electronics

MDPI

Article

Performance Analysis of Several Intelligent Algorithms for Class Integration Test Order Optimization

Wenning Zhang [1,2,*], Qinglei Zhou [3], Li Guo [2], Dong Zhao [2] and Ximei Gou [2]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
[2] Software College, Zhongyuan University of Technology, Zhengzhou 450000, China
[3] School of Information Engineering, Zhengzhou University, Zhengzhou 450000, China
* Correspondence: zhangwn@zut.edu.cn

Abstract: Integration testing is an essential activity in software testing, especially in object-oriented software development. Determining the sequence of classes to be integrated, i.e., the class integration test order (CITO) problem, is of great importance but computationally challenging. Previous research has shown that meta heuristic algorithms can devise class integration test orders with lower test stubbing complexity, resulting in software testing cost reduction. This study focuses on the comparable performance evaluation of ten commonly used meta heuristic algorithms: genetic algorithm (GA), particle swarm optimization (PSO), cuckoo search algorithm (CS), firefly algorithm (FA), bat algorithm (BA), grey wolf algorithm (GWO), moth flame optimization (MFO), sine cosine algorithm (SCA), salp swarm algorithm (SSA) and Harris hawk optimization (HHO). The objective of this study is to identify the most suited algorithms, narrowing down potential avenues for future researches in the field of search-based class integration test order generation. The standard implementations of these algorithms are employed to generate integration test orders. Additionally, these test orders are evaluated and compared in terms of stubbing complexity, convergence speed, average runtime, and memory consumption. The experimental results suggest that MFO, SSA, GWO and CS are the most suited algorithms. MFO, SSA and GWO exhibit excellent optimization performance in systems where fitness values are heavily impacted by attribute coupling. Meanwhile, MFO, GWO and CS are recommended for systems where the fitness values are strongly influenced by method coupling. BA and FA emerge as the slowest algorithms, while the remaining algorithms exhibit intermediate performance. The performance analysis may be used to select and improve appropriate algorithms for the CITO problem, providing a cornerstone for future scientific research and practical applications.

Keywords: integration testing (IT); object-oriented testing (OOT); meta heuristic algorithms; class integration testing order (CITO); performance analysis

Citation: Zhang, W.; Zhou, Q.; Guo, L.; Zhao, D.; Gou, X. Performance Analysis of Several Intelligent Algorithms for Class Integration Test Order Optimization. *Electronics* **2023**, *12*, 3733. https://doi.org/10.3390/electronics12173733

Academic Editor: Antonio Brogi

Received: 14 August 2023
Revised: 28 August 2023
Accepted: 1 September 2023
Published: 4 September 2023

1. Introduction

Software testing is an important and labor-intensive process carried out to ensure software quality, accounting for about 40–50% of the overall project cost [1]. Effective integration testing helps to verify that the interface implements the desired functions. Traditional integration testing strategies include non-incremental one-time integration methods and bottom-up or top-down incremental integration methods [2]. With the wide application of object-oriented technology and the improvement of requirement complexity, the number of classes in a system increases. Consequently, the intricate control and dependency interrelations among classes are complex and pose challenges in precise delineation. Conducting object-oriented integration testing by conventional approaches becomes impractical. How to decide the order of classes that are developed independently is called the class integration test order problem (CITO). The CITO problem is computationally difficult, making accurate solutions very difficult to obtain [3].

Generally, people are required to construct test stubs for target classes to emulate their necessary functionalities, an important factor that impacts the cost of integration testing.

Although several test tools (such as JUnit) can reduce the test stub cost, the stub construction remains complex and cumbersome with the risk of introducing new defects. Theoretically, CITO is a NP complete problem, rendering the identification of globally optimal class integration test orders impractical. However, Kung et al. proved that the number of test stubs and the corresponding construction complexity are different for various CITO [4]. Reasonable class integration test orders can reduce the test stub cost [5].

In the literature, various approaches and strategies for the CITO problem have been proposed. Most of the approaches are based on graph theory, focusing on breaking class dependency cycles, and determining orders via reverse topological sorting. However, graph-based approaches have difficulties in identifying strongly connected components, especially when a system contains numerous dependency cycles. In the field of software engineering, some popular meta heuristic algorithms such as genetic algorithm, particle swarm optimization algorithm, and firefly algorithm have been used to automate the process of test data generation with the goal of covering target paths [1]. Due to the computational complexity, the CITO problem has become a subject of the search-based software engineering (SBSE) [6]. Researchers have transformed the CITO problem into an optimization problem. More recent approaches for CITO are search-based methods by using meta heuristic algorithms.

In recent years, many new algorithms have emerged because of their simplicity and robustness. A survey conducted by Hussain [7] collected 1222 publications from 1983 to 2016 and revealed around 140 meta heuristic algorithms applied in a diverse range of fields broadly in the area of science and technology, economics, and daily life. The famous No Free Lunch (NFL) theorem states that the performance of all algorithms is equal across all possible optimization problems. To identify the most appropriate algorithms for CITO problem and streamline forthcoming research in the field of search-based class integration test order generation, we conducted a comparative performance analysis.

The remainder of this paper is structured as follows. Section 2 describes the past literature on CITO problem. Section 3 describes the entire research methodology with detailed information of algorithm selection, problem representation and the optimization process. Section 4 analyzes and discusses the experimental results, including stubbing complexity, attribute complexity, method complexity, convergence performance, average runtime, and memory consumption. Section 5 summarizes the internal validity and external validity of the experiment. Section 6 presents the conclusion as well as the theoretical and practical contributions for the study.

2. Literature Review

Object-oriented software engineering is the application of object-oriented method in the field of software engineering. The object relationship graphs and class design models provide a solid working foundation for generating class integration test orders. Several papers have proposed various strategies to derive test orders.

Kung et al. [4] were the first researchers to address the CITO problem. They put forward the graph-based method to minimize the number of test stubs. If there was no cycle in an object relationship graph, reverse topology sorting was used to generate class integration test orders. If there were cycles, the cycles were destroyed by deleting associated edges with the smallest number of test stubs. Based on Kung's work, Tai and Daniels [8] created the primary integration test order in strong inheritance and aggregation dependencies, and then constructed the secondary integration order by deleting edges with the largest weight. Traon et al. [9] measured class relationships based on the leaf edge counts and deleted loops containing nodes with the maximum weights. Briand et al. [10] used Tarjan's algorithm to explore object relationship graph in depth. Based on recursive identification of strong connected components, they calculated the weight of each association dependency, and then deleted the edges with the largest weight to break cycles.

All the aforementioned endeavors employed graphs models to portray class dependencies, subsequently applying graph-based algorithms to break cycles. The objective of

these graph-based approaches is to minimize the number of test stubs, as perceived as a major cost factor in integration testing.

Subsequent studies indicated that the test order with the fewest stubs may not be the best choice [11]. Briand et al. [12] pointed out that the solutions based on graph theory would be difficult or impossible to fully solve CITO problem especially when assuming equal development cost for each stub. Consequently, various meta heuristic algorithms, such as genetic algorithm and particle swarm optimization, were employed to solve the CITO problem, yielding promising performance and efficiency outcomes.

Briand et al. [12] first proposed the search-based method to obtain appropriate test orders. They proposed the concept of stubbing complexity based on attribute coupling and method coupling. Under the constraint of not breaking strong dependencies, they combined the inter class stubbing complexity and genetic algorithm to minimize the stubbing effort and formulate optimal class integration orders. The results on five real-world applications were encouraging and reliable.

Borner et al. [13] presented an approach to measure the test focus of a given integration test order. They selected the fault prone dependencies as the test focus, integrated the test focus in early integration steps, and then applied the simulated annealing algorithm to minimize the effort to simulate not yet integrated components of the system. They pointed out that the disadvantage of simulated annealing and genetic algorithm was the long duration for deriving test orders.

Cabral et al. [6] pointed out the existing graph-based approaches failure to consider various factors or metrics, leading to suboptimal solutions. They proposed a multi-objective optimization representation of CITO problem and implemented the Pareto ant colony algorithm to strike a balance between different metrics.

Based on Cabral's work, Vergilio et al. [14] introduced a multi-objective optimization approach and implemented three different multi-objective optimization algorithms to generate a set of good solutions: Pareto ant colony, multi-objective tabu search and non-dominated sorting genetic algorithm. The experimental results showed that meta heuristic algorithms were suitable for addressing software engineering complex problems, like CITO.

Mariani et al. [15] proposed an offline hyper-heuristic algorithm named GEMOITO to reduce efforts in choosing, implementing, and configuring of search algorithms. Grammatical evolution was used to automatically generate a multi-objective evolutionary algorithm. These generated optional algorithms were distinguished by components and parameters values to solve CITO problem.

Czibula et al. [16] proposed an improved genetic algorithm with stochastic acceptance to optimize class integration test orders. In the proposal, the complexity of stub construction was estimated by assigning weights to various types of dependencies in object relation diagram.

Given the successful application and validation of particle swarm optimization in addressing complex optimization problems within search-based software engineering, Zhang et al. [17] reformulated the CITO problem into a one-dimensional optimization problem and introduced the standard particle swarm optimization algorithm to devise optimal test orders.

Zhang et al. [18] proposed an improved particle swarm optimization algorithm that incorporates an individual's dream ability, thereby improving population diversity, optimization accuracy, and convergence speed. The proposed algorithm could converge slowly and avoid falling into local optimum too early.

Zhang et al. [3,19] assessed test stub cost from two aspects: data coupling and control coupling. Based on the extended object relationship diagram, they calculated testing costs and test profits to set class selection priorities. Furthermore, they proposed a strategy for combining similar classes, reducing class count and thereby greatly simplifying the optimization complexity.

Zhang et al. [20] extended the search-based class integration test order approach to the context of service-oriented architecture. They employed integration priority to denote the

importance of service software and proposed an improved genetic algorithm to generate integration test sequence between service software.

Zhang et al. [21] adopted coupling-based complexity to gauge test stub cost. They proposed a hybrid algorithm of grey wolf optimizer and arithmetic optimization algorithm to provide a proper balance between exploration and exploitation, improving the convergence speed and optimization accuracy.

It is meaningful to conduct performance analysis in a certain research field, which helps follow-up researchers to quickly understand the research status, research results, and the effectiveness of corresponding solutions. Harman and Mcminn [22] conducted theoretical and practical research on test data generation, analyzed the applicability and effectiveness of local search algorithm, global search algorithm and hybrid search algorithm. Harman et al. [23] summarized the existing achievements in the field of search-based software testing and put forward the problems or challenges in this field. Khari et al. [24] analyzed and summarized the research results of search-based software testing from 1996 to 2016. Zhang et al. [25] comprehensively analyzed the technical characteristics and research status of graph-based and search-based methods for the CITO problem. Zhang et al. [26] conducted a convergence analysis for some related algorithms from the perspective of Markov chain theory. Khari et al. [1] analyzed and compared the effectiveness and efficiency of several meta heuristic algorithms over the automatic test suite generation.

3. Research Methodology

The research methodology and main processes are defined for this comparative study, as shown in Figure 1. The main steps undertaken for this study include algorithm selection, problem representation, CITO optimization and comparative performance evaluation.

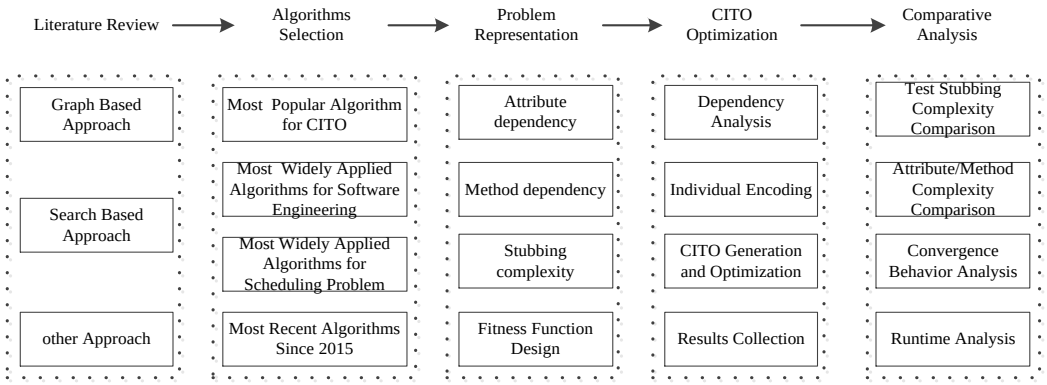


Figure 1. Research methodology.

The first phase involves conducting a comprehensive literature review on the CITO problem, outlined in Section 2. Both graph-based and search-based approaches have been widely used to obtain class integration test orders. Differently from the reverse topological sorting used in graph-based approaches, search-based approaches are flexible and practical. Researchers have pointed out that the CITO problem, which is very much related to the so-called routing or scheduling problem, is a multi-objective optimization problem. Theoretically, the search space for the CITO problem encompasses all possible integration test orders, and the optimization process could be fraught with numerous local minima in which a stepwise algorithm may get stuck. These unique characteristics of CITO may have a significant impact on the optimization ability of meta heuristics algorithms to converge towards optimal test orders.

