An Iterated Local Search Strengthened by a Q-learning-based Hyper-heuristic for Software Modularization

Mahjoubeh Tajgardan [1], Ph.D. Candidate, Habib Izadkhah [2*], Associate Professor, Shahriar Lotfi [3], Associate Professor

[1,2,3] Faculty of Mathematics, Statistics, and Computer Science, University of Tabriz, Tabriz, Iran, m.tajgardan@tabrizu.ac.ir, izadkhah@tabrizu.ac.ir, shahriar_lotfi@tabrizu.ac.ir

Abstract:

Software comprehension plays an important role during its improvement and maintenance process. Software modularization is a key activity for recovering the software architecture, which improves software understanding. Since the software modularization problem is NP-hard, meta-heuristics such as evolutionary algorithms (EAs) are usually used to solve it. EAs are problem-dependent, and they also require considerable space and time. Recently, the use of hyper-heuristic approaches growing to obtain more generality. This paper proposes an iterated local search (ILS) strengthened by a Q-learning-based hyper-heuristic for software modularization that overcomes the limitations of EAs. In the proposed algorithm, two main components of ILS, i.e., perturbation and local search components, are intelligently selected using a Q-learning-based hyper-heuristic in each iteration. The performance of the proposed algorithm is evaluated on eleven real-world software systems with small and medium sizes. The results of the experiments demonstrate that the proposed ILS produces modularizations that have higher or equal quality compared to the quality of the modularizations obtained by selected algorithms.

Keywords:

Software modularization; Iterated local search; Hyper-heuristic; Q-learning; Evolutionary algorithms.

* Habib Izadkhah, izadkhah@tabrizu.ac.ir

1. Introduction

As software systems progress, their structure deviates from their original architecture, making software maintenance a significant challenge for developers. When the software designer is unavailable and there is insufficient documentation, this task becomes even more complex. Software modularization, an essential task in software maintenance, extracts well-structured components from source code. A well-structured software system is easier to maintain, improve, and understand.

In software modularization, program artifacts are organized into modules based on their similarities. This means that the artifacts within a module are typically more alike to each other than they are to artifacts in other modules [1]. Graphs are a usual tool used to represent software systems and help to reduce their complexity [2]. One well-established type of graph used for this purpose is the artifact dependency graph (ADG) [1], [3], which provides an abstract view of the software architecture. ADG is frequently used as an input for modularization algorithms. Let ADG = (V, E) represent an artifact dependency graph, where $V= \{v_1, v_2, \ldots, v_n\}$ is the set of n artifacts, and $E \subseteq V\times V=\{(v_i, v_j)|v_i, v_j\in V$ and $i\neq j\}$ is the set of links between artifacts, such as call dependencies, inheritance relationships, or semantic similarities. In software modularization, all source code artifacts must be divided into k non-overlapping modules, denoted as $M_1, M_2, \ldots, M_k$. This means that $M_1\cup M_2\cup \ldots \cup M_k =V$, where $M_i \neq \emptyset$, $M_i\cap M_j = \emptyset$, $i, j = 1, 2, \ldots, k$, and $i \neq j$.

The literature contains numerous algorithms for software modularization. Due to the complexity of the software modularization problem (SMP), researchers tend to favor search-based techniques, such as local search methods and evolutionary algorithms (EAs), for solving this problem.

Several studies, such as [4]–[10], have utilized EAs to address the SMP. EAs achieve near-optimal solutions but have some limitations that prevent their efficiency. The performance of EAs is significantly impacted by various parameters, and determining the optimal values for these parameters can be a time-intensive process. Furthermore, due to time and space constraints, EAs are not particularly effective for large-scale software systems [11]. Additionally, EAs are problem-specific techniques, which can be a limitation in certain contexts.

Certain studies, such as [9], [12], have employed heuristic techniques like local search methods to obtain good-quality modularization results in a reasonable timeframe. However, such methods are prone to becoming trapped in local optima, resulting in solutions of lower quality when compared to those provided by EAs. To circumvent this issue, meta-heuristic (MH) techniques have been developed, which integrate diversification approaches with heuristics like local search methods to break free from local optima [13]. This integration results in achieving high modularization quality in a more reasonable time than EAs.

This paper utilizes iterated local search (ILS) [14], a meta-heuristic that combines a perturbation technique with the local search method, for software modularization. When introducing the ILS approach, we must address the following issues that need to be resolved:

- Defining the perturbation component is one of the most critical challenges [14]. The design of this component is crucial, as it directly impacts the effectiveness of the ILS approach. The perturbation component must be potent enough to break free from the local optima yet not so potent that it reduces ILS to a simple random restart algorithm [14]. To overcome this limitation, we will employ multiple perturbation techniques from various perspectives instead of relying on a single perturbation method.
- Given the numerous local search and perturbation methods available for SMP, ILS can potentially select from various pairwise combinations of these methods. In manually designing an ILS approach, selecting an appropriate combination of these two components from all possible combinations is a significant challenge. Trial and error is a straightforward approach, but it is time-consuming. Moreover, the optimal configuration of these two components may change during the modularization process. To address this

limitation, we will leverage a reinforcement learning (RL) technique as an intelligent means of selecting the optimal combination.

Certain studies, such as [8], [15]–[23], have used machine learning (ML) techniques to improve search performance. ML techniques are used to extract useful knowledge from the data generated during the search process, which can aid MHs in making better decisions [13]. MHs leverage this knowledge to conduct intelligent searches and enhance their performance. RL [13] is a type of ML technique in which a learner progressively learns from interactions with the environment to select optimal actions that either maximize rewards or minimize risks. In this paper, we utilize an RL algorithm to address the limitations of RL techniques employed in certain studies, as outlined below:

- The RL technique possesses two crucial attributes: trial and error and delayed reward [21]. Despite featuring a reward/penalty scheme, some RL-based algorithms, such as [16], [17], [20], do not adhere to these characteristics. To address this limitation, we will utilize an RL technique that takes both features into account.
- In some existing RL-based algorithms, such as [16], [17], the set of states defined does not encompass all potential states that may arise during the search process. Additionally, the defined states possess varying parameters, and setting these parameters correctly can be a time-intensive task. To address this limitation, we will define a set of states that includes all possible conditions and is also problem-independent.

Hyper-heuristic (HH) [8], [21] approaches have recently garnered attention for their effectiveness in solving optimization problems such as SMP [8]. HHs function at a higher level of abstraction than MHs, operating on the space of low-level heuristics (LLHs) rather than the solution space. A HH automatically selects (selection HH) or generates (generation HH) a set of LLHs for solving optimization problems. A selection HH is defined as "heuristics to select heuristics". It comprises two levels: the low level consists of a problem representation, evaluation function(s), and a set of problem-specific LLHs, while the high level contains an LLH selection method and a move acceptance method. An LLH selection method chooses an LLH to produce a new solution. Selection HHs can leverage RL for the automatic selection of LLHs. RL-based HHs can be integrated with modular MHs, such as ILS, to intelligently select their components.

