Journal of Southeast University (English Edition) Vol. 34, No. 1, pp. 28 – 35 Mar. 2018 ISSN 1003—7985

A hybrid algorithm based on ILP and genetic algorithm for time-aware test case prioritization

Sun Jiaze[1,2] Wang Gang[1]

([1]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China)
([2]Shaanxi Key Laboratory of Network Data Intelligent Processing, Xi'an University of Posts and Telecommunications, Xi'an 710121, China)

Abstract: To solve the problem of time-aware test case prioritization, a hybrid algorithm composed of integer linear programming and the genetic algorithm (ILP-GA) is proposed. First, the test case suite which can maximize the number of covered program entities and satisfy time constraints is selected by integer linear programming. Secondly, the individual is encoded according to the cover matrices of entities, and the coverage rate of program entities is used as the fitness function and the genetic algorithm is used to prioritize the selected test cases. Five typical open source projects are selected as benchmark programs. Branch and method are selected as program entities, and time constraint percentages are 25% and 75%. The experimental results show that the ILP-GA convergence has faster speed and better stability than ILP-additional and ILP-total in most cases, which contributes to the detection of software defects as early as possible and reduces the software testing costs.

Key words: test case prioritization; integer linear programming(ILP); genetic algorithm; time constraint
DOI: 10.3969/j.issn.1003 – 7985.2018.01.005

Regression testing is costly in software evolution, which consumes 80% of the overall testing budgets[1]. Test case prioritization (TCP) is an efficient and practical regression testing technique, and it can be used to reorder the test cases to achieve the goal of improving the test efficiency and reducing the test cost. Over the past few years, many test case prioritization techniques have been proposed to study various prioritization approaches [2-4] and coverage criteria[5]. However, these approaches do not definitely consider the time budget and the execution time difference of test cases. Executing the entire test suite is sometimes time-consuming, which does not allow for the execution of all the test cases. Therefore, the test cases are usually executed under time constraints.

Many researchers have focused on time-aware test case prioritization. Walcott et al.[6] reduced time-aware test case prioritization to the zero/one knapsack problem, and used the genetic algorithm to solve the problem. Zhang et al.[7] selected a subset of original test suites by integer linear programming (ILP), and prioritized the selected test cases by traditional total prioritization and additional prioritization. You et al.[8] used the SIEMENS suite and space program as empirical research objects to prioritize test suites by random ordering, total strategy, additional strategy, ILP-total, and ILP-additional, respectively. Do et al.[9] found that the best effective TCP technology under a certain time constraint cannot ensure the best effect under other time constraints. Recently, Hao et al.[3] presented a unified test case prioritization approach that encompassed both the total and additional strategies. Lu et al.[10] found that the genetic algorithm can obtain better results than other techniques, such as random, total, additional, and search-based techniques.

As the above literature shows, greedy strategy and random ordering are not always effective under different time constraints. The convergence speed of the genetic algorithm is not fast, and the initial value of TCP is sensitive. The test case selection involves a subset which is selected from the original test suite. Broadly speaking, it is related to the research on regression test case selection[11] and test case reduction[12]. Meanwhile, ILP is an effective optimization method for the zero/one knapsack problem[7]. Therefore, this paper uses ILP for the test case selection and the genetic algorithms for prioritizing the selected test cases. We compare ILP-GA with two greedy-based approaches under different time constraints, and the experimental results show that our approach is superior to greedy-based approaches. In summary, the paper makes the following contributions: It is the first attempt to combine ILP and genetic algorithms for time-aware test case prioritization, and the empirical evaluation of the proposed approach and approaches of the greedy strategy based ILP is carried out in detail; the time complexity of ILP-GA is analyzed.

1 Time-Aware Test-Case Prioritization

Time-aware test case prioritization is usually formalized

Received 2017-10-15, **Revised** 2018-01-21.

Biography: Sun Jiaze (1980—), male, doctor, associate professor, sunjiaze@126.com.

