The Cuckoo Search and Integer Linear Programming Based Approach to Time-Aware Test Case Prioritization Considering Execution Environment

Yu Wong[1,2], Hongwei Zeng[1], Huaikou Miao[1,2], Honghao Gao[1,3], and Xiaoxian Yang[4](✉)

[1] School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China
{joye_wong, zenghongwei, hkmiao, gaohonghao}@shu.edu.cn
[2] Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai 200444, China
[3] Computing Center, Shanghai University, Shanghai 200444, China
[4] School of Computer and Information Engineering, Shanghai Polytechnic University, Shanghai, China
xxyang@sspu.edu.cn

Abstract. Regression testing plays an important role in software development process. The more mature software system development is, the greater the proportion of regression testing during software life cycle takes. To this point, test case prioritization techniques are proposed to detect more faults as early as possible and improve the effectiveness of regression testing. However, it is often performed in a time constrained execution environment. This paper introduces a new method of time-aware test case prioritization. First of all, it takes advantage of the cuckoo search algorithm to reorder test suite. Then, integer linear programming model is employed to test selection in light of time budget. At last, a novel fitness function is designed focusing on code coverage that from method-call information perspective. Experimental results show that our method improves the effectiveness of fault detection compared with traditional fault detection techniques especially time is constrained.

Keywords: Test case prioritization · Time-aware · Cuckoo search

1 Introduction

Regression testing takes up a large proportion of workload during the software testing process, which should be completely executed in all stages of software development. It is expensive in most circumstances. As we known, researcher had stated that a test suite of software having 20,000 lines of code requires 7 weeks to run and check [1]. As a solution, test case prioritization (TCP) is an effective solution that aims to make test suite detect more faults as early as possible. Therefore, kinds of TCP strategies have been published, such as greedy algorithm, meta-heuristic algorithm. However, effective

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
H. Gao et al. (Eds.): CollaborateCom 2018, LNICST 268, pp. 734–754, 2019.
https://doi.org/10.1007/978-3-030-12981-1_51

meta-heuristic methods are not always applicable to TCP, requiring making related adjustment according to different test scenario.

Most of the existing approaches to prioritization did not incorporate a testing time budget. Li et al. [2] firstly applied hill climbing and genetic algorithm (GA) to the TCP. Though their meta-heuristic searches algorithms have no difference with the additional greedy algorithm in performance, time constraint is still not incorporated in their proposed algorithms. Walcott et al. [3] used the GA to reorder test suites in light of testing time constraints. But the effectiveness of their methods has yet to be verified. Cuckoo Search (CS) is a relatively new meta-heuristic algorithm proposed in 2009 [4, 5]. It aims to effectively solve the optimization problem by simulating parasitic brooding behavior of cuckoo. Many experimental studies [4, 5] have proved that CS is more effective than other optimization algorithms such as particle swarm optimization (PSO) algorithm. However, there is no valuable literature to demonstrate the superiority of the CS algorithm in TCP. Some existing CS approaches in testing [6, 7] have been applied for TCP, but the effectiveness of their approaches was not proved by experiments.

In response to the above issues, the TCP problem is transformed into finding an optimal solution of the test case prioritization in our method, and the performance of the method is verified by multi-level comparison experiments. Thus, this paper presents a test case prioritization technique that combine CS algorithm and the integer linear programming (ILP). The ILP is applied to selecting test cases when time is limited. A new fitness function based on code coverage and method-call information is designed to solve the multi-object problem. Through empirical evaluation, the parameter combination of CS algorithm is optimized. Then our approach is evaluated and compared with previous TCP techniques. The experimental results demonstrate that our proposed TCP algorithm detect faults more and earlier than traditional techniques and other heuristic-based prioritization algorithms especially time is constrained.

The rest of the paper is organized as follows. Section 2 discusses related work for test case prioritization techniques. Section 3 describes how to combine the integer linear programming with cuckoo search algorithm to solve the time-aware test case prioritization problem and introduces a new fitness function. Section 4 describes the design, results, and analysis of experiments. Section 5 comes conclusion and future work.

2 Related Work

Techniques for test case prioritization aim to improve the rate of fault detection through reordering test cases for execution. Many TCP strategies have been raised, the existing TCP techniques are classified into three categories: source code, requirement, and model. The TCP technologies based on source code are also divided into greedy algorithm, machine learning method, fusion expert method and others. This section describes other exiting relevant contributions in TCP strategies.

Rothermel et al. [8] firstly proposed the complete definition of the TCP problem. They empirically evaluated several test case prioritization techniques such as statement coverage prioritization, function coverage prioritization and so on. These researches

focused on code-coverage TCP methods,and had been relatively mature [8, 9]. In 2001, Elbaum [10] conducted specific research for TCP metrics, including APFD and APFDc. APFD metric proclaims that all faults have the same severity and all test cases have equal costs. APFDc considers test case costs and fault severities. Their study primarily focused on white-box testing but not black-box testing.

In addition, the multi-objective is also gradually used in the TCP problem. Islam et al. [11] proposed a multi-objective test case prioritization approach based on latent semantic indexing, that is the method of information retrieval (IR). Saini and Tyagi [12] proposed a multi-objective test case prioritization algorithm (MTCPA) which is based on two objective functions. The objective functions include test case execution time used in the GA and statement coverage. The proposed method has been compared with different prioritization techniques to find the optimal solution. Experimental results showed that the proposed algorithm returns a test case suite with maximum fault coverage and minimum execution time with maximum APFD criteria as the solution.

There are many other meta-heuristic search algorithms applied to solve multi-objective TCP problem. Schultz and Radloff [13] were committed to combining PSO algorithm with multi-target TCP problems. However, they compared their own approaches with just Random by experiments. The credibility of the effectiveness of their approaches seemed not strong enough. Cuckoo Search (CS) is proposed by Yang and Deb [4, 5] in 2009. Many researches of various fields have proved that CS algorithm shows greater effectiveness than other heuristics. For example, CS was extended to solve multi-objective problems in [5]. In specific, the proposed MOCS (multi-objective cuckoo search) was tested on a subset of well-chosen test functions. And it performs better for almost all these test problems in comparison with the PSO algorithm and the GA. A few researchers have attempted to use the cuckoo algorithm in test field. Nagar et al. [6] proclaimed that they use the CS algorithm in TCP. But actually they applied the CS algorithm to test case selection, instead of the test case prioritization. Prior to them, Srivastava et al. [7] also applied CS algorithm to TCP, but the algorithm converged slowly and the effectiveness of their approach was not proved by experiments. Thus, the CS algorithm is applied to solve the time-constrained TCP problem, and its effectiveness in fault detection is verified by experiments in this paper.

