



Article

Program Source-Code Re-Modularization Using a Discretized and Modified Sand Cat Swarm Optimization Algorithm

Bahman Arasteh ¹, Amir Seyyedabbasi ^{1,*} , Jawad Rasheed ²  and Adnan M. Abu-Mahfouz ^{3,4} 

¹ Department of Software Engineering, Faculty of Engineering and Natural Science, Istinye University, Istanbul 34396, Turkey

² Department of Software Engineering, Nisantasi University, Istanbul 34398, Turkey

³ Council for Scientific and Industrial Research (CSIR), Pretoria 0184, South Africa

⁴ Department of Electrical and Electronic Engineering Science, University of Johannesburg, Johannesburg 2006, South Africa

* Correspondence: amir.seyyedabbasi@istinye.edu.tr

Abstract: One of expensive stages of the software lifecycle is its maintenance. Software maintenance will be much simpler if its structural models are available. Software module clustering is thought to be a practical reverse engineering method for building software structural models from source code. The most crucial goals in software module clustering are to minimize connections between created clusters, maximize internal connections within clusters, and maximize clustering quality. It is thought that finding the best software clustering model is an NP-complete task. The key shortcomings of the earlier techniques are their low success rates, low stability, and insufficient modularization quality. In this paper, for effective clustering of software source code, a discretized sand cat swarm optimization (SCSO) algorithm has been proposed. The proposed method takes the dependency graph of the source code and generates the best clusters for it. Ten standard and real-world benchmarks were used to assess the performance of the suggested approach. The outcomes show that the quality of clustering is improved when a discretized SCSO algorithm was used to address the software module clustering issue. The suggested method beats the previous heuristic approaches in terms of modularization quality, convergence speed, and success rate.

Keywords: software module clustering; cohesion; coupling; modularization quality; sand cat swarm optimization algorithm



Citation: Arasteh, B.; Seyyedabbasi, A.; Rasheed, J.; M. Abu-Mahfouz, A. Program Source-Code Re-Modularization Using a Discretized and Modified Sand Cat Swarm Optimization Algorithm. *Symmetry* **2023**, *15*, 401. <https://doi.org/10.3390/sym15020401>

Academic Editor: Zhixun Su

Received: 22 December 2022

Revised: 17 January 2023

Accepted: 20 January 2023

Published: 2 February 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

It is unavoidable that software evolves to meet user requirements, but any modifications must be performed with an eye toward maintaining both functional and non-functional qualities. On average, 60% of software costs go toward maintenance [1–4]. Program comprehension has a significant impact on software maintenance costs when the source code is the only product offered. Therefore, when faced with huge and complicated software source code, and also when design documentation and models are not available, extracting and comprehending the software structural models is required. Using the reverse engineering technique of clustering source code modules, one may understand the architecture and behavior of software before making changes. After extracting the software modules (components, classes, and methods) from the source code, this approach groups the modules with related characteristics. In this manner, the modularization quality (MQ) requirements are computed based on the number of connections inside (cohesion) and between the clusters (coupling) [1,2]. In fact, MQ criteria are used to analyze a clustering of software modules. According to this criterion, better clustering is the result of more cohesion (the links between modules into a cluster) and less coupling (the links between different clusters).

In the software module clustering (SMC) problem, the source-code product is partitioned into m parts (clusters). Let's assume that S (a source code) consists of n modules, $M1, M2, \dots, Mn$; each module has programming structures such as functions, methods, and properties. The possible combinations for clustering the n modules of a program into m clusters are represented by set π . Each member of π indicates a combination of all feasible clustering combinations. The number of feasible combinations for clustering a program with n modules into m clusters is shown by variable $S_{n,m}$ (Stirling numbers) that grow exponentially with respect to the number of modules. For example, a program with 5 modules has 52 distinct clustering combinations, and a program with 25 modules (15-node module dependency graph) has 1,382,958,545 clustering combinations. In a clustering combination, the intersection of m clusters is empty, and the union of m clusters equals all source code (S). Hence, the SMC problem is formally an NP-complete problem [2,4–8]. This encourages researchers to use heuristic methods to identify the optimal clustering.

Currently, various artificial intelligence and machine-learning algorithms are being used to solve different computer engineering problems [9–16]. A number of search-based heuristic techniques have been used to tackle SMC as a discrete search-based optimization problem [1–3,11,17–24]. Lower MQ, poor stability, and lower convergence speed, specifically in large software products, are the main limitations of the previous methods. Other significant shortcomings of the earlier techniques include a local optimum and lower success rate. The following are the study's main objectives:

1. Increasing the modularization quality (MQ);
2. Increasing the success rate of generating the optimal clusters;
3. Increasing the stability of the SMC method during several executions;
4. Increasing the convergence speed to attain the optimal clusters.

The first contribution of this study is the development of an SMC method based on the discretized and modified sand cat swarm optimization (SCSO) algorithm; since SMC is a discrete optimization problem, the SCSO algorithm was modified and discretized [16]. The developed, discretized SCSO can be used for other discrete optimization problems that are similar to SMC. The proposed discrete SCSO algorithms have higher performance than the genetic algorithm (GA), particle swarm optimization algorithm (PSO), and PSO-GA in regard to the SMC problem. Another major contribution is the creation of a software tool to automatically generate a clustered design model for a complex software system. This paper is divided into the following five sections: An overview of significant SMC research is given in Section 2; the proposed approach is shown and explained in Section 3; the first half of Section 4 describes the setting and instrument that will be used to carry out the suggested strategy, and presents evaluation standards and data sets; the study's results are given and discussed in the second half of Section 4; and the study's findings and recommendations for further research are presented in Section 5.

2. Related Works

A hill climbing (HC) method is suggested in [19] for creating clustered structural models of software source code. Due to the fact that an MDG has N modules, it may produce up to N initial clusters. Each software module is given a random cluster assignment before the first hill climber starts. The MQ of the clusters that have formed is evaluated using a fitness function. The goal of this approach is to produce clusters with the lowest coupling and maximum cohesion possible. Each hill climber tries to reach the nearest neighboring cluster with a higher MQ at each stage of the procedure. The hill climber looks for another neighbor as soon as it spots one in the new cluster (climber with a higher MQ). The search process in this way terminates when none of the clustering's closest neighbors can find a superior MQ value. The initial set of hills from the first stage eventually combines to form a group of hills. This approach has the potential to fall in the local optimum, which makes it more challenging to locate the global optimum (the optimal clustering).

The genetic algorithm (GA) is an evolutionary-based heuristic search method. The main drawbacks of the HC method for the SMC problem have been solved by the GA. There

is no probability of finding the ideal clustering while using direct search strategies such as the HC algorithm. Large search spaces cannot be handled by such algorithms. Contrarily, the search procedure in the GA proceeds simultaneously with the chromosomes of the initial population. In the GA, each chromosome corresponds to a clustering combination. The initial population is made up of random chromosomes (random clustering combinations). In each iteration, the chromosomes are developed by the fitness function; the fitness function is used to try to create chromosomes (clustering combinations) with a high degree of cohesion and a low degree of coupling. The search is then expanded by updating the selected chromosomes using the crossover and mutation operators [25–27]. According to the results, the GA is a suitable solution for the SMC problem for small- and medium-scale software clusters.

The firefly algorithm (FA) was proposed by XinShe-Yang in 2008 [28]. It uses swarm intelligence as its foundation. Each firefly in this technique represents a clustering combination with a certain light intensity (clustering quality). A fitness function may be used to quantify this level of light intensity (MQ). In order to attain an ideal clustering combination and even the best time, fireflies try to go toward better fireflies (brighter ones) and alter their placement. A non-discrete optimization technique called the firefly algorithm is employed to resolve problems; it may be employed, nonetheless, for specific issues such as the SMC problem. Experimental findings reveal that firefly beats HC and GA on the majority of benchmarks. Local optimum probability and low convergence speed are the method's primary drawbacks, particularly in large software systems.

In [4], a PSO-based method for choosing the best clusters for a software product's modules was proposed. Each particle represents how software components (modules) are grouped together. The two primary characteristics of each particle are its position vector and its speed vector. In contrast, the particles' present locations are moved in a specific direction using the speed vector. Particularly, the positions of the particles vary with increasing speed. The initial population is a collection of particles. Based on the difficulty of the SMC problem, the particle placement and speed of the PSO approach were developed. PSOMC is a modified version of the PSO algorithm that enhances particle position using a local search technique. PSOMC's effectiveness has been demonstrated in a number of SMC benchmarks. The testing results show that the suggested method generates higher-quality software clusters.

Arasteh and colleagues introduced a hybrid PSO–GA approach (Araz) [29] to identify the best clusters for a software application. This approach aims to solve the flaws in earlier software clustering approaches (inadequate MQ, low convergence, inadequate success rate, and low stability). This method takes advantage of the strengths of both heuristic methods. This hybrid approach, which incorporates both PSO and GA, improves the quality of the clustering and delivers quick data convergence when compared to PSO and GA algorithms. Crossover and mutation were utilized to update and optimize the position of the particles throughout the stage of updating the speed vector of all the particles. The convergent PSO–GA approach is superior to the standard method, according to experiments on 10 common benchmark module dependency graphs (MDGs); experiments show that, 90% of the time, the convergent PSO–GA method is superior to the GA and PSO strategies. The results demonstrate that the PSO–GA method outperforms the PSO and GA algorithms in 50% of the benchmark programs. In addition, all three algorithms had the same success rate in 30% of the benchmark applications. The results suggest that the PSO–GA approach is more stable than the PSO and GA algorithms in more than 60% of the benchmark programs. A free download of the correspondingly implemented code is available.

The ant colony optimization (ACO) method was applied in [30] to cluster software components in the best possible way. Each cluster's (subsystem) modules are intricately linked to one another. The ACO algorithm uses swarm intelligence to address a number of search-based optimization problems. In the suggested method, each ant is a clustering combination. A high-quality clustering has the maximum degree of coherence and the least coupling. This strategy was tested using a number of benchmark datasets. The

outcomes show convergence and a high rate of finding the best clusters in the benchmarks. Additionally, the ACO-based approach typically outperforms GA and PSO in terms of convergence speed and MQ value. In terms of convergence speed, GA was found to perform better than PSO. In terms of stability, all three approaches could be able to address the issue of SMC

In [23], an SMC method (Bölen) was proposed to cluster software modules using a combination of the shuffle frog-leaping algorithm (SFLA) and GA approaches. Quicker data convergence, higher modularization quality (MQ), enhanced stability, and a higher success rate are the main merits of this method. Similar to other methods, MDG is used to show how various software components (modules) are connected. SFLA includes local and global searches. In the SMC problem, each frog in the SFLA is seen as a clustering array (clustering combination). In this method, the best frogs in each memplex are copied from the poorest frogs using the crossover operator. Additionally, the mutation operator is applied to each memplex member with the option of optimizing the weakest memplex member. In 80% of the benchmark cases, the SFLA–GA method outperformed the other methods in terms of average MQ. According to the convergence criteria, the SFLA–GA technique converges to the optimal answer more quickly than the HC, GA, and PSO algorithms in 90% of the programs.

In [14], a hybrid single-objective method has been suggested using the combination of the gray wolf algorithm (GWOA) and GA. The proposed method combines a swarm-based algorithm with an evolutionary algorithm to sort out the SMC problem. The classic GWOA was discretized and adapted for the SMC problem. Experiments on 14 standard benchmarks confirmed the higher performance of this single-objective hybrid method over the GA, PSO, and PSO–GA in regard to the SMC problem; a higher MQ and higher convergence speed are the main merits of this method, specifically in large software products. In [31], different chaos-based heuristic algorithms, such as bat, cuckoo, teaching–learning-based, black widow, and grasshopper, have been suggested for the SMC problem. Additionally, experimental research has been performed to determine the implications of chaos theory on how well various algorithms perform in this scenario. The performance of the BWO, PSO, and TLB algorithms in regard to the SMC problem is higher than that of the other algorithms, according to the results of real-world applications. Additionally, the performance of these algorithms improved when their initial populations were generated using the logistic chaos method rather than the random method. The created clusters for the chosen benchmark set by BWO, PSO, and TLB have average MQ values of 3.155, 3.120, and 2.778, respectively.