The motivation behind this research is to present an ILS algorithm that not only addresses the time and space constraints of EAs and the issue of becoming trapped in local optima associated with local search methods but also enhances the quality of modularization solutions by conducting an intelligent search of the search space with the aid of an RL-based HH.

This paper introduces an ILS algorithm strengthened by a Q-learning-based HH for software modularization. The key feature of the proposed algorithm is its utilization of a Q-learning-based HH to intelligently select ILS components, including perturbation and local search components. Additionally, we present an informative perturbation method for SMP.

The experimental results conducted on eleven small and medium-scale software systems demonstrate that the proposed algorithm generates modularizations of superior or equivalent quality to the compared algorithms while requiring less execution time.

The primary objective of this paper is to improve both modularization quality (MQ) and running time. The key contributions of this paper are outlined below:

- Introducing a general ILS approach strengthened by a Q-learning-based HH that can also be employed for other combinatorial optimization problems in addition to SMP;
- Presenting a fast modularization algorithm that concurrently improves MQ;
- Using a Q-learning-based HH to intelligently choose the perturbation and local search components of ILS;
- Introducing an informative perturbation method for SMP.

The remainder of this paper is structured as follows. Section 2 provides an overview of related modularization algorithms. Q-learning is examined in section 3. The proposed algorithm is outlined in section 4. Section 5 presents the performance evaluation. Section 6 concludes the paper and highlights future research directions.

2. Related works

Within the realm of literature, various algorithms exist for solving SMP. Typically, these algorithms can be grouped into two distinct categories: hierarchical and non-hierarchical. In the following, we will address a few of these methodologies.

2.1. Hierarchical methods

This subsection presents several popular agglomerative hierarchical modularization methods. In agglomerative hierarchical algorithms [2], [24], each artifact is initially placed in a separate module. Then, at each stage, two artifacts with a higher degree of similarity are merged, and this process is repeated until all artifacts are contained within a single module. The similarity between two artifacts in these algorithms is evaluated using similarity measures [1]. Although hierarchical algorithms achieve solutions in a reasonable amount of time, their modularization quality is often suboptimal due to the use of local similarity measures [1]. In the following section, we will discuss some of the existing hierarchical modularization methods. Table (1) summarizes these methods, along with their features, including the similarity metric used and its type.

There are several classic hierarchical methods, including Single linkage (SL) [2], complete linkage (CL) [2], average linkage (AL) [2], and weighted average linkage (WAL) [2].

Maqbool and Babri proposed two software modularization algorithms, the Combined Algorithm (CA) [25] and Weighted Combined Algorithm (WCA) [26]. WCA is a well-known hierarchical modularization algorithm that has two variations, WCA-UE and WCA-UENM, which employ Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM), respectively.

Andritsos and Tzerpos [27] introduced another popular hierarchical modularization algorithm called scaLable InforMation BOttleneck (LIMBO). LIMBO utilizes information theory and entropy principles to achieve software modularization.

Naseem et al. [28] proposed the cooperative clustering technique (CCT) for software modularization. CCT employs multiple similarity measures that collaborate throughout the hierarchical modularization procedure.

Table (1): Some hierarchical algorithms

Method	Similarity metric	Type of metric
SL	Jaccard	Local
CL	Jaccard	Local
AL	Jaccard	Local
WAL	Jaccard	Local
CA	Jaccard	Local
WCA	Ellenberg	Local
CCT	Jaccard-NM and Unbiased Ellenberg-NM	Local
LIMBO	Entropy	Local

2.2. Non-hierarchical methods

As SMP is an NP-hard problem, search-based algorithms are commonly employed to solve it. In search-based modularization algorithms, SMP is treated as a search problem [11]. An objective function guides the modularization process in these algorithms. TurboMQ and BasicMQ [1], [2], [5], [6], [8], [10], [11], [15], [29], [30] are two well-known metrics utilized to assess modularization quality. Despite achieving near-optimal modularization, search-based algorithms have limitations in terms of running time and search space when applied to large-scale software systems. In the following, we will introduce various existing search-based modularization algorithms, which possess distinct features, such as global search (GS), local search (LS), single-objective (SO), multi-objective (MO), structured-based (S) methods, and non-structured-based (Non-S) methods. Table (2) summarizes these methods and their features.

Mitchell [9] introduced a single-objective algorithm called Bunch for software modularization that employs a genetic algorithm, Next Ascent Hill Climbing (NAHC) algorithm, and Steepest Ascent Hill Climbing (SAHC) algorithm. Bunch utilizes real-valued encoding. However, the efficiency of this algorithm diminishes in large-scale software systems due to the vast search space and significant presence of duplicate solutions.

Parsa et al. [4] proposed a single-objective genetic algorithm called DAGC for software modularization, which significantly reduces the search space compared to Bunch. DAGC utilizes permutation-based encoding.

Praditwong et al. [6] introduced two multi-objective genetic algorithms, ECA and MCA. Both algorithms have five objectives that are similar to each other except for one case.

Huang et al. [7] proposed an objective function called MS for software modularization that takes into account global modules and edge directions between two modules. They also utilized three algorithms, namely the hill-climbing algorithm (HC-SMCP), genetic algorithm (GA-SMCP), and multi-agent evolutionary algorithm (MAEA), to optimize the proposed objective function.

Jalali et al. [10] introduced an objective function that considers structural and non-structural properties for software modularization. They also proposed three algorithms, namely the genetic algorithm, hill-climbing algorithm, and estimation of distribution algorithm, to optimize their objective function.

Kargar et al. [12] presented SHC, a hill-climbing algorithm that employs a semantic dependency graph to achieve software architecture in a programming language-independent manner. The search process in SHC is guided by the TurboMQ quality function.

Prajapati et al. [31] introduced a multi-dimensional information-driven many-objective search-based algorithm for solving SMP. Their algorithm optimizes various versions of coupling and cohesion metrics, including structural-based, lexical-based, and changed-history-based, simultaneously using a tailored many-objective artificial bee colony (MaABC) to produce a modularization.

Arasteh et al. [32] proposed SCSO, a discretized sand cat swarm optimization method, for solving SMP. The modified SCSO aims to identify high-quality regions in the search space by learning the correlations between decision factors. At each iteration of the algorithm, the search space is sampled based on a probability distribution.

Kumari and Srinivas [8] introduced MHypEA, a Multi-objective Hyper-heuristic Evolutionary Algorithm, for addressing SMP. This genetic algorithm utilizes a hyper-heuristic approach to select genetic operators, such as selection, crossover, and mutation, based on reinforcement learning coupled with roulette-wheel selection.

In addition to search-based algorithms, there are other non-hierarchical algorithms for software modularization. In the following, we introduce some of these methods.

Pourasghar et al. [1] introduced a modularization algorithm called GMA that utilizes the depth of relationships to compute the similarity between artifacts. They also introduced seven new metrics to assess the quality of modularization.