In recent years, test Case Prioritization has also made progress in other aspects. During 2016, Alves and Machado et al. [14] proposed another novel refactoring-based approach, which reordered an existing test sequence utilizing a set of refactoring fault models. This approach promoted early detection of refactoring faults. Eghbali and Tahvildari [15] proposed a new heuristic for breaking ties in coverage based techniques using the notion of lexicographical ordering. This technology is a positive solution to the problem of how to choose a better one once the test cases cover the same number of statement. In 2017, Lachmann et al. [16] introduced a technique for test case prioritization of manual system-level regression testing based on supervised machine learning, and used SVM Rank to evaluate their approach by means of two subject systems. Kim et al. [17] proposed a test case prioritization method based on failure history data.

3 Time-Aware Test Case Prioritization

Test suite usually could not be run in a limited time, so the test cases that detect more faults should be run as much as possible. Although faults could not be predicted, test cases that cover more code usually have more potential to detect more faults from the perspective of code coverage. Therefore, these test cases that cover more code need to be selected to run when time is constrained. Among them, the test case that covers more code should be executed earlier. The test case prioritization problem is incorporated with time budget, and it is called time-aware test case prioritization problem. According to Walcott et al. [3], this problem is defined as follows.

Definition 1 (Time-aware Test Case Prioritization):

Given: A test suite T, the set of permutations of T's powerset PT, the time budget $Time_{max}$, and two functions from PT to the real numbers *fit*() and *Time*().

Problem 1: Find the test tuple $\sigma_{max} \in$ PT such that $Time(\sigma_{max}) \leq Time_{max}$ and $\forall\sigma \in$ PT where $\sigma_{max} \neq \sigma$, $Time(\sigma) \leq Time_{max}$ and $fit(\sigma) \leq fit(\sigma_{max})$.

In Definition 1, PT represents the collection of all possible tuples and subtuples of T. The time budget $Time_{max}$ is expected limited test case execution time in actual test scenario. The execution time of each test case is obtained in our experiments in Sect. 4. $Time_{max}$ is simulated by the percentage of total execution time of all test cases. The test tuple σ contains several test cases, *Time*(σ) is total execution time of them. The function *fit*() is usually designed according to fault-related code information, requirement information and so on. It measures the potential fault detection capabilities of test tuples.

Table 1. Test suite and faults exposed.

Test cases (execution time)	Faults									
	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
T_1 (9 min)	X	X		X	X		X	X		X
T_2 (1 min)	X									
T_3 (3 min)	X				X					
T_4 (4 min)		X	X				X			
T_5 (4 min)				X		X			X	

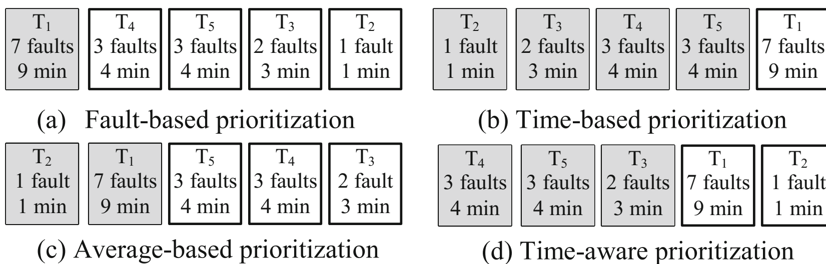


Fig. 1. Prioritizations based on different principles.

Consider an example program with 10 faults and a test suite of 5 test cases, with the faults detecting abilities as shown in Table 1. Meanwhile, the time budget is 12 min. As illustrated in Fig. 1, if test cases are prioritized based on faults, the execution order of them is <T_1> and 7 faults are detected within 9 min. If test cases are prioritized based on time, the execution order of them is <T_2, T_3, T_4, T_5> and 8 faults are detected within 12 min. If test cases are prioritized based on average time, the execution order of them is <T_2, T_1> and 7 faults are detected within 10 min. Time-aware test case prioritization problem aims to design an intelligent method to find a great execution order that detect more faults in less time. For example, in Fig. 1(d), executing test cases in <T_4, T_5, T_3> detect 8 faults within 11 min.

Moreover, definition 1 is separated into the following two definitions to design an intelligent technique in this paper.

Definition 2 (Test Case Selection in Time Budget):

Given: A test suite T, the time budget $Time_{max}$, and two functions from T to the real numbers *fit*1() and *Time*().

Problem 2: Find the subset $T'_{max} \subset T$ such that $Time(T'_{max}) \leq Time_{max}$ and $\forall T' \subset T$ where $T'_{max} \neq T'$, $Time(T') \leq Time_{max}$ and $fit1(T') \leq fit1(T'_{max})$.

The function *fit*1() in Problem 2 also be designed by known fault-related code information, requirement information, or others. In experiments, method coverage information is considered in *fit*1(). More specifically, the integer linear programming (ILP) is used for test case selection in time budgets and it will be elaborated in Sect. 3.1.

Definition 3 (Test Case Prioritization of Test Subset):

Given: A subset obtained by solving Problem 2 T'_{max}, the collection of T'_{max}'s all possible tuples and sub tuples PT, and a function from PT to the real numbers *fit*2().

Problem 3: Find the test tuple $\sigma_{max} \in PT$ such that $\forall \sigma \in PT$ where $\sigma_{max} \neq \sigma$ and $fit2(\sigma) \leq fit2(\sigma_{max})$.

In Definition 3, the fitness function *fit*2() is designed by code information and execution time. It helps us to find the test tuple that detect more faults as soon as possible and is referred to in Sect. 3.3.

3.1 ILP-Based Test Case Selection Introduction

For the time-aware test case prioritization problem, test case selection in time budget should be first solved to obtain an optimal test case subset that run in limit time. In our software test environment, execution time of test cases and the specific methods covered by each test case are available. The test case set, execution time, and covered method meet a corresponding linear relationship. In order to select a test subset covering more methods under certain linear constraints (Eqs. 5 and 6), a linear programming model is undoubtedly suitable. In this section, test case selection in time budget is viewed as a 0/1 integer linear programming (ILP) problem to solve as follows [18].

Suppose that test suite T contains n test cases $\{T_1, T_2, \ldots, T_n\}$, the execution time of each test case is *time*(T_i). The code unit set of the program is denoted as C = $\{c_1, c_2, \ldots, c_m\}$. A code unit may be a method, a class, or a statement block. The problem is to select a subset $T'_{\max}$ that not only covers the biggest code unit subset of C but also the sum execution time of it does not exceed the time budget.