Foundation items: The Natural Science Foundation of Education Ministry of Shaanxi Province (No. 15JK1672), the Industrial Research Project of Shaanxi Province (No. 2017GY-092); Special Fund for Key Discipline Construction of General Institutions of Higher Education in Shaanxi Province.

Citation: Sun Jiaze, Wang Gang. A hybrid algorithm based on ILP and genetic algorithm for time-aware test case prioritization[J]. Journal of Southeast University (English Edition), 2018, 34(1): 28 – 35. DOI: 10.3969/j.issn.1003 – 7985.2018.01.005.

as follows.

Given: A test suite, S; the set of permutations of all subsets of S, P; objective function F and time cost function T, which range from permutations to real numbers; time constraint, $t_{\max}$.

Problem: Time-aware test case prioritization aims to find a permutation $S' \in P$ satisfying that for any element $S'' \in P(S'' \neq S')$, $F(S') \geqslant F(S'')$, $T(S'') \leqslant t_{\max}$, and $T(S') \leqslant t_{\max}$.

In the problem, P is the set of all possible ordering of S; T is a function, and it yields the execution time of that ordering when it is applied to any ordering of S, so we can decide whether an ordering satisfies the time constraint through judging the inequality $T(S') \leqslant t_{\max}$; the function F is applied to any ordering and returns a fitness value of that ordering. In this paper, the function F measures a test sequence's ability of detecting bugs as early as possible, and we stipulate that a test sequence with a higher fitness value is superior to that with a lower fitness value.

2 Proposed Approach

In this paper, we propose a new approach for time-aware test case prioritization by combining integer linear programming and the genetic algorithm. As shown in Fig. 1, our approach is composed of two steps. First, we use the ILP model to select a subset of original test suites which satisfies the time constraint. Secondly, we prioritize the subset by the genetic algorithm.

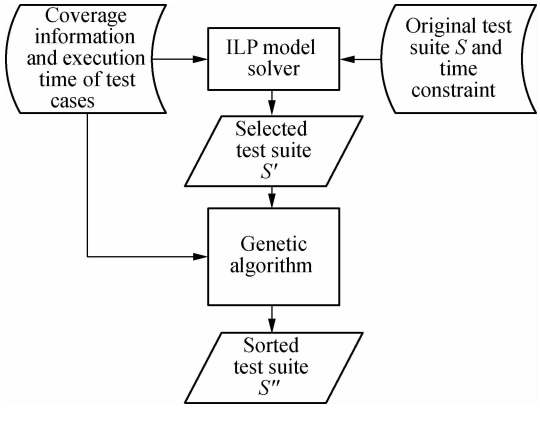

Fig. 1 The description of ILP-GA

We consider branch coverage criteria and method coverage criteria in this paper. For the ease of presentation, we present our approach only in terms of branch coverage.

2.1 Test case selection

Test case selection can be formalized as two ILP models, and each model is composed of decision variables, an objective function and a constraint system. As shown in Fig. 2, we denote the original test suite as S and two subsets of the original test suite are denoted as S_1 and S_2, respectively. S_1 satisfies the time constraint and maximizes the number of covered branches, and it is selected by the first ILP model. We denote the sum of time of S_1 as t_1, so the remaining time from the time constraint is $t_{\max} - t_1$. Furthermore, S_2 will be selected by the second ILP model. S_2 has the maximum sum of the number of branches covered by each test case that belongs to $S_3(S_3 = S - S_1)$ and the total time of S_2 does not exceed $t_{\max} - t_1$. Thus, after solving the two ILP models, all the test cases that have been selected can be denoted as $S'(S' = S_1 \cup S_2)$, and we denote the number of these test cases as n. The first ILP model will be built as follows.