An autonomous method (Savalan) for the SMC problem has been developed in [32] and is based on a multi-objective evolutionary algorithm and a novel set of objective functions. The primary goal of this research is to simultaneously enhance all clustering aims (cohesion, coupling, modularization quality, cluster size, and number). Six distinct objective criteria were considered as optimization goals in this study. In the suggested technique, the multi-objective genetic algorithm was driven by the PESA (Pareto envelope-based selection algorithm). It is possible to apply this strategy for both small and large applications. The 14 benchmark programs' results demonstrate that this method's main benefit is that it simultaneously advances every clustering aim. The results show that the Savalan approach simultaneously improves all clustering criteria. Savalan produces clusters of greater quality than comparable technologies such as Bunch [17], CIA [27], and chava [33], graphviz [34] and it performs significantly better (MQ) in large software systems. For scholars and developers, Savalan was made available as free software in [35]. As a result, this research has implications for the computer society both theoretically and practically. JavaScript programming language was used in the development of this utility. Table 1 lists the key traits of recent studies and innovative techniques in complex software systems. The most difficult regions remain those with sluggish convergence, local optimal stability, and low stability.

Table 1. Characteristics of prior studies and tools.

Type of Objective Function	Characteristics	Algorithm	Year	Researcher
Single objective	Software structure recovery method	GA, HC, SA	1999	Mancordis [17]
Single objective	Software re-modularization methods using heuristic search techniques	GA, HC, SA	2002	Mitchell [1]
Single objective	Usage of two fitness factors for module clustering	HC	2005	Harmen [18]
Single objective	Usage of evolutionary algorithm for module clustering	GGA	2011	Wang [36]
Multi-objective	Multi-objective search algorithm for clustering the modules of software	Two-archive	2011	Wang [2]
Single objective	Re-modularization of a software source code using an interactive GA	GA	2012	Bavota [20]
Multi-objective	Re-modularization of a software source code using a hyper-heuristic algorithm	Hyper-heuristic	2013	Kumari [37]
Multi-objective	Object-oriented software clustering by weighted class connections and multi-objective algorithm	NSGA-II	2015	Chhabra [38]
Single objective	Harmony-search based modularization of object-oriented software modules	HS	2016	Amerjit [39]
Single objective	Combination of SFLA and GA to modularize the software source code	SFLA-GA	2019	Arasteh [23]
Single objective	Combination of PSO and GA to modularize the software source code	PSO-GA	2020	Arasteh [29]
Single objective	Proposing ACO to generate software design model from the source code	ACO	2020	Arasteh [30]
Multi-objective	Using the two-archive ABC algorithm for software module clustering problem	TA-ABC	2021	Pourasghar [22]
Multi-objective	Proposing a multi-objective homogeneous clustering approach (Savalan) to software module clustering problem	PESA-GA	2022	Arasteh [32]
Hybrid single objective	Combination of discretized GWO algorithm and GA to cluster the modules of large software products	Hybrid Gray Wolf	2022	Arasteh [14]
Chaos-based single objective	Combination of chaos theory and heuristic algorithms to generate design models from the software source code	Chaos-based metaheuristic algorithms	2022	Arasteh [40]
Accessibility	Name of Tool and Characteristics	Name of Tool and Characteristics	Name of Tool and Characteristics	Year
Free desktop tool [17]	Bunch: Desktop standalone tool, single objective,	Bunch: Desktop standalone tool, single objective	Bunch: Desktop standalone tool, single objective	1999
Free web-based tool link [35]	Savalan: Web-based, multi-objective, and homogeneous tool, written in JavaScript	Savalan: Web-based, multi-objective, and homogeneous tool, written in JavaScript	Savalan: Web-based, multi-objective, and homogeneous tool, written in JavaScript	2022

3. Proposed Method

3.1. Inspiration and Mathematical Model

As a result of the sand cat's ability to detect low-frequency noise, the sand cat swarm optimization (SCSO) algorithm was developed based on sand cat behavior [16]. In addition to their superior ability to locate prey on the ground or underground, sand cats can also quickly locate and catch prey. In order to emphasize the concept of swarm intelligence, the authors assumed that sand cats live in herds, since they live alone in nature. Therefore, the number of sand cats can be declared for the initialization of an algorithm to optimize a

minimization or maximization problem. The first step is to create the initial population and define the problem.

3.2. Initialization

In the SCSO algorithm, each sand cat indicates the solution and a one-dimensional array representing the solution of a d-dimensional optimization problem. Consequently, each variable value (x_1, x_2, \dots, x_d) has a floating-point value attached to it (Figure 1). As a first step in the SCSO algorithm, the algorithm creates a candidate matrix containing the number of sand cat populations necessary to solve the problem based on the problem’s size. A fitness function for each sand cat is also evaluated. A fitness function is a mathematical function that defines the necessary parameters (variables) for solving the problem and determines how they should be set for the solution. Thus, each sand cat will be assigned a value based on the fitness function as a result. Based on the problem goal of minimization or maximization, the best search agent (Sand Cat) in each iteration has the optimum cost value. In this way, the search agents try to update their positions based on the best search agent position in the upcoming iterations. Therefore, the best solution in each iteration can be used to represent the sand cat closest to the prey. As a result of improved solutions, the previous solution is not stored in memory unnecessarily, allowing memory to be used efficiently.

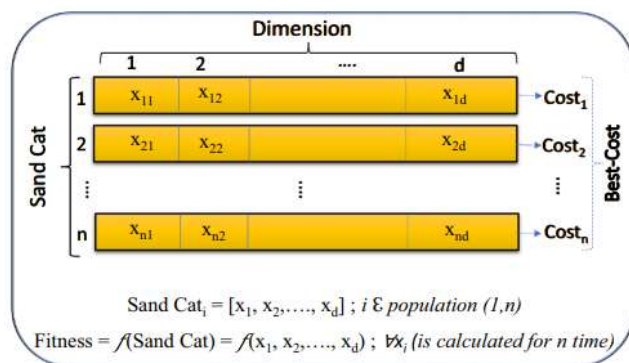


Figure 1. The initial and definition phases of SCSO.

3.3. Exploration and Exploitation

Utilizing the SCSO algorithm, sand cats are able to find prey by detecting low frequency noise emissions. The SCSO algorithm utilizes the sand cat’s hearing capacity to detect low frequencies. In this way, the sensitivity ranges of each cat can be compared. The sand cat senses low frequencies below 2 kHz. It is predicted that the value r_G^{\rightarrow} parameter will decrease linearly from two to zero as the iterations progress, as the algorithm seeks to approach the hunt it is seeking and not to lose or pass (not to move away from) it. Hence, the SM value is assumed to be 2, since sand cats have a hearing range between 2 kHz and 0 (Equation (1)). As a result, the sand cat is expected to have a sensitivity range from 2 kHz to 0 (Equation (1)). There are a variety of problems that can affect the speed at which search agents react. This is a good indication of the flexibility and versatility of the equation presented. If you have a maximum number of iterations of 100, the value will be greater than 1 in the first half of the iterations and less than 1 in the last half. In order to control the transition between exploration and exploitation phases, R is the final and most important parameter. Equation (2) provides an adaptive strategy that will reduce the imbalance between the transitions and opportunities.

Search agents’ positions are updated randomly during the search step. The search space is randomly initialized between defined boundaries, allowing the search agents to explore new areas. As a result of Equation (3), the general sensitivity range decreases linearly from 2 to 0, since each sand cat has a different sensitivity range, thus avoiding

the local optimum trap. Additionally, $iter_c$ represents the current iteration and $iter_{Max}$ represents the maximum iteration, also demonstrating the sensitivity range of each cat.

$$\vec{r}_G = s_M - \left(\frac{s_M - iter_c}{iter_{Max}} \right) \quad (1)$$

$$\vec{R} = 2 \times \vec{r}_G \times rand(0,1) - \vec{r}_G \quad (2)$$

$$\vec{r} = \vec{r}_G \times rand(0,1) \quad (3)$$

According to the best-candidate position of the search agent (\vec{Pos}_{bc}), as well as the current position (\vec{Pos}_c) of the search agent and sensitivity range (\vec{r}), each sand cat updates its own position. Therefore, the sand cats are able to determine another possible best-prey position (Equation (4)). As a result of this equation, the algorithm has another opportunity to discover new local optima within the search area, resulting in a position between the prey's current position and the cat's current position. A further benefit of this algorithm is that it is low in operation cost and moderate in complexity, because it is based on random rather than exact methods. By giving the search agents in the algorithm the benefit of randomness, the algorithm is high in efficiency and low in operation cost.

$$\vec{Pos}(t+1) = \vec{r} \cdot \left(\vec{Pos}_{bc}(t) - rand(0,1) \cdot \vec{Pos}_c(t) \right) \quad (4)$$

After finding prey (exploration), the SCSO moves on to exploitation of the prey that has been discovered after being searched for (exploitation) in the preceding stages of the process. Using Equation (5), we are able to calculate the distance between the best position of each search agent and its current position. The SCSO is designed to provide random angles for the hunt of its prey, so the sand cats will determine where to move based on a roulette wheel in the SCSO. As a result of this angle being random (θ) within a range of 0 to 360 degrees, the cosine of the angle is between -1 and 1 . This will allow the search agents to move circularly. Using Equation (5), the positions are determined through the best solution (\vec{Pos}_b) and random positions (\vec{Pos}_{rnd}).

$$\vec{Pos}(t+1) = \vec{Pos}_b(t) - \vec{r} \cdot \vec{Pos}_{rnd} \cdot \cos(\theta) \quad (5)$$

3.4. Modified SCSO

The sand cat swarm optimization algorithm is one of the newly proposed algorithms described in the previous section. As one of the most metaheuristic algorithms, this algorithm tries to find an optimal solution in a continuous search space. In this type of algorithm, the optimization problem must be in a continuous search space. In this way, for a discrete optimization problem, this kind of algorithm must be modified to fit a discrete search space; so, researchers prefer to use a discrete optimization algorithm. Because there are no more discrete optimization algorithms, researchers attempt to convert continuous research space to discrete. However, the inspiration for the appropriate metaheuristic algorithm is kept. In this paper, the authors used the SCSO algorithm in a discrete version. Most of the algorithms that achieve successful results in a continuous search space do not have the same efficiency in the discrete. According to our results, our proposed algorithm is as effective as the continuous version.

As part of this paper, we propose a modified SCSO algorithm for software module clustering. Although the algorithm is based on the main mechanism of the SCSO algorithm, it has been modified to make use of the position-updating phase of the algorithm in order to achieve optimal results. Consequently, the modified SCSO algorithm is able to achieve optimal results as a result of using the mutation concept of the genetic algorithm (GA). As in the GA, mutations are used to increase diversity and have the opportunity to find

the worst solutions. As a result of this, certain regions of the worst solution can be used to improve the performance of the system. Thus, a controlled mutation method is also incorporated into the modified SCSO algorithm. However, since the mutation is used in discrete systems, the modified SCSO algorithm is also capable of utilizing discrete search space. The modified SCSO algorithm flowchart and pseudocode are given in Figure 2 and Algorithm 1, respectively.