After the step of meta heuristic algorithms selection, the reformulation of the CITO problem is presented with mapping strategies between a test order and an individual location. The subsequent step is the execution of each meta heuristic algorithm to search class integration test orders following a set of rules, which are the core of each algorithm. This iterative process is repeated hundreds of times and the stubbing complexity, method

complexity, attribute complexity, average time, etc., are recorded for each execution. After this, the optimization performance is analyzed and discussed.

3.1. Algorithms Selection

According to their popularity and how recently they were developed, we selected ten algorithms: genetic algorithm (GA), particle swarm optimization algorithm (PSO), cuckoo search (CS), firefly algorithm (FA), bat algorithm (BA), grey wolf optimizer (GWO), moth flame algorithm (MFO), sine cosine algorithm (SCA), salp swarm optimization algorithm (SSA), and Harris hawk optimization algorithm (HHO). The primary rationale for the selection is based on their simplicity, flexibility, and capability of preventing the optimization process from getting trapped into local optimum. The detailed reasons are as follows.

(1) Genetic algorithm and particle swarm optimization algorithm are widely recognized and commonly employed for generating integration test orders.
(2) Cuckoo search algorithm, firefly algorithm, and bat algorithm have been widely used to automate the process of test data generation with the goal of covering target paths in the research field of search-based software engineering [1,27].
(3) Grey wolf optimization algorithm, moth flame algorithm, sine cosine algorithm, salp swarm algorithm, and Harris hawk optimization algorithm have emerged as more efficient solutions since 2015. They have been successfully applied to scheduling or routing problems and various combinatorial optimization problems with a huge and noisy search space [7]. The relevance of these problems to the test order problem makes these algorithms suitable candidates in the CITO context.

A brief introduction of each algorithm is described below.

Genetic algorithm [12] is a global optimization techniques that mimic biological evolution. Each chromosome evolves under the influence of natural selection and recombination. Natural selection determines which individuals are selected, survive, and reproduce. Recombination ensures the chromosome will be mixed to form a new one. In the context of genetic algorithm, the encoding scheme, crossover, and mutation operators should be designed for a special optimization problem.

Particle swarm optimization [28] is a meta heuristic algorithm proposed by Kennedy in 1995 to simulate the predatory behavior of birds. Each particle represents a candidate solution with its own velocity and spatial position. In the process of evolution, a particle improves its quality under the guidance of individual historical optimal solution p and the global optimal solution g. At the $t + 1$ iteration, the velocity and position update equations of the dth dimension of individual x are shown in Equations (1) and (2), where w represents the inertia weight, c_1 and c_2 are learning factors, r_1 and r_2 are random numbers between 0 and 1.

$$v_{id}^{t+1} = w^t v_{id}^t + c_1 r_1 \left(p_{id}^t - x_{id}^t\right) + c_2 r_2 \left(g_{id}^t - x_{id}^t\right) \tag{1}$$

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \tag{2}$$

The cuckoo search algorithm [29] is a meta heuristic algorithm proposed by Yang in 2009, inspired by the hatching parasitism characteristic of cuckoo. Each bird nest improves its quality through random walk. The optimization process discards poor solutions with a certain probability. The position update formula of the ith nest of cuckoo at $t + 1$ iteration is shown in Equation (3), where $\oplus$ represents point-to-point multiplication, α represents the step size subject to normal distribution and $L(\lambda)$ represents the random step size following to Levy distribution.

$$x_i^{t+1} = x_i^t + \alpha \oplus L(\lambda) \tag{3}$$

The firefly algorithm [30] is a meta heuristic algorithm proposed by Yang in 2010 inspired by the attracting and information exchanging behavior of fireflies. In the process of evolution, fireflies are attracted by brighter ones. In the $t + 1$ generation, the dth dimen-

sional update formula of firefly x is defined in Equation (4), where β represents the relative attraction between two fireflies, α represents the step factor, ε represents a random number following the uniform distribution.

$$x_{id}(t+1) = x_{id}(t) + \beta\Big(x_{jd}(t) - x_{id}(t)\Big) + \alpha_i(t)\varepsilon \tag{4}$$

The bat algorithm [31] is a meta heuristic algorithm proposed by Yang in 2010 to simulate the precise search and prey hunting behavior of bats using ultrasound. In the process of evolution, the best bat locates the accurate prey position through ultrasonic pulse. The speed and position update formula for each individual are given in Equations (5) and (6), where x^* represents the global best position and f_i represents the search pulse frequency.

$$v_i^t = v_i^{t-1} + \left(x_i^t - x^*\right)\cdot f_i \tag{5}$$

$$x_i^t = x_i^{t-1} + v_i^t \tag{6}$$

The grey wolf optimization algorithm [32] is a meta heuristic algorithm proposed by Mirjalili in 2014 to simulate the hierarchical structure and hunting behavior of grey wolves. The social groups of wolves are divided into α wolves, β wolves, δ wolves and ω wolves, representing the leader, assistant decision-maker, executor, and lowest level of wolves, respectively. Each ω wolf represents a candidate solution and moves to a new position under the guidance of α, β and δ wolf, as shown in Equations (7) and (8), where D represents the distance between the wolf and the prey, C represents the swing factor, $X_p(t)$ represents the prey position, and A represents the convergence factor.

$$D = \left|C\cdot X_p(t) - X(t)\right| \tag{7}$$

$$X(t+1) = X_p(t) - A\cdot D \tag{8}$$

The moth flame algorithm [33] is a meta heuristic optimization algorithm proposed by Mirjalili in 2015 to simulate the spiral flight path of moths at night. In the process of evolution, the individual position is updated based on its nearby flames, as shown in Equations (9) and (10), where D_i represents the distance between the moth and prey, F_j represents the flame j, M_i represents the moth i, and b is the constant coefficient defining the logarithmic spiral shape, and θ is the random path coefficient in $[-1, 1]$.

$$D_i = \left|F_j - M_i\right| \tag{9}$$

$$M_i = S\left(M_i, F_j\right) = D_i\cdot e^{b\theta}\cdot cos(2\pi t) + F_j \tag{10}$$

The sine cosine algorithm [34] is a meta heuristic algorithm proposed by Mirjalili in 2016. Candidate solutions fluctuate towards the best solution using a mathematical model based on sine and cosine functions. At the t + 1 iteration, the position update equation of the individual x is shown in Equation (11), where r_2, $r_{3,}$ and r_4 are random numbers. In order to balance the exploration and exploitation ability, r_2 is viewed as the control parameter of linear decreasing according to (12), in which $\propto$ is a constant.

$$x_{id}^{t+1} = \begin{cases} x_{id}^t + r_1 \times sin(r_2) \times \left|r_3 P_{id}^t - x_{id}^t\right|, r_4 < 0.5 \\ x_{id}^t + r_1 \times cos(r_2) \times \left|r_3 P_{id}^t - x_{id}^t\right|, r_4 \geq 0.5 \end{cases} \tag{11}$$

$$r_1 = \propto \times (1 - t/T) \tag{12}$$

The salp swarm algorithm [35] is a meta heuristic algorithm proposed by Mirjalili in 2017 to simulate the navigation and foraging chain behavior of salp swarm. The group is

divided into leaders and followers, representing the exploration and exploitation ability separately. In the process of evolution, leaders guide followers to move toward the target food, and each follower interacts with the before and after individuals. The position update formula of dimension d of leader x^1 is shown in Equation (13), where F_d represents the food position, ub and lb represent the upper and lower bounds of corresponding dimensions, respectively. C_2 and C_3 are random numbers in [0, 1]. C_1 is the convergence factor shown in Equation (14), where t represents the current iterations and T is the max iterations. The update formula of the follower x^i is shown in Equation (15).

$$x_d^1 = \begin{cases} F_d + c_1(c_2(ub - lb) + lb), c_3 \geq 0.5 \\ F_d - c_1(c_2(ub - lb) + lb), c_3 < 0.5 \end{cases} \tag{13}$$

$$c_1 = 2e^{-(4t/T)^m} \tag{14}$$

$$X_d^i = (X_d^i - X_d^{i-1})/2 \tag{15}$$

The Harris hawk optimization [36] is a meta heuristic algorithm proposed by Heidari in 2019 to simulate the cooperative behavior and chasing style of Harris' hawks in nature. In view of its large number of position update equations, only the workflow of the algorithm is briefly presented here. The algorithm selects exploration or exploitation stage according to the comparison result between escape energy of prey and the random number set. Further, according to the escape behavior of prey and the chase strategy of the Harris hawk, the algorithm can carry out the process of soft and hard siege and timely switch between soft and hard siege.

3.2. Problem Representation

3.2.1. Stubbing Complexity

Due to complex dependencies among classes, a lot of effort is required to build stubs to simulate services required by the tested object. However, stubs are not a real part of the software and will not be used in the final software. Naturally, people hope that the fewer stubs constructed in the process of class integration testing the better, and the lower construction cost the better. Thus, the cost of a class integration test order can be evaluated by test stub cost.

However, in many cases, the number of stubs is not directly proportional to the test stub cost; hence, the accuracy of using the number of stubs to evaluate the test stub cost is low. Since the stubbing effort cannot be directly measured or estimated, we adopt the coupling-based fitness function, which captures coupling relationships in object-oriented systems, in a fashion similar to the previous studies.

There are various dependencies among classes in the object-oriented software development. Inheritance and composition include not only control coupling but also data coupling, constituting strong dependencies [2]. Breaking such strong relationships would likely lead to complex stubs with high cost. For instance, if B is a subclass of A, A should be integrated before B. Usage, association, and simple aggregations are considered to be weak dependencies with low cost. Two simple and intuitive measures of coupling are used to quantify weak dependencies:

Attribute complexity [12] is quantified as follows: The number of attributes locally declared in the target class when references/pointers to instances of the target class appear in the argument list of some methods in the source class, as the type of their return value, in the list of attributes of the source class, or as local parameters of methods. The number of attributes between target class i and source class j that would be handled in a stub is called attribute complexity A (i, j).

Method complexity [12] is quantified as follows: The number of methods locally declared in the target class which are invoked by the source class methods. The number of

methods between target class i and source class j that would be handled in a stub is called method complexity $M(i, j)$.

After the static analysis of the system under test, the attribute complexity matrix A and method complexity matrix M among classes can be constructed. Taking account both attribute complexity and method complexity, we can define stubbing complexity between class i and class j through linear scalarization method.

Further, in order to avoid the strong influence of any complexity measure unit on the stubbing complexity, any complexity measure $Cplx(i, j)$ is normalized according to Equation (16), expressed as $\overline{Cplx(i, j)}$.

$$\overline{Cplx(i,j)} = Cplx(i,j)/(Cplx_{max} - Cplx_{min}) \tag{16}$$

where $Cplx_{max}$ and $Cplx_{min}$ represent the maximum and minimum value of matrix elements, respectively. When there is no coupling between two classes, the value of the matrix element is specified as 0, that is, the minimum value of the matrix is 0. Therefore, Equation (16) can be simplified to Equation (17).

$$\overline{Cplx(i,j)} = Cplx(i,j)/Cplx_{max} \tag{17}$$

For a pair of classes (i, j) with dependency linking, the stubbing complexity $SCplx(i, j)$ can be defined and computed as a weighted geometric average of the two normalized complexity measures, as shown in Equation (18).

$$SCplx(i,j) = \left(W_A * \overline{A(i,j)}^2 + W_M * \overline{M(i,j)}^2\right)^{1/2} \tag{18}$$

where W_A and W_M represent the weights of attribute complexity and method complexity, respectively, and $W_A + W_M = 1$.

3.2.2. CITO Formulation

For a class integration test order o containing n classes, its fitness function can be expressed as the stubbing complexity of the total class dependency pairs (i, j) contained in the testing order o, as shown in Equation (19).

$$fitness(o) = Ocplx(o) = \sum_{i=1,j=1}^{n} SCplx(i,j) \tag{19}$$

Assuming that the classes are represented as a set $C = \{C_1, \ldots, C_m\}$ to be integrated, then the CITO problem can be viewed as the problem of constructing a permutation π of {1, 2, …, m} that minimizes the stubbing complexity when classes are integrated in the order given by π: $C_{\pi 1,\ldots,m} = (C_{\pi 1}, C_{\pi 2}, \ldots, C_{\pi m})$.

The search space to the CITO problem is the set of all possible integration test orders. Therefore, the optimization process of generating class integration test orders is to search one or a group of testing orders that can minimize the test stub cost under the guidance of the fitness function, as shown in Equation (20).

$$min\ fitness(C_{\pi 1,\ldots,m}) = min\ ocplx(C_{\pi 1}, C_{\pi 2}, \ldots, C_{\pi m}) \tag{20}$$

3.3. CITO Optimization

This is the main step wherein each selected meta heuristic algorithm is implemented to generate a set of optimal test orders. The main difference between heuristic algorithm and meta-heuristic algorithm is that the heuristic algorithm needs to be modified according to actual problems, while meta heuristic algorithm can solve most kinds of optimization problems without modification [1]. So we use the standard form of these algorithms to solve the CITO problem. The general search-based CITO optimization process is shown in Figure 2.

