940	0.5927	0.08041	0.04133	0.03596	0.01848	0.434	0.975	–	0.976
Genetic with permutation-based encoding. Browser	0.9033	0.8502	0.06424	0.9836	0.02873	0.04399	0.650	0.341	–	0.346
EOD.Browser	0.9980	0.9990	0.00447	0.00224	0.00200	0.00100	0.267	0.667	–	0.671
CGH.Browser	0.9522	0.9550	0.00743	0.00000	0.00332	0.00000	0.032	0.424	–	0.447
CGoH.Browser	0.8650	0.8662	0.01789	0.01521	0.00800	0.00680	0.768	0.912	–	0.912
Genetic with value-based encoding.Db	0.1385	0.1412	0.00778	0.01039	0.00348	0.00465	0.625	0.654	–	0.655
Genetic with permutation-based encoding. Db	0.2967	0.3393	0.07113	0.01995	0.03181	0.00892	0.099	0.233	–	0.257
EOD.Db	0.9980	0.9986	0.00447	0.00313	0.00200	0.00140	0.531	0.812	–	0.813
CGH.Db	0.7614	0.7590	0.00537	0.00000	0.00240	0.00000	0.029	0.347	–	0.374
CGoH.Db	0.5690	0.5674	0.00000	0.00358	0.00000	0.00160	0.029	0.347	–	0.374

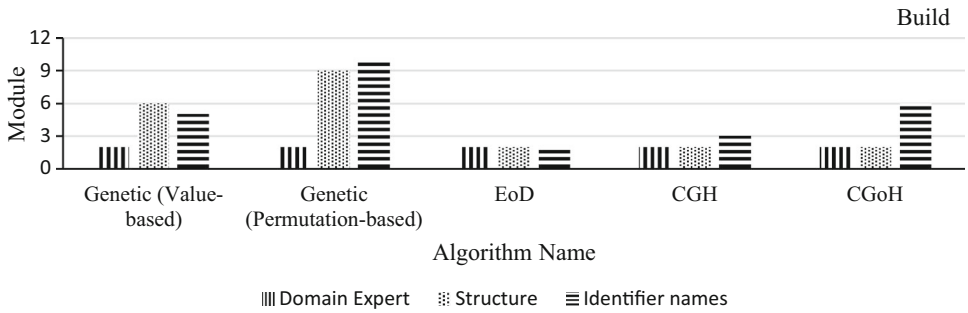


Fig. 33 The number of modules obtained in the proposed algorithms for the Build

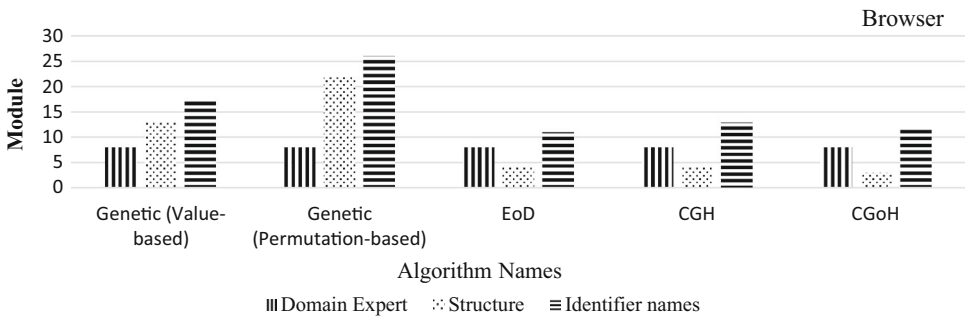


Fig. 34 The number of modules obtained in the proposed algorithms for the Browser

MoJoFM. The best results in Table 15 are highlighted in bold. As can be seen in this table, in folders of Db and Build, the EoD outperforms other algorithms. In contrast, in the Browser folder, the MCA and Neighborhood tree algorithms outperform other algorithms. The reasons why the proposed algorithm, in this folder, failed to perform better than these two algorithms are: (1) by checking this folder, we found that call dependency, as a structural