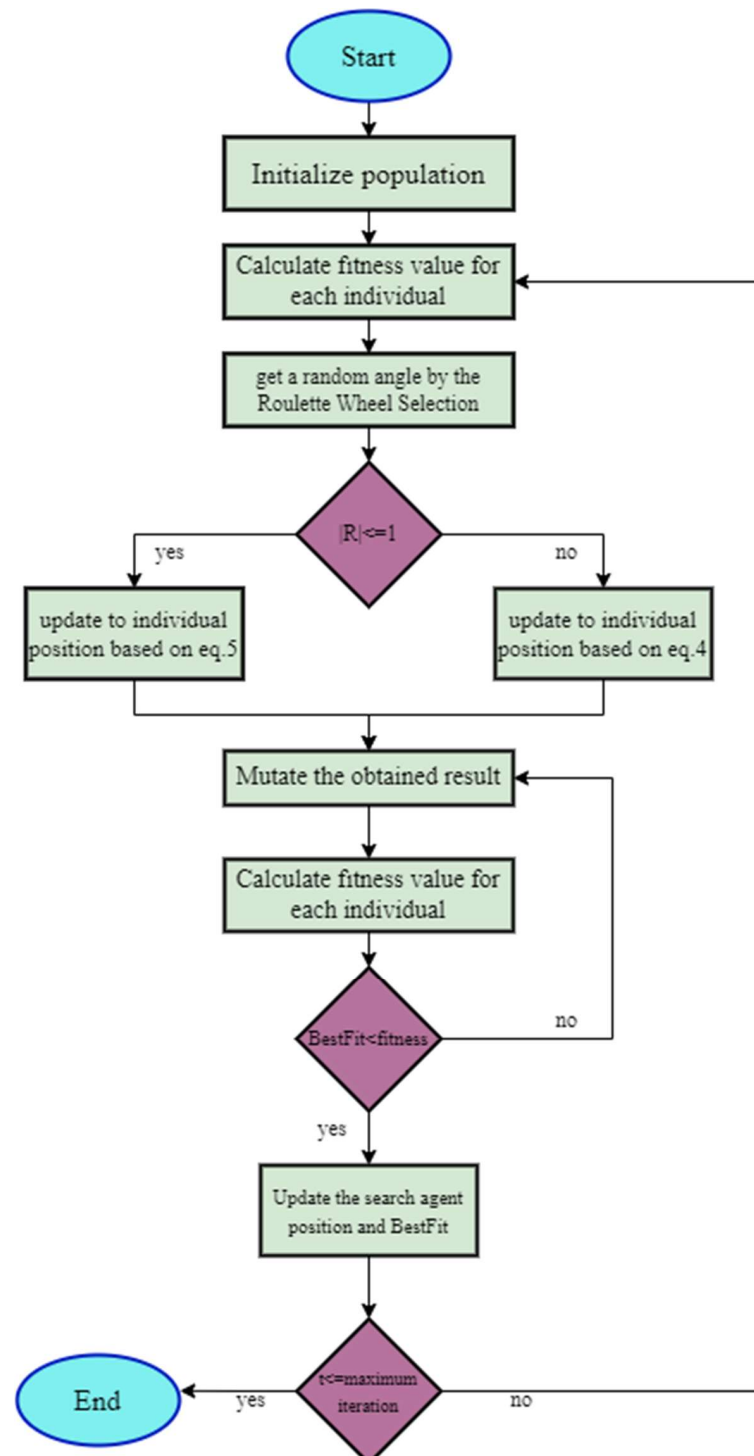


Figure 2. The flowchart of the modified SCSO algorithm.

Algorithm 1 Modified sand cat swarm optimization algorithm pseudocode

```

Population and parameter initialization
Based on the objective function, calculate the fitness function
While (itert <= maxIter)
For each search agent
Get a random angle based on the roulette wheel selection ( $0^\circ \leq \theta \leq 360^\circ$ )
If (abs(R) <= 1)
Update the search agent position based on Equation (5)
Else
Update the search agent position based on Equation (4)
End
Mutation of the obtained results with predefined values
If(obtainedFitness < BestFitness)
Update the search agent position and BestFitness
Else
Mutation of the obtained results with predefined values
End
End
t = t++
End

```

3.5. Fitness Function

The MQ (modularization quality) criterion was used to direct participants in the suggested procedure. This criterion is used by the SCSO algorithm to guide its search for the ideal software clusters. This criterion was suggested by Mancoridis et al. as a gauge of clustering quality [17]. High cohesion (internal connection) and low coupling characterize high-quality clusters (external connection). A cluster with significant cohesion (internal connectivity) is an indication of a good clustering arrangement since the modules within a cluster are highly interconnected. Equation (1) illustrates the method for calculating the clustering quality (MQ) for cluster k . In Equation (6), variable I represents the number of internal connections and variable j the number of exterior connections for a particular cluster. Equation (7) is used to calculate the average quality of all the clusters that were produced. In this equation, the term m stands for the number of clusters. The trade-off between intra-connectivity (cohesion) and interconnectedness is shown via the MQ function (coupling). This function assesses the clustering quality by balancing coupling and cohesion. In this trade-off, the cohesiveness of the individual modules into a cluster must be improved at the expense of the architecture's coupling. While coupling cannot be entirely removed, MQ aims to drastically decrease it. The "ideal" system would consist of a single cluster that houses every module if coupling is thought to be undesirable. Such a solution would not have any coupling. Consequently, coupling and cohesiveness must be balanced using the MQ criteria.

$$MFk = \begin{cases} 0 & \text{if } i = 0 \\ \frac{i}{i+\frac{1}{2}j} & \text{if } i > 0 \end{cases} \quad (6)$$

$$MQ = \sum_{k=1}^m MFk \quad (7)$$

Figure 3 shows the MDG extracted from the source code. The nodes indicate the modules of a software product, and the edges indicate the connections (calls, inheritance, and association) among the modules. There are six modules in the software product shown in Figure 3. The related dependency matrix of the program is shown in Figure 3. The binary values of the matrix cells indicate the module connections. The dependency matrix can be generated from the source code or MDG. Each cat in the SCSO is a clustering array. The length of the clustering array is equal to the number of modules in the software.

Figure 4a shows an MDG of a software product that includes six modules. Figure 4b shows a clustering of the MDG with one cluster; its MQ is evaluated using Equation (7). Figure 4c shows another clustering of the MDG shown in Figure 4a; its MQ is evaluated using Equation (8). The MQ of the second method is about 0.67 lower than the MQ of the first method (shown in Figure 4b). MQ is used to evaluate the quality of the generated clusters using the method.

$$MQ_{Clustering\ 1} = \frac{6}{6 + \frac{0}{2}} = 1 \tag{8}$$

$$MF_{Cluster\ C} = \frac{2}{2 + \frac{2}{2}} = \frac{2}{3}$$

$$MF_{Cluster\ B} = \frac{2}{2 + \frac{2}{2}} = \frac{2}{3} \tag{9}$$

$$MQ_{Clustering\ 2} = MF_{Cluster\ B} + MF_{Cluster\ C} \simeq 0.67$$

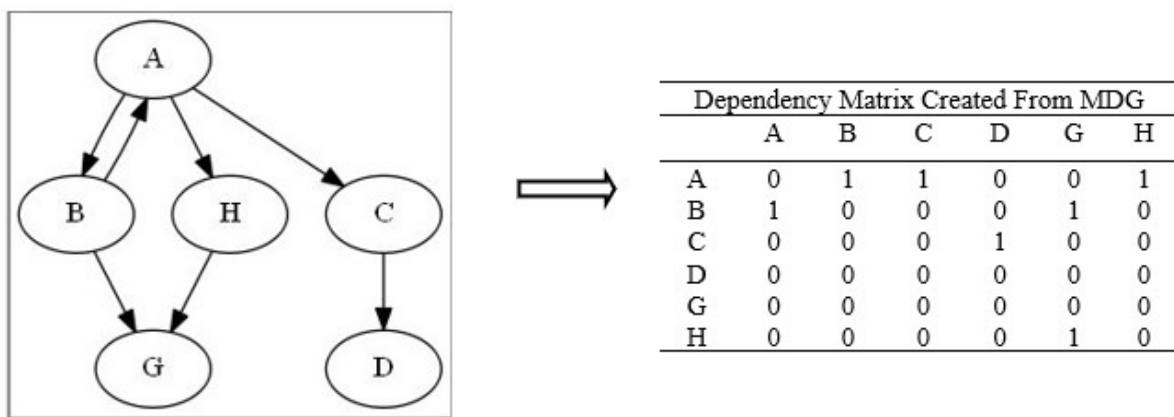


Figure 3. The MDG and related dependency matrix of a software product.

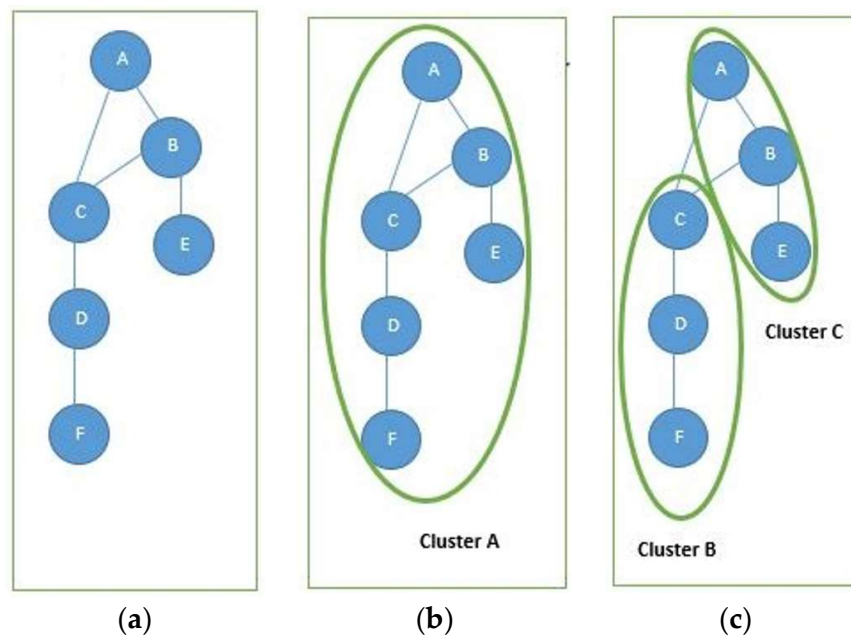


Figure 4. Two different clusterings and their MQ values of the input MDG shown in part (a). (a) MDG without any clustering. (b) MDG with one cluster. (c). MDG with two clusters.

4. Experiments and Results

In order to evaluate the performance of the proposed method, a large number of experiments have been performed on the implemented platform in MATLAB. The proposed

method, along with the GA, PSO, and GWO, was implemented in MATLAB to compare the provided results. The calibration parameters of GA, PSO, GWO, and SCSO were adjusted during the experiments conducted in this study. The optimal values of these parameters for the SMC problem are shown in Table 2. To obtain reliable results, all of the experiments were performed on the same hardware and software platform. In the experiments, ten standard and real-world benchmark MDGs were used. The exploited MDGs include the module dependency graph of small, mid-sized, and large-sized software systems with different numbers of modules and links. The specifications of the benchmark programs are illustrated in Table 3. The selected benchmark programs include real-world complexity in terms of the number of nodes (modules or classes) and the number of connections (edges) among the modules. Figure 5 shows the clustered form of the ispell MDG benchmark (as a small benchmark) that was used in the experiments. This program includes 24 modules and 103 connections among the modules. In Figure 5, the similar modules have been grouped into the same cluster. In this study, the similarity of the modules is defined by the fitness function (Equation (7)). The proposed method tries to group the most similar modules into the same cluster. The modularization quality (MQ) for the clusters shown in Figure 5 is 2.076. The higher the MQ criterion, the better the clustering quality.

Table 2. Best value for the SMC method’s parameters.

Algorithms	Parameters	Value
SCSO	Population size	40
	Sensitivity range (rG)	[0, 2]
	Phases control range (R)	[−2 rG, 2 rG]
	P_c	0.8
	P_m	0.04
Genetic Algorithm (GA)	Number of chromosomes	40
	Cross-mutate rate	0.8
	Mutation rate	0.05
Particle Swarm Optimization (PSO)	Population size	40
	W	[0.7, 0.8]
	C1	[1.5, 1.7]
	C2	[1.5, 1.7]

Table 3. The benchmark programs.