Let Boolean variable x_i ($1 \leq i \leq n$) indicates if test case T_i is selected or not, and Boolean variable s_{ij} is defined as whether a test case T_i covers c_j. So, for the set of code units, m Boolean variables y_i ($1 \leq i \leq m$) are used to represent whether code unit c_j is covered by at least one test case. The problem is reduced to the 0/1 ILP described as follows:

0/1 variables:

$$x_i = \begin{cases} 1, & \text{if } T_i(1 \leq i \leq n) \text{ is selected} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$S_{ij} = \begin{cases} 1, & \text{if } T_i \text{ covers } c_j \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$y_j = \begin{cases} 1, & \text{if one or more selected test cases cover } c_j \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

A linear function to be maximized:

$$\max \sum_{j=1}^{m} y_j \tag{4}$$

Problem constraints:

$$\sum_{i=1}^{n} time(T_i) * x_i \leq Time_{\max} \tag{5}$$

$$\sum_{i=1}^{n} s_{ij} * x_i \geq y_j \quad (i < j < m) \tag{6}$$

In Eq. 4, the linear function $\sum_{j=1}^{m} y_j$ amounts to the function *fit*1() in Problem 2, and it aims to pick out a subset of test cases that cover the most code in finite time $Time_{\max}$. The *time*(T_i) is the execution time of the test case T_i. Although this test subset has reached the maximum code coverage, there may still have spare time to run unselected test cases after executing the selected ones. In order to detect more faults, the remaining unchosen test cases need to be further selected according to the remaining time. For further selection, the total number of methods that are covered by

each test case is calculated, then select the maximum coverage test case that run in the remaining time. The optimal subset of test suite T is obtained by the ILP and further selection.

3.2 Test Case Prioritization Using CS

The suitable test case subset that could be executed in a limited time has been selected, but many faults detected by the test cases are duplicated. If the test case set is not prioritized, a lot of time will still be wasted. Thus, the test case subset that have potential to detect more faults should be run first. This TCP problem is absolutely considered as the global optimized problem that solved by Cuckoo Search (CS) algorithm. CS is a meta-heuristic algorithm that effectively solve optimization problem by simulating the parasitic brooding behavior of certain species of cuckoo. Based on the fitness value of each iteration, a fraction of worse nests will be abandoned, and each generation infinitely close to an optimized solution by replacing the solutions to the better ones [4, 5]. The optimal solution is based on the fitness function which measures the potential fault detect ability of the test sequence.

For simplicity in describing the CS, CS usually uses the following three idealized rules: (a) Each cuckoo lays one egg at a time, and dumps it in a randomly chosen nest; (b) The high-quality eggs will be carried over to the next generations; (c) The number of available host nests is fixed, and the egg laid by a cuckoo is discovered by the host bird with a probability p_{a}. In this case, the host bird either get rid of the egg, or simply abandon the nest and build a completely new nest [4, 5].

According to Yang [4], based on above three rules, the basic steps of the cuckoo search algorithm for TCP are described as Algorithm 1.

Algorithm 1 CS Algorithm for TCP [4]

1: **Begin**
2: *Generate initial test tuple population* (n *host nests* $\boldsymbol{x}_i$ ($i = 1,2, \ldots, n$))
3: **while** ($t < MaxGeneration$)
4: *Get a test tuple(cuckoo)* $\boldsymbol{x}_i$ *randomly by Lévy flights and evaluate it;*
5: *Choose a test tuple (nest) among n* (*say*, j) *randomly;*
6: **if** ($f(\boldsymbol{x}_i) > f(\boldsymbol{x}_j)$)
7: *replace* $\boldsymbol{x}_j$ *by the new solution* $\boldsymbol{x}_i$*;*
8: **end if**
9: *Rank the solutions and find the current best;*
10: *A fraction* (p_a) *of worse nests are abandoned and new ones are built to replace them;*
11: *Keep the best solutions (or nests with quality solutions);*
12: **end while**
13: *Obtain the best solution.*
14: **End**

In CS algorithm for TCP, the initial population is generated randomly from the permutation of the test subset obtained by ILP. Each d-dimensional vector $\boldsymbol{x}_i$ called test tuple represents one of the ordering results of d test cases. The parameter t represents the t-th generation of nest transformation and *MaxGeneration* is set to terminated the algorithm. The objective function $f(\boldsymbol{x}_i)$ is just *fit*2() in Problem 3 and will be defined in Sect. 3.

In the loop, Lévy flights is the core of the CS, and it is performed to generate new solutions. For a nest $\boldsymbol{x}_i^{(t)}$, a Lévy flight is described as follows:

$$\boldsymbol{x}_i^{(t+1)} = \boldsymbol{x}_i^{(t)} + \boldsymbol{\alpha} \oplus \text{Lévy}(\lambda) \tag{7}$$

In (7), $\boldsymbol{\alpha}$ is the step size, in this paper $\boldsymbol{\alpha} = 1$ is setted here. The product $\oplus$ means entrywise multiplication. In test case prioritization, a test tuple is a sequence of test cases (non-numerical values), so the location numbers of test cases in a test tuple are used to form a nest. Suppose vector $\mathbf{I}$ marks the initial locations of test cases in test tuple $\boldsymbol{x}_i$, a Lévy flight of $\boldsymbol{x}_i$ is defined as

$$\mathbf{I}^{(t+1)} = \mathbf{Hash}(\mathbf{I}^{(t)} + \boldsymbol{\alpha} \oplus \text{Lévy}(\lambda)) \tag{8}$$

In (8), $\boldsymbol{\alpha} \oplus$ Lévy(λ) creates a new d-dimensional vector which elements are d random integers based on Lévy distribution. To get these d random numbers, Levy Random Number Generator (LRNG) is designed. The final results of $\mathbf{I}^{(t+1)}$ are a set of integers between 0 to d got by modulo operation. These d integers represent locations, and they are highly likely to cause hash conflicts. Hash function is applied to solve this hash conflict. It computes new sequence numbers of test cases in $\boldsymbol{x}_i$, and quadratic probing method to solve conflict. At last, a new vector $\mathbf{I}^{(t+1)}$ is obtained and a new test tuple $\boldsymbol{x}_i^{(t+1)}$ generates according to $\mathbf{I}^{(t+1)}$.

To better elaborate, an example is given to explain the specific process of Lévy flight. In Fig. 2(a), the original test tuple $\boldsymbol{x}_i^{(t)} = (x_0, x_1, x_2, x_3, x_4)^{\mathrm{T}} = (T_3, T_4, T_1, T_5, T_2)^{\mathrm{T}}$, so $\mathbf{I}^{(t)} = (2, 4, 0, 1, 3)$, elements in I are the location of test cases in the tuple. Then update the $\mathbf{I}^{(t)}$ with the Lévy vector generated by LRNG in Fig. 2(b), but there are conflicts in the new $\mathbf{I}^{(t)}$ like "3" and "1". Then quadratic probing method is used to solve hash conflicts in Fig. 2(c). In addition, the element "n_i" in Fig. 2(c) means the i-th repetition of value "n". At last, a new subscripts vector $\mathbf{I}^{(t+1)}$ is got without conflict and the new test tuple vector $[T_5, T_4, T_3, T_1, T_2]$ is obtained according to $\mathbf{I}^{(t+1)}$ as shown in Fig. 2(d).