Teymourian et al. [11] proposed a fast clustering algorithm called FCA for software modularization. FCA performs some operations on the dependency matrix and extracts other matrices based on a set of features. These matrices are used during the software modularization process.

Tzerpos and Holt [33] introduced a pattern-based algorithm called ACDC for software modularization, which utilizes multiple patterns to modularize program artifacts. Previous research has demonstrated that ACDC consistently outperforms other algorithms.

3. Q-learning technique

Q-learning [15], [21]–[23] is a commonly used reinforcement learning algorithm that requires a set of actions and states to be defined. A Q-value, which represents the total cumulative reward, is assigned to each state-action pair and is calculated using Equation (1) (Q-function). Suppose S= [s_1, s_2, s_3, ..., s_n] and A= [a_1, a_2, a_3, ..., a_m] denote the set of possible states and selectable actions, respectively. The Q-value at time t, Q(s_t, a_t), is calculated by Equation (1), where r_{t+1} is the immediate reinforcement signal, and $\alpha \in [0,1]$ and $\gamma \in [0,1]$ are the learning rate and discount factor, respectively. The Q-values are stored in a Q-table.

$$Q_{t+1}(s_t,a_t)= Q(s_t,a_t)+\alpha[r_{t+1}+\gamma *\max_a Q(s_{t+1}, a)-Q(s_t, a_t)] \quad (1)$$

4. The proposed algorithm

This section presents the proposed algorithm for software modularization. The algorithm implements an ILS procedure that utilizes a Q-learning-based hyper-heuristic to select perturbation and local search components at each iteration. Algorithm 1 provides a high-level pseudo-code of the proposed algorithm. As shown in Algorithm 1, the proposed approach differs from formal ILS procedures in that the

Table (2): Some search-based methods

Method	Type	SO / MO	LS / GS	S / non-S features	Encoding type	Learning-based	Main disadvantage
Bunch	GA	SO	GS	S	real-valued	No	Time and space limitations
NAHC	HC	SO	LS	S	real-valued	No	Trapping in local optima
SAHC	HC	SO	LS	S	real-valued	No	Trapping in local optima
DAGC	GA	SO	GS	S	permutation-based	No	Time and space limitations
ECA	Two archive GA	MO	GS	S	real-valued	No	Time and space limitations
GA-SMCP	GA	SO	GS	S	real-valued	No	Time and space limitations
EoD	EDA	MO	GS	S, non-S	real-valued	Yes	Time and space limitations
SHC	HC	SO	LS	non-S	real-valued	No	Trapping in local optima
Many-objective approach	Artificial Bee Colony	MO	LS	S, non-S	real-valued	No	Trapping in local optima
Modified SCSO	Sand Cat Swarm Optimization Algorithm	SO	GS	S	floating-point value	No	Time and space limitations
MHypEA	Hyper-heuristic-based GA	MO	GS	S	real-valued	Yes	Time and space limitations

Algorithm 1. The high-level pseudo-code of the proposed algorithm

```
Ensure: Improved solution GlobalBest
1: Q-table = Initialize_Q-table( )
2: $Sol_{initial}$ = Generate_initial_solution( )
3: $Sol_{current}$ = NAHC-local_search($Sol_{initial}$)
4: GlobalBest = $Sol_{current}$
5: $S_t$ = Specify_initial_state($f_{current}$, avg($f_{current}$))
6: while (the stopping condition is not reached) do
7:    Selected_components = HH(Q-table, $S_t$)
8:    $Sol_p$ = Selected_ perturbation_method($Sol_{current}$)
9:    $Sol_{new}$ = Selected_local_search_method($Sol_p$)
10: Update_O-table( )
11: Move_acceptance _ method($Sol_{new}$, $Sol_{current}$ )
12: $S_{t+1}$ = Specify_next_state($f_{current}$, avg($f_{current}$))
13: $S_t$ = $S_{t+1}$
14: Update_Global_best($Sol_{new}$, GlobalBest)
15:end while
```

perturbation and local search components are automatically selected using a hyper-heuristic.

In Algorithm 1, the Q-learning-based hyper-heuristic selects the best action, which comprises a perturbation method and a local search method with the highest Q-value, at each iteration. The selected methods are then implemented, starting from the last accepted local optimum solution. Following the execution, the Q-value for the action is updated based on its performance. A move acceptance method determines whether the new solution, i.e., the new local optimum, is accepted. The next state is determined based on the normalized fitness of the last accepted local optimum solution, and subsequently, the state and global best solution are updated. In the following, the details of the proposed algorithm are addressed. Then, the next iteration of ILS commences by selecting an appropriate action once again. This process of iteration and action selection continues until the stopping condition is satisfied.

4.1. Encoding type

Modularization solutions are represented using real-valued encoding, proposed by [9]. In this encoding, the maximum number of modules is *n*, where *n* is the number of nodes.

4.2. Objective function

Cohesion [34] and coupling [34] are widely recognized metrics in software engineering that describe the relationships between artifacts within the same module and between artifacts across different modules, respectively [2]. A software system is well-designed if it has high cohesion and low coupling [2], [35]. This paper measures the modularization quality (MQ) during the search process using the TurboMQ metric [2], which aims to achieve low coupling and high cohesion. To compute MQ, the module factor (MF) is first calculated for all modules using Equation (2), where μ_i and ε_{ij} represent the internal connections within module *i* and the external connections between modules *i* and *j*, respectively. Finally, MQ is computed using Equation (3), where k denotes the number of modules.

$$\mathrm{MF_i} = \frac{2 * \mu_i}{2 * \mu_i + \varepsilon_{ij}} \qquad (2)$$

$$\mathrm{MQ} = \sum_{i=1}^{k} \mathrm{MF_i} \qquad (3)$$

4.3. Initial solution

The initial solution is generated using a hierarchical method called single linkage [2].

4.4. Perturbation methods

Three perturbation methods are employed to perturb the local optimum solutions.

The first perturbation method involves randomly selecting a node from solution *s* and moving it into either a distinct module or an existing module.

The second perturbation method involves performing multiple perturbation moves to transform a local optimum into a perturbed solution. Algorithm 2 provides the pseudo-code for this method. As demonstrated in the algorithm, an integer *t* is randomly generated between 1 and *n*/2, where *n* is the number of nodes, to represent the number of iterations. In each iteration, two nodes from different modules are randomly selected, and their module numbers are swapped. Each pair of nodes can only be selected for swapping once.

The third perturbation method is informative and exploits information about nodes and their neighbors. Algorithm 3 outlines the pseudo-code for this method. It uses the notions of vertex cohesion and vertex coupling discussed in [36] to purposefully identify nodes whose module numbers should be modified. Assume that s is the current solution, k represents the number of modules in s, and TurboMQ is the fitness of s. This method first verifies the relation vertex cohesion-vertex coupling<TurboMQ/k for all nodes in s.