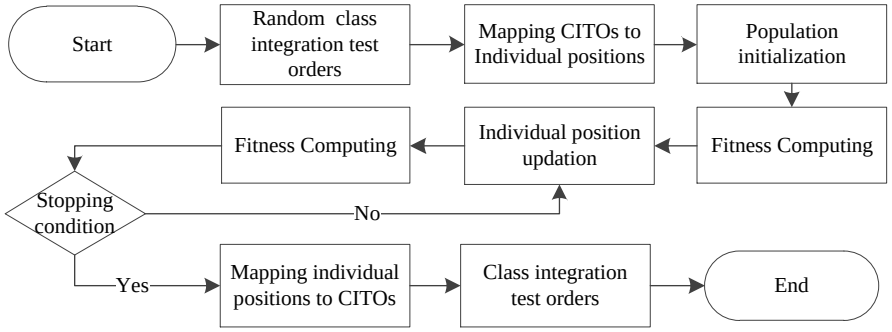


Figure 2. The general flow of search-based CITO optimization process.

There are three modules in the general flow of search-based CITO optimization process. Firstly, a group of class integration test orders are generated randomly. By following the mapping algorithm as shown in Section 3.3.1, each test order is mapped to an individual position in one dimensional space. Secondly, the position of each individual is calculated and improved under the guidance of fitness function as described in Section 3.2, following the optimization rules of different algorithms. Finally, the algorithm stops and the optimum individual position is obtained, and mapped to the optimal class integration test order as described in Section 3.3.2.

3.3.1. Mapping CITO to Individual Position

The search space to the CITO problem is the set of all possible integration test orders for the system under test. Referring to the approach proposed by Zhang [17], any class integration test order can be regarded as an individual in one-dimensional space, while different test orders correspond to different positions in one dimensional space. The mapping pseudo code from a class integration test order to individual position is shown in Algorithm 1.

Algorithm 1: Map a class integration test order to individual position

Input: class integration testing order *o*
list of classes in descending order *list*
number of classes *n*
Output: *position*

```
1.  BEGIN
2.  Initialize the individual position = 0;
3.  FOR (int i = 0; i < n; i++)
4.    temp_class = o.get(i);/*get ith class in o*/
5.    index = list.getIndex(temp_class);
6.    position = position + index *(n-i-1)!
7.  END FOR
8.  END
```

Given a class integration test order *o*, the classes list in descending order based on the sum of method and attribute coupling *list*, and the number of classes *n*, Algorithm 1 can generate the corresponding individual position for *o* in one dimensional space, *position*.

Algorithm 1 begins with the initialization step (line 2) where the output *position* is initialized and set to 0. Every class in a given test order *o* would be traversed and the individual position of these *n* classes in test order *o* would be determined (lines 3–7). For each class in *o*, the class information and its index in the ordered list are obtained (lines 4–5), then its location can be calculated (line 6). By summing locations of these *n* classes, we can get the position in one dimensional space for a class integration test order through the above steps.

3.3.2. Mapping Individual Position to CITO

Any individual in search space can be converted to a class integration test order, as shown in Algorithm 2. In the process of evolution, the algorithm generates new class integration test orders by constantly changing their own spatial positions. At the end of the iteration, the algorithm obtains the optimal individual position, which would be further mapped into a class integration test order. The mapping pseudo code from an individual position to a class integration test order is shown in Algorithm 2.

Algorithm 2: Map individual position to a class integration test order

Input: *position*
list of classes in descending order *list*
number of classes *n*
Output: a class integration test order *o*

```
1.  BEGIN
2.  FOR (int i = 0; i < n; i++)
3.    index = position/(n-i)!
4.    temp_class = list.get(index)
5.    o.add(temp_class)
6.    list.remove(temp_class)
7.    position = position % (n-i)!
8.  END FOR
9.  o.add(list.get(0))
10. END
```

Given a position of an individual in one dimensional space *position*, the sorted classes list in descending order based on the sum of method and attribute coupling *list*, and the number of classes *n*, Algorithm 2 determines the class integration test order *o*.

Algorithm 2 calculates the class integration test order that the individual corresponds to, in the form of adding each class in the test order one by one (lines 2–8). The position of a class in *list* can be calculated through the location information divided by the permutations number of classes at this location (line 3). Then, a class is added to *o* and removed from *list* (lines 5–6). After circular computations, the remaining classes can be added to *o* (lines 2–8). When there is only one class in *list*, the last class is added to *o* finally (line 9).

3.3.3. Constraints

A number of dependencies can be found among classes in the object-oriented software development. Different dependencies may lead to stubs of widely varying complexity [12]. Among these various relationships, inheritance and composition relationships usually entail tight attribute coupling or method coupling between child/parent and container/contained classes, constituting strong dependencies. Breaking such strong relationships would likely lead to complex stubs with high cost.

According to the coupling level involved and fitness function designed in Section 3.2, our strategy does not construct stubs for inheritance and composition dependencies. This implies that the parent/container classes must precede child/contained classes in any class integration order.

Based on the static analysis of software under test, a precedence table outlines the conditions requisite for an individual to be accepted. Under these constraints, any integration test order that violates the inheritance and composition dependency relations would be rejected and regenerated until the order fulfills the constrains in the precedence table.

3.4. Comparative Analysis

The execution process of each meta heuristic algorithm to search class integration test orders is done hundreds of times repeatedly to make sure the consistent optimization

results. Once the end criteria are achieved, the stubbing complexity, method complexity, etc., are stored for each execution.

Then, the comparative optimization performance is analyzed and discussed to compare the algorithms relative to each other. Here, the metrics of stubbing complexity, method complexity, attribute complexity, average runtime and memory are compared to deduce the suited algorithms for the CITO problem.

4. Experimentation

Taking the stubbing complexity as the fitness function, we design two questions to focus on the performance analysis of these ten meta heuristic algorithms for the CITO problem.

Q1: Which algorithms can devise class integration test orders with low stubbing complexity?

The performance metrics for hundreds executions of each meta heuristic algorithm are collected, analyzed, and compared, including stubbing complexity, attribute complexity, and method complexity. The smaller value of the stubbing complexity, the lower the test stub cost required.

Q2: How efficient are the algorithms for generating class integration test orders?

Different algorithms have different convergence speeds and different iteration runtime. Therefore, this paper attempts to analyze the convergence speed, runtime and memory required of each algorithm to find which algorithms can obtain the same or better class integration test order with less computational resource.

4.1. Experiment Subjects

In order to verify the performance of these meta heuristic algorithms for the CITO problem, ten classic benchmark systems are selected for testing [3,11,17–19,21]. These case studies are real-world systems with varying complexities written in Java. Elevator is an elevator operation simulation system. SPM is a patrol monitoring simulation system. ATM is an automated teller machine simulation system. Daisy is a network file system. ANT is a Java program update and build tool supplied by the open source Apache project, which is a part of the Jakarta project. DEOS is a detection error system for concurrency bugs of multi-threaded programs. Email is a supporting system for email operations. BCEL is a byte code engineering library, a part of Jakarta project too. DNS is a domain name service system, and Notepad_SPL (Notepad for short) is a source code editor.

For SPM, ATM, ANT, BCEL and DNS, the same benchmark used by Briand et al. [12] is employed. We take the inter class relationship as input, but not source code. The class information, dependency information are identified and analyzed based on reverse engineering of object relation diagram.

For Elevator, Daisy, DEOS, Email and Notepad, the source code is analyzed by using the open source tool SOOT for detailed relationships among classes. Using the information of variable, method, class name and package name, etc., we can obtain the object relation diagram, attribute complexity, method complexity and precedence relationship. Different from the SPM, ATM, ANT, BCEL and DNS, we have not re-implemented the graph-based algorithm for Elevator, Daisy, DEOS, Email and Notepad, and therefore, we do not know the number of cycles, which is expressed by "N/A" in Table 1.

The detailed information about these systems is shown in Table 1. Column 2 shows the total class numbers, Columns 3–9 give the number of usages, associations and aggregations, compositions, inheritance, the total dependencies, the total elementary circuits, and code lines, respectively. These systems come from various application fields and have different functions. The number of classes ranges from 21 to 65, the cycles from 4 to 416,091, and the code lines from 934 to 6710, making these cases representative.

Table 1. Detailed information of the ten systems.

System	Classes	Usages	Associations & Aggregations	Compositions	Inheritance	Dependencies	Cycles	#LOC
Elevator	12	8	3	16	0	27	N/A	934
SPM	19	24	34	10	4	72	1178	1198
ATM	21	39	9	15	4	67	30	1390
Daisy	23	31	2	3	0	36	N/A	1148
ANT	25	54	16	2	11	83	654	4093
DEOS	25	52	15	6	5	78	N/A	2215
Email	39	44	5	12	2	63	N/A	2276
BCEL	45	18	226	4	46	294	416,091	3033
DNS	61	211	23	12	30	276	16	6710
Notepad	65	122	8	11	0	141	N/A	2419

In order to further describe the detailed coupling distribution, the attribute and method coupling values between classes are summarized in Table 2. Recall that the CITO optimization process should satisfy the inheritance and composition dependency, the value of attribute coupling and method coupling in Table 2 are the coupling value of associations, simple aggregations, and usage dependencies. Column 2 and column 5 are the max values, column 3 and column 6 are the average values, and the sum values are shown in column 4 and column 7.

Table 2. Coupling summary of the ten systems.

System	Attribute Coupling			Method Coupling			Total
	Max	Average	Sum	Max	Average	Sum	
Elevator	3	1.62	21	25	6.33	158	179
SPM	21	7.97	462	8	2.41	135	597
ATM	13	6.02	283	7	2.33	84	367
Daisy	11	3.78	34	16	4.22	135	169
ANT	31	9.14	585	14	2.9	177	762
DEOS	4	2.04	26	15	3.28	223	249
Email	22	3.13	72	40	4.18	222	204
BCEL	8	2.52	454	4	1.55	369	823
DNS	10	4.35	766	8	1.92	328	1094
Notepad	8	1.88	102	37	1.74	181	283

4.2. Parameters Settings

Ten intelligent algorithms, genetic algorithm, particle swarm optimization algorithm, cuckoo search algorithm, firefly algorithm, bat algorithm, grey wolf optimization algorithm, moth flame algorithm, sine cosine algorithm, salp swarm algorithm and Harris hawk optimization algorithm, are selected for performance analysis.

Since genetic algorithm and particle swarm optimization algorithm have been employed to generate CITO in the literature, the values of their relevant parameters in the literature are used. While other algorithms have not been applied to the CITO problem, their standard implementation form and corresponding parameter values are adopted to minimize the parameters dependence. Although there are many variants of the remaining eight algorithms, most of them were designed to solve certain special engineering problems. Essentially, the standard form of each algorithm embodies the core optimization principles with excellent stability and performance [7]. Additionally, for the NP complete CITO problem, we have not conduct in-depth theoretical research on its mathematical characteristics, hence we use the standard form of these algorithms to generate test orders.

The detailed parameter settings for these ten meta heuristic algorithms are shown in Table 3.

Table 3. Parameter settings for various algorithms.

Algorithm	Parameters	Reference
GA	$p_m = 0.01$, $p_c = 0.5$	Briand 2002 [12]
PSO	$w_{max} = 0.9$, $w_{min} = 0.4$, $c_1 = 2$, $c_2 = 2$	Zhang 2018 [17]
CS	$p = 0.02$, $\alpha = 0.15$, $\lambda = 1.5$	Yang 2009 [29]
FA	$\alpha = 0.2$, $\beta_0 = 1$, $\gamma = 1$	Yang 2010 [30]
BA	$\alpha = 0.9$, $\gamma = 0.7$, $f_{min} = 0$, $f_{max} = 2$	Yang 2010 [31]
GWO	$\alpha_{first} = 2$, $\alpha_{final} = 0$	Mirjalili 2014 [32]
MFO	$b = 1$, $\theta \in [-1, 1]$	Mirjalili 2015 [33]
SCA	$\alpha = 2$, $r_2 \in [0, 2\pi]$, $r_3 \in [-2, 2]$, $r_4 \in [0, 1]$	Mirjalili 2016 [34]
SSA	$m = 2$, $c_2, c_3 \in [0, 1]$	Mirjalili 2017 [35]
HHO	$\beta = 1.5$, $E_0 \in [-1, 1]$, $J \in [0, 2]$	Heidari 2019 [36]

4.3. Experiment Design

The experiment environment is Intel i7 CPU 2.6 GHz. RAM 8 GB. The software used is Windows 10 with minconda3 and Pycharm development environment. For each algorithm, the detail steps are implemented as follows to obtain optimal test orders.

Step 1. Randomly initialize the population, where each individual represents an integration test order.

Step 2. Map each individual in the initialized population into a individual location of one dimensional space, hence each position is used to represent an integration test order instead, as designed in Section 3.3.1.

Step 3. Update the individual location according to the evolve strategy designed by each algorithm, and gain the optimal individual location once achieving the end criteria, as described in Section 3.1.

Step 4. Map the output individual locations to class integration test orders, as designed in Section 3.3.2.

Considering the recommended value of population size rang of 25 to 100, the population size is set to 100 and the maximum iteration is set to 200 to maintain the population diversity in this paper. Additionally, the values of W_A and W_M in the fitness function are both set to 0.5, referring to previous research findings. To mitigate the impact of randomness, the experimental results are averaged over 20 independent runs.