Original test tuple:

$$\boldsymbol{x}_i^{(t)} = (x_0, x_1, x_2, x_3, x_4)^{\mathrm{T}} = \begin{bmatrix} T_3 \\ T_4 \\ T_1 \\ T_5 \\ T_2 \end{bmatrix} \quad \Rightarrow \quad \mathbf{I}^{(t)} = \begin{bmatrix} 2 \\ 4 \\ 0 \\ 1 \\ 3 \end{bmatrix}$$

(a) Original test tuple

$$\boxed{\begin{matrix} \mathbf{L} \\ \mathbf{R} \\ \mathbf{N} \\ \mathbf{G} \end{matrix}} \oplus \alpha \rightarrow \mathbf{L} = \begin{bmatrix} 5 \\ 1 \\ 3 \\ 4 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{I}^{(t)} + \mathbf{L} = \begin{bmatrix} 2 \\ 4 \\ 0 \\ 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 5 \\ 1 \\ 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 3_1 \\ 1_1 \\ 0 \end{bmatrix}$$

(b) Test case location update

Hash($\mathbf{I}^{(t)}$ + L) and Quadratic probing:

0	1	2	3	4
0	1	1_1	3	3_1

(c) Quadratic probing

New test tuple:

$$\mathbf{I}^{(t+1)} = \begin{bmatrix} 3 \\ 1 \\ 4 \\ 2 \\ 0 \end{bmatrix} \qquad \boldsymbol{x}_i^{(t+1)} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} T_5 \\ T_2 \\ T_4 \\ T_1 \\ T_3 \end{bmatrix}$$

(d) Lévy test tuple

Fig. 2. Create new test tuple by the Lévy flight.

3.3 Fitness Function Design

In meta-heuristic algorithm, fitness function play an important role. To use the cuckoo search algorithm to search the best test tuple, a new fitness function of the CS is designed for TCP algorithm.

$$Fitness(\sigma, t_c, \omega_1, \omega_2) = \omega_1 F_{cover}(\sigma, t_c) + \omega_2 F_{mc}(\sigma) \tag{9}$$

The fitness of a test tuple σ is evaluated by code coverage information and method-call information, and $\omega_1, \omega_2 \in [0, 1]$ are weights that satisfy $\omega_1 + \omega_2 = 1$. In which, the code coverage information is obtained on a given test coverage adequacy criteria t_c.

$F_{cover}(\sigma, t_c)$ is related to code coverage, giving precedence to test tuples which cover more code earlier. F_{cover} is calculated in two parts: $F_{c\text{-}actual}$, $F_{c\text{-}max}$. First, $F_{c\text{-}actual}$ is computed by summing the products of execution time $time(T_i)$ and the code coverage of the sub tuple $\sigma_{\{1,i\}}$. In which, the sub tuple $\sigma_{\{1,i\}} = <T_1, T_2, \ldots, T_i>$ for each test case $T_i \in \sigma$. Formally, for each σ,

$$F_{c\text{-}actual}(\sigma, t_c) = \sum_{i=1}^{|\sigma|} \left\{ time(\langle T_i \rangle) * ccover(\sigma_{\{1,i\}}, t_c) \right\} \tag{10}$$

$F_{c\text{-}max}$ represents the maximum code coverage of the test tuple σ at maximum time limit (i.e. the maximum value that $F_{c\text{-}actual}$ takes). For each σ,

$$F_{c\text{-}max}(\sigma, t_c) = ccover(\sigma, t_c) * \sum_{i=1}^{n} time(\langle T_i \rangle) \tag{11}$$

Then,

$$F_{cover}(\sigma, t_c) = \frac{F_{c\text{-}actual}(\sigma, t_c)}{F_{c-max}(\sigma, t_c)} \tag{12}$$

The F_{cover} evaluates the code coverage of the test tuple σ, but larger code coverage may have smaller number of method calls. $F_{mc}(\sigma)$ is associated with method call number. More method calls covered by the test tuple make fault detection more effective. Similarly, $F_{mc}(\sigma)$ is also calculated in two parts. Meanwhile, for each σ,

$$F_{m\text{-}actual}(\sigma) = \sum_{k=1}^{|\sigma|} \left\{ \sum_{i=1}^{k} time(\langle T_i \rangle) * mc(\sigma_{\{1,k\}}) \right\} \tag{13}$$

$$F_{m\text{-}max}(\sigma) = mc(\sigma) * \sum_{i=1}^{|\sigma|} time(\langle T_i \rangle) \tag{14}$$

where $mc(\sigma_{\{1,i\}})$ means the method call number of the sub tuple $\sigma_{\{1,i\}} = <T_1, T_2, \ldots, T_i>$ for each test case $T_i \in \sigma$.

Then, $F_{mc}(\sigma)$ is obtained by these two parameters. Specifically, for each σ,

$$F_{mc}(\sigma) = \frac{F_{m\text{-}actual}(\sigma)}{F_{m\text{-}max}(\sigma)} * \frac{2}{|\sigma|} \tag{15}$$

where $2/|\sigma|$ is to neutralize the gap between numerator and denominator.

3.4 Algorithm Implementation About TCP Using CS

Our approach combines the CS with ILP for the test case prioritization problem under time constraint. To conclude above all, this section further describes the specific implementation process of our approach in Algorithm 2.

Algorithm 2 CSPrioritize

Input: Program P, Test suite T, s, g_{max}, p_a, p_t, t_c, ω_1, ω_2
Output: Maximum fitness tuple $F_{max} \in F$ in set σ_{max}

1: $Time_{max} \leftarrow p_t * \sum_{i=0}^{n} time(T_i)$
2: $T'_{max} \leftarrow ILP(T, Time_{max}, p_t)$
3: $POP \leftarrow Permute(T'_{max})$
4: $g \leftarrow 0$
5: $R_g \leftarrow RandomPerplr(POP, s)$
6: **repeat**
7: $\quad \sigma_i \leftarrow SelectOne(R_g)$
8: $\quad \sigma_j \leftarrow ApplyLevy(\sigma_i)$
9: $\quad$ **if** $(Fitness(\sigma_j, t_c, \omega_1, \omega_2) > Fitness(\sigma_i, t_c, \omega_1, \omega_2))$
10: $\quad\quad \sigma_i \leftarrow \sigma_j$
11: $\quad Fitness \leftarrow \phi$
12: $\quad$ **for** $\sigma_k \in R_g$
13: $\quad\quad Fitness \leftarrow Fitness \cup Fitness(\sigma_i, t_c, \omega_1, \omega_2)$
14: $\quad R_g \leftarrow Rank(R_g, Fitness)$
15: $\quad R_{g+1} \leftarrow ApplyLevy(R_g, p_a)$
16: $\quad g \leftarrow g + 1$
17: **until** $g \geq g_{max}$
18: $\sigma_{max} \leftarrow SelectBest(R_{g+1}, Fitness)$
19: **return** σ_{max}