Algorithm 2. The pseudo-code of the second perturbation method

```
Ensure: Perturbated solution s′
1: s′ =  current_solution
2: t = generate an integer number between 1 to
        n/2 randomly (n: the number of nodes)
3: i = 1
4: while (i<=t) do
5:   x = generate an integer between 1 to n
6:   y = generate an integer between 1 to n
7:   s′ = swap(s′ (x), s′ (y))
8:   i = i+1
9: end while
```

If node *i* satisfies this relation, its module number is randomly changed to the module number of one of its neighbors in *s*. If node *i* and all of its neighbors have the same module number in *s*, the module number of node *i* is randomly changed to one of the module numbers in *s*. If none of the nodes meet this relation, a node is randomly chosen.

Algorithm 3. The pseudo-code of the third perturbation method

```
Ensure: Perturbated solution s′
1: s = current_solution
2: s′ = s
3: k = number of modules in s
4: TurboMQ = fitness of s ($f_s$)
5: i = 1
6: while (i<=n   n: the number of nodes) do
7:   compute vertex cohesion for node i of the s
8:   compute vertex coupling for node i of the s
9:   if (vertexcohesion-vertexcoupling<TurboMQ/k)
10:     s′(i) ← choose from the s, the module number
        of one of  the dissimilar-module neighbors of
        node i, randomly
11: end if
12:   i = i+1
13:end while
```

4.5. Local search methods

Two versions of the hill climbing algorithm, called NAHC [9] and SAHC [9], serve as the local search components.

4.6. Q-learning-based hyper-heuristic

The proposed ILS employs a hyper-heuristic to automatically select a pair containing a perturbation method and a local search method at each iteration. This hyper-heuristic utilizes Q-learning as the LLH selection method, thus requiring the definition of a set of actions and states.

The proposed algorithm treats a pair having a perturbation method and a local search method as an action. With three perturbation methods and two local search methods provided in this paper, there are six possible actions to choose from.

In Q-learning, the state represents the environmental condition for deciding an action. In the proposed algorithm, the state specifies the last accepted local optimum solution, i.e., the first solution of the next iteration. This paper categorizes the state space into three distinct categories based on fitness value. The fitness value is normalized using Equation (4), where Avg($f_{local\text{-}optimum}$) represents the average fitness value of local optimum solutions from the first to the current iteration. The state space is partitioned into three aggregate states: *normal(f)*∈ [0,0.67), [0.67,1.33), and [1.33,∞).

$$\text{normal(f)}=\frac{\text{f}}{\text{Avg}(f_{local\text{-}optimum})} \quad (4)$$

In this paper, the parameters of the Q-function (Equation (1)) are determined as follows:

- The reinforcement signal, *r*, represents the reward or penalty and is specified by the user. The selected action, comprising a perturbation method and a local search method, is implemented. If the global best solution improves following the execution of the chosen action, the state-action pair is rewarded with *r*=1. Conversely, if the global best solution doesn't improve, the state-action pair is penalized with *r*=-1.
- The discount factor, γ, determines the impact of future rewards and is recommended to be set at 0.8 [21], [22].
- The learning rate, α, determines the ratio of accepting the newly learned information. In this paper, α is dynamically adjusted using Equation (5) [21], where $t_{current}$ denotes the current time of the algorithm and t_{max} represents the total execution time of the algorithm.

$$\alpha_t=1-\left(0.9*\frac{t_{current}}{t_{max}}\right) \quad (5)$$

The Q-table is initialized with zero for all its elements. During each iteration, the action with the highest Q-value is chosen for implementation.

The All Moves method is utilized as the move acceptance method [21]. This method always accepts the newly generated local optimum solution.

4.7. The stopping condition

The algorithm is designed to terminate after 50 seconds of execution. However, if the global best solution fails to improve for 30 consecutive iterations, the algorithm terminates before reaching the maximum time limit.

4.8. A numerical example

We provide a numerical example for a better understanding of the proposed algorithm. Given three perturbation methods and two local search methods, there are six possible actions to choose from, as delineated in Table (3).

Table (3): The selectable actions of the example

A_1	Perturbation method 1 + NAHC
A_2	Perturbation method 2 + NAHC
A_3	Perturbation method 3 + NAHC
A_4	Perturbation method 1 + SAHC
A_5	Perturbation method 2 + SAHC
A_6	Perturbation method 3 + SAHC

The Q-table is initialized to zero, as demonstrated in Table (4). Assuming that the fitness value of the first local optimum solution, $f_{current}$, is 0.5, norm($f_{current}$) is 1, and the current state (s_t) is the *norm(f)∈ [0.67,1.33)*. At this stage, the fitness value of the global best solution is also 0.5.

Table (4): The initialization of the Q-table

	A_1	A_2	A_3	A_4	A_5	A_6
norm(f)∈ [0,0.67)	0	0	0	0	0	0
norm(f)∈ [0.67,1.33)	0	0	0	0	0	0
norm(f)∈ [1.33,∞)	0	0	0	0	0	0

Initially, the RL-based HH approach selects an action with the highest Q-value to determine the components of ILS. Assume that A_1, i.e., Perturbation method 1 + NAHC, is chosen for this iteration. The selected components are executed, resulting in the generation of a new local optimum solution (Sol_{new}) with a fitness value of 1.5. Following this, the Q-table is updated. Assuming that α=0.9 (α is specified by Equation (5) in the real execution) and γ=0.8, the Q-value of ($norm(f)\in$ [0.67,1.33), A_1) is computed using Equation (1), where r_{t+1}=1 since the execution of A_1 results in a solution with a better fitness value than the global best solution. Table (5) displays the updated Q-table.

Table (5): The Q-table after one iteration of ILS

	A_1	A_2	A_3	A_4	A_5	A_6
norm(f)∈ [0,0.67)	0	0	0	0	0	0
norm(f)∈ [0.67,1.33)	0.9	0	0	0	0	0
norm(f)∈ [1.33,∞)	0	0	0	0	0	0

Subsequently, the newly generated solution (Sol_{new}) is adopted as the current solution ($Sol_{current}$), and the next state (s_{t+1}) is specified using the current solution's fitness value, $f_{current}$, and the average fitness values of local optimum solutions obtained from the initial iteration to the current one as indicated by Equation (4): $Avg(f_{current})$ = (0.5+2.5)/2=1.5, $norm(f)$ = 2.5/1.5=1.667. Therefore, s_{t+1} is the $norm\ (f)\in[1.33,\infty)$.

Lastly, the next state, i.e., s_{t+1}, is considered as the current state, i.e., s_t ($s_t = norm(f)\in[1.33,\infty)$). The fitness value of the global best solution is updated to 2.5. Subsequently, the next iteration of ILS commences by selecting an appropriate action once again. This process of iteration and action selection continues until the stopping condition is satisfied.

5. Performance evaluation

This section assesses the performance of the proposed algorithm. Initially, the experimental setup is outlined, followed by a description of the experimental results.