Programs	# Modules	# Edges	Size
mtunis	20	57	Small
spdb	21	16	Small
ispell	24	97	Small
Rcs	29	155	Mid
bison	37	117	Mid
Cia	38	216	Large
Dot	42	248	Large
Php	62	163	Large
Grappa	86	252	Large
Incle	174	360	Large

MQ: 2.07678
Cohesion: 34
Coupling: 63
Number of edges: 103
Number of modules: 24
Number of clusters: 5

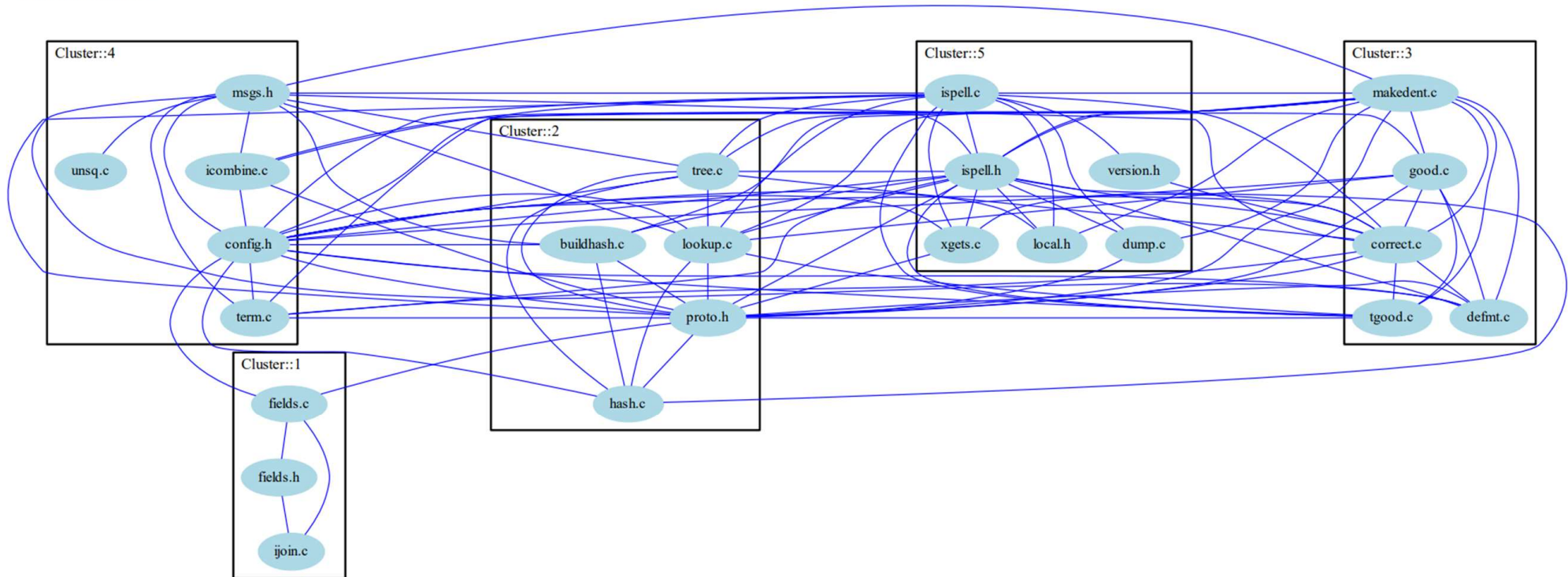


Figure 5. The clustered form of the *ispell* benchmark with 20 modules and 57 edges.

In the experiments, different evaluation criteria were taken into account. MQ is the first performance criteria for SMC methods. The quality of the generated clusters using SMC methods is evaluated through the value of the MQ. All the SMC methods try to find the clusters with the highest MQ. The MQ of a generated cluster for an MDG is measured using Equation (7). The clusters with lower coupling and higher cohesion have higher MQ. The number of clusters can be one and the number of modules. The best value for the number of clusters is determined experimentally. Convergence speed is the other performance criterion that was taken into consideration. The convergence speed is related to the time required for finding the best clustering using an SMC algorithm. Success rate is another performance criterion measured for SMC methods. The probability of finding the best clusters using an SMC method indicates its success rate. To this end, each SMC method was executed 10 times on each benchmark program. The number of times an SMC method reaches the optimal solution divided by 10 indicates the success rate. Stability is the other performance criterion that is measured for SMC methods. SMC methods use different heuristic algorithms. The standard deviation (STDV) of the results obtained from different executions of an SMC algorithm is used to evaluate the stability of the algorithm. The lower the value of the STDV, the higher the stability. The results obtained using more stable methods are more reliable.

Results

The proposed method was executed in ten benchmark programs. The clusters generated by the proposed method were evaluated using the MQ criteria. Tables 4 and 5 show the MQ of the generated clusters for each benchmark program. The methods were executed 20 times for each benchmark. Selecting the number of clusters in each benchmark is an important parameter of the SMC problem. The best value for the cluster number is determined experimentally. The number of clusters depends on different parameters such as the number of modules, the number of edges in its MDG, and the correlation of the modules; hence, it should be determined experimentally. As shown in Tables 4 and 5, the proposed method was executed on each benchmark with a different number of clusters. Each execution of the proposed method on a benchmark included 100 iterations.

In order to evaluate the performance of the proposed method, the obtained data (some of them shown in Tables 4 and 5) were analyzed. In the SMC problem, the methods are compared in terms of the best MQ, average MQ, convergence speed, and stability. Figure 6 shows the best performance of the proposed method in different benchmark programs. The performance of the SMC method depends on the MQ of the clusters generated by the method. Figure 6 shows the MQ of the best clusters generated by the proposed method. The proposed method has a higher MQ specifically in the large benchmark programs. Indeed, the MQ of the generated clusters for the large software programs is higher than the other methods. The average MQ of the generated clusters in the experiments in the best case is about 3.320, whereas the MQ of the PSO–GA is about 3.310. PSO–GA combines the PSO and GA algorithms to exploit the advantages of both algorithms. In small programs, the SMC algorithms exhibits similar performance. Figure 7 shows the MQ of the generated clusters by different SMC algorithms in the worst cases. In the executed experiments, the average MQ of the worst clusters generated during 20 executions was about 2.885. Indeed, the quality of the worst clusters generated using the proposed method is higher than the other methods.

Table 4. The MQ of the generated clusters for *mtunis*, *spdb*, *ispell*, *rcs* and *bison* benchmarks in 20 runs of the proposed method.

Runs Number	Benchmark	Mtunis			SPDb			ispell			rcs		Bison		
	Num of Cluster	2	3	5	2	6	9	3	5	7	4	6	8	4	6
1	1.5788	1.8329	2.3145	5.7412	5.7412	5.7412	1.9343	2.086	2.1475	1.9326	2.069	2.1314	1.9091	2.3096	2.4982
2	1.5788	1.8588	2.3145	5.7412	5.7412	5.7412	1.8869	2.1368	2.1581	1.9243	2.1035	2.12	1.9206	2.2934	2.4254
3	1.5788	1.8588	2.3145	5.7412	4.7474	5.7412	1.8869	2.1158	2.1622	1.9242	2.1004	2.1499	1.9316	2.199	2.312
4	1.5788	1.8588	2.3145	5.7412	5.7412	5.7412	1.9343	2.1368	2.0548	1.9354	2.0509	2.1321	1.9372	2.1843	2.3933
5	1.5788	1.8329	2.3145	5.7412	5.2235	5.7412	1.9057	2.1292	2.1216	1.9516	2.083	2.1708	1.9466	2.1739	2.4846
6	1.5788	1.8586	2.3145	5.7412	5.7412	5.7412	1.9343	2.1292	2.1654	1.9304	2.0884	2.1109	1.9216	2.1435	2.2634
7	1.4913	1.8588	2.3145	5.7412	4.7474	5.7412	1.9343	2.1368	2.1756	1.9195	2.1394	2.1194	1.7801	2.2736	2.4653
8	1.5788	1.7926	2.3145	5.7412	5.7412	5.7412	1.9343	2.1167	2.1538	1.8102	2.0778	2.1074	1.933	2.2896	2.4225
9	1.5788	1.8329	2.3145	5.7412	5.7412	5.7412	1.8585	2.1368	2.1018	1.9272	2.058	2.0966	1.9285	2.2239	2.3819
10	1.5788	1.8588	2.3145	5.7412	5.7412	5.7412	1.9193	2.1223	2.1899	1.9238	2.0992	2.126	1.8055	2.2789	2.258
11	1.5788	1.8588	2.3145	5.7412	4.7983	5.7412	1.9193	2.0658	2.1484	1.9283	2.1343	2.1234	1.9461	2.1413	2.3657
12	1.4913	1.8329	2.3145	5.7412	5.7412	5.7412	1.9051	2.0965	2.1625	1.9382	2.079	2.1382	1.8016	2.2948	2.4423
13	1.5788	1.7926	2.3145	5.7412	5.7412	4.7474	1.9051	2.1419	2.1492	1.9156	2.1059	2.0983	1.8898	2.0944	2.3949
14	1.5788	1.8329	2.3145	5.7412	5.7412	5.7412	1.9057	2.063	2.1579	1.9212	2.0854	2.1025	1.9497	2.2307	2.3493
15	1.5788	1.8588	2.3145	5.7412	5.7412	5.7412	1.9343	2.0715	2.168	1.9242	2.1155	2.1537	1.9299	2.1941	2.3421
16	1.5788	1.8329	2.3145	5.7412	5.7412	5.7412	1.9343	2.1522	2.1494	1.9334	2.018	2.101	1.9212	2.2034	2.2627
17	1.5788	1.8286	2.3145	5.7412	5.7412	5.7412	1.8766	2.1292	2.1899	1.8175	2.0471	2.0734	1.9572	2.2969	2.4546
18	1.5788	1.8588	2.3145	5.7412	5.7412	5.7412	1.8623	2.1201	2.173	1.9213	2.1219	2.1428	1.8952	2.2888	2.3208
19	1.3354	1.8588	2.3145	5.7412	5.7412	5.7412	1.8766	2.1522	2.1682	1.9239	2.1215	2.1262	1.9677	2.3114	2.4174
20	1.5788	1.8588	2.3145	5.7412	4.7983	5.7412	1.9343	2.1243	2.1605	1.9298	2.118	2.1539	1.9208	2.1866	2.3357

Table 5. The MQ of the generated clusters for *cia*, *dot*, *php*, *grappa*, and *incle* benchmarks in 20 runs of the proposed method.