In Algorithm 2, the inputs of the CS for TCP algorithm are the program P, each $T_i \in \{T_1, T_2, \ldots, T_n\}$, and the following user specified parameters: (1) s, maximum number of candidate test tuples generated during each iteration, (2) g_{max}, maximum number of iterations, (3) p_a, the fraction (p_a) of worse nests will be abandoned, (iv) p_t, percent of the execution time of T allowed by the time budget, (4) t_c, test coverage adequacy criteria, like class, method, block and so on, (5) ω_1, the program coverage weight, (6) ω_2, the program method calls weight, which $p_t, \omega_1, \omega_2 \in [0, 1]$, and $\omega_1 + \omega_2 = 1$.

The cuckoo search algorithm uses heuristic search to identify the test tuple $\sigma_{max} \in Permute(T'_{max})$, which T'_{max} is the subset of T and it may have the highest rate of fault detection in provided limit time. The subset T'_{max} get by the ILP. The initial test tuples get from $Permute(T'_{max})$. Finally, after the iterative process of the algorithm, the output is the best test tuple, that is, the test tuple with the greatest value. In general, any $\sigma_j \in Permute(T'_{max})$ has the form $\sigma_j = \, < T_1, \ldots, T_n >$ where $u \leq n$.

The first two lines of the algorithm are the key to our time-aware processing and aim to solve the Problem 2. Line 1 is the time acquisition and time constraint step. Next, the ILP is used to filter out the optimal set T'_{max} of test cases which are executed in restricted time. The next two lines are the process to generate the initial population of test case prioritization using CS. Before the algorithm run for the loop on line 6, the algorithm creates a set R_0 containing s random test tuples σ from $Permute(T'_{max})$. R_0 is the first generation of s potential "best" test tuple.

Then a test tuple is randomly chosen by $SelectOne(R_g)$ and Lévy flights is performed by $ApplyLevy(\sigma_i)$ to obtain a new solution. After that, the fitness value of them

are compared and the better one is selected to stay in $R_{g.}$ Once a set of test tuples is created, the *Fitness*(σ_j, t_c, ω_1, ω_2) method on line 9 will use coverage information and method-call information to calculate the fitness value of $\sigma_j \in R_g$. On line 14, the test tuples in R_g are reordered according to their fitness value. p_a of worse nests will be abandoned, and should be rebuilt by Lévy flights as well. After iterating over *MaxGeneration* times, the test tuple with the highest fitness value in the last test tuple set R_g is the global optimal solution obtained by the algorithm.

4 Empirical Evaluation and Discussion

Test case prioritization aims to detect more faults as early as possible. The APFD metrics is commonly used as a standard to evaluate this performance, and the specific meaning of it is described later in this section. Experiments are designed to compare the calculated APFD value and the number of detected faults, which proves that our method has better fault detection capability than the traditional methods.

In this section, the part one describes experiment environment, design and evaluation metrics. In the next part, it presents the parameters of our method in the next comparative experiments. Then, the first experiment compares the performance of CS with traditional techniques and the genetic algorithm. The second one compares them under time constraints.

4.1 Experiment Configuration

Test cases run on the Linux virtual machine. The execution time of them is obtained by the tool time.pl [19] in the SIR system and specifically defined as the average value of its 500 tests. All experiments are conducted on the same computer which is configured as the 64-bit Windows 7 operating systems, Pentium(R) dual-core CPU and 4 GB memory. The heuristic algorithm for each different parameter configuration is performed 200 times, and the random TCP experiment is performed 500 times to reduce the uncertainty of the experimental results. Experiments are conducted many times to avoid occasional abnormal data posing a threat to the validity of the experimental results.

Table 2. Case study application.

Apache-xml-security	
Classes	115
Methods	505
LOC	16800
Test cases	10
Seeded faults	12

Case Study. To validate the effectiveness of our approach, a Java program Apache-xml-security from the SIR infrastructure [20] is used as an analysis object. The differences between versions do not need to be considered. Version 0 is just chosen as our research object.

Table 2 shows the details of our case study. Apache-xml-security is a component library implementing XML signature and encryption standards, supplied by the XML subproject of the open source Apache project. It currently provides a mature implementation of Digital Signatures for XML, with implementation of encryption standards in progress. There are several sequential, previously-released versions of XML-security, each provided with a developer supplied JUnit test suite [20]. In each version, faults are seeded using fault seeding procedure described in Java Fault Seeding Process. Moreover, new faults are implanted with the original basis, and some faults that are not detected by any test cases or are detected by each test case should also be removed. Eventually, twelve faults are retained after screening. In addition, the number of test cases described in the table is also obtained by deleting redundancy.

Table 3. Study tools.

Tools	
Emma	A free Java code coverage tool
Source monitor	A tool for measuring code written in a variety of languages
time.pl	Calculate the time taken by test cases to execute the program, and it is used to collect timings
Lingo	An integrated tool for building and solving linear, nonlinear and integer optimization models

Implementation and Environment. The open source software program Apache-xml-security run in a Linux system environment. It is built by creating a virtual Ubuntu system on VMware. In this virtual environment, faults are injected in the source code and test cases are run on the research object.

There are tools should be used as shown in Table 3. They are related to two aspects. One is code information and the other is time constraint. In the CS algorithm, every time when the initialization or iteration executed, the fitness function of each test tuple should be calculated with test case code coverage information and the number of method calls. In which, the code coverage information is obtained by a tool called Emma. Emma instruments classes for class, method and block coverage. Emma is quite fast: the runtime overhead of added instrumentation is small (5–20%) and the byte-code instrumentor itself is fast, mostly limited by file I/O speed. Memory overhead is a few hundred bytes per Java class [21]. However, in addition to the code in the experiment coverage information, the method-call information of each test case is also indispensable. In this regard, the code metric tool Source Monitor is used to feed us back to a series of code-related information.

Meanwhile, the calculation of time cost for the test case use the tool time.pl [19] in the SIR system. This tool calculates the time taken by test cases to execute the program, and it is used to collect timings. Code coverage and test case execution time should be combined to filter out the better and more test cases that run in a limited time. This operation takes advantage of Lingo because it is easier to understand and handle than other tools when dealing with integer linear programming problems.