5.1. Experimental setup

This subsection outlines the necessary setup for conducting the experiments.

5.1.1. Software system

Eleven small and medium-sized applications have been chosen for evaluation and comparison purposes. Table (6) illustrates the characteristics of these software systems.

5.1.2. Expert's decomposition

A modularization algorithm is considered reliable when its outcomes closely align with expert modularization. To evaluate the modularization algorithms in this paper, the "mtunis" application, for which expert decomposition is available, is employed.

Table (6): Description of selected software systems

Applications	#Files	#Links
mtunis	20	57
compiler	13	32
nos	16	52
boxer	18	29
spdb	21	33
ispell	24	103
ciald	26	64
rcs	29	163
star	36	89
bison	37	179
cia	38	87

5.1.3. Assessment of results

There exist several algorithms for software modularization. However, as there is no universally accepted definition of optimal modularization features, external and internal criteria are employed to evaluate modularization quality and compare the outcomes of different algorithms [2]. External criteria evaluate the proximity of the obtained modularization to expert decomposition, which is carried out by an individual who has decomposed the software system. A modularization algorithm is considered reliable when its outcomes are close to expert modularization [1]. Conversely, internal criteria assess the accuracy with which modules are separated based on various metrics.

This study employs TurboMQ [2] (Equation (3)), an internal criterion, to assess the performance of modularization methods. Additionally, the external criteria MoJo [2], Edge MoJo [2], and F_m [2] are used to measure the proximity of the obtained modularization to the expert's decomposition. Lower values of MoJo and Edge MoJo are indicative of better performance, while higher values of Fm and TurboMQ suggest superior outcomes.

5.1.4. Algorithmic parameters

The parameter setting suggested in [11] is used for experiments.

5.1.5. Research questions (RQs)

We answer the following research questions to evaluate the proposed algorithm.

1) Does the proposed algorithm achieve modularization close to the expert decomposition?
2) Does the proposed algorithm produce modularizations with a higher MQ (TurboMQ) compared to hierarchical and non-hierarchical algorithms?
3) Is the proposed algorithm stable?
4) Does the proposed algorithm converge to the answer (solution)?
5) Does the execution time of the proposed algorithm less than EAs such as Bunch and DAGC?
6) Does the proposed informative perturbation method is beneficial and leads to improve search performance?
7) Does the proposed ILS algorithm have better performance than classic (formal) ILS?

5.2. Experimental results

This section describes the experimental results. For comparison with the proposed algorithm, we selected several methods discussed in section 2, including Complete Linkage (CL) [2], Single Linkage (SL) [2], Average Linkage (AL) [2], WCA-UE [2], fast clustering algorithm (FCA) [11], Bunch [9], DAGC [4], NAHC [9], and SAHC [9]. Additionally, the k-means algorithm [2] and ACDC are chosen for comparison. We address the research questions below.

5.2.1. RQ 1

To address RQ1, the MoJo, Edge MoJo, and Fm measures are calculated for the modularization outcomes produced by each algorithm. These external criteria are used to determine the proximity of the obtained modularization to the expert decomposition. Table (7) presents the values of these criteria for the compared search-based algorithms on the “mtunis” application. The most favorable outcomes in this table are highlighted in bold. Regarding MoJo, the proposed algorithm performs better than DAGC and is equivalent to other methods. In terms of Edge MoJo and Fm, it is equivalent to Bunch and superior to other methods. The proposed algorithm exhibits similar behavior to Bunch, which is a well-known method for software modularization. These findings confirm the reliability of the proposed algorithm.

Table (7): The external criteria values

Algorithms	MoJo	Edge MoJo	F_m
Bunch	**5**	**7.47**	**0.57**
DAGC	7	10.33	0.48
NAHC	**5**	13.14	0.53
SAHC	**5**	10.81	0.55
Proposed algorithm	**5**	**7.47**	**0.57**

5.2.2. RQ 2

To address RQ2, the TurboMQ (Equation (3)) value is calculated for the modularization outcomes produced by each algorithm on ten applications. Table (8) presents a comparison of the proposed algorithm with some hierarchical methods, FCA, ACDC, and k-means, while Table (9) demonstrates its comparison with some search-based algorithms. The most favorable outcomes in these tables are highlighted in bold. As shown in Table (8), the proposed algorithm outperforms all compared hierarchical and non-hierarchical methods. In Table (9), the proposed algorithm performs better than NAHC, SAHC, and DAGC in most cases and is superior to Bunch in half of the cases while being equivalent to Bunch in the remaining cases. Therefore, the proposed algorithm exhibits good performance in terms of TurboMQ.

5.2.3. RQ 3

To address RQ3, ten applications are selected, and the proposed algorithm is executed 30 times for each case. If the algorithm's results are close enough to each other, it is stable.

To analyze the stability of the outcomes, the t-test statistical technique is employed. For this, the obtained results are divided into two groups of equal size, named G1 and G2. Subsequently, descriptive and inferential statistics are derived from G1 and G2. Table (10) displays the outcomes, where the first three columns present descriptive statistics, and the last two columns depict the output of the inferential statistics. The descriptive statistics consist of the mean, standard deviation, and standard error between the mean of the two groups.

Table (8): Comparison of the proposed algorithm with some hierarchical and non-hierarchical methods in terms of TurboMQ

Applications	WCA-UE	AL	CL	SL	FCA	ACDC	k-means	Proposed algorithm
Compiler	0.836	0.527	0.527	0.933	1.220	1.000	0.850	**1.506**
Boxer	1.343	0.964	0.983	0.964	3.020	2.820	0.790	**3.101**
Ispell	1.489	1.739	1.639	0.995	1.970	1.750	1.200	**2.190**
Bison	0.994	0.994	0.994	0.994	2.250	1.000	1.000	**2.684**
Cia	0.997	0.997	0.997	0.997	2.049	1.860	1.780	**2.790**
Ciald	0.984	0.487	1.093	0.984	1.720	1.700	0.780	**2.851**
Nos	0.969	0.990	0.990	0.990	1.080	1.000	0.980	**1.636**
Rcs	0.977	0.990	1.018	1.018	1.810	1.000	1.260	**2.201**
Spdb	0.933	0.933	0.933	0.933	5.000	5.000	1.150	**5.741**
Star	1.388	0.989	0.805	0.989	3.048	2.090	0.810	**3.832**

Table (9): Comparison of the proposed algorithm with some search-based methods in terms of TurboMQ

Applications	NAHC	SAHC	Bunch	DAGC	Proposed algorithm
Compiler	1.442	1.465	**1.506**	**1.506**	**1.506**
Boxer	2.931	**3.101**	**3.101**	**3.101**	**3.101**
Ispell	2.013	2.043	2.177	1.997	**2.190**
Bison	2.565	2.632	2.606	1.763	**2.684**
Cia	2.729	2.670	2.706	1.833	**2.790**
Ciald	2.721	2.838	**2.851**	2.463	**2.851**
Nos	1.547	1.550	**1.636**	1.606	**1.636**
Rcs	2.066	2.103	2.175	1.894	**2.201**
Spdb	**5.741**	**5.741**	**5.741**	5.314	**5.741**
Star	3.536	3.798	3.809	2.831	**3.832**