For Q1, the stubbing complexity is used to measure the test stub cost as defined in Section 3.2. In order to understand the effectiveness of search-based approaches for CITO, we analyze the relationship between optimization results and the number of dependencies, as well as the relationship between optimization results and coupling distribution. Subsequently we conduct statistical Wilcoxon analysis for performance comparison. Furthermore, we discuss the optimization capability on attribute complexity and method complexity separately, to determine which algorithms perform better in attribute coupling guided systems and which algorithms fare better in method coupling guided systems.

For Q2, the convergence behavior, average runtime and memory consumption are used to measure the efficiency of these meta heuristic algorithms. For the convergence behavior analysis, we discuss the influence of dependency cycle on optimization performance. In terms of computational resource, the values of average runtime and memory of each algorithm are stored and analyzed.

4.4. Effectiveness Analysis

4.4.1. Stubbing Complexity

As described in Section 3.2, the fitness function value $Ocplx(o)$ for a given test order o is the sum of a set of weighted geometric average of the normalized attribute complexity $\overline{A(i,j)}$ and method complexity $\overline{M(i,j)}$, computed according to Equation (17), respectively, ensuring that the resulting stubbing complexity is insensitive to the two measurement units.

During the experiment, each algorithm ran independently 20 times, and the minimum, maximum and average values of the stubbing complexity were recorded. The resulting

stubbing complexity implemented by the genetic algorithm, particle swarm optimization algorithm, cuckoo search algorithm, firefly algorithm, bat algorithm, grey wolf optimization algorithm, moth flame algorithm, salp swarm algorithm, sine cosine algorithm and Harris hawk optimization algorithm is shown in Table 4 and Figure 3. In Table 4, for each system under test, the first and second rows represent the best and worst value, the third row provides the average stubbing complexity value, the fourth row represents the standard deviation of the optimization results, then the fifth row represents the ranking of the average optimization results. In Figure 3, the horizontal axis indicates the systems under test and the vertical axis indicates the average stubbing complexity of the CITO generated by each algorithm.

Table 4. Stubbing complexity values of the ten systems.

System	Stats.	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO
Elevator	best	1.76	1.79	1.76	1.85	1.97	1.76	1.76	1.76	1.76	1.94
	worst	1.85	1.95	1.89	2.27	2.37	1.91	1.82	1.88	1.93	3.91
	mean	1.81	1.86	1.84	2.03	2.20	1.84	1.79	1.79	1.85	3.52
	Stdev	0.02	0.05	0.04	0.09	0.13	0.04	0.02	0.03	0.04	0.54
	rank	3	7	4	8	9	5	1	2	6	10
SPM	best	3.05	3.40	2.99	3.35	4.05	3.04	2.45	2.55	3.58	2.99
	worst	3.96	4.16	3.80	4.99	6.37	3.80	3.54	3.77	4.19	4.08
	mean	3.51	3.75	3.37	4.03	4.88	3.45	2.97	3.19	3.82	3.61
	Stdev	0.28	0.22	0.27	0.46	0.62	0.21	0.23	0.32	0.19	0.30
	rank	5	7	3	9	10	4	1	2	8	6
ATM	best	2.32	2.17	2.29	2.26	3.02	2.29	2.17	2.17	2.32	2.27
	worst	3.18	2.82	2.74	3.52	4.90	2.77	2.55	2.94	2.94	2.98
	mean	2.74	2.53	2.54	3.00	3.83	2.50	2.37	2.42	2.65	2.69
	Stdev	0.25	0.16	0.12	0.17	0.44	0.12	0.09	0.16	0.18	0.21
	rank	8	4	5	9	10	3	1	2	6	7
Daisy	best	0.32	0.32	0.35	0.43	0.55	0.22	0.19	0.48	0.26	0.39
	worst	0.59	0.74	0.61	1.68	1.22	0.51	3.86	0.77	0.67	0.67
	mean	0.46	0.54	0.48	0.65	0.94	0.43	0.72	0.61	0.50	0.54
	Stdev	0.09	0.13	0.07	0.28	0.18	0.09	1.00	0.09	0.13	0.08
	rank	2	6	3	8	10	1	9	7	4	5
ANT	best	2.12	2.16	2.14	2.13	2.69	1.86	1.75	1.93	2.17	2.31
	worst	2.83	2.79	2.65	2.92	4.36	2.61	2.34	2.68	2.57	2.82
	mean	2.51	2.59	2.45	2.99	3.41	2.44	2.03	2.30	2.39	2.55
	Stdev	0.21	0.15	0.14	0.21	0.48	0.19	0.16	0.22	0.12	0.15
	rank	6	8	5	9	10	4	1	2	3	7
DEOS	best	3.42	3.28	3.20	3.96	4.18	3.24	2.84	3.20	2.92	3.33
	worst	4.06	4.18	4.02	4.84	5.39	3.85	3.73	3.68	4.05	4.46
	mean	3.74	3.81	3.63	4.32	4.94	3.57	3.18	3.47	3.71	3.86
	Stdev	0.17	0.28	0.24	0.62	0.36	0.20	0.29	0.15	0.33	0.28
	rank	6	7	4	9	10	3	1	2	5	8
Email	best	0.70	0.66	0.65	1.10	1.05	0.73	0.56	0.75	0.70	0.62
	worst	0.86	0.89	0.81	1.85	1.43	0.91	0.80	1.10	0.91	0.92
	mean	0.78	0.81	0.73	1.33	1.20	0.82	0.69	0.90	0.78	0.81
	Stdev	0.05	0.06	0.05	0.20	0.11	0.06	0.07	0.10	0.06	0.07
	rank	3	5	2	10	9	7	1	8	4	6
BCEL	best	10.19	10.90	10.45	11.00	11.98	10.35	11.98	10.00	10.71	10.33
	worst	11.65	11.98	11.95	13.63	11.98	11.47	11.98	11.85	11.59	11.98
	mean	10.89	11.62	11.29	11.75	11.98	10.79	11.98	10.83	11.14	11.16
	Stdev	0.56	0.51	0.40	0.60	0.00	0.44	0.00	0.63	0.26	0.47
	rank	3	7	6	8	9	1	10	2	4	5

Table 4. *Cont.*

System	Stats.	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO
DNS	best	6.31	7.34	7.20	6.53	8.63	6.97	6.30	6.68	7.34	6.95
	worst	9.45	9.63	9.11	10.27	12.80	8.95	8.37	8.71	10.05	9.65
	mean	7.98	8.63	8.14	9.31	10.39	8.06	7.33	7.70	8.94	8.64
	Stdev	0.92	0.73	0.70	0.86	1.57	0.69	0.81	0.67	0.76	0.75
	rank	3	6	5	9	10	4	1	2	8	7
Notepad	best	1.75	1.83	1.77	1.76	1.90	1.53	1.65	1.43	1.80	1.77
	worst	1.93	2.17	1.96	2.41	3.09	1.97	1.85	2.21	1.94	2.17
	mean	1.84	1.98	1.85	2.07	2.50	1.77	1.73	1.92	1.85	1.90
	Stdev	0.05	0.10	0.07	0.24	0.30	0.13	0.06	0.20	0.04	0.15
	rank	3	8	4	9	10	2	1	7	5	6

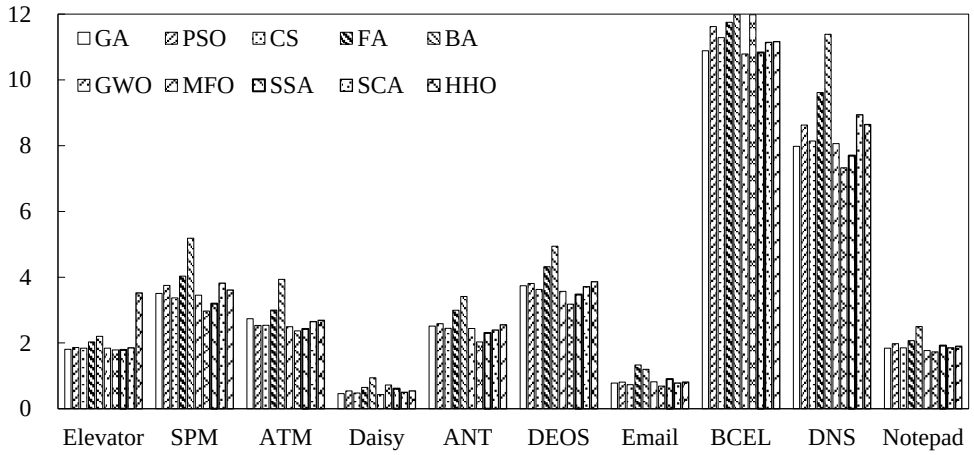


Figure 3. Histogram for test stubbing complexity comparison.

The stubbing complexity is affected by many factors, such as the number of classes, the dependency distribution, the coupling distribution, etc. Some typical influencing factors on performance result are analyzed as follows.

(1) Dependencies Distribution Factor Analysis

The relationship between the optimization results and the number of dependencies is analyzed as follows:

- Systems with few dependencies

For the Elevator and Daisy with few class dependencies, the difference of optimization results among various algorithms is not obvious.

- Systems with mid-sized dependencies

Systems with mid-sized dependencies between 61 and 83 include SPM, ATM, ANT, DEOS and Email.

For SPM with 72 dependencies, the algorithms with better optimization results in order are moth flame algorithm, salp swarm algorithm, grey wolf optimization algorithm and particle swarm optimization algorithm.

For ATM with 67 dependencies, the algorithms with better optimization results are moth flame algorithm, salp swarm algorithm, cuckoo search algorithm and grey wolf optimization algorithm in order.

For ANT with 83 dependencies, the algorithms with better optimization results are moth flame algorithm, salp swarm algorithm, sine cosine algorithm and grey wolf optimization algorithm, respectively.

For DEOS with 73 dependencies, the algorithms with better optimization results are moth flame algorithm, salp swarm algorithm, grey wolf optimization algorithm and cuckoo search algorithm in order.

The algorithms with better optimization results are moth flame algorithm, cuckoo search algorithm, genetic algorithm, and sine cosine algorithm in order for Email with 61 dependencies.

From the perspective of ranking results, it can be seen that moth flame algorithm and salp swarm algorithm rank first and second in SPM, ATM, ANT and DEOS, respectively. Grey wolf optimization algorithm appears four times in these five systems, cuckoo search algorithm three times and sine cosine algorithm two times. These data fully illustrate the effectiveness of moth flame algorithm, salp swarm algorithm, grey wolf optimization algorithm and cuckoo search algorithm in the software with medium number of class dependencies.

- Systems with large-sized dependencies

For BCEL, DNS and Notepad with large-sized class dependencies, moth flame algorithm continues to rank high in DNS and Notepad. However, it shows obvious stagnation and obtains the worst result in BCEL. Grey wolf optimization algorithm is the best algorithm with the highest precision of optimization results for BCEL. Additionally, it is the better algorithm with the higher precision of optimization results in Notepad. Additionally, it performs well in DNS system.

(2) Coupling Distribution Factor Analysis

The relationship between the optimization results and the coupling distribution is analyzed as follows.

Systems with relatively low stubbing complexity include Daisy, Email and Notepad. The reason behind is that the sum of method coupling and attribute coupling of these systems is low. It is worth noting that although Notepad contains the most classes, the stubbing complexity is small because of its max value 37 of method coupling and lots of simple attribute and method coupling. BCEL contains 45 classes with 823 coupling which reflects the high coupling of classes, hence its stubbing complexity is also the highest. While 61 classes and 276 dependencies with total 1094 attribute and method coupling in DNS increases the difficulty of test stub construction, the average stubbing complexity is high. The higher coupling degrees in a system, the higher the stubbing complexity.

Among all optimization results for these ten systems, the stubbing complexity of class integration test orders generated by bat algorithm and firefly algorithm is high. Most notably, the standard deviation of bat algorithm is the largest, which indicates the unstable iterative optimization process. The fact that individuals in bat algorithm lack the ability to mutate reduces the population diversity and makes the search process easy to fall into stagnation.

(3) Wilcoxon Analysis

Taking the experimental results of 20 times of independent running with 200 maximum iteration each time as training samples, we conduct the statistical tests for performance comparison. Due to the small sample size, the stats.shapiro() method in python was used to perform the Shapiro–Wilk normality test on the sample data. The normality test result shows that the sample data does not meet the normality condition, thus we can apply the non-parametric Wilcoxon test on the average stubbing complexity value between any two algorithms. At a significance level of 0.05, the null hypothesis (H_0) states that the two algorithms at the row and column have equivalent performance and the alternative hypothesis (H_1) states that the two algorithms have significant performance. If the p-value < 0.05, then H_0 cannot be valid.

The detailed results of Wilcoxon signed rank test with a p-value threshold of 0.05 are reported in Table 5. Columns and rows represent different meta heuristic algorithms to generate integration test orders. Each cell contains the p-value obtained between the

two algorithms at the row and column of this cell. While the values in last column indicate the number of cases where p-value < 0.05. In those cases, significant p-values denote that the algorithms identified by their associated rows are significantly better than the other algorithms being compared. From Table 5, we can observe that grey wolf optimization algorithm is significantly better than 6 algorithms, salp swarm algorithm is significantly better than 5 algorithms, whereas cuckoo search algorithm and moth flame algorithm are significantly better than 4 algorithms. Therefore, we conclude that grey wolf optimization algorithm, salp swarm algorithm, moth flame optimization algorithm and cuckoo search algorithm are the four algorithms achieving better optimization results for the CITO problem of all these ten systems under test.