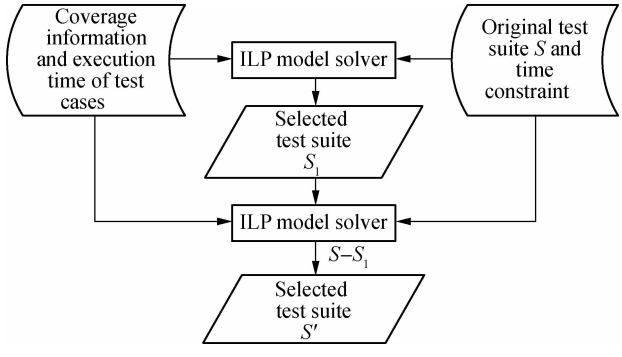

Fig. 2 The technological process of ILP

2.1.1 Decision variables

For each test case, there is a Boolean decision variable to represent whether the test case is selected or not. The test suite is denoted as $S = \{s_1, s_2, \ldots, s_n\}$, and n Boolean decision variables are denoted as $x_i(1 \leqslant i \leqslant n)$. x_i is defined as

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

Furthermore, there is another group of Boolean decision variables to represent whether each branch is covered by one or more test cases. A set of branches are denoted as $B_R = \{b_1, b_2, \ldots, b_m\}$, and y_i is defined as

$$y_j = \begin{cases} 1 & \text{if some selected test cases cover } b_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

2.1.2 Objective function

Our goal of test case selection is to maximize the number of covered branches, so the objective function can be defined as

$$F = \max \sum_{j=1}^{m} y_j \tag{3}$$

In the objective function, if any selected test case does not cover b_j, the value y_j is 0. Thus, Eq. (3) guarantees to count each covered branch just once.

2.1.3 Constraint system

To ensure that the selected test cases satisfy the time constraint, an inequality is defined as

$$\sum_{i=1}^{n} T(s_i)\, x_i \leqslant t_{\max} \tag{4}$$

Formula (4) indicates that the sum of execution time of all the selected test cases is no more than $t_{\max}$. Furthermore, there is a group of inequalities to ensure that, if $y_j = 1 (1 \leqslant j \leqslant m)$, at least one test case covering b_j is selected. There is a need to represent whether a test case covers a branch, and it can be denoted as

$$c_{ij} = \begin{cases} 1 & \text{if } s_i \text{ cover } b_j \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Using the coverage information in Eq. (5), the inequalities can be defined as

$$\sum_{i=1}^{n} c_{ij} x_i \geqslant y_j \qquad 1 \leqslant j \leqslant m \tag{6}$$

2.1.4 Further test case selection

A subset of test cases that can satisfy the time constraint and maximize the number of branches has been selected by the first ILP model, but there may be some time left over from time budget, so the second ILP model will be built for later selecting test cases to enhance the fault detection in the time constraint. It is noted that further selecting does not necessarily increase the number of covered branches, but these test cases may contribute to detecting faults.

The unselected test cases in S are denoted as $C(C = \{c_1, c_2, \ldots, c_L\})$ and time left from $t_{\max}$ is denoted as t_{left}. The L Boolean decision variables are denoted by $z_k (1 \leqslant k \leqslant L)$, and z_k is defined as

$$z_k = \begin{cases} 1 & \text{if } c_k (1 \leqslant k \leqslant L) \text{ is further selected} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

The objective function shows that a subset of test suite that has the maximum sum of the number of branches covered by each test case should be selected from C, and it is defined as

$$\max \sum_{k=1}^{L} \text{count}(c_k)\, z_k \tag{8}$$

The constraint system indicates that the sum of execution time of all the selected test cases from C is no more than t_{left}, and it is defined as

$$\sum_{k=1}^{l} T(c_k)\, z_k \leqslant t_{\text{left}} \tag{9}$$

2.2 Test case prioritization

After the test case selection, the time-aware test case prioritization is transformed into the traditional test case prioritization. Test case prioritization is a NP-complete problem. The genetic algorithm is a heuristic search algorithm which provides a general framework for solving complex problems. The genetic algorithm is used to prioritize test cases, as described in Algorithm 1. It should be noted that the cover matrix $\boldsymbol{A}$ is used to record the coverage information of program entities of a program. If the j-th program entity is covered by the i-th test case, then $a_{ij} = 1$, otherwise $a_{ij} = 0$.