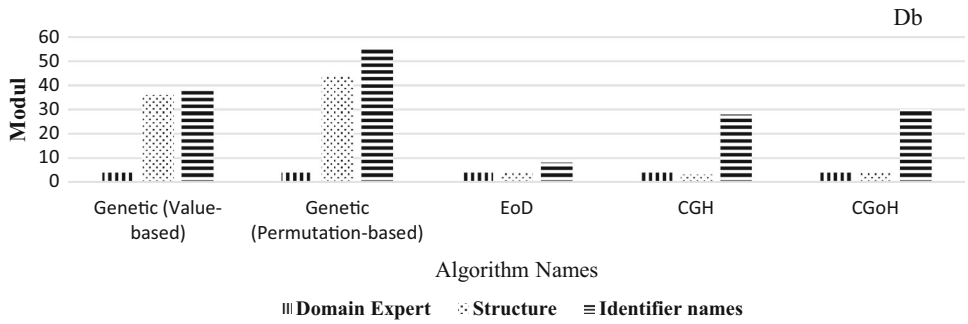


Fig. 35 The number of modules obtained in the proposed algorithms for the Db

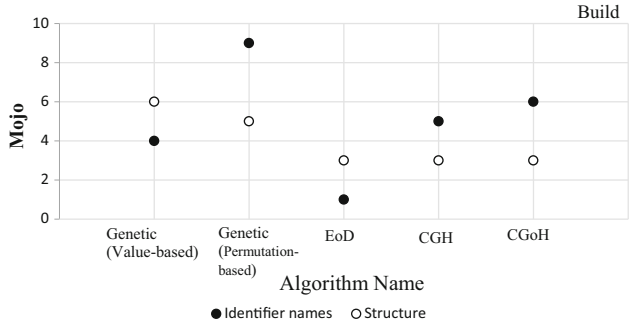


Fig. 36 MoJo in the proposed algorithms for the Build

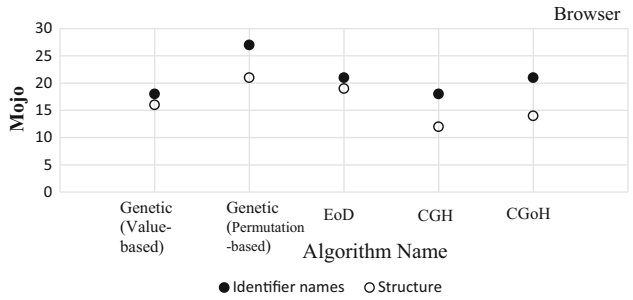


Fig. 37 MoJo in the proposed algorithms for the Browser

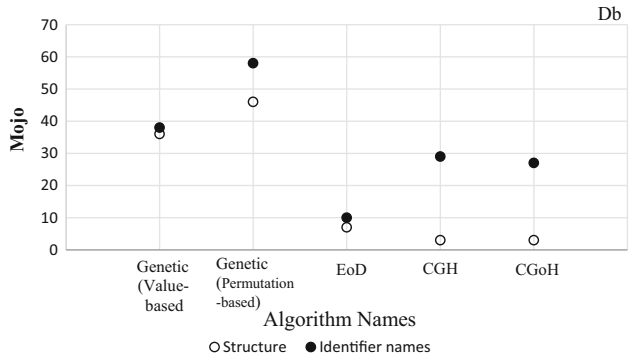


Fig. 38 MoJo in the proposed algorithms for the Db

features, in this folder is high and effects on other features; and (2) also the artifacts names are not named meaningfully by developers.

The results from this study indicate:

Table 15 Comparison of EoD with a number of state-of-the-art algorithms in terms of MoJoFM

Algorithm	Db	Browser	Build
HC	94.68	50	84.21
Bunch	94.68	70	84.21
DAGC	50	60	78.95
ECA	96.81	60	78.95
MCA	96.81	**72.50**	78.95
E-CDGM	94.68	50.19	84.21
EDA	94.98	51.78	61.09
HC-SMCP	93.21	68.80	60
Neighborhood tree algorithm	90.43	**72.50**	21.05
EoD	**96.91**	52.5	**94.74**

(1) In software systems where the comments and identifier names are chosen meaningful by developers, the modularization achieved by the proposed multi-objective fitness function will have a higher quality.