Table (10): t-test for analyzing the stability of the proposed algorithm

Case Study	Descriptive Statistics						Inferential Statistics			
	Mean		Standard Deviation		Standard Error between Mean		levene's test		t-test	
	G1	G2	G1	G2	G1	G2	F	Sig.	T	Sig.
Compiler	1.480	1.493	0.035	0.029	0.016	0.013	1.524	0.252	-0.632	0.545
Boxer	3.067	3.033	0.076	0.093	0.034	0.042	1.524	0.252	0.632	0.545
Ispell	2.155	2.161	0.079	0.066	0.035	0.029	0.121	0.737	-0.130	0.899
Bison	2.660	2.674	0.053	0.023	0.024	0.010	1.893	0.206	-0.516	0.620
Cia	2.778	2.766	0.027	0.054	0.012	0.024	1.366	0.276	0.438	0.673
Ciald	2.848	2.846	0.006	0.007	0.003	0.003	1.524	0.252	0.632	0.545
Nos	1.618	1.619	0.040	0.038	0.018	0.017	0.004	0.950	0.024	0.981
Rcs	2.174	2.154	0.060	0.065	0.027	0.029	0.491	0.503	0.493	0.635
Spdb	5.656	5.741	0.191	0.000	0.085	0.000	7.111	0.029	-1.000	0.347
Star	3.825	3.766	0.015	0.129	0.007	0.058	5.263	0.051	1.016	0.339

Levene's test is utilized as an inferential statistic to evaluate the equality of variances for a variable calculated for two groups. If the p-value (indicated in the “Sig.” column in the table) is greater than a certain significance level (0.05 in our experiments), the null hypothesis of equal variances cannot be rejected. The last columns of Table (10) refer to the outcomes of an independent two-sample t-test with equal sample sizes and equal variances (based on the results of Levene's test) on two randomly separated groups of proposed algorithm outcomes. All the p-values are greater than 0.05, indicating that the proposed algorithm exhibits stability.

5.2.4. RQ 4

To address RQ4, the convergence of the proposed algorithm is assessed. To accomplish this, three software systems are chosen as samples. Figures (1-3) depict the convergence plots of our algorithm for the “Cia,” “Ispell,” and “Nos” applications, respectively. In these figures, the line denotes the best solution generated by the algorithm up to the current iteration. These figures substantiate that the proposed algorithm converges to the solution after several iterations.

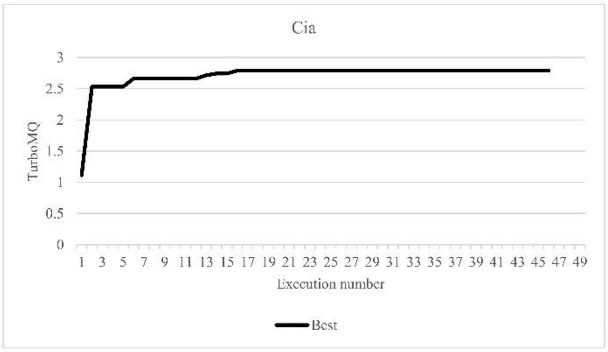

Figure (1): Convergence diagram of the proposed algorithm for the "Cia" application

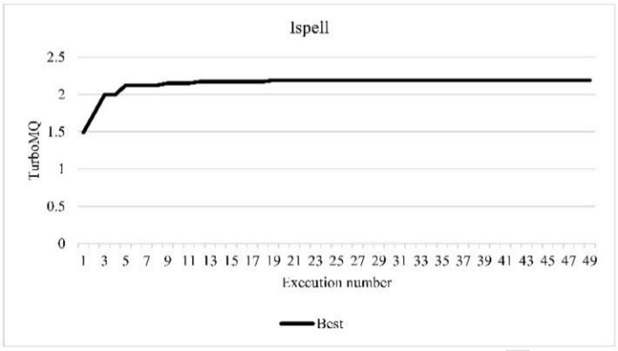

Figure (2): Convergence diagram of the proposed algorithm for the "Ispell" application

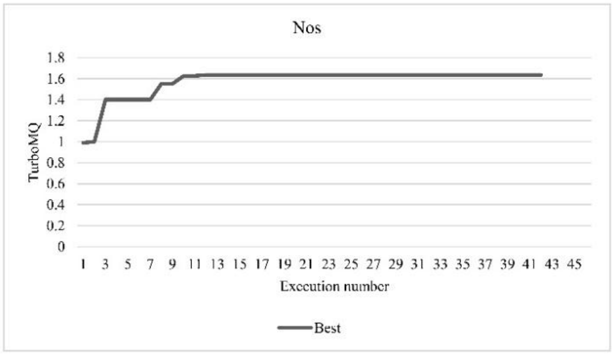

Figure (3): Convergence diagram of the proposed algorithm for the "Nos" application

5.2.5. RQ 5

To address RQ5, we compared the execution time of the proposed algorithm with Bunch and DAGC, two well-known representatives of evolutionary algorithms. Figures (4) and (5) display the results for selected applications. It is worth noting that the execution times of Bunch and DAGC have been obtained from [37]. These figures illustrate that the execution time of the proposed algorithm is significantly lower than that of Bunch and DAGC.

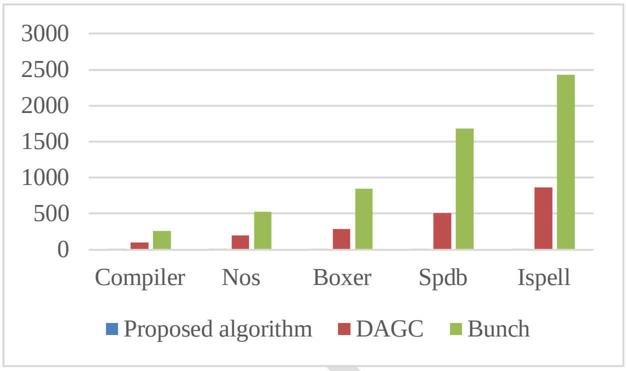

Figure (4): Execution time of the algorithms on five applications (second)

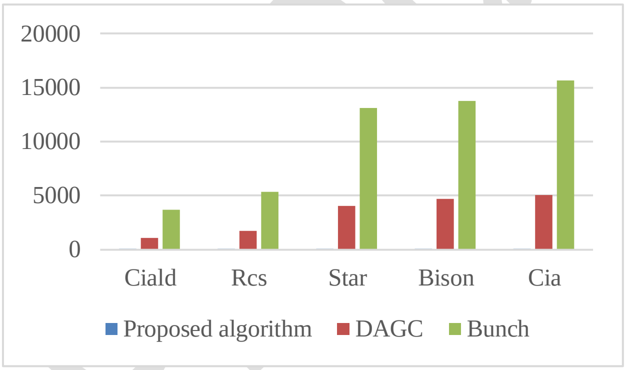

Figure (5): Execution time of the algorithms on five other applications (second)

5.2.6. RQ 6

To address RQ6, we experimented to assess the effectiveness of the proposed perturbation method. For this, we compared the proposed algorithm and its variant with the proposed informative perturbation method disabled in terms of TurboMQ. Both algorithms were executed under the same conditions. The outcomes are presented in Table (11), where the bold values indicate the best outcomes. This table reveals that in most cases (70 percent of cases), our algorithm, without the proposed informative perturbation method, failed to find the best solution. These findings demonstrate that the proposed informative perturbation method enhances the search performance of the proposed algorithm.