Table 5. Wilcoxon analysis results.

Algorithm	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO	Cases Where p-Value < 0.05
GA	-	0.053	0.722	0.001	0.001	0.958	0.920	0.947	0.239	0.042	3
PSO	0.958	-	0.999	0.001	0.001	0.999	0.976	0.993	0.813	0.528	2
CS	0.313	0.002	-	0.001	0.001	0.858	0.920	0.947	0.096	0.010	4
FA	1.000	1.000	1.000	-	0.002	1.000	0.997	1.000	1.000	0.968	1
BA	1.000	1.000	1.000	0.999	-	1.000	0.997	1.000	1.000	0.976	0
GWO	0.042	0.002	0.171	0.001	0.001	-	0.903	0.784	0.010	0.002	6
MFO	0.097	0.032	0.097	0.005	0.005	0.116	-	0.096	0.065	0.042	4
SSA	0.065	0.010	0.065	0.001	0.001	0.216	0.922	-	0.042	0.014	5
SCA	0.797	0.216	0.922	0.001	0.001	0.993	0.947	0.968	-	0.161	2
HHO	0.968	0.528	0.993	0.042	0.032	0.999	0.968	0.990	0.862	-	2

Combined with the above analysis, moth flame algorithm, salp swarm algorithm, grey wolf optimization algorithm and cuckoo search algorithm algorithms can obtain high-precision solutions for most systems under test. Among them, moth flame algorithm and salp swarm algorithm show good optimization performance for systems with small search space, low dependence, and low complexity. However, they are prone to large optimization errors and unstable performance with the expansion of search space and complex improvement of classes coupling. The optimization performance of grey wolf optimization algorithm and cuckoo search algorithm is exciting for SPM, ATM, ANT, DEOS and Email. With the increment of classes and dependencies, grey wolf optimization algorithm performs more and more prominently. In general, firefly algorithm and bat algorithm have poor optimization performance for the CITO problem and their iterative optimization process is unstable.

4.4.2. Attribute Complexity and Method Complexity

The CITO optimization problem is a multi objective problem with two objectives: attribute complexity and method complexity. According to multi objective researches [15], the linear weights combination approach could not ensure find Pareto front (formed by non-dominated solutions) points of interest. Furthermore, the aggregation fitness function we used is in fact a multiple cost function. Thus, it is impossible to obtain the detailed optimization results of the two single objectives from the normalized weighted average fitness value.

How the two factors, attribute complexity and method complexity, influence the stubbing complexity need to be discussed. In this section, we focus on the attribute complexity A (i, j) and method complexity M(i, j) of test orders generated to find out which algorithms perform better in attribute coupling guided systems and which algorithms perform better in method coupling guided systems.

Observing the coupling distribution information described in Table 2, the systems under test are deemed to be of varying coupling. Some systems have more method coupling

while some have more attribute coupling. In order to further analyze the influence of the two factors on stubbing complexity, the range of attribute complexity and method complexity for a given class integration test order are collected and discussed.

The detailed attribute complexity and method complexity for each system are summarized in Tables 6 and 7, where columns 2–11 are the attribute complexity or method complexity of class integration test orders generated by genetic algorithm, particle swarm optimization algorithm, cuckoo search algorithm, firefly algorithm, bat algorithm, grey wolf optimization algorithm, moth flame algorithm, salp swarm algorithm, sine cosine algorithm and Harris hawk optimization algorithm. Each cell contains the complexity intervals as the optimization results for each system are not consistent across the repeated executions of each algorithm.

Table 6. Attribute complexity values of the ten systems.

System	Stats.	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO
Elevator	min	9	8	9	9	9	9	9	9	9	9
	max	9	9	9	10	11	9	9	9	10	19
	mean	9.00	8.90	9.00	9.30	10.40	9.00	9.00	9.00	9.40	15.70
	Stdev	0.00	0.30	0.00	0.57	0.66	0.00	0.00	0.00	0.40	3.03
	rank	2	1	2	3	5	2	2	2	4	6
SPM	min	52	58	49	60	68	54	45	47	52	63
	max	71	89	72	103	132	76	63	74	111	128
	mean	61.60	68.18	61.20	83.73	100.27	63.13	53.60	56.47	86.36	95.45
	Stdev	5.38	9.21	7.89	12.05	17.89	6.63	4.67	8.27	18.64	19.78
	rank	4	6	3	7	10	5	1	2	8	9
ATM	min	31	30	31	35	44	31	30	30	40	31
	max	51	38	39	52	74	38	34	35	138	149
	mean	36.30	33.40	33.45	43.70	55.00	33.10	31.15	31.35	81.91	70.27
	Stdev	5.52	2.33	2.33	4.26	7.48	1.80	1.18	1.27	37.03	43.29
	rank	6	4	5	7	8	3	1	2	10	9
Daisy	min	0	0	0	1	0	0	0	0	0	0
	max	3	1	1	21	4	2	1	2	23	28
	mean	0.36	0.75	0.50	9.91	1.27	0.40	0.18	0.91	8.45	8.82
	Stdev	0.92	0.46	0.53	5.68	1.27	0.70	0.41	0.70	7.59	10.33
	rank	2	5	4	10	7	3	1	6	8	9
ANT	min	45	48	47	63	55	42	34	37	70	50
	max	66	76	74	95	111	69	55	82	105	124
	mean	57.53	64.4	56.93	73.60	85.07	57.73	42.20	53.73	89.00	91.36
	Stdev	6.20	7.86	7.67	8.62	15.61	7.93	5.21	10.79	8.89	21.10
	rank	4	6	3	7	8	5	1	2	9	10
DEOS	min	6	6	7	14	8	6	6	8	13	10
	max	12	12	13	31	15	11	9	10	29	32
	mean	8.8	8.7	9.1	21.1	12.9	8.9	7.4	8.9	20.5	19.8
	Stdev	1.87	1.89	2.08	6.91	2.08	1.73	1.17	0.88	5.21	7.91
	rank	3	2	5	9	6	4	1	4	8	7
Email	min	5	7	6	8	7	9	7	5	11	11
	max	13	13	10	34	21	13	11	16	57	44
	mean	9.64	9.33	8.57	24.73	14.27	11	8.33	11	28.18	30.64
	Stdev	2.21	2	1.62	8.25	4.47	1.41	1.51	3.5	14.13	9.48
	rank	4	3	2	7	6	5	1	5	8	9
BCEL	min	66	78	53	72	93	65	78	44	116	128
	max	102	100	111	155	140	98	93	106	281	347
	mean	83.72	90.39	86.37	107.73	111.64	78.54	81.91	67	151.36	168.72
	Stdev	8.82	6.05	17.04	25.20	11.74	5.73	13.71	19	21.52	22.73
	rank	4	6	5	7	8	2	3	1	9	10

Table 6. *Cont.*

System	Stats.	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO
DNS	min	69	68	73	113	122	76	70	69	121	154
	max	107	116	112	154	170	100	99	99	502	318
	mean	89.13	92.5	93.2	131.63	140.75	88.88	83	81	218.36	203.27
	Stdev	11.67	15.37	14.62	13.11	17.88	10.25	12	7.3	31.71	49.58
	rank	4	5	6	7	8	3	2	1	10	9
Notepad	min	5	6	5	14	9	5	5	6	10	13
	max	9	12	7	66	20	11	6	13	26	64
	mean	5.91	8.2	6.2	35.64	12.73	7.9	5.36	9.4	17.8	31.71
	Stdev	1.14	2.35	0.84	22.45	3.52	2.51	0.51	2.5	5.18	22.30
	rank	2	5	3	10	7	4	1	6	8	9

Table 7. Method complexity values of the ten systems.

System	Stats.	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO
Elevator	min	17	18	17	20	25	17	17	17	17	27
	max	23	33	25	30	34	24	22	22	32	125
	mean	20.30	22.30	21.50	27.34	28.70	21	18.90	18.60	24.70	76.70
	Stdev	1.87	4.01	2.81	3.75	3.28	2.20	2.12	2.06	4.82	26.8
	rank	3	6	5	8	9	4	2	1	7	10
SPM	min	22	22	20	24	30	21	18	19	26	30
	max	31	32	31	37	48	31	27	28	45	56
	mean	27.33	28.09	25.26	28.93	37.93	25.60	22.06	24.13	35.27	40.73
	Stdev	2.74	2.74	2.89	3.45	4.63	3.02	3.08	2.51	5.29	7.32
	rank	5	6	3	7	9	4	1	2	8	10
ATM	min	11	11	11	13	13	12	11	11	14	13
	max	20	19	15	23	27	15	14	17	45	36
	Stdev	14.05	13.75	13.50	17.65	19.4	13.20	12.70	13.25	24.82	20.09
	mean	2.09	1.89	1.15	2.66	3.93	1.01	0.75	1.33	9.26	8.85
	rank	6	5	4	7	8	2	1	3	10	9
Daisy	min	10	10	9	13	17	7	6	13	48	24
	max	18	21	19	81	34	16	17	22	83	78
	mean	13.82	15.5	14	54	27.09	12.9	12.20	17.46	61.36	47.55
	Stdev	2.48	4.00	2.75	18.12	5.48	3.14	3.28	2.66	13.03	16.40
	rank	3	5	4	9	7	2	1	6	10	8
ANT	min	32	30	33	34	36	30	29	30	47	40
	max	47	42	42	51	70	42	36	40	74	73
	mean	38.27	37.07	36.67	42.60	48.53	36.47	32.53	34.53	58.20	58.09
	Stdev	3.94	3.41	2.79	3.78	8.94	2.88	2.47	2.88	9.07	12.22
	rank	6	5	4	7	8	3	1	2	10	9
DEOS	min	42	48	43	66	55	40	39	37	73	64
	max	60	61	67	126	84	54	53	52	109	118
	mean	52.8	54	50.8	92.9	70.2	48.9	45.5	46.9	90.2	89.9
	Stdev	6.27	4.74	7.31	18.75	8.55	4.18	4.93	4.68	14.71	15.76
	rank	5	6	4	10	7	3	1	2	9	8
Email	min	22	24	22	38	34	26	20	22	71	62
	max	34	39	34	145	57	34	33	46	131	129
	mean	30	32.11	28.57	105.64	46.36	29.33	26.3	34	96.64	102.36
	Stdev	3.61	4.48	4.16	30.96	9.71	3.08	4.32	6.8	20.85	20.28
	rank	4	5	2	10	7	3	1	6	8	9

Table 7. *Cont.*

System	Stats.	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO
BCEL	min	87	92	96	102	109	86	91	89	104	99
	max	109	109	109	121	123	106	109	109	198	241
	mean	91.55	103.28	101.10	113.36	117.45	90.63	105.82	100.72	150.64	160.18
	Stdev	4.85	5.57	4.01	10.99	4.63	5.74	22.32	6.89	26.71	38.37
	rank	2	5	4	7	8	1	6	3	9	10
DNS	min	50	59	53	80	75	52	47	54	92	88
	max	77	84	74	100	114	85	67	79	163	152
	mean	64.25	68.38	61.6	87.75	95.13	62.6	59	66.5	116.64	127.91
	Stdev	8.92	7.39	7.70	6.48	13.82	10.7	7.54	9.4	20.90	21.10
	rank	4	6	2	7	8	3	1	5	9	10
Notepad	min	69	60	73	67	61	53	56	56	80	63
	max	80	82	78	119	94	73	77	80	100	100
	mean	76.10	74.3	76	92.73	82.27	64.1	70.3	66	89.2	90.14
	Stdev	3.59	8.08	1.87	17.24	10.68	7.42	7.48	9.4	7.55	13
	rank	6	4	5	10	7	1	3	2	8	9

Overall, the optimization results of attribute complexity and method complexity are consistent with the coupling distribution information in Table 2. Generally, for systems with more attribute coupling, the attribute complexity value and its interval are large. While for systems with more method coupling, the method complexity value and its interval are large. From the two types of coupling, the characteristics of optimization process for systems with more attribute coupling and more method coupling are analyzed, respectively.

(1) Attribute Coupling Factor Analysis

According to the coupling distribution information in Table 2, the attribute and method coupling of SPM are 462 and 135, respectively, ATM are 283 and 84, respectively, ANT are 585 and 177, respectively, and DNS are 766 and 328, respectively. The number of attributes coupling in these systems far exceeds the number of methods coupling, thus the optimization process of each algorithm is guided mainly by attribute coupling.

Based on the analysis result of stubbing complexity in Section 4.4.1, the first four algorithms are grey wolf optimization algorithm, salp swarm algorithm, cuckoo search algorithm and moth flame algorithm. As for SPM, ATM, ANT, DNS, moth flame algorithm and salp swarm algorithm perform best in these four systems, grey wolf optimization algorithm appears four times in different order, cuckoo search algorithm, particle swarm optimization algorithm, sine cosine algorithm and genetic algorithm appear once, respectively. Additionally, the method complexity intervals of CITO generated by all algorithms for these systems are relatively small. The attribute complexity intervals of moth flame algorithm, salp swarm algorithm and grey wolf optimization algorithm are narrow, while the attribute complexity intervals of salp swarm algorithm in SPM and ANT are wide, indicating the unstable optimization process.