(2) Due to the maintenance of building blocks during the evolutionary process, in most cases estimation of distribution algorithms achieve higher-quality solutions, in contrast to the genetic algorithm and hill climbing and their combination.

(3) By studying the structural and non-structural features separately, we realized that the modularization obtained by an algorithm which considers structural features has a higher quality than those which consider non-structural features. Of course, this difference is not high. Therefore, non-structural features can be used if it is not possible to extract structural features from a program.

5 Conclusions

Software architecture recovery plays an increasingly essential role in software maintenance tasks. Source code comprehension (source code understanding) is a vital

software maintenance activity concerned with the ways software engineers maintain existing source code. To recover software architecture, various modularization techniques have been employed to automatically partition a software system into meaningful subsystems, to support source code comprehension. This paper presented a new multi-objective fitness function aiming to modularize a software system from its source code. The presented objective function enables evolutionary algorithms to modularize a source code taking into account structural non-structural features. The structural features used in this paper are call dependency graph and inheritance relationships, and non-structural features used are textual, identifier names, and comment. We presented the equations to calculate these features from a source code. Also, to simplify the interpretation of information, the dimensionality reduction is applied to non-structural features. To modularize a software system using these features, five algorithms namely genetic algorithms with two encodings, distribution estimation algorithm, a combination of the genetic and hill climbing algorithms in two ways, are adapted to optimize the presented multi-objective fitness function. The results of the proposed algorithms are evaluated with different criteria such as Mojo and MoJoFM. The results of the evaluations indicated that the modularization produced by distribution estimation algorithm is close to the expert modularization. Also, we evaluated the results in terms of convergence and stability. Using these evaluations and the slight differences between the features, it is observed that in the absence of structural features, non-structural features could be used instead. It is concluded that identifier names features could be a better alternative in the absence of structural features.

5.1 Future work

The following suggestions are made for future works:

- Presenting semantic and nominal dependency graphs, to support multi-programming software systems modularization;
- Creating a thesaurus for programming languages and using it to calculate the similarity between comments and identifier names;
- Using the information theory in the objective function
- Using feature selection methods such as algorithms presented in (Abualigah and Khader 2017; Abualigah et al. 2016) for extracting features from the source code.
- In the estimation of distribution algorithms, the quality of the solutions depends on the probabilistic model that has been made. We believe that the Gaussian probability distribution function can improve the accuracy of the constructed probability model.

Funding There is no funding for this study.

Compliance with ethical standards

Conflict of interest All authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants performed by any of the authors.

References

Abualigah LM, Khader AT (2017) Unsupervised text feature selection technique based on hybrid particle swarm optimization algorithm with genetic operators for the text clustering. J Supercomput 73(11):4773–4795

Abualigah LM, Khader AT, Al-Betar MA (2016, July) Unsupervised feature selection technique based on genetic algorithm for improving the text clustering. In: Computer science and information technology (CSIT), 2016 7th international conference on. IEEE, pp 1–6

Abualigah LM, Khader AT, Hanandeh ES, Gandomi AH (2017a) A novel hybridization strategy for krill herd algorithm applied to clustering techniques. Appl Soft Comput 60:423–435

Abualigah LM, Khader AT, Al-Betar MA, Alomari OA (2017b) Text feature selection with a robust weight scheme and dynamic dimension reduction to text document clustering. Expert Syst Appl 84:24–36

Abualigah LM, Khader AT, Hanandeh ES (2018) Hybrid clustering analysis using improved krill herd algorithm. Appl Intell 48(11):4047–4071. https://doi.org/10.1007/s10489-018-1190-6

Alswaitti M, Albughdadi M, Isa NAM (2018) Density-based particle swarm optimization algorithm for data clustering. Expert Syst Appl 91:170–186

Andritsos P, Tzerpos V (2005) Information-theoretic software clustering. IEEE Trans Softw Eng 31(2):150–165

Bavota G, Carnevale F, De Lucia A, Di Penta M, Oliveto R (2012, September) Putting the developer in-the-loop: an interactive GA for software re-modularization. In: International symposium on search based software engineering. Springer, Berlin, pp 75–89