Evaluation Metrics. To calculate the fault detection ability, APFD (Average Percentage Fault Detection) metric [10] should be considered to apply to. It evaluates the optimization degree of test case prioritization set, and measures the effectiveness of different test case prioritization techniques under different time constrains.

Assume there is a test case set T with n test cases and m defects. Given a test case prioritization tuple, TF_i represents the location of the first test case in which fault i-th was detected. APFD is calculated as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n} \quad (16)$$

and the larger the value of APFD, the higher the efficiency of the test sorting.

At the same time, not all test cases usually should be run due to time constraints. It will most likely cause some faults might not be detected by any test case. In this regard, counting $TFi = n + 1$ if there are no test cases detect the i-th fault.

Take the test tuple $<T_4, T_5, T_3>$ in Fig. 1. (d) as an example, its $APFD = 1 - (3 + 1 + 1 + 2 + 3 + 2 + 1 + 6 + 2 + 6)/5 * 10 + 1/2 * 5 = 0.56$.

Although the execution time of the test case is important, the other evaluation criteria APFDc that takes the severity of faults and time cost into account is not used as our evaluation criteria. Because, this paper puts more emphasis on detecting more faults in a limited time when comparing the different techniques under different time constraints.

4.2 Experiments and Results

Table 4. Parameters used in CS configurations.

CS parameters	
P	Apache-xml-security
$(g_{\max}, s)$	(25, 60), (50, 30), (75, 15)
p_a	0.8, 0.5, 0.25
(ω_1, ω_2)	(1, 0), (0, 1), (0.85, 0.15)
t_c	class, method
p_t	0.85, 0.75, 0.5, 0.25

Optimal Parameters of CS Algorithm. As shown in Table 4, there are many uncertain parameters in the CS algorithm. So the fault detection effectiveness of our CS

algorithm are compared under different parameter configurations. It aims to obtain a greatest parameter configuration for the next comparative experiments. Different parameter values of the four variable groups need to be combined. Finally, there are $3 * 3 * 3 * 2 * 4 = 216$ different parameter configurations.

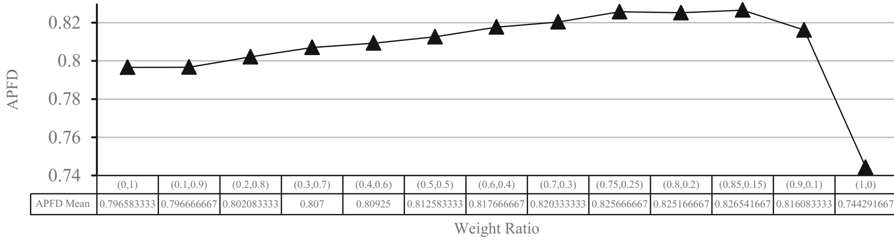


Fig. 3. Variation trend of CS prioritization algorithm's APFD value at different weight ratios.

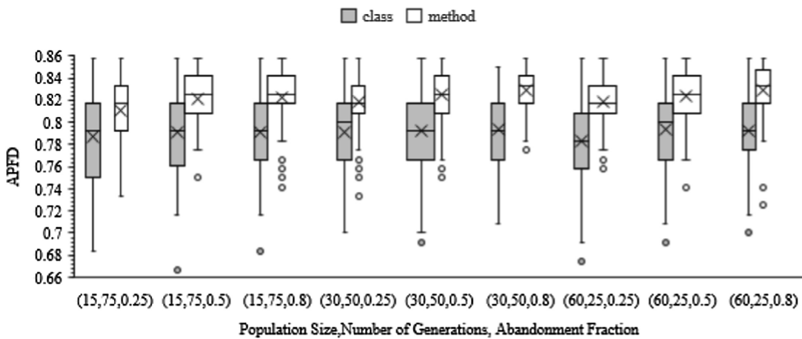


Fig. 4. Code coverage adequacy criteria comparison: method vs. class.

Table 5. Coverage adequacy criteria comparison t-test.

APFD mean		p-value	t-value
Class	Method		
0.790634	0.822273	<0.01	−31.652

Among them, the parameter combination (ω_1, ω_2) is related to fitness function calculation and the value (0.85, 0.15) is obtained by trying different weight ratios. The weight ratio is adjusted gradually by the step size of 0.1 from (0, 1) to (1, 0), then the relative optimal one is obtained. Experiments performed under each weight ratio and the results is shown in Fig. 3. According to the trend of the curve in Fig. 3, the fault detection rate of the CS algorithm get the best results when the ratio of weights is (0.85,

0.15). As is shown in Fig. 3, the TCP considers the code coverage without method calls when (ω_1, ω_2) is (0, 1). And its fault detection performance is significantly worse than the one which took method calls into consideration as well. This shows that adding the number of method invocations in fitness function gets a more rigorous fitness value, which indeed to improve the quality of test tuples.

The optimal weight ratio of the fitness function has been determined. Now combine and compare the experimental results of other uncertain parameters. First, the effect of different coverage adequacy criteria on experimental results is compared. There are 18 combinations of those 2 code coverage criteria with 9 different generations number, population size and abandonment fraction. Figure 4 illustrates box diagrams about experimental results. Table 5 shows the t test result of experiments, and the p-value and t-value are obtained by 2-tail t test at significant level $\alpha = 0.05$.

The p-value reveals the degree of difference of two targets. These two targets have difference when $p < 0.1$ and the difference is significant when $p < 0.05$. If $p > 0.1$, they have no difference. The t-value shows which is better. If p-value has told us these two targets have difference and t-value < 0, the latter one is better. According to the p-values and t-value in Table 5, fine-grained method-level experimental results are significantly better than the class-level. Although the experiment might not proceed with the level of the block, there is a great possibility of believing that the performance of the experiment might be further enhanced if the code coverage criterion is block.

The method has been chosen to be the code coverage criterion in next experiments. The white boxes in Fig. 4 are extracted and compared to determine other parameters. These parameters include the population size, number of generations and abandonment fraction. Figure 5. Shows those nine boxes and verifies that the abandonment fraction had better not choose the value of 0.25.

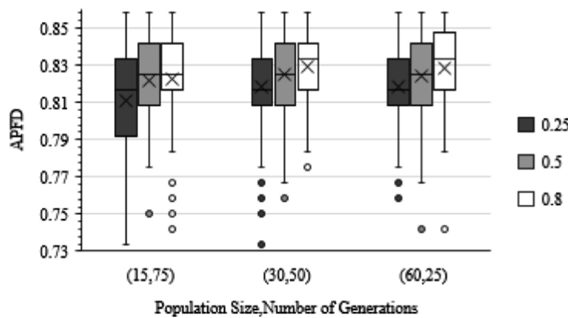


Fig. 5. Population size, generations number and abandonment fraction comparison.