(2) Method Coupling Factor Analysis

Similarly, the attribute and method coupling of Elevator are 21 and 158, respectively, Daisy are 34 and 135, respectively, DEOS are 26 and 223, respectively, and Email are 72 and 222, respectively. The method coupling of these systems are much greater than the attribute coupling, thus the optimization process is greatly affected by method coupling.

For Elevator, Daisy, DEOS and Email, the first four algorithms with excellent performance can be counted. Moth flame algorithm, grey wolf optimization algorithm and cuckoo search algorithm algorithms appear four times, salp swarm algorithm and sine cosine algorithm appear twice, and genetic algorithm appears once. The ranking of grey wolf optimization algorithm is higher than that of optimization process guided by attribute coupling, showing its good optimization ability. As seen from Table 6, there is little difference among attribute complexity intervals of CITO generated by all algorithms.

Further, the method complexity intervals of moth flame algorithm, grey wolf optimization algorithm, cuckoo search algorithm and salp swarm algorithm for these four systems are narrow, as shown in Table 7. Meanwhile, the method complexity intervals of sine cosine algorithm for Daisy, DEOS and Email are wide. Combined with the standard deviation of sine cosine algorithm's fitness function value in Table 4, we can see the instability of its optimization performance.

To sum up, the optimization process guided by the coupling-based fitness function can generate encouraging class integration test orders. When using *OCplx* with $W_A = W_M = 0.5$ to measure the stub cost, the designed fitness function can give a proper balance between attribute complexity and method complexity.

For systems with great attribute coupling, moth flame algorithm, salp swarm algorithm and grey wolf optimization algorithm are recommended. However, the intervals of salp swarm algorithm are wide because of SSA's unstable optimization process.

For systems with great method coupling, moth flame algorithm, grey wolf optimization algorithm, cuckoo search algorithm and sine cosine algorithm are recommended, but the optimization result of sine cosine algorithm is unstable.

4.5. Efficiency Analysis

4.5.1. Convergence Behavior Analysis

Generally, more dependency cycles indicate high coupling among classes. In the context of search-based approaches, optimization involves searching for test orders that minimizes the coupling-based fitness function. Due to the highly non-linear nature of this fitness function, the search space exhibits a curvy, noisy landscape with numerous local minima. In this section, we focus on the ability of these algorithms to avoid getting stuck and discuss how dependency cycles influence the optimization performance.

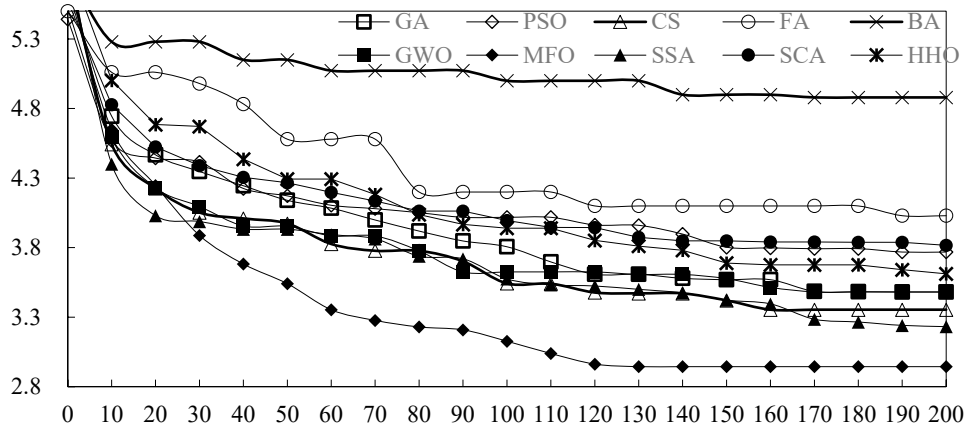
In order to observe the convergence behavior to devise test orders for systems in the presence of dependency cycles, we select the systems with dependency cycles, SPM, ATM, ANT, BCEL and DNS. We collect their stubbing complexity values obtained at the 10th, 20th, ... until 200th iteration, respectively. Then, the average value is calculated across 20 executions. The search history and convergence curve of the optimal results obtained by each algorithm are illustrated in Figure 4. The horizontal axis indicates the number of iterations, and the vertical axis indicates the average value of stubbing complexity.

It can be observed that the stubbing complexity generally decreases with the increasing number of iterations, reflecting the effectiveness of the search-based approach for the CITO problem. Generally speaking, the convergence speed of various algorithm for different systems is affected by many factors. Under the same population size and running environment, the convergence behavior is analyzed from following two aspects: individual encoding and optimization performance.

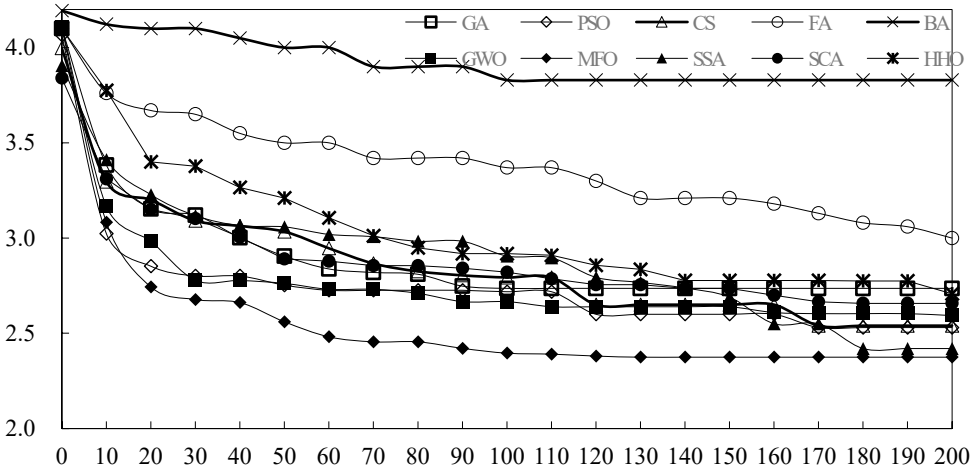
From the perspective of individual encoding strategy, this paper encodes any class integration test order into one individual in one-dimensional space through Algorithm 1, and maps it into CITO through Algorithm 2. The method of using one-dimensional spatial locations to represent individuals reduces the spatial dimension, simplifies the location update process, and reduces the computational complexity. It can be seen that most meta algorithms can gradually improve the quality of solutions and converge faster. However, we also find that this encoding strategy has some limitations. Generally, the search space for the CITO problem increases with the number of classes, so the position coordinate of an individual may be very large. Consequently, it is easy to reduce the precision of the obtained class integration test order because of the large search space.

Illustrated by the optimization trend of these ten algorithms, the convergence speed of bat algorithm and firefly algorithm is comparatively slow, and their convergence curve is relatively flat as a whole. The precision of the optimal solution generated by the firefly algorithm is higher than that of bat algorithm and significantly lower than that of other eight algorithms. The reason behind the slow convergence of the bat algorithm is that the speed term with constant coefficient reduces individual flexibility and population diversity,

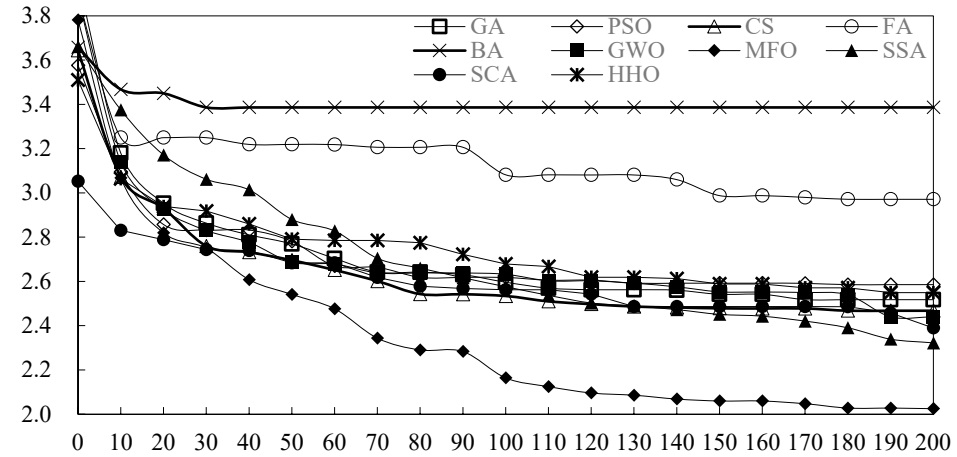
which make the algorithm converge rapidly in early stage but slow in later stage. The pairwise learning mechanism of firefly algorithm causes oscillation of optimization results, especially in the larger search space caused by more classes.



(**a**) SPM

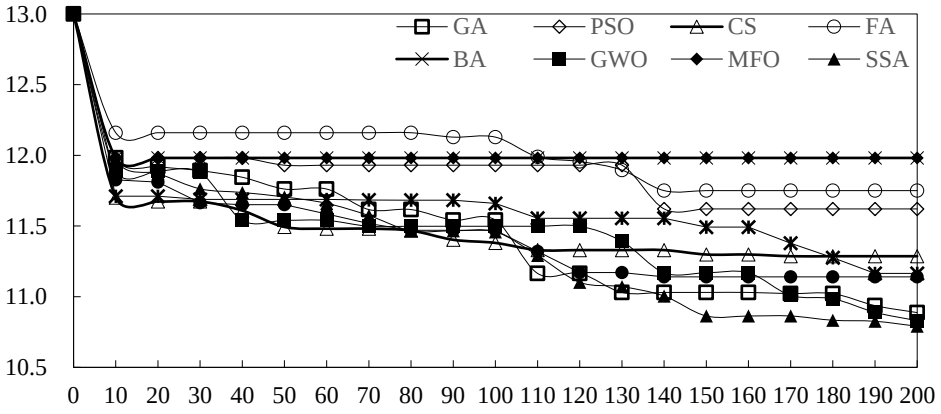


(**b**) ATM

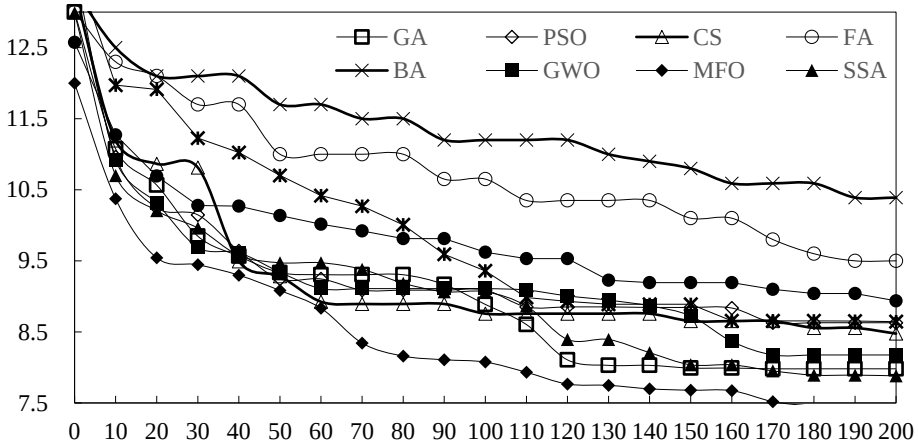


(**c**) ANT

Figure 4. *Cont*.



(**d**) BCEL



(**e**) DNS

Figure 4. Graph for convergence analysis.

The top 4 fastest algorithms with excellent optimization results for these five systems are shown in Table 8. The detailed information is described as follows.

SPM contains 19 classes, 72 dependencies and 1178 cycles. There are abrupt changes in the movement of the search individuals over the first 40 iterations of the firefly algorithm and first the 80 iterations of bat algorithm. The precision of optima solutions obtained by both firefly algorithm and bat algorithm is slightly improved in the later stage. The other eight intelligent algorithms can produce promising stubbing complexity within about 120 iterations, among which moth flame algorithm has the fastest convergence speed with best accuracy. At the early stage, the precision of the optima solutions generated by the moth flame algorithm is much higher than that of other algorithms and precision of optima solutions of the salp swarm algorithm, grey wolf optimization algorithm and cuckoo search algorithm is higher than that of other remaining algorithms. When the same precision of optima solutions is required, the moth flame algorithm needs the least iterations, while the salp swarm algorithm, grey wolf optimization algorithm and cuckoo search algorithm need less iterations than other algorithms. Among all ten algorithms, bat algorithm needs the most iterations and firefly algorithm needs more iterations. The convergence speed of the moth flame algorithm, grey wolf optimization algorithm, cuckoo search algorithm and salp swarm algorithm is consistent with the optimization results in Table 4.

Table 8. The top 4 fastest algorithms with excellent optimization results.