Candela I, Bavota G, Russo B, Oliveto R (2016) Using cohesion and coupling for software remodularization: is it enough? ACM Trans Softw Eng Methodol (TOSEM) 25(3):24

Chang L, Li W, Qin L, Zhang W, Yang S (2017) pSCAN: fast and exact structural graph clustering. IEEE Trans Knowl Data Eng 29(2):387–401

Chhabra JK (2015) Search-based object-oriented software re-structuring with structural coupling strength. Procedia Comput Sci 54:380–389

Chhabra JK (2017) Harmony search based remodularization for object-oriented software systems. Comput Lang Syst Struct 47:153–169

Cincotti A, Cuttelo V, Pavone M (2002, May) Graph partitioning using genetic algorithms with ODPX. In: Proceedings of the world congress on computational intelligence

Corazza A, Di Martino S, Scanniello G (2010, March) A probabilistic based approach towards software system clustering. In: Software

maintenance and reengineering (CSMR), 2010 14th European conference on. IEEE, pp 88–96
Dumais ST (2004) Latent semantic analysis. Ann Rev Inf Sci Technol 38(1):188–230
Garcia J, Krka I, Mattmann C, Medvidovic N (2013, May) Obtaining ground-truth software architectures. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 901–910
Huang J, Liu J (2016) A similarity-based modularization quality measure for software module clustering problems. Inf Sci 342:96–110
Huang J, Liu J, Yao X (2017) A multi-agent evolutionary algorithm for software module clustering problems. Soft Comput 21(12):3415–3428
Isazadeh A, Izadkhah H, Elgedawy I (2017) Source code modularization: theory and techniques. Springer. ISBN 978-3-319-63346-6
Izadkhah H, Elgedawy I, Isazadeh A (2016) E-CDGM: an evolutionary call dependency graph modularization approach for software systems. Cybern Inf Technol 16(3):70–90
Jeet K, Dhir R (2016a) Software module clustering using hybrid socio-evolutionary algorithms. Int J Inf Eng Electron Bus 8(4):43
Jeet K, Dhir R (2016) Software clustering using hybrid multi-objective black hole algorithm. In: SEKE, pp 650–653
Kumari AC, Srinivas K (2016) Hyper-heuristic approach for multi-objective software module clustering. J Syst Softw 117:384–401
Liu S, Zhou B, Huang D, Shen L (2017) Clustering mixed data by fast search and find of density peaks. Math Probl Eng 2017:1–7
Lutellier T, Chollak D, Garcia J, Tan L, Rayside D, Medvidovic N, Kroeger R (2017) Measuring the impact of code dependencies on software architecture recovery techniques. IEEE Trans Softw Eng 44:159–181
Mahdavi K (2005) A clustering genetic algorithm for software modularisation with a multiple hill climbing approach. Doctoral Dissertation, Brunel University
Mahdavi K, Harman M, Hierons R (2003, July) Finding building blocks for software clustering. In: Genetic and evolutionary computation conference. Springer, Berlin, pp 2513–2514
Mamaghani AS, Hajizadeh M (2014, September). Software modularization using the modified firefly algorithm. In: Software engineering conference (MySEC), 2014 8th Malaysian. IEEE, pp 321–324
Mamaghani AS, Meybodi MR (2009, October) Clustering of software systems using new hybrid algorithms. In: Computer and information technology, 2009. CIT'09. Ninth IEEE international conference on, vol 1. IEEE, pp 20–25
Maqbool O, Babri HA (2004, March) The weighted combined algorithm: A linkage algorithm for software clustering. In: Software maintenance and reengineering, 2004. CSMR 2004. Proceedings. Eighth European conference on. IEEE, pp 15–24
Maqbool O, Babri H (2007) Hierarchical clustering for software architecture recovery. IEEE Trans Softw Eng 33(11):759–780
Misra J (2012) Java source-code clustering: unifying syntactic and semantic features. arXiv preprint arXiv:1208.6408
Misra J, Annervaz KM, Kaulgud V, Sengupta S, Titus G (2012, October) Software clustering: unifying syntactic and semantic features. In: Reverse engineering (WCRE), 2012 19th working conference on. IEEE, pp 113–122