Benchmark	cia			Dot			Php			grappa			Incle			
	Num of Cluster	4	5	8	5	8	10	5	8	10	6	9	12	6	9	12
Runs Number	1	1.9779	2.3221	2.4402	2.0372	2.2345	2.3691	2.4569	3.1658	3.2426	3.1125	3.7766	4.8342	2.8566	3.1288	4.0849
	2	2.0294	2.2014	2.378	1.9539	2.2719	2.4234	2.2761	3.1039	3.4534	3.0087	4.0463	5.1269	3.1009	3.2741	4.1521
	3	2.0339	2.2063	2.4405	1.9395	2.2131	2.3251	2.4341	2.8972	3.4954	3.2639	3.9741	4.4655	2.9586	3.3348	3.8131
	4	1.9394	2.2739	2.2721	1.9765	2.2533	2.4244	2.5243	3.1991	3.5236	2.9187	3.7832	4.4747	2.4894	3.1647	4.0183
	5	2.0095	2.3423	2.3313	1.9654	2.413	2.3318	2.3795	3.0004	3.4165	3.0295	3.549	4.5945	2.7704	3.2089	4.254
	6	2.0339	2.2465	2.4171	1.9793	2.2635	2.2771	2.5577	3.187	3.6409	3.2625	3.8471	4.8048	2.8157	3.3483	3.965
	7	1.9357	2.2067	2.4209	1.9891	2.2847	2.4597	2.5374	3.0952	3.5331	2.7447	3.4325	4.2638	2.4817	3.4964	3.6061
	8	1.9843	2.275	2.4604	2.0238	2.2203	2.4309	2.5403	2.9074	3.4525	2.817	3.8644	4.3635	2.3992	3.2879	3.8646
	9	2.0121	2.3186	2.4522	1.9687	2.2701	2.3266	2.2593	3.2362	3.3616	2.9667	4.0904	4.8428	2.8425	3.4349	3.8800
	10	2.0175	2.0938	2.4329	1.9054	2.2859	2.4113	2.4445	3.0447	3.1577	3.1282	3.8296	4.5456	2.7469	3.5857	4.0520
	11	1.9995	2.3227	2.4400	1.9313	2.3345	2.495	2.2958	3.0752	3.3336	2.9957	3.4981	4.435	2.6722	3.3484	4.1005
	12	1.9497	2.2889	2.3240	2.0172	2.1968	2.3014	2.2821	3.0896	3.7083	3.3946	4.4796	5.0126	2.7414	3.4878	3.9198
	13	1.9763	2.1975	2.5003	1.9707	2.233	2.4340	2.462	3.0392	3.6035	3.0771	3.9823	4.6013	2.5326	3.3311	3.7315
	14	2.0241	2.2869	2.3938	1.6494	2.2821	2.3773	2.4063	3.2853	3.254	3.1588	3.7945	5.1012	2.7827	3.4141	3.6892
	15	1.9833	2.3608	2.3578	1.9598	2.2403	2.3834	2.326	3.0582	3.4204	3.1035	3.4801	4.9125	2.6992	3.4968	3.7882
	16	2.0222	2.1962	2.3988	1.9604	2.3514	2.374	2.5321	3.17	3.1605	3.3943	4.1102	4.8747	2.9511	3.4265	3.7478
	17	2.0023	2.2744	2.3956	1.9632	2.1814	2.3757	2.452	3.2065	3.5177	3.0367	3.8518	4.8804	2.8871	3.0117	3.9116
	18	2.0097	2.3103	2.3239	1.9646	2.1457	2.3043	2.4934	3.2302	3.394	3.281	3.995	4.6482	2.943	3.5177	4.3516
	19	1.9656	2.2387	2.4261	2.0496	2.2482	2.3989	2.2812	3.2239	3.4599	2.8828	4.2824	5.012	2.8368	3.2773	3.4710
	20	2.0500	2.2289	2.3961	2.0216	2.3033	2.3203	2.1992	3.1698	3.4634	3.149	3.9254	4.4157	2.9281	3.1679	3.8497

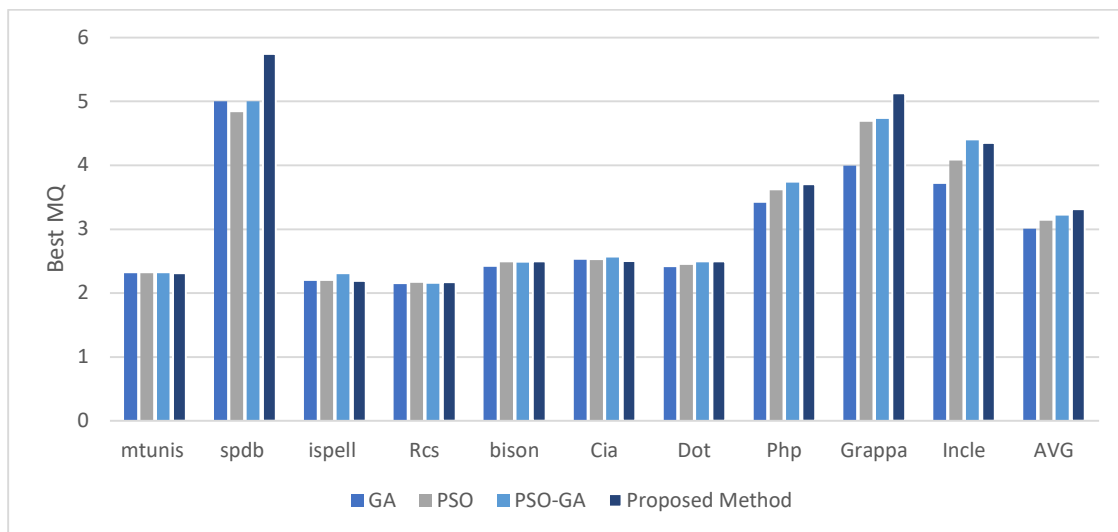


Figure 6. The best MQ obtained using different SMC algorithms for all benchmark programs.

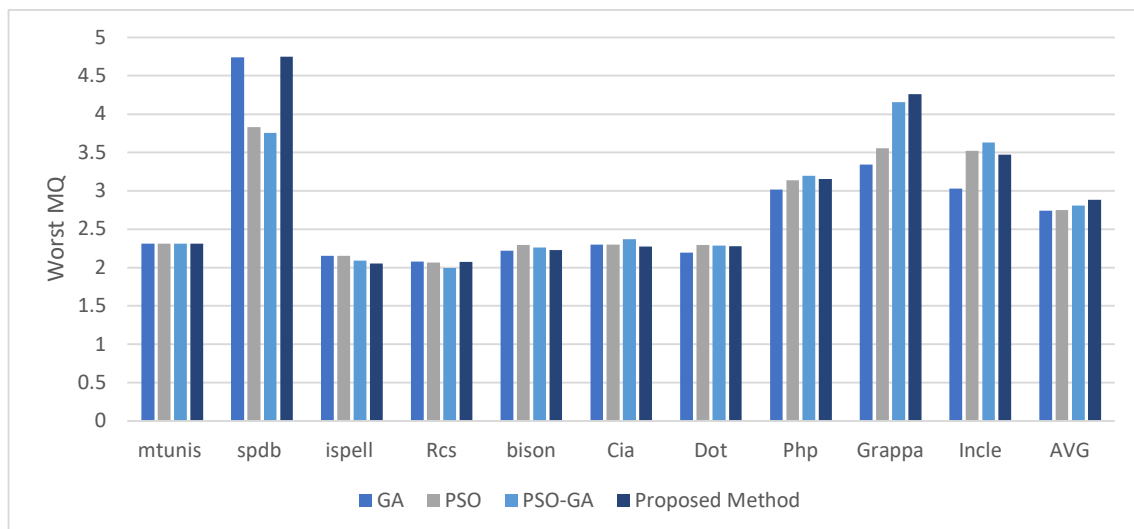


Figure 7. The worst MQ obtained using different SMC algorithms for all benchmark programs.

Figure 8 shows the MQ of the generated clusters for the mtunis, spdb, ispell, and rcs benchmarks using the box diagram. The proposed method has a higher MQ for all of these benchmarks. Indeed, the quality of the generated clusters for the mtunis, spdb, and rcs benchmarks has a higher MQ than the clusters generated using GA, PSO, and PSO-GA algorithms. Only in the spell benchmark is the average performance of the PSO algorithm better than that of the other algorithms. As shown in Figure 9, the MQ of the generated clusters for the bison, cia, and dot benchmarks using the proposed method is higher than the other methods. In the cia benchmark, PSO-GA shows higher performance than the other algorithms. The best clusters for the ispell benchmark were generated using the PSO-GA. Figure 10 shows the MQ of the generated clusters using the SMC algorithms for the grappa and incle benchmarks. SCSO and PSO-GA show similar performances in these benchmarks. In the grappa benchmark, SCSO shows a higher performance in terms of MQ. In the incle benchmark, PSO-GA and SCSO exhibit good results.

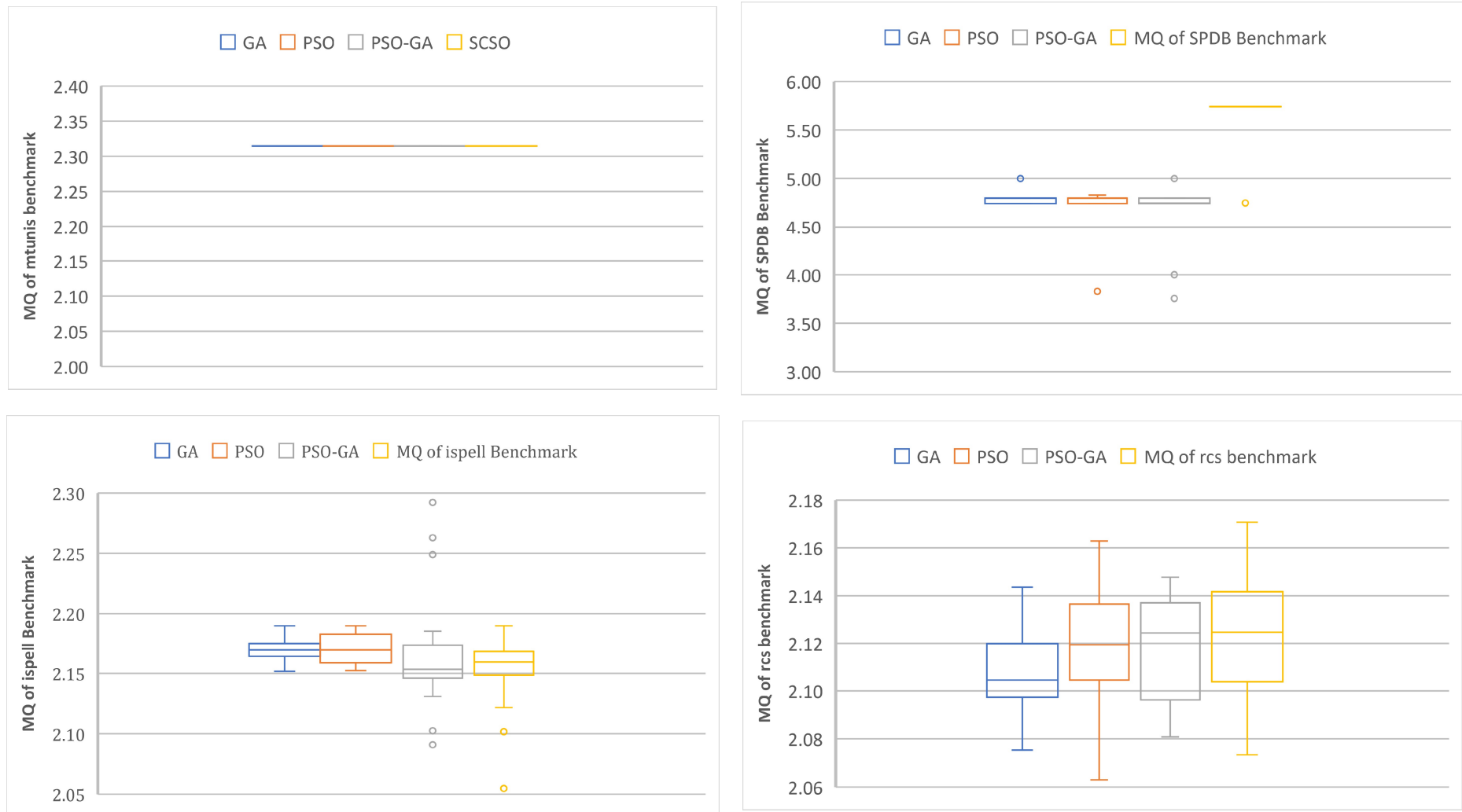


Figure 8. The MQ values obtained using different SMC algorithms for mtunis, spdb, ispell, and rcs benchmarks.