In Table 6, compares to other ones, whatever the $(s, g_{\max})$ is, the p-value is always <0.01 when $p_a = 0.25$. Though $p_a = 0.5$ shows no significant difference between $p_a = 0.8$ in Fig. 5, Table 6 proves that $p_a = 0.5$ and $p_a = 0.8$ have significant differences ($p = 0.0366$), and $p_a = 0.8$ is better than $p_a = 0.5$ ($t = -2.467$) when the $(s, g_{\max})$ is (60, 25). The dark shadows in Table 6 show that the t test result has no significant difference when not partitioning $(s, g_{\max})$, and it prove the $p_a = 0.8$ is better as well.

When focus on the experimental results of different (s, $g_{\max}$) combinations in Table 7, differ from the GA, the difference is not reflected on the CS algorithm. Except (15, 75) shows obvious disadvantage over (30, 50) and (60, 25). The $(s, g_{\max}) = (30, 50)$ shows there is no significant different from $(s, g_{\max}) = (60, 25)$. Thus, a relatively good combination (50, 30) is chose for the next experiments. Finally, the optimal parameter is $(s, g_{\max}, p_a, \omega_1, \omega_2, t_c) = (30, 50, 0.8, 0.85, 0.15, \text{ method})$.

Table 6. Abandonment fraction comparison t-test.

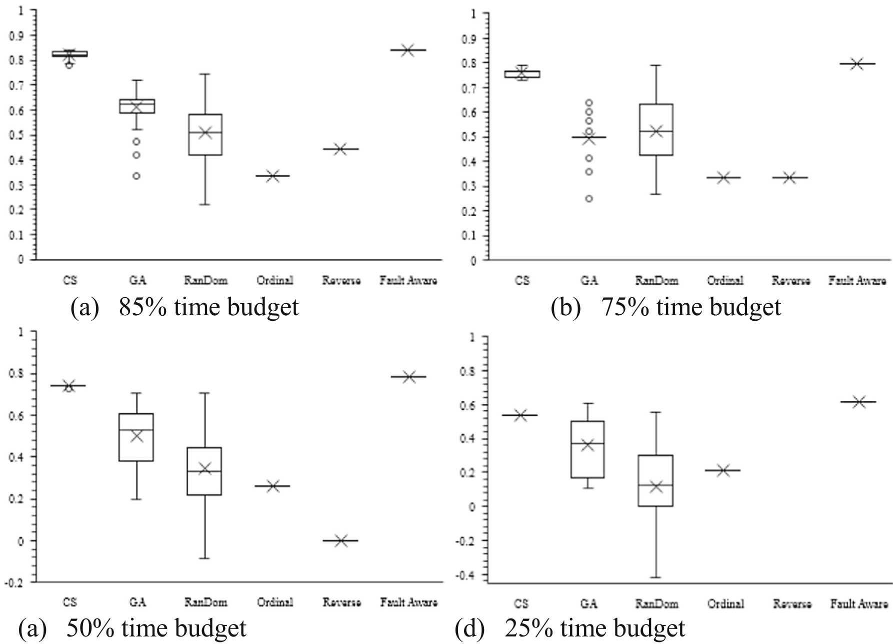
(s, $g_{\max}$)	p_a (1)	APFD mean	p_a (2)	APFD mean	p-value	t-value
(15, 75)	0.25	0.81092	0.5	0.82175	<0.01	−3.615
	0.5	0.82175	0.8	0.82296	0.6378	−0.471
	0.25	0.81092	0.8	0.82296	<0.01	−4.074
(30, 50)	0.25	0.81854	0.5	0.8255	<0.01	−2.957
	0.5	0.8255	0.8	0.8295	0.0533	−1.938
	0.25	0.81854	0.8	0.8295	<0.01	−4.982
(60, 25)	0.25	0.81829	0.5	0.82429	<0.01	−2.805
	0.5	0.82429	0.8	0.82856	0.0366	−2.097
	0.25	0.81829	0.8	0.82856	<0.01	−4.822
Total	0.25	0.81592	0.5	0.82385	<0.01	−5.422
	0.5	0.82385	0.8	0.82706	**0.0143**	**−2.454**
	0.25	0.81592	0.8	0.82706	<0.01	−7.771

Table 7. Population size, generations number t-test.

p_a	(s, g_{max}) (1)	APFD mean	(s, g_{max}) (2)	APFD mean	p-value	t-value
0.25	(15, 75)	0.81092	(30, 50)	0.81854	<0.01	−2.596
	(30, 50)	0.81854	(60, 25)	0.81829	0.9149	0.107
	(15, 75)	0.81092	(60, 25)	0.81829	<0.01	−2.612
0.5	(15, 75)	0.82175	(30, 50)	0.8255	0.1232	−1.545
	(30, 50)	0.8255	(60, 25)	0.82429	0.5757	0.5602
	(15, 75)	0.82175	(60, 25)	0.82429	0.2828	−1.075
0.8	(15, 75)	0.82296	(30, 50)	0.8295	<0.01	−2.941
	(30, 50)	0.8295	(60, 25)	0.82856	0.694	0.3937
	(15, 75)	0.82296	(60, 25)	0.82856	0.014	−2.467
Total	(15, 75)	0.81854	(30, 50)	0.82451	<0.01	−3.992
	(30, 50)	0.82451	(60, 25)	0.82376	0.5573	0.587
	(15, 75)	0.81854	(60, 25)	0.82376	<0.01	−3.535

Alternative Prioritization Comparisons. To verify the effectiveness of our approach using integer linear programming at different time budgets, time constraint is added on the last experiments. This section compares our approach with the GA, several traditional techniques under different time constraints.

Our CS algorithm combined with the ILP model described in Sect. 3.1. The genetic algorithm is implemented according to the technique designed by Walcott et al. [3].



(a) 85% time budget (b) 75% time budget

(a) 50% time budget (d) 25% time budget

Fig. 6. CS vs other techniques APFDs under different time-constraint.

Analyzing experimental results in general according to Fig. 6. Compare with the GA and other traditional techniques, our CS algorithm obviously improves the efficiency of fault detection to a certain extent, except the reverse order algorithm due to contingency. In addition, by observing the first three boxes of 4 pictures, it may be found that APFD value of our CS algorithm is not only high but also concentrated. It shows that the fault detection performance of our approach is more stable than the latter two techniques.