Mitchell BS, Mancoridis S (2002) A heuristic search approach to solving the software clustering problem. Drexel University, Philadelphia
Mitchell BS, Mancoridis S (2006) On the automatic modularization of software systems using the bunch tool. IEEE Trans Softw Eng 32(3):193–208
Mitchell BS, Mancoridis S (2008) On the evaluation of the Bunch search-based software modularization algorithm. Soft Comput 12(1):77–93
Mkaouer W, Kessentini M, Shaout A, Koligheu P, Bechikh S, Deb K, Ouni A (2015) Many-objective software remodularization using NSGA-III. ACM Trans Softw Eng Methodol (TOSEM) 24(3):17
Mohammadi S, Izadkhah H (2018) A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code. Inf Softw Technol 105:252–256. https://doi.org/10.1016/j.infsof.2018.09.001
Monçores MC, Alvim AC, Barros MO (2018) Large neighborhood search applied to the software module clustering problem. Comput Oper Res 91:92–111
Naseem R, Maqbool O, Muhammad S (2013) Cooperative clustering for software modularization. J Syst Softw 86(8):2045–2062
Naseem R, Deris MBM, Maqbool O (2014, December) Software modularization using combination of multiple clustering. In: Multi-topic conference (INMIC), 2014 IEEE 17th international. IEEE, pp 277–281
Parsa S, Bushehrian O (2005) A new encoding scheme and a framework to investigate genetic clustering algorithms. J Res Pract Inf Technol 37(1):127
Pfleeger SL (2001) Software engineering: theory and practice, 2nd edn. Prentice Hall, Upper Saddle River
Pigoski TM (1996) Practical software maintenance: best practices for managing your software investment. Wiley, Hoboken
Praditwong K, Harman M, Yao X (2011) Software module clustering as a multi-objective search problem. IEEE Trans Softw Eng 37(2):264–282
Qiu D, Zhang Q, Fang S (2015) Reconstructing software high-level architecture by clustering weighted directed class graph. Int J Softw Eng Knowl Eng 25(04):701–726
Rathee A, Chhabra JK (2017) Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering. In: Advanced informatics for computing research. Springer, Singapore, pp 94–106
Saeed M, Maqbool O, Babri HA, Hassan SZ, Sarwar SM (2003, March) Software clustering techniques and the use of combined algorithm. In: Software maintenance and reengineering, 200 Proceedings. Seventh European conference on. IEEE, pp 301–306
Shokoufandeh A, Mancoridis S, Maycock M (2002) Applying spectral methods to software clustering. In Reverse engineering, 2002. Proceedings. Ninth working conference on. IEEE, pp 3–10
Shokoufandeh A, Mancoridis S, Denton T, Maycock M (2005) Spectral and meta-heuristic algorithms for software clustering. J Syst Softw 77(3):213–223
Srinivas C, Radhakrishna V, Rao CG (2013a) Clustering software components for program restructuring and component reuse using hybrid XOR similarity function. AASRI Procedia 4:319–328
Srinivas C, Radhakrishna V, Rao CV (2013, December) Clustering software components for component reuse and program restructuring. In Proceedings of the second international conference on innovative computing and cloud computing. ACM, p 261
Tajgardan M, Izadkhah H, Lotfi S (2016) Software systems clustering using estimation of distribution approach. J Appl Comput Sci Methods 8(2):99–113
Tzerpos V, Holt RC (1999, October) MoJo: A distance metric for software clusterings. In: Reverse engineering, 1999. Proceedings. Sixth working conference on. IEEE, pp 187–193
Wen Z, Tzerpos V (2004, June) An effectiveness measure for software clustering algorithms. In: Program comprehension, 2004. Proceedings. 12th IEEE international workshop on. IEEE, pp 194–203
Wen D, Qin L, Zhang Y, Chang L, Lin X (2017) Efficient structural graph clustering: an index-based approach. Proc VLDB Endow 11(3):243–255