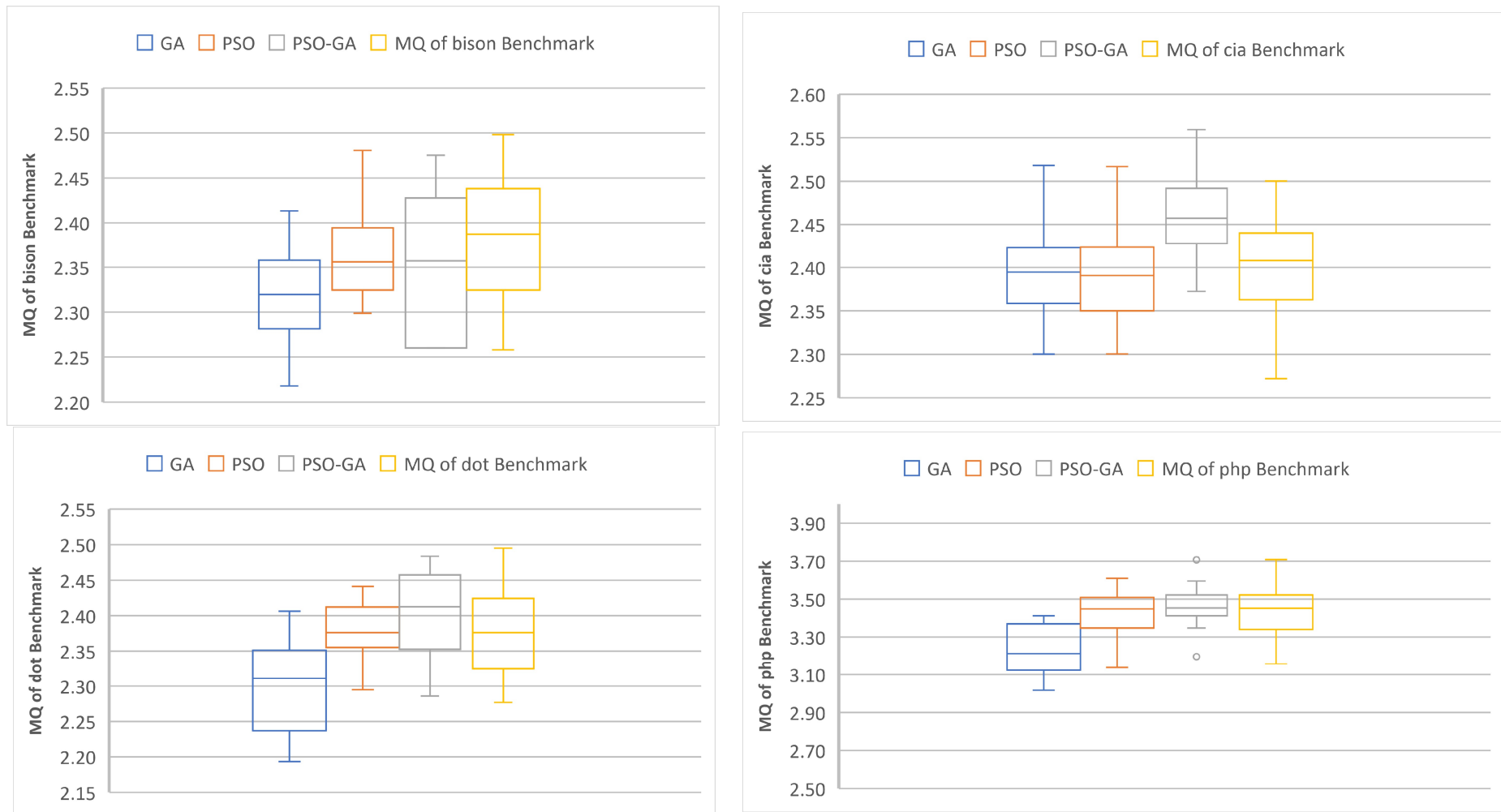


Figure 9. The MQ values obtained using different SMC algorithms for bison, cia, dot, and php.

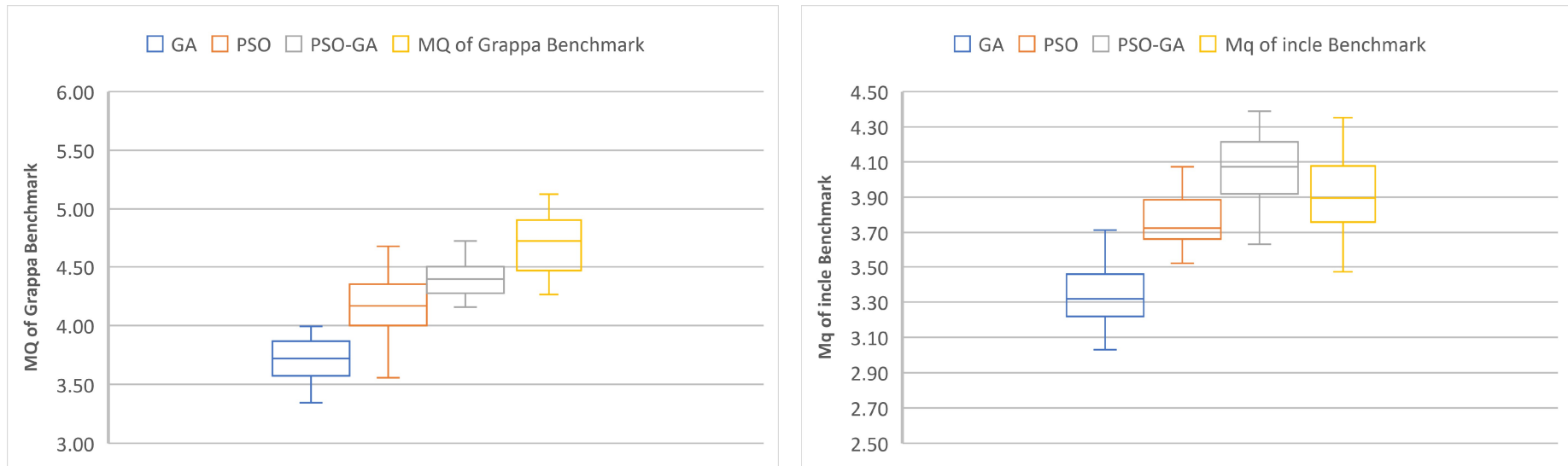


Figure 10. The MQ values obtained using different SMC algorithms for grappa and inle benchmarks.

Figure 11 shows the average MQ of the generated clusters in 20 executions. According to the results, the average MQ of the generated clusters using SCSSO in the experiments is about 3.148. The average MQ of SCSSO is higher than for other SMC algorithms. Indeed, in the proposed method, the modules grouped in the same cluster have higher similarity in terms of functionality and relation. Figures 9 and 10 indicate that the discretized SCSSO is a successful SMC algorithm, especially in large software products. Most algorithms have similar performance in programs with a small number of modules and edges. The SCSSO has a higher MQ value in the best (Figure 6), worst (Figure 7), and average cases (Figure 11). In fact, in all cases, the discretized SCSSO has higher performance in regard to the SMC problem.

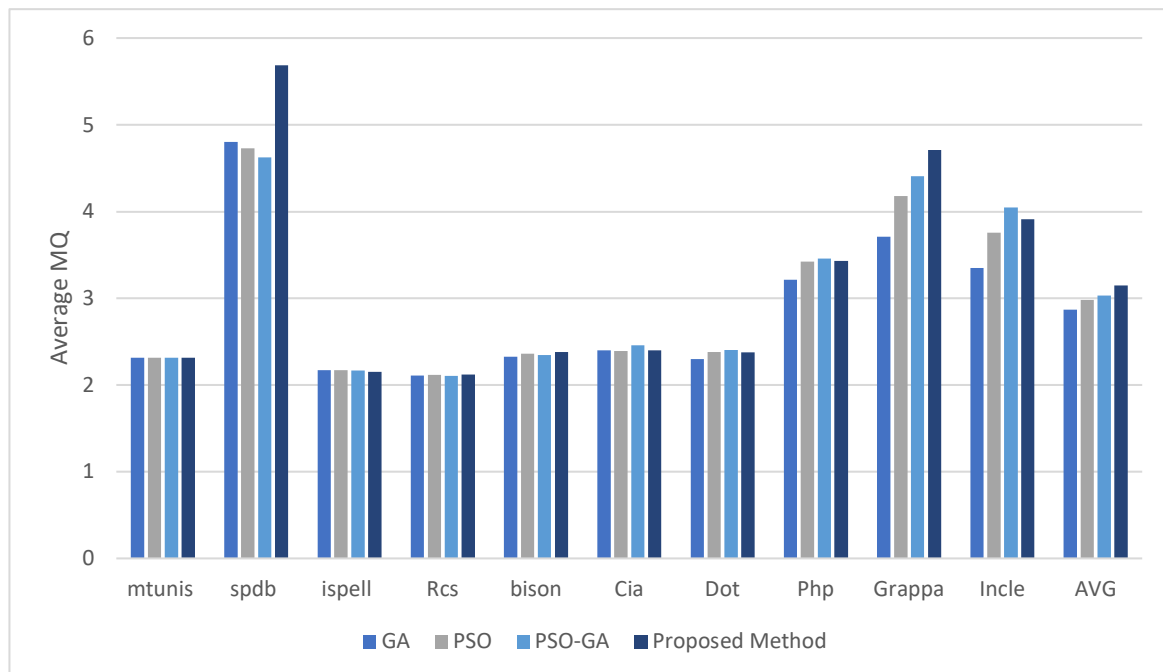


Figure 11. The average MQ obtained using different SMC algorithms for all benchmark programs.

The other important concern regarding heuristic algorithms is their reliability. Heuristic algorithms may have different results for the same inputs during different executions; this is due to the use of random coefficients in heuristic algorithms. Hence, the reliability of the results generated using a heuristic algorithm should be taken into account. To this end, the standard deviation (STDV) among the MQ of the generated clusters in 20 executions of each SMC algorithm was evaluated in this study. Figure 12 shows the STDV of different algorithms in different benchmark programs. The STDV of SCSSO in the conducted experiments is about 0.104, which is lower than PSO and PSO-GA. In these experiments, the STDV of the GA is lower than SCSSO. The lower the STDV, the higher the stability of the SMC algorithm.

Figures 13–15 show the MQ of the generated clusters using the SMC algorithms during 20 executions. In the mtunis benchmark, all the SMC algorithms generated the same results with the same stability (STDV value). This benchmark is related to a small software product that includes 20 modules with 57 edges among them. As shown in Figure 13, in the spdb benchmark, SCSSO has higher MQ and lower deviation. All SMC algorithms have similar MQ in the ispell benchmark, but the stability of SCSSO is higher than the other algorithms. In the rcs benchmark, the performance of all algorithms is similar in terms of MQ and stability. Figure 16 depicts the MQs obtained using the proposed method and up-to-date optimizers.

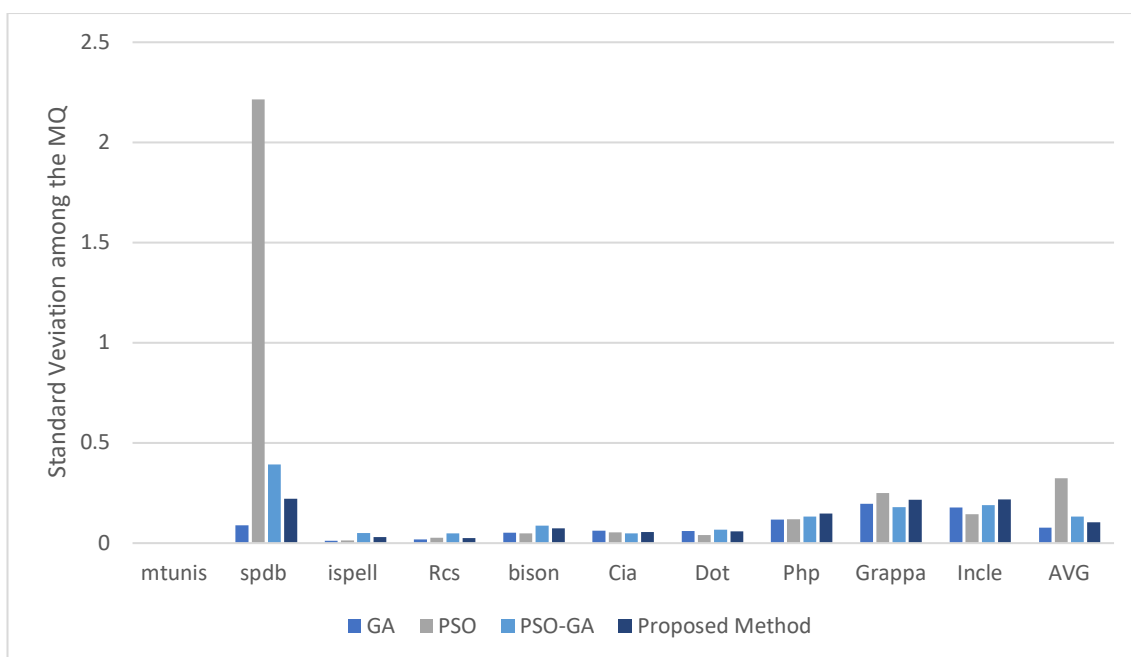


Figure 12. The standard deviation among the results obtained from 20 times execution of each SMC algorithm.