System	Classes	Dependencies	Cycles	Top 4 Fastest Algorithms with Excellent Optimization Results
SPM	19	72	1178	MFO (moth flame optimization algorithm) GWO (grey wolf algorithm) CS (cuckoo search algorithm) SSA (salp swarm algorithm)
ATM	21	67	30	MFO (moth flame optimization algorithm) GWO (grey wolf optimization algorithm) PSO (particle swarm optimization algorithm) SSA (salp swarm algorithm) with slow convergence speed at early stage and fast convergence speed at later stage
ANT	25	83	654	MFO (moth flame optimization algorithm) SCA (sine cosine algorithm) GWO (grey wolf optimization algorithm) SSA (salp swarm algorithm) with slow convergence speed at early stage and fast convergence speed at later stage.
BCEL	45	294	416,091	GWO (grey wolf optimization algorithm) GA (genetic algorithm) SSA (salp swarm algorithm) SCA (sine cosine algorithm)
DNS	61	276	16	MFO (moth flame optimization algorithm) GA (genetic algorithm) SSA (salp swarm algorithm) GWO (grey wolf algorithm)

ATM contains 21 classes, 67 dependencies and 30 cycles. Most algorithms can find promising solutions in the first 80 iterations. Among them, the convergence speed of the moth flame algorithm is the fastest, the Harris hawk optimization algorithm converges more quickly than the firefly algorithm and the bat algorithm, but more slowly than other intelligent algorithms. At the early stage, the three algorithms with the highest precision under the same iterations are the moth flame algorithm, grey wolf optimization algorithm and particle swarm optimization algorithm. When the same precision of optima solutions is required, the iterations required by the moth flame algorithm, grey wolf optimization algorithm and particle swarm optimization algorithm are significantly less those required by other algorithms, while the firefly algorithm and bat algorithm need the most iterations. Additionally, the convergence speed of moth flame optimization algorithm, grey wolf optimization algorithm and particle swarm optimization algorithm is consistent with the optimization results in Table 4. In particular, salp swarm algorithm shows the characteristics of slow convergence speed at early stage and fast convergence speed at later stage.

ANT contains 25 classes, 83 dependencies and 654 cycles. Most algorithms can find high-precision solutions in the first 110 iterations. At the early stage of optimization process, the convergence speed of sine cosine algorithm and grey wolf optimization algorithm is not as fast as moth flame algorithm. Additionally, the convergence speed and precision of solutions obtained by the salp swarm algorithm are higher than firefly algorithm and bat algorithm, lower than other algorithms. At the same iterations, the algorithm with the highest optimization accuracy is the moth flame algorithm. When the same precision is needed, the iterations required by moth flame algorithm is significantly less than that of other algorithms, but the firefly algorithm and bat algorithm need the most iterations. Additionally, the convergence behavior of moth flame algorithm, sine cosine algorithm and grey wolf optimization algorithm is consistent with the optimization results in Table 4. Although the final optimization results of the salp swarm algorithm rank second, the convergence speed of the salp swarm algorithm is slow at the early stage and becomes fast at later optimization stage.

BCEL contains 45 classes, 294 dependencies and 416,091 cycles, which improves the complexity of the system. As can be seen from the convergence curve, the optimization ability of most algorithms in the first 100 iterations is low. Most algorithms can find stable solutions at 190 iterations. The reason behind this behavior is that the dependencies in BCEL are complex and the search space is large, which makes individuals easy to oscillate in the search space. Among all these algorithms, the wolf optimization algorithm, genetic algorithm, salp swarm algorithm and sine cosine algorithm show exciting convergence speed between 100 and 170 iterations, consistent with the optimization results in Table 4.

DNS contains 61 classes, 276 dependencies and 16 cycles. Although the number of classes in DNS is the most among these five systems, the dependencies among classes are weak. It can be seen that most algorithms can find stable solutions at 150 iterations and maintain a good convergence speed at the early stage. Among these ten algorithms, the optimization performance of the moth flame algorithm is outstanding and the convergence speed is the fastest. In the first 110 iterations, the convergence speed of other algorithms is almost the same except the fastest convergence speed of moth flame algorithm and the slow convergence speed of the bat algorithm, firefly algorithm, Harris hawk optimization and sine cosine algorithm. At the later stage of optimization process, the genetic algorithm, salp swarm algorithm and grey wolf optimization algorithm show their excellent search ability, consistent with the optimization results in Table 4.

Based on the above analysis, the convergence speed is affected by the number of classes and dependency cycles. The more the number of classes and cycles are, the more difficult the optimization process is, and the more iterations are required to achieve higher precision of optima solutions. Combing the consistency between convergence speed and optima solutions obtained, algorithms with fast convergence speed can obtain solutions with high accuracy, except for the salp swarm algorithm.

Generally, the salp swarm algorithm shows the characteristics of slow convergence speed at early stage and fast convergence speed at later stage. Based on the analysis of individual update mechanism in the salp swarm algorithm, the leader's update process is affected by the maximum number of iterations. The greater the maximum number of iterations, the slower the leader searches in the early stage. Compared with SPM, BCEL and DNS, the number of classes in ATM and ANT is small with low coupling; thus, their maximum iteration is relatively high, resulting in low convergence at early stage.

4.5.2. Runtime Analysis

To ensure the fairness of the experimental results, each algorithm run independently in the same environment. The actual average runtime of each iteration of these algorithms for each system is recorded and shown in Table 9 and Figure 5. In Figure 5, the horizontal axis indicates the software system and the vertical axis indicates the actual average runtime of each iteration.

Table 9. Average runtime (ms).

System	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO	Average
Elevator	2578	1004	2067	2652	2759	1003	1010	993	1085	779	1593
SPM	5919	2489	4816	6335	8159	2394	2432	2428	2668	1826	3947
ATM	10,503	2948	5723	7653	8199	2918	2905	3027	3225	2268	4937
Daisy	7540	3566	6962	8027	9320	3476	3514	3546	3877	2675	5250
ANT	8426	4189	8011	9422	11,874	4069	4151	4157	4560	3149	6201
DEOS	8323	4225	8037	9678	11,720	4054	4156	4158	4602	3108	6206
Email	20,216	9975	19,495	22,876	28,365	9704	10,061	9985	10,958	7608	14,924
BCEL	26,855	13,232	25,821	29,350	36,527	13,157	13,140	13,181	14,150	10,358	19,577
DNS	81,973	24,306	47,800	60,892	72,791	23,698	24,191	24,964	26,141	17,212	40,397
Notepad	74,445	27,413	52,760	61,349	78,079	26,831	27,334	28,084	29,509	24,475	43,028

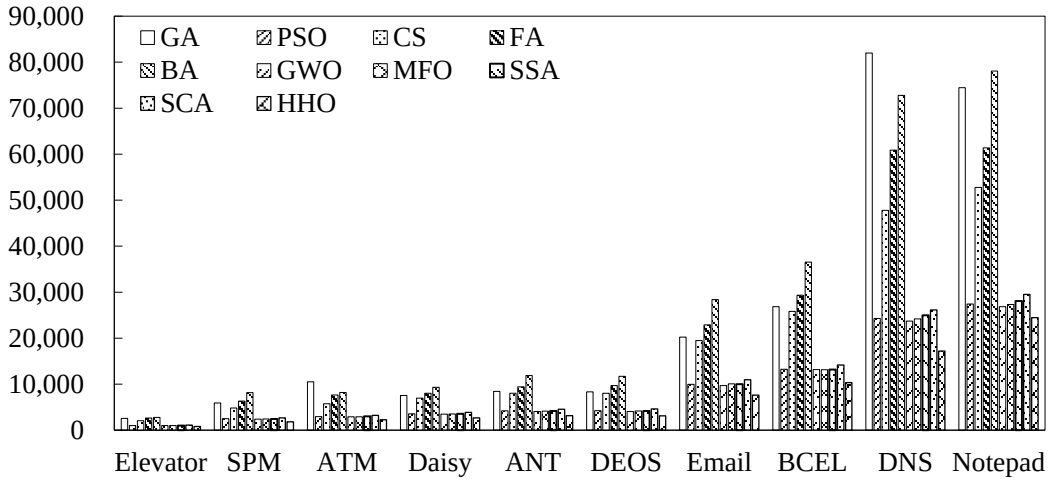


Figure 5. Histogram for average time analysis.

Search-based software testing is the process of automatically generating test data according to a test adequacy criterion using search-based optimization algorithms. The search-based approaches for the CITO problem means that meta heuristic algorithms need to search acceptable class integration test orders with low stubbing complexity among all possible combinations of classes. As we can see from the growth trend of running time, systems with more classes have larger search space which increases the optimization difficulty and requires more time to obtain optima solutions.

Elevator contains 12 classes, with the average runtime of 1593 ms, minimum runtime of 779 ms and maximum runtime of 2759 ms. ATM contains 21 classes, with the average runtime of 4937 ms, minimum runtime of 2268 ms and maximum runtime of 8199 ms. BCEL contains 45 classes, with the average runtime of 19,577 ms, minimum runtime of 10,358 ms and maximum runtime of 36,527 ms. Notepad contains 65 classes, with the average runtime of 41,028 ms minimum runtime of 24,475 ms and maximum runtime of 78,079 ms. It can be seen from the statistical data that the average runtime is directly proportional to the number of classes in the systems. The more classes in system, the more running time is required.

From the perspective of the running time required by the different algorithms, the Harris hawk optimization spends the least runtime for each iteration. The runtime per each iteration of the grey wolf optimization algorithm, moth flame algorithm, particle swarm optimization algorithm, salp swarm algorithm and sine cosine algorithm is roughly the same. Grey wolf optimization algorithm ranked the second least running time for seven systems. Moth flame algorithm ranked the third least runtime for six systems. Therefore, grey wolf optimization algorithm and moth flame algorithm show their obvious time advantages. Additionally, the running time of the cuckoo search algorithm, firefly algorithm, genetic algorithm and bat algorithm is significantly higher than that of other algorithms. The bat algorithm is the algorithm with most runtime per each iteration.

4.5.3. Memory Analysis

Similarly, the memory of each iteration of these algorithms for each system is collected and shown in Table 10 and Figure 6. In Figure 6, the horizontal axis indicates systems under test and the vertical axis indicates the actual average memory of each iteration.

It can be seen from the statistical data that the used memory of generating optimal test orders is rising with the increasing number of classes. That is, more classes in systems require more memory. For systems with almost the same number of classes, the more dependencies there are, the more memory is required.

Table 10. Average memory (KB).

System	GA	PSO	CS	FA	BA	GWO	MFO	SSA	SCA	HHO	Average
Elevator	216	256	236	332	208	264	336	320	332	248	275
SPM	252	312	160	416	244	166	332	360	380	368	299
ATM	148	132	200	364	160	184	348	332	264	388	252
Daisy	196	208	184	360	192	302	328	332	320	456	288
ANT	208	160	580	592	209	268	328	332	328	496	350
DEOS	288	140	220	488	384	360	324	520	320	360	340
Email	176	248	532	640	236	392	332	692	368	436	405
BCEL	612	616	656	604	644	556	544	572	768	476	605
DNS	968	876	956	1084	1324	1152	1088	1392	1296	1248	1138
Notepad	844	524	844	628	372	832	688	856	840	840	727

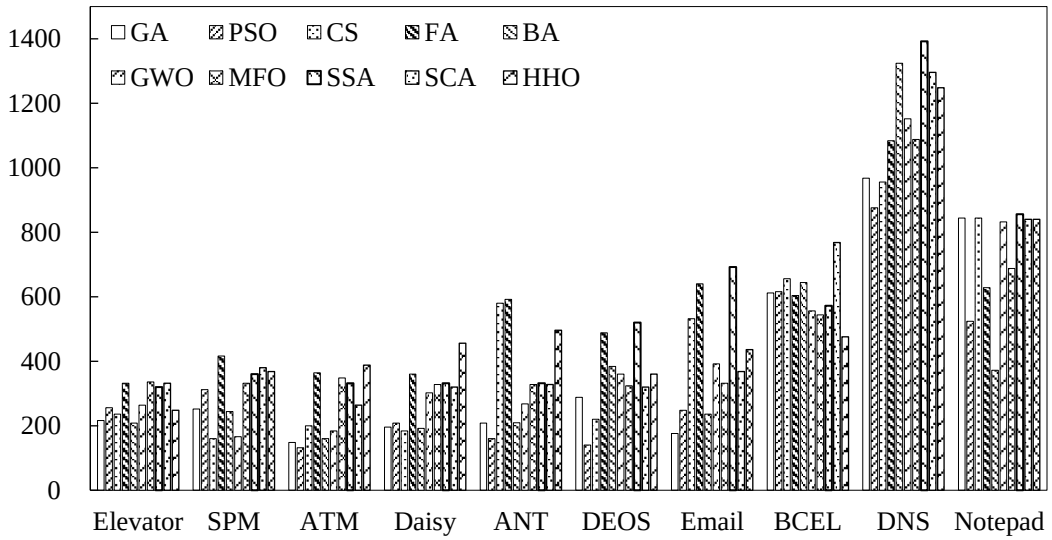


Figure 6. Histogram for memory analysis.

It also can be seen that for all systems, the memory by the genetic algorithm, particle swarm optimization algorithm and bat algorithm is the least. The average memory of the grey wolf optimization algorithm, moth flame algorithm and cuckoo search algorithm is less than that generated by the genetic algorithm, particle swarm optimization algorithm and bat algorithm. The average memory of the firefly algorithm, sine cosine algorithm, salp swarm algorithm and Harris hawk optimization is small.