In Fig. 6, the fault aware is an strategy under ideal conditions, its premise is that the faults detected by the test case are predicted in advance. This strategy runs the test cases that detect more faults as early as possible. Although it is no practical meaning, it is used as a reference to the performance of our approach. By comparing with the data diagram of this idealized technology, it shows that applying the cuckoo algorithm to TCP problem does not match the fault aware strategy. But the difference is not obvious, the CS algorithm shows good fault detection performance.

Analyzing experimental results in detail according to Fig. 6 and Table 8. In Fig. 6, our CS algorithm is superior in fault detection efficiency to the other four techniques except fault aware strategy, whether there is no time constraint (100%), loose time constraint (85%) or compact time constraint (25%). The p-values (all p-values < 0.01)

and t-values obtained by the t-test in Table 8 prove that this superiority is extremely significant. When compare with fault-aware strategy, the difference between our approach and the fault-aware is significant under most of the time constraints, but is not significant when the time constraint is 85%.

Table 8. CS vs other techniques under different time constraints and t-test.

Time percent	CS APFD mean	Detected number	Other algorithms	APFD mean	Detected number	p-value	t-value
0.85	0.82796	12	GA	0.61428	11(±1)	<0.01	33.084
			Random	0.50648	11	<0.01	27.562
			Order	0.33333	11	<0.01	30.894
			Reserve	0.44444	8	<0.01	23.846
			Fault aware	0.84259	12	**0.1602**	**−1.4096**
0.75	0.77019	11	GA	0.49486	9(+1)	<0.01	46.212
			Random	0.52204	11	<0.01	18.007
			Order	0.33333	11	<0.01	23.676
			Reserve	0.33333	8	<0.01	23.676
			Fault aware	0.7963	11	0.045	−2.0169
0.5	0.74	11	GA	0.49864	9(±1)	<0.01	18.807
			Random	0.34517	7(±4)	<0.01	23.780
			Order	0.26042	7	<0.01	158.26
			Reserve	0	6	<0.01	244.2
			Fault aware	0.78125	11	<0.01	−13.631
0.25	0.53571	9	GA	0.36057	8(+1)	<0.01	11.012
			Random	0.11802	5(±4)	<0.01	17.623
			Order	0.21429	7	<0.01	1.3E+14
			Reserve	−∞	0	<0.01	+∞
			Fault aware	0.61905	9	<0.01	−3E+13

In Table 8, the "Detected number" records the number of faults that are detected by the current technology under different time constraints. Table 8 presents that our approach always detect as many faults as the fault aware strategy, and the number of faults detected by our approach is always not less than other techniques. The more compact the time constraint, the more obvious the advantage of the CS algorithm that detects more faults.

In general, the effective of our approach has been proved by the comparison about the number of fault detections and the evaluation standard APFD in above experiments. The conclusion is that applying CS Algorithm and integer linear programming to the time constraint TCP problem not only detects more faults but also detects faults as early as possible.

5 Conclusion

This paper proposes a time-aware test case prioritization technique, using cuckoo search algorithm and integer linear programming. The integer linear programming model is used to generate the best test case subset when the time is limited. A new fitness function based on code coverage and method-call information is also designed to improve the effectiveness of fault detection. Experimental analysis demonstrates that a time-aware test case order is created by our approach. And it significantly outperforms ones obtained from other prioritization techniques, even the genetic algorithm.

In the future work, the mature industrial projects will be further selected as experimental objects to verify the feasibility and effectiveness of the algorithm. Then, block, even sentence will be considered as our code coverage criterion in the later experiments. Finally, parallel computing and test case clustering will be studied in the future because of the consumption of time and space that the test suite should be permutated dynamically.

Acknowledge. This work is supported by the National Natural Science Foundation of China under grant No. 61572306 and No. 61502294, the CERNET Innovation Project under Grant No. NGII20170513, and the Youth Foundation of Shanghai Polytechnic University under Grant No. EGD18XQD01.

References

1. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)
2. Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. Acta Paediatrica **33**(4), 225–237 (2007)
3. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M.: Time aware test suite prioritization. In: ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July, DBLP, pp. 1–12 (2006)
4. Yang, X.S., Deb, S.: Cuckoo search via levey flights. In: World Congress on Nature & Biologically Inspired Computing, pp. 210–214. IEEE Xplore (2010)
5. Yang, X.S., Deb, S.: Multi-objective cuckoo search for design optimization. Comput. Oper. Res. **40**(6), 1616–1624 (2013)
6. Nagar, R., Kumar, A., Singh, G.P.: Test case selection and prioritization using cuckoos search algorithm. In: International Conference on Futuristic Trends on Computational Analysis and Knowledge Management, pp. 283–288. IEEE (2015)
7. Srivastava, P.R., Reddy, D.V.P.K., Reddy, M.S.: Test case prioritization using cuckoo search. Adv. Autom. Softw. Test. Framew. Refin. Pract. **28**(2), 159–182 (2012)
8. Rothermel, G., Untch, R.H., Chu, C.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)
9. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. In: ACM (2000)
10. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: ICSE, pp. 329–338. IEEE Computer Society (2001)

11. Islam, M.M., Marchetto, A., Susi, A.: A multi objective technique to prioritize test cases based on latent semantic indexing. In: Proceedings 16th European Conference Software Maintenance and Reengineering, pp. 21–30 (2012)
12. Saini, A., Tyagi, S.: MTCPA: multi-objective test case prioritization algorithm using genetic algorithm. In: International Journal of Advanced Research in Computer Science and Software Engineering (2015)
13. Schultz, M., Radloff, M.: Test case prioritization using multi objective particle swarm optimizer. In: International Conference on Signal Propagation and Computer Technology, pp. 390–395. IEEE (2014)
14. Alves, E.L., Machado, P.D., Massoni, T., Kim, M.: Prioritizing test cases for early detection of refactoring faults. Softw. Test. Verif. Reliab. **26**, 402–426 (2016)
15. Eghbali, S., Tahvildari, L.: Test case prioritization using lexicographical ordering. IEEE Trans. Softw. Eng. **42**(12), 1178–1195 (2016)
16. Lachmann, R., Schulze, S., Nieke, M.: System-level test case prioritization using machine learning. In: IEEE International Conference on Machine Learning and Applications, pp. 361–368. IEEE (2017)
17. Kim, J., Jeong, H., Lee, E.: Failure history data-based test case prioritization for effective regression test. In: Symposium on Applied Computing, pp. 1409–1415. ACM (2017)
18. Zhang, L., Hou, S.S., Guo, C.: Time-aware test-case prioritization using integer linear programming. In: Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, pp. 213–224 (2009)
19. Time.pl:a tool to collect timings. http://sir.unl.edu/content/tools.php
20. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir. Softw. Eng. **10**, 405–435 (2005)
21. Roubtsov, V.: EMMA: a free Java code coverage tool. http://emma.sourceforge.net/