In order to analyze the significance of the difference (improvement) among the results obtained using the proposed method and other SMC methods, an ANOVA test, as parametric statistical analysis, has been conducted on the results obtained for two benchmarks. ANOVA, which stands for analysis of variance, is a statistical test used to analyze the difference between the means of more than two groups. Tables 6 and 7 indicate the results of the ANOVA test. In the SPDB benchmark, the f-ratio value is 75.11321 and the p -value is <0.00001 . Indeed, in this benchmark, the proposed method makes a significant improvement in the values of MQ. Furthermore, in the incle benchmark, the f-ratio value is 53.87688 and the p -value is <0.00001 . Similar to the previous benchmark, the null hypothesis was rejected, and there are significant differences among the obtained results. Similar results have been obtained for the other benchmarks.



Figure 13. The stability diagram of different SMC algorithms during 20 times execution for mtunis, spdb, spwill, and rcs.

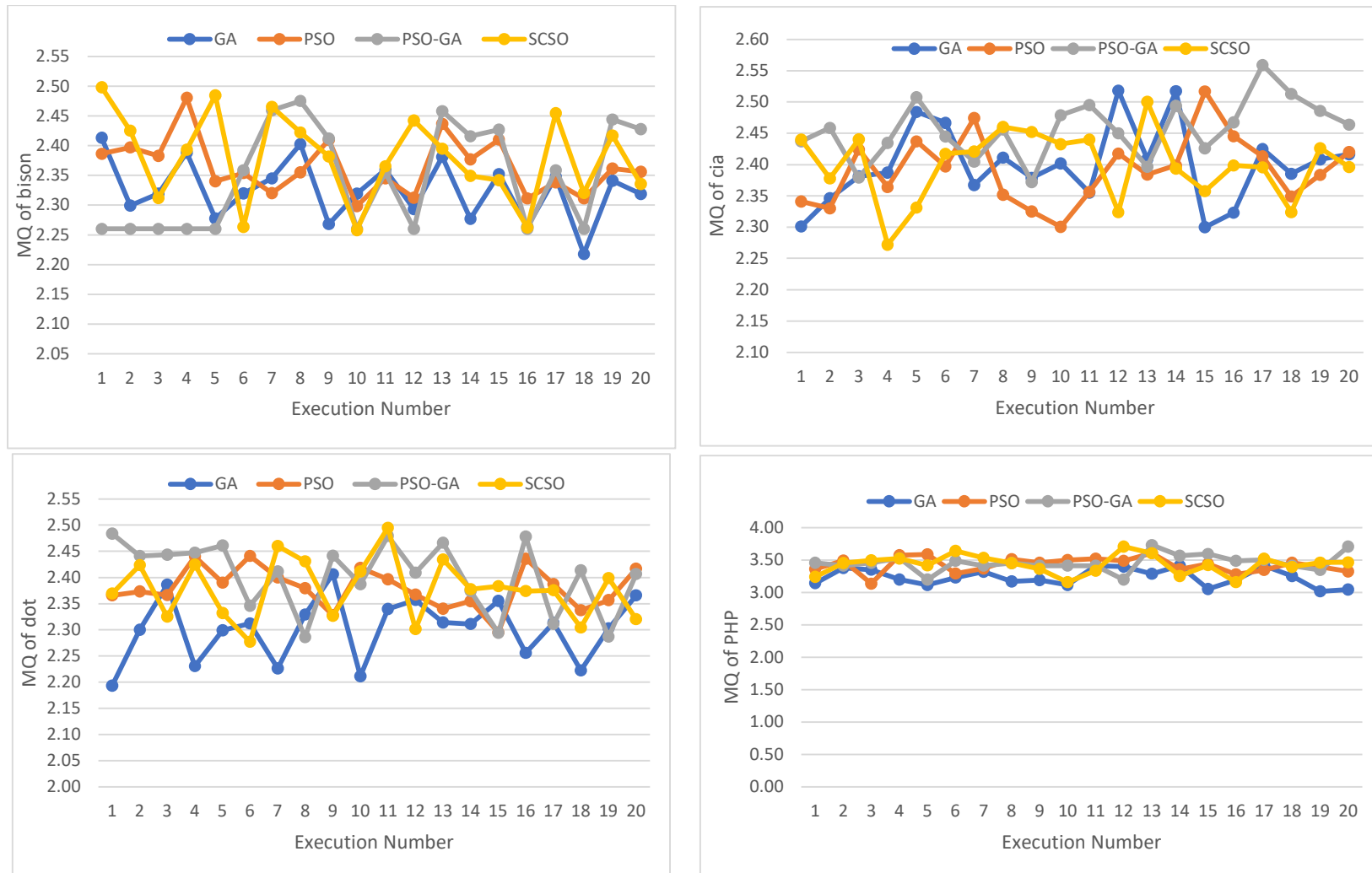


Figure 14. The stability diagram of different SMC algorithms during 20 times executions for bison, cia, dot, and php.

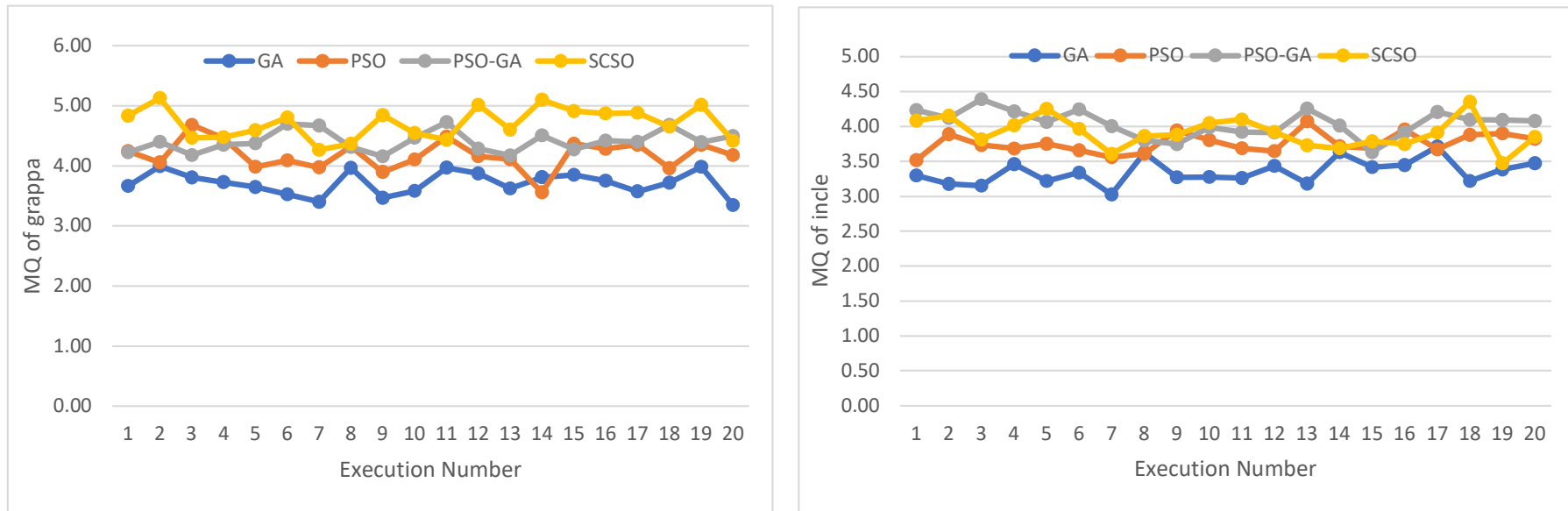


Figure 15. The stability diagram of different SMC algorithms during 20 times executions for grappa and incle.



Figure 16. The comparison of the proposed method with the up-to-date optimizers [2,7,23,40].

Table 6. The results of ANOVA test on SPDB and inle benchmarks in 20 runs of the SMC methods.

The Statistical Analysis of Obtained Results for SPDB Dataset				
Source	SS	df	MS	
Between methods	14.4385	3	4.8128	F = 75.11321
Within methods	4.8697	76	0.0641	
Total	19.3082	79		
<ul style="list-style-type: none"> Null hypothesis was rejected, and there is significant difference among the results in SPDB benchmark. 				
The statistical analysis of the obtained results for inle dataset				
Between treatments	5.455	3	1.8183	F = 53.87688
Within treatments	2.565	76	0.0338	
Total	8.02	79		
<ul style="list-style-type: none"> Null hypothesis was rejected, and there is significant difference among the results in inle benchmark. 				

Table 7. Post hoc Tukey analysis of the results obtained for SPDB and inle benchmarks.

The Tukey Analysis of Obtained Results for SPDB Dataset			
Pairwise Comparisons		HSD _{.05} = 0.2103, HSD _{.01} = 0.2577	Q _{.05} = 3.7149, Q _{.01} = 4.5530
T ₁ :T ₂	M ₁ = 4.80 M ₂ = 4.73	0.07	Q = 1.28 (p = 0.80117)
T ₁ :T ₃	M ₁ = 4.80 M ₃ = 4.63	0.18	Q = 3.11 (p = 0.13252)
T ₁ :T ₄	M ₁ = 4.80 M ₄ = 5.69	0.89	Q = 15.68 (p = 0.00000)
T ₂ :T ₃	M ₂ = 4.73 M ₃ = 4.63	0.10	Q = 1.83 (p = 0.57020)
T ₂ :T ₄	M ₂ = 4.73 M ₄ = 5.69	0.96	Q = 16.96 (p = 0.00000)
T ₃ :T ₄	M ₃ = 4.63 M ₄ = 5.69	1.06	Q = 18.79 (p = 0.00000)

Table 7. Cont.

The Tukey analysis of obtained results for inclc dataset			
Pairwise Comparisons		HSD _{.05} = 0.1526, HSD _{.01} = 0.1870	Q _{.05} = 3.7149 Q _{.01} = 4.5530
T ₁ :T ₂	M ₁ = 3.35 M ₂ = 3.76	0.41	Q = 9.94 (<i>p</i> = 0.00000)
T ₁ :T ₃	M ₁ = 3.35 M ₃ = 4.05	0.70	Q = 16.95 (<i>p</i> = 0.00000)
T ₁ :T ₄	M ₁ = 3.35 M ₄ = 3.91	0.56	Q = 13.67 (<i>p</i> = 0.00000)
T ₂ :T ₃	M ₂ = 3.76 M ₃ = 4.05	0.29	Q = 7.00 (<i>p</i> = 0.00003)
T ₂ :T ₄	M ₂ = 3.76 M ₄ = 3.91	0.15	Q = 3.73 (<i>p</i> = 0.04853)
T ₃ :T ₄	M ₃ = 4.05 M ₄ = 3.91	0.13	Q = 3.27 (<i>p</i> = 0.10396)

• T1: GA, T2: PSO, T3: PSO-GA, T4: SCSO

5. Conclusions

A huge software system's structure may not always be apparent from its source code. One of the toughest areas of software engineering is figuring out the affected code sections of a source code during the maintenance process. Clustering source code modules can save maintenance costs by making it easier to comprehend how a program is put together. The goal of this work was to create clustered structural models that outperform earlier approaches in terms of cohesion, coupling, and MQ value. In this study, the best clusters were created using a discretized and modified version of SCSO. Ten common applications were changed to MDG versions for this study's benchmark programs. The selected benchmark programs contained both a large and small real-world software product. The developed clustering techniques have been used in a huge number of studies. The GA, PSO, and PSO-GA approaches to addressing the SMC problem were contrasted with the suggested approach.