4.6. Answering the Research Questions

Based on the above experimental results and analysis of the test stubbing complexity, method complexity, attribute complexity, convergence speed, average runtime and memory consumption, the conclusions to the research questions are as follows:

Answer to question 1: Comparatively speaking, the moth flame optimization algorithm, salp swarm algorithm, grey wolf optimization algorithm, cuckoo search algorithm and sine cosine algorithm can obtain class integration test orders with low test stubbing complexity. Among these promising algorithms, the moth flame optimization algorithm and salp swarm algorithm are unstable especially in large search space.

Answer to question 2: The moth flame optimization algorithm, grey wolf algorithm, cuckoo search algorithm and sine cosine algorithm have faster convergence speed, but salp swarm algorithm shows the characteristics of slow convergence at the early stage and fast convergence at the late stage. In terms of computational resource consumption, the moth flame optimization algorithm and grey wolf optimization algorithm requires less runtime and memory. The cuckoo search algorithm requires more runtime and more memory than the moth flame optimization algorithm and grey wolf optimization algorithm.

5. Threats to Validity

The main threats to the experiment validity can be summarized into internal validity and external validity.

Internal validity refers to whether the evaluation measure is appropriate and whether the experiment data is accurate. At present, the stubbing complexity has been widely used to evaluate the performance of various approaches for the CITO problem. For these ten algorithms, the choice of parameter values is another threat to the internal validity. Due to the lack of mathematical characteristic analysis of the CITO problem, we adopt the standard form of these algorithms except genetic algorithm and particle swarm optimization algorithm. At the same time, the time() methods are inserted into each iteration to ensure the accuracy. Furthermore, the fact that the mapping from individual location to the stubbing complexity is limited by data accuracy, not only affects the objectivity of evaluation results, but also limits the scope of application of the method.

The external validity refers to the generalization of experimental conclusion. Although these intelligent optimization algorithms have their own unique performance in the experiment, there is no guarantee that the conclusion can be extended to other systems. We selected benchmark systems with various complexity to verify the performance, but we need to do more work to further verify the effectiveness, including conducting some experiments on systems of other language such as C#, python. Additionally, more datasets need to be selected from different fields or scales for testing to further validate the findings reported in this study.

6. Conclusions

This paper propose a testing framework of meta heuristic algorithms for CITO problem. It analyzes the optimization performance of ten typical algorithms: genetic algorithm, particle swarm optimization algorithm, cuckoo search algorithm, firefly algorithm, bat algorithm, grey wolf optimization algorithm, moth flame algorithm, salp swarm algorithm, sine cosine algorithm, and Harris hawk optimization algorithm.

In general, the bat algorithm and firefly algorithm exhibit less effectiveness and efficiency in CITO compared to the other eight algorithms. For systems with fewer dependencies, the moth flame algorithm, salp swarm algorithm, grey wolf optimization algorithm, and cuckoo search algorithm show their excellent optimization performance. In cases of systems with significant class coupling, the grey wolf optimization algorithm and cuckoo search algorithm demonstrate impressive optimization performance with high precision of optima solutions. However, the moth flame algorithm and salp swarm algorithm display instability.

The coupling-based fitness function achieves a proper balance between attribute complexity and method complexity. For systems greatly affected by attribute coupling, the moth flame algorithm, salp swarm algorithm and grey wolf optimization algorithm are recommended. For systems greatly affected by method coupling, the moth flame algorithm, grey wolf optimization algorithm, cuckoo search algorithm and sine cosine algorithm algorithms are recommended. Among these algorithms, the sine cosine algorithm and salp swarm algorithm are unstable. Furthermore, most algorithms with high-precision optima solutions exhibit faster convergence speed, need less runtime and memory.

This study is helpful in selecting appropriate meta heuristic algorithms to generate class integration test orders, thereby laying a foundation for further scientific research and practical applications. However, certain challenges remain, including issues of lack of accuracy, space explosion caused by encoding mechanism and dependency cycles quoted from related works. In the future, we will focus on coding mechanism improvement, space reduction, adaptive optimization and so on. In addition, the CITO problem is a constrained multi-objective optimization problem. We intend to conduct further Pareto analyses in multi-objective optimization algorithms such as NSGA-II to identify better solutions in future work.

Author Contributions: Conceptualization, W.Z.; Data curation, W.Z. and L.G.; Formal analysis, W.Z.; Funding acquisition, Q.Z.; Investigation, W.Z.; Methodology, W.Z.; Project administration, Q.Z.; Resources, W.Z. and D.Z.; Software, W.Z. and L.G.; Supervision, Q.Z.; Validation, W.Z. and X.G.; Visualization, W.Z.; Writing—original draft, W.Z.; Writing—review and editing, W.Z. and D.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This search was supported in part by the Science and Technology Planning Program of Henan Province, under grant numbers 172102210592 and 212102210417.

Data Availability Statement: The dataset used to support the findings of this study is available from the corresponding author upon request.

Conflicts of Interest: The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

1. Khari, M.; Sinha, A.; Verdu, E.; Crespo, R.G. Performance Analysis of Six Meta-heuristic Algorithms over Automated Test Suite Generation for Path Coverage based Optimization. *Soft Comput.* **2021**, *24*, 9143–9160. [CrossRef]
2. Le Hanh, V.; Akif, K.; Le Traon, Y.; Jézéque, J.M. Selecting an Efficient OO Integration Testing Strategy: An Experimental comparison of Actual Strategies. In Proceedings of the 15th European Conference on Object-Oriented Programming, Budapest, Hungary, 18–22 June 2001; pp. 381–401.
3. Jiang, S.J.; Zhang, M.; Zhang, Y.M.; Wang, R.; Yu, Q.; Keung, J.W. An Integration Test Order Strategy to Consider Control Coupling. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1350–1367. [CrossRef]
4. Kung, D.; Gao, J.; Hsia, P.; Toyoshima, Y.; Chen, C. A Test Strategy for Object Oriented Programs. In Proceedings of the 19th Annual International Computer Software and Applications Conference, Dallas, TX, USA, 9–11 August 1995; pp. 239–244.
5. Zhang, M.; Keung, J.W.; Chen, T.Y.; Xiao, Y. Validating Class Integration Test Order Generation Systems with Metamorphic Testing. *Inf. Softw. Technol.* **2021**, *132*, 106507. [CrossRef]
6. da Veiga Cabral, R.; Pozo, A.; Vergilio, S.R. A Pareto Ant Colony Algorithm Applied to the Class Integration and Test Order Problem. In Proceedings of the esting Software & Systems-ifip Wg 61 International Conference, Natal, Brazil, 8–10 November 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 16–29.
7. Hussain, K.; Mohd, M.S.; Cheng, S.; Shi, Y. Metaheuristic Research: A Comprehensive Survey. *Artifical Intell. Rev.* **2019**, *52*, 2191–2233. [CrossRef]
8. Tai, K.C.; Daniels, F.J. Interclass Test Order for Object-Oriented Software. In Proceedings of the 21st International Computer Software and Applications Conference, Washington, DC, USA, 11–15 August 1997; pp. 602–607.
9. Traon, Y.L.; Jezequel, J.M. Efficient Object-Oriented Integration and Regression Test. *IEEE Trans. Reliab.* **2000**, *49*, 12–25. [CrossRef]
10. Briand, L.C.; Labiche, Y.; Wang, Y. An Investigation of Graph-Based Class Integration Test Order Strategies. *IEEE Trans. Softw. Eng.* **2003**, *29*, 594–607. [CrossRef]
11. Zhang, M.; Keung, J.; Xiao, Y.; Kabir, M.A.; Feng, S. A Heuristic Approach to Break Cycles for the Class Integration Test Order Generation. In Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference, Milwaukee, WI, USA, 15–19 July 2019; pp. 47–52.
12. Briand, L.; Feng, J.; Labiche, Y. *Experiment with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders*; Technical report SCE-02-03; Carleton University: Ottawa, ON, Canada, 2002.
13. Borner, L.; Paech, B. Integration Test Order Strategies to Consider Test Focus and Simulation Effort. In Proceedings of the International Conference on Advances in System Testing and Validation Lifecycle, Porto, Portugal, 20–25 September 2009; pp. 80–85.
14. Vergilio, S.R.; Pozo, A.; Árias, J.C.G.; Cabral, R.d.V.; Nobre, T. Multi-objective Optimization Algorithms Applied to the Class Integration and Test Order Problem. *Softw. Tools Technol. Transf.* **2012**, *14*, 461–475. [CrossRef]
15. Mariani, T.; Guizzo, G.; Vergilio, S.R.; Pozo, A.T. Grammatical Evolution for the Multi-Objective Integration and Test Order Problem. In Proceedings of the Genetic and Evolutionary Computation Conference, Denver, CO, USA, 20–24 July 2016; pp. 1069–1076.
16. Czibula, G.; Czibula, G.; Marian, Z. Identifying Class Integration Test Order Using an Improved Genetic Algorithm based Approach. In Proceedings of the International Conference on Software Technologies, Madrid, Spain, 24–26 July 2018; pp. 163–187.
17. Zhang, Y.M.; Jiang, S.J.; Chen, R.Y.; Wang, X.Y.; Zhang, M. Class Integration Testing Order Determination Method based on Particle Swarm Optimization Algorithm. *Chin. J. Comput.* **2018**, *41*, 931–945.
18. Zhang, Y.N.; Jiang, S.J.; Zhang, Y.M. Approach for Generating Class Integration Test Sequence based on Dream Particle Swarm Optimization Algorithm. *Comput. Sci.* **2019**, *46*, 159–165.
19. Zhang, M.; Keung, J.W.; Xiao, Y.; Kabir, M.A. Evaluating the Effects of Similar Class Combination on Class Integration Test Order Generation. *Inf. Softw. Technol.* **2021**, *129*, 106438. [CrossRef]
20. Zhang, B.Q.; Fei, Q.; Wang, Y.C.; Yang, Z. Study on Integration Test Order Generation Algorithm for SOA. *Comput. Sci.* **2022**, *49*, 24–29. [CrossRef]

21. Zhang, W.N.; Zhou, Q.L.; Jiao, C.Y.; Xu, T. Hybrid Algorithm of Grey Wolf Optimizer and Arithmetic Optimization Algorithm for Class Integration Test Order Generation. *Comput. Sci.* **2023**, *50*, 72–81.
22. Harman, M.; Mcminn, P. A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. *IEEE Trans. Softw. Eng.* **2010**, *36*, 226–247. [CrossRef]
23. Harman, M.; Yue, J.; Zhang, Y. Achievements, Open Problems and Challenges for Search Based Software Testing. In Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation, Graz, Austria, 13–17 April 2015.
24. Khari, M. An extensive evaluation of search-based software testing: A review. *Soft Comput.* **2019**, *23*, 1933–1940. [CrossRef]
25. Zhang, Y.; Jiang, S.; Zhang, M.; Ju, X. Survey of Class Test Order Generation Techniques for Integration Test. *Chin. J. Comput.* **2018**, *41*, 670–694.
26. Zhang, M.; Jiang, S.J.; Zhang, Y.M. Research on Multi-objective optimization in Class Integration Test Order. *J. Chines Comput. Syst.* **2017**, *38*, 1772–1777.
27. Khari, M.; Kumar, P.; Burgos, D.; Crespo, R.G. Optimized Test Suites for Automated Testing Using Different Optimization Techniques. *Soft Comput.* **2018**, *22*, 8341–8352. [CrossRef]
28. Kennedy, J.; Eberhart, R. Particle swarm optimization. In Proceedings of the IEEE International Conference on Neural Networks, Perth, Australia, 27 November–1 December 1995; pp. 1942–1948.
29. Yang, X.S.; Deb, S. Cuckoo Search via Levy Flights. In Proceeding of the World Congress on Nature and Biologically Inspired Computing (NaBIC2009), Coimbatore, India, 9–11 December 2009; pp. 210–214.
30. Yang, X.S. *Nature-Inspired Metaheuristic Algorithms*; Luniver Press: London, UK, 2010; pp. 81–89.
31. Yang, X.S. A New Metaheuristic Bat-inspired Algorithm. *Comput. Knowl. Technol.* **2010**, *284*, 65–74.
32. Mirjalili, S.; Mirjalili, S.M.; Lewis, A. Grey Wolf Optimizer. *Adv. Eng. Softw.* **2014**, *69*, 46–61. [CrossRef]
33. Mirjalili, S. Moth-flame Optimization Algorithm: A Novel Nature Inspired Heuristic Paradigm. *Knowl.-Based Syst.* **2015**, *89*, 228–249. [CrossRef]
34. Mirjalili, S. SCA: A Sine Cosine Algorithm for Solving Optimization Problems. *Knowl.-Based Syst.* **2016**, *96*, 120–133. [CrossRef]
35. Mirjalili, S.; Gandomi, A.H.; Mirjalili, S.Z.; Saremi, S.; Faris, H.; Mirjalili, S.M. Salp Swarm Algorithm: A Bio-Inspired Optimizer for Engineering Design Problems. *Adv. Eng. Softw.* **2017**, *114*, 163–191. [CrossRef]
36. Heidari, A.A.; Mirjalili, S.; Faris, H.; Aljarah, I.; Mafarja, M.; Chen, H. Harris hawks optimization: Algorithm and applications. *Future Gener. Comput. Syst.* **2019**, *97*, 849–872. [CrossRef]