Table (11): Comparison of the proposed algorithm and its variant without the proposed informative perturbation method

Applications	Proposed algorithm without informative perturbation method	Proposed algorithm
Compiler	**1.506**	**1.506**
Boxer	2.931	**3.101**
Ispell	**2.190**	**2.190**
Bison	2.627	**2.684**
Cia	2.729	**2.790**
Ciald	2.539	**2.851**
Nos	1.628	**1.636**
Rcs	2.175	**2.201**
Spdb	**5.741**	**5.741**
Star	3.481	**3.832**

5.2.7. RQ 7

To address RQ7, we experimented with comparing the performance of the proposed ILS algorithm with classic ILS in terms of TurboMQ. To accomplish this, we executed classic ILS by considering different combinations of the local search method and perturbation method. Since this paper presents three perturbation methods and two local search methods, there are a total of six selectable combinations. The outcomes are demonstrated in Table (12), where per1, per2, and per3 denote the first, second, and third perturbation methods defined in this paper, respectively. In this table, the bold values indicate the best outcomes. The results reveal that, unlike the proposed ILS, none of the implemented classic ILS algorithms achieve the best outcome in all cases. Therefore, the proposed ILS algorithm exhibits superior performance compared to classic ILS.

6. Conclusion

Modularization is a technique in reverse engineering that aids in understanding software during the software maintenance process. In this paper, we introduced a modified iterated local search (ILS) for software modularization. The proposed ILS employs a Q-learning-based hyper-heuristic to automatically select the perturbation method and local search method in each iteration. To apply Q-learning, we utilized a fitness-based state categorization that is independent of the problem. We also presented an informative perturbation method for the software modularization problem. The outcomes of experiments and comparisons with other methods on eleven small and medium-scale applications revealed that the proposed algorithm generates better modularizations in most cases. Furthermore, the proposed algorithm is general and can be easily adapted to other optimization problems.

In the following, some improvements of this paper that can be performed in future research are listed.

1. Testing the proposed algorithm on large-scale software systems such as Mozilla Firefox.
2. Using other metrics to evaluate the modularization quality.
3. Employing the proposed algorithm to other optimization problems.
4. Employing reinforcement learning-based hyper-heuristic in other modular meta-heuristic to automatically select their components.

Table (12): Comparison of the proposed ILS algorithm with the classic ILS algorithm

Applications	ILS (per1+SAHC)	ILS (per2+SAHC)	ILS (per3+SAHC)	ILS (per1+NAHC)	ILS (per2+NAHC)	ILS (per3+NAHC)	Proposed ILS algorithm
Compiler	1.497	1.273	**1.506**	**1.506**	1.273	**1.506**	**1.506**
Boxer	**3.101**	1.781	**3.101**	**3.101**	1.781	**3.101**	**3.101**
Ispell	2.120	1.739	**2.190**	2.174	1.934	2.176	**2.190**
Bison	2.601	2.345	2.675	2.590	2.504	2.664	**2.684**
Cia	2.636	2.586	2.768	2.779	2.634	2.787	**2.790**
Ciald	2.731	1.532	**2.851**	**2.851**	1.532	**2.851**	**2.851**
Nos	**1.636**	1.000	**1.636**	1.627	1.000	**1.636**	**1.636**
Rcs	2.166	1.953	2.166	2.157	1.953	2.136	**2.201**
Spdb	**5.741**	2.000	**5.741**	**5.741**	3.000	**5.741**	**5.741**

Star	**3.832**	1.690	**3.832**	3.038	1.690	3.803	**3.832**

References

[1] Pourasghar, B., Izadkhah, H., Isazadeh, A., Lotfi, S., "*A graph-based clustering algorithm for software systems modularization*", Information and Software Technology 133: 106469 (2021). https://doi.org/10.1016/j.infsof.2020.106469

[2] Isazadeh, A., Izadkhah, H., Elgedawy, I., *Source Code Modularization Theory and Techniques*, Springer International Publishing, 2017.

[3] Aghdasifam, M., Izadkhah, H., Isazadeh, A., "*A new metaheuristic-based hierarchical clustering algorithm for software modularization*", Complexity 2020: 1-25 (2020). https://doi.org/10.1155/2020/1794947

[4] Parsa, S., Bushehrian, O., "*A New Encoding Scheme and a Framework to Investigate Genetic Clustering Algorithms*", Journal of Research and Practice in Information Technology 37(1): 127–143 (2005).

[5] Izadkhah, H., Tajgardan, M., "*Information theoretic objective function for genetic software clustering*", Multidisciplinary Digital Publishing Institute Proceedings 46(1): 18 (2019). https://doi.org/10.3390/ecea-5-06681

[6] Harman, M., Yao, X., "*Software module clustering as a multi-objective search problem*", IEEE Transactions on Software Engineering 37(2): 264–282 (2010). https://doi.org/10.1109/TSE.2010.26

[7] Huang, J., Liu, J., "*A similarity-based modularization quality measure for software module clustering problems*", Information Sciences 342: 96–110 (2016). https://doi.org/10.1016/j.ins.2016.01.030

[8] Kumari, A. C., Srinivas, K., "*Hyper-heuristic approach for multi-objective software module clustering*", Journal of Systems and Software 117: 384–401 (2016). https://doi.org/10.1016/j.jss.2016.04.007

[9] Mitchell, B. S., *A heuristic search approach to solving the software clustering problem*, Ph.D. Thesis, Drexel University, 2002.