As part of the proposed technique, it is implied or explicit that it attempts to discover high-quality regions of the search space by learning the correlations between decision factors. At each iteration of our proposed approach (SCSO), the search space is sampled according to a probability distribution. This work proposes an explicit technique for escaping local minima and balancing phase changes. The proposed modified SCSO algorithm should be applied to a high dimension algorithm in order to assess performance. Generally speaking, SCSO performs better than other algorithms in a variety of situations, especially in large software projects. The implementation of algorithms in a way that is independent of the software product's size is one of the suggested future efforts. For small and large software products, several methodologies call for distinct calibrations. The impact of chaotic equations on the effectiveness of SMC algorithms should be the subject of further study. In the SMC problem, integrating several swarm and evolution-based methods may also produce superior outcomes. Future research is advised for improving the fitness function to take the other needs into consideration. Global modules are not included in MQ, despite it being often used as a universal criterion in recent research. Global modules are those that receive calls from more than two different modules but do not themselves make any calls; future research can be performed on this issue.

Author Contributions: Conceptualization, B.A. and A.S.; methodology, B.A.; software, B.A., A.S. and J.R.; validation, B.A. and A.S.; formal analysis, B.A.; investigation, B.A., A.S., J.R. and A.M.A.-M.; resources, B.A. and A.M.A.-M.; data curation, B.A., J.R. and A.M.A.-M., writing—original draft preparation, B.A. and A.S.; writing—review and editing, B.A., A.S., J.R. and A.M.A.-M.; visualization,

B.A.; supervision, B.A.; project administration, B.A. and A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mitchell, B.S. *A Heuristic Search Approach to Solving the Software Clustering Problem*; Drexel University: Philadelphia, PA, USA, 2002.
2. Praditwong, K.; Harman, M.; Yao, X. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Trans. Softw. Eng.* **2011**, *37*, 264–282. [[CrossRef](#)]
3. Pressman, R. *Software Engineering: A Practitioner's Approach: European Adaptation*; McGraw-Hill: New York, NY, USA, 2000.
4. Paikray, H.K.; Das, P.K.; Panda, S. Optimal Multi-robot Path Planning Using Particle Swarm Optimization Algorithm Improved by Sine and Cosine Algorithms. *Arab. J. Sci. Eng.* **2021**, *46*, 3357–3381. [[CrossRef](#)]
5. Amarjeet; Chhabra, J.K. An empirical study of the sensitivity of quality indicator for software module clustering. In Proceedings of the 2014 Seventh International Conference on Contemporary Computing (IC3), Noida, India, 7–9 August 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 206–211. [[CrossRef](#)]
6. Amarjeet; Chhabra, J.K. Improving modular structure of software system using structural and lexical dependency. *Inf. Softw. Technol.* **2017**, *82*, 96–120. [[CrossRef](#)]
7. Keshtgar, S.A.; Arasteh, B.B. Enhancing Software Reliability against Soft-Error using Minimum Redundancy on Critical Data. *Int. J. Comput. Netw. Inf. Secur.* **2017**, *9*, 21–30. [[CrossRef](#)]
8. Garey, M.R.; Johnson, D.S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*; W. H. Freeman & Co.: New York, NY, USA, 1979.
9. ZadahmadJafarlou, M.; Arasteh, B.; YousefzadehFard, P. A Pattern-Oriented and Web-Based Architecture to Support Mobile Learning Software Development. *Procedia-Soc. Behav. Sci.* **2011**, *28*, 194–199. [[CrossRef](#)]
10. Bouyer, A.; Arasteh, B.; Movaghar, A. A New Hybrid Model Using Case-Based Reasoning and Decision Tree Methods for Improving Speedup and Accuracy. In Proceedings of the IADIS International Association for Development of the Information Society, Salamanca, Spain, 18–20 February 2007; pp. 787–789.
11. Arasteh, B.; Karimi, M.B.; Sadegi, R. Düzen: Generating the structural model from the software source code using shuffled frog leaping algorithm. *Neural Comput. Appl.* **2023**, *35*, 2487–2502. [[CrossRef](#)]
12. Jalali, M.; Bouyer, A.; Arasteh, B.; Moloudi, M. The Effect of Cloud Computing Technology in Personalization and Education Improvements and its Challenges. *Procedia-Soc. Behav. Sci.* **2013**, *83*, 655–658. [[CrossRef](#)]
13. Ghaemi, A.; Arasteh, B. SFLA-based heuristic method to generate software structural test data. *J. Softw. Evol. Process* **2020**, *32*, 22–28. [[CrossRef](#)]
14. Arasteh, B.; Abdi, M.; Bouyer, A. Program source code comprehension by module clustering using combination of discretized gray wolf and genetic algorithms. *Adv. Eng. Softw.* **2022**, *173*, 103252. [[CrossRef](#)]
15. Kirtil, H.S.; Seyyedabbasi, A. A Hybrid Metaheuristic Algorithm for the Localization Mobile Sensor Nodes. In *Forthcoming Networks and Sustainability in the IoT Era. FoNeS-IoT 2021. Lecture Notes on Data Engineering and Communications Technologies*; Al-Turjman, F., Rasheed, J., Eds.; Springer: Cham, Switzerland, 2022; Volume 130, pp. 40–52. [[CrossRef](#)]
16. Seyyedabbasi, A.; Kiani, F. Sand Cat swarm optimization: A nature-inspired algorithm to solve global optimization problems. *Eng. Comput.* **2022**, 1–25. [[CrossRef](#)]
17. Mancoridis, S.; Mitchell, B.S.; Chen, Y.; Gansner, E.R. Bunch: A clustering tool for the recovery and maintenance of software system structures. In Proceedings of the IEEE International Conference on Software Maintenance-1999 (ICSM'99). "Software Maintenance for Business Change" (Cat. No.99CB36360), Oxford, UK, 30 August–3 September 1999; IEEE: Piscataway, NJ, USA, 1999; pp. 50–59. [[CrossRef](#)]
18. Harman, M.; Swift, S.; Mahdavi, K. An empirical study of the robustness of two module clustering fitness functions. In Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA, 25 June 2005; ACM: New York, NY, USA, 2005; pp. 1029–1036. [[CrossRef](#)]
19. Mahdavi, K.; Harman, M.; Hierons, R.M. A multiple hill climbing approach to software module clustering. In Proceedings of the International Conference on Software Maintenance, 2003. ICSM 2003, Amsterdam, The Netherlands, 22–26 September 2003; IEEE: Piscataway, NJ, USA, 2003; pp. 315–324. [[CrossRef](#)]
20. Bavota, G.; Carnevale, F.; De Lucia, A.; Di Penta, M.; Oliveto, R. Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization. In *Search Based Software Engineering. SSBSE 2012*; Fraser, G., Teixeira de Souza, J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7515, pp. 75–89. [[CrossRef](#)]
21. Amarjeet; Chhabra, J.K. TA-ABC: Two-Archive Artificial Bee Colony for Multi-objective Software Module Clustering Problem. *J. Intell. Syst.* **2018**, *27*, 619–641. [[CrossRef](#)]
22. Pourasghar, B.; Izadkhah, H.; Isazadeh, A.; Lotfi, S. A graph-based clustering algorithm for software systems modularization. *Inf. Softw. Technol.* **2021**, *133*, 106469. [[CrossRef](#)]

23. Arasteh, B.; Sadegi, R.; Arasteh, K. Bölen: Software module clustering method using the combination of shuffled frog leaping and genetic algorithm. *Data Technol. Appl.* **2021**, *55*, 251–279. [[CrossRef](#)]
24. Sun, J.; Ling, B. Software Module Clustering Algorithm Using Probability Selection. *Wuhan Univ. J. Nat. Sci.* **2018**, *23*, 93–102. [[CrossRef](#)]
25. Maletic, J.I.; Marcus, A. Supporting program comprehension using semantic and structural information. In Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, ON, Canada, 19 May 2001; IEEE: Piscatway, NJ, USA, 2001; pp. 103–112. [[CrossRef](#)]
26. de Oliveira Barros, M. An analysis of the effects of composite objectives in multiobjective software module clustering. In Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA, 7–11 July 2012; ACM: New York, NY, USA, 2012; pp. 1205–1212. [[CrossRef](#)]
27. Chen, Y.-F. Reverse Engineering. In *Practical Reusable UNIX Software*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 1995; pp. 177–208.
28. Mamaghani, A.S.; Hajizadeh, M. Software modularization using the modified firefly algorithm. In Proceedings of the 2014 8th Malaysian Software Engineering Conference (MySEC), Langkawi, Malaysia, 23–24 September 2014; IEEE: Piscatway, NJ, USA, 2014; pp. 321–324. [[CrossRef](#)]
29. Arasteh, B.; Sadegi, R.; Arasteh, K. ARAZ: A software modules clustering method using the combination of particle swarm optimization and genetic algorithms. *Intell. Decis. Technol.* **2021**, *14*, 449–462. [[CrossRef](#)]
30. Hatami, E.; Arasteh, B. An efficient and stable method to cluster software modules using ant colony optimization algorithm. *J. Supercomput.* **2020**, *76*, 6786–6808. [[CrossRef](#)]
31. Seyyedabbasi, A. WOASCALF: A new hybrid whale optimization algorithm based on sine cosine algorithm and levy flight to solve global optimization problems. *Adv. Eng. Softw.* **2022**, *173*, 103272. [[CrossRef](#)]
32. Arasteh, B.; Fatolahzadeh, A.; Kiani, F. Savalan: Multi objective and homogeneous method for software modules clustering. *J. Softw. Evol. Process* **2022**, *34*, e2408. [[CrossRef](#)]
33. Korn, J.; Chen, Y.-F.; Koutsofios, E. Chava: Reverse engineering and tracking of Java applets. In Proceedings of the Sixth Working Conference on Reverse Engineering (Cat. No.PR00303), Atlanta, GA, USA, 8 October 1999; IEEE: Piscatway, NJ, USA, 1999; pp. 314–325. [[CrossRef](#)]
34. Graphviz. Available online: <https://graphviz.org/> (accessed on 10 January 2023).
35. Savalan. Available online: <http://savalan-smct.com/> (accessed on 30 October 2022).
36. Praditwong, K. Solving software module clustering problem by evolutionary algorithms. In Proceedings of the 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), Nakhonpathom, Thailand, 11–13 May 2011; IEEE: Piscatway, NJ, USA, 2011; pp. 154–159. [[CrossRef](#)]
37. Kumari, A.C.; Srinivas, K.; Gupta, M.P. Software module clustering using a hyper-heuristic based multi-objective genetic algorithm. In Proceedings of the 2013 3rd IEEE International Advance Computing Conference (IACC), Ghaziabad, India, 23 February 2013; IEEE: Piscatway, NJ, USA, 2013; pp. 813–818. [[CrossRef](#)]
38. Amarjeet; Chhabra, J.K. Improving package structure of object-oriented software using multi-objective optimization and weighted class connections. *J. King Saud Univ. Comput. Inf. Sci.* **2017**, *29*, 349–364. [[CrossRef](#)]
39. Amarjeet; Chhabra, J.K. Harmony search based modularization for object-oriented software systems. *Comput. Lang. Syst. Struct.* **2017**, *47*, 153–169. [[CrossRef](#)]
40. Arasteh, B. Clustered design-model generation from a program source code using chaos-based metaheuristic algorithms. *Neural Comput. Appl.* **2022**, 1–23. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.