[10] Sadat Jalali, N., Izadkhah, H., Lotfi, S., "*Multi-objective search-based software modularization: structural and non-structural features*", Soft Computing 23(21): 11141–11165 (2019). https://doi.org/10.1007/s00500-018-3666-z

[11] Teymourian, N., Izadkhah, H., Isazadeh, A., "*A fast clustering algorithm for modularization of large-scale software systems*", IEEE Transactions on Software Engineering 48(4): 1451-1462 (2020). https://doi.org/10.1109/TSE.2020.3022212

[12] Kargar, M., Isazadeh, A., Izadkhah, H., "*Semantic-based software clustering using hill climbing*", International Symposium on Computer Science and Software Engineering Conference (CSSE) 55-60 (2017). 10.1109/CSICSSE.2017.8320117

[13] Karimi-Mamaghan, M., Mohammadi, M., Meyer, P., Karimi-Mamaghan, A. M., Talbi, E. G., "*Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art*", European Journal of Operational Research 296(2): 393-422 (2022). https://doi.org/10.1016/j.ejor.2021.04.032

[14] Chiarandini, M., Stützle, T., "*An application of iterated local search to graph coloring problem*", Proceedings of the computational symposium on graph coloring and its generalizations 112-125 (2002).

[15] Tajgardan, M., Izadkhah, H., Lotfi, S., "*A Reinforcement Learning-based Iterated Local Search for Software Modularization*", 8th Iranian Conference on Signal Processing and Intelligent Systems 1-6 (2022). 10.1109/ICSPIS56952.2022.10043949

[16] Sghir, I., Ben Jaafar, I. B., Ghédira, K., "*A multi-agent based optimization method for combinatorial optimization problems*", International Journal on Artificial Intelligence Tools 27(05):

1850021 (2018). https://doi.org/10.1142/S0218213018500215

[17] Sghir, I., Hao, J. K., Jaafar, I. B., Ghédira, K., “*A multi-agent based optimization method applied to the quadratic assignment problem*”, Expert Systems with Applications 42(23): 9252-9262 (2015). https://doi.org/10.1016/j.eswa.2015.07.070

[18] Asta, S., *Machine learning for improving heuristic optimisation* Doctoral dissertation, University of Nottingham, 2015.

[19] Meignan, D., Koukam, A., Créput, J.-C., “*Coalition-based metaheuristic: a self-adaptive metaheuristic using reinforcement learning and mimetism*”, Journal of Heuristics 16(6): 859–879 (2010). https://doi.org/10.1007/s10732-009-9121-7

[20] Özcan, E., Misir, M., Ochoa, G., Burke, E. K., “*A reinforcement learning: great-deluge hyper-heuristic for examination timetabling*”, Modeling, analysis, and applications in metaheuristic computing: advancements and trends 34-55 (2012). https://doi.org/10.4018/978-1-4666-0270-0.ch003

[21] Choong, S. S., Wong, L. P., Lim, C. P., “*Automatic design of hyper-heuristic based on reinforcement learning*”, Information Sciences 436: 89–107 (2018). https://doi.org/10.1016/j.ins.2018.01.005

[22] Ahmed, B. S., Enoiu, E., Afzal, W., Zamli, K. Z., “An evaluation of Monte Carlo-based hyper-heuristic for interaction testing of industrial embedded software applications”, Soft Computing 24(18): 13929-13954 (2020). https://doi.org/10.1007/s00500-020-04769-z

[23] Cheng, L., Tang, Q., Zhang, L., Yu, C., “*Scheduling flexible manufacturing cell with no-idle flow-lines and job-shop via Q-learning-based genetic algorithm*”, Computers & Industrial Engineering 169 108293 (2022) https://doi.org/10.1016/j.cie.2022.108293

[۲۴] نبی‌لو، مریم، دانش‌پور، نگین، «**ارائه یک الگوریتم خوشه‌بندی برای داده‌های دسته‌ای با ترکیب معیارها**»، مجله محاسبات نرم، جلد ۵، شماره ۱، ص ۱۴–۲۵، دانشگاه کاشان، بهار و تابستان ۱۳۹۵.

[25] Saeed, M., Maqbool, O., Babri, H. A., Hassan, S. Z., Sarwar, S. M., “*Software clustering techniques and the use of combined algorithm*”, Seventh European Conference on Software Maintenance and Reengineering 301-306 (2003). https://doi.org/10.1109/CSMR.2003.1192438

[26] Maqbool, O., Babri, H., “*Hierarchical clustering for software architecture recovery*”, IEEE Transactions on Software Engineering 33(11): 759-780 (2007). https://doi.org/10.1109/TSE.2007.70732

[27] Andritsos, P., Tzerpos, V., “*Information-theoretic software clustering*”, IEEE Transactions on Software Engineering 31(2): 150-165 (2005). https://doi.org/10.1109/TSE.2005.25

[28] Naseem, R., Maqbool, O., Muhammad, S., “*Cooperative clustering for software modularization*”, Journal of Systems and Software 86(8): 2045–2062 (2013). (2013). https://doi.org/10.1016/j.jss.2013.03.080

[29] Tajgardan, M., Izadkhah, H., “*Critical Review of the Bunch: A Well-Known Tool for the Recovery and Maintenance of Software System Structures*”, Critical Review 6(3): 363–367 (2017). DOI10.17148/IJARCCE.2017.6383

[30] Tajgardan, M., Izadkhah, H., “*Software Systems Clustering Using Estimation of Distribution Approach*”, Journal of Applied Computer Science Methods 8(2): 99–113 (2016). http://dx.doi.org/10.1515/jacsm-2016-0007

[31] Prajapati, A., Parashar, A., Rathee, A., “*Multi-dimensional information-driven many-objective software remodularization approach*”, Frontiers of Computer Science 17(3): 173209 (2023). https://doi.org/10.1007/s11704-022-1449-2

[32] 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 15(2): 401 (2023). https://doi.org/10.3390/sym15020401

[33] Tzerpos, V., Holt, R. C., “*Accd: an algorithm for comprehension-driven clustering*”, Proceedings Seventh Working Conference on Reverse Engineering 258-267 (2000). https://doi.org/10.1109/WCRE.2000.891477

[۳۴] رسول‌زادگان، عباس، بصیری، محدثه، «**اندازه‌گیری کمی کیفیت در مهندسی نرم‌افزار سرویس‌گرا: روش‌ها، کاربردها و چالش‌ها**»، مجله محاسبات نرم، جلد ۳، شماره ۱، ص ۲–۱۹، دانشگاه کاشان، بهار و تابستان ۱۳۹۳.

[۳۵] غلامشاهی، شبنم، هاشمی‌نژاد، سید محمدحسین، «**روشی برای تشخیص مؤلفه‌های نرم‌افزاری مبتنی بر الگوریتم ژنتیک مرتب‌سازی نامغلوب**»، مجله محاسبات نرم، جلد ۷، شماره ۲، ص ۴۷–۶۴، دانشگاه کاشان، پاییز و زمستان ۱۳۹۷.

[36] Izadkhah, H., Elgedawy, I., Isazadeh, A., “*E-CDGM: An Evolutionary Call-Dependency Graph Modularization Approach for Software Systems*”, Cybernetics and Information Technologies 16(3) (2016). https://doi.org/10.1515/cait-2016-0035

[37] Mohammadi, S., Izadkhah, H., “*A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code*”, Information and Software Technology 105: 252-256 (2019). https://doi.org/10.1016/j.infsof.2018.09.001