Algorithm 1 Genetic algorithm

Input: Test suite S; the number of initial sequences of test cases N; the maximum number of iterations $g_{\max}$; crossover probability P_c; mutation probability P_m; coverage information of program entities M_c; the execution time of test cases in S, T_S.

Output: A test sequence with the highest fitness value $\sigma_{\max}$.

1. $R_0 \in \varnothing$
2. repeat
3. $R_0 \leftarrow R_0 \cup \{\text{InitializeRandomPopulation}(S, M_c)\}$
4. until $|R_0| = N$
5. $g \leftarrow 0$
6. repeat
7. $F \leftarrow \varnothing$
8. for $\sigma_i \in R_g$
9. $F \leftarrow F \cup \{\text{computeFitness}(\sigma_i, M_c, T_S)\}$
10. $\sigma_1, \sigma_2 \leftarrow \text{EliteTwoBest}(R_g, F)$
11. $R_{g+1} \leftarrow \sigma_1, \sigma_2$
12. repeat
13. $\sigma_j, \sigma_k \leftarrow \text{Select}(R_g - \{\sigma_1, \sigma_2\}, F)$
14. $\sigma_q, \sigma_r \leftarrow \text{Crossover}(P_c, \sigma_j, \sigma_k)$
15. $\sigma'_q \leftarrow \text{Mutation}(P_m, \sigma_q)$
16. $\sigma'_r \leftarrow \text{Mutation}(P_m, \sigma_r)$
17. $R_{g+1} = R_{g+1} \cup \{\sigma'_q\} \cup \{\sigma'_r\}$
18. until $|R_{g+1}| = N$
19. $g \leftarrow g + 1$
20. until $g > g_{\max}$
21. $\sigma_{\max} \leftarrow \text{FindMaxFitnessSequence}(R_{g-1}, F)$
22. return $\sigma_{\max}$

2.2.1 Genetic algorithm framework

We denote the subset S' in section 2.1 as S. All of probabilities P_c, $P_m \in [0, 1]$. In general, any $\sigma_i \in \text{perms}(2^S)$ has the form $\sigma_i = \langle S_j, \cdots, S_n \rangle$, and n is the number of test cases in σ_i. $\text{perms}(2^S)$ represents the set of all possible sequences and subsequence of T. A test sequence is composed of n numbers and we denote a test sequence as $\langle w_1, \cdots, w_i, w_{i+1}, \cdots, w_n \rangle$ ($1 \leqslant w_i \leqslant n$ and $w_i \neq w_{i+1}$). A test sequence is an integer array and there is a number w_i in each position of the integer array.

As shown in Algorithm 1, in the loop beginning on line 3, the algorithm creates a set R_0 containing N random test sequences σ_i from $\text{perms}(2^S)$. R_0 is the first generation in the iteration. If a test sequence is created, then $\text{computeFitness}(\sigma_i, M_c, T_S)$ will be used to compute the fitness of this test sequence. We denote F_i as the fitness value of σ_i. We also use $F = \langle F_1, F_2, \cdots, F_N \rangle$ to denote the sequence of fitness for each $\sigma_i \in R_g, 0 \leqslant g \leqslant g_{\max}$.

The EliteTwoBest(R_g, F) on line 10 chooses the two best test sequences in R_g to be elements for the next generation R_{g+1}, which applies the elitist selection technique. On line 13, Select($R_g - \{\sigma_1, \sigma_2\}, F$) identifies pairs of sequences $\{\sigma_j, \sigma_k\}$ from R_g through a roulette wheel selection technique. The Crossover(P_c, σ_j, σ_k) on line 14 may form two new test sequences $\{\sigma_q, \sigma_r\}$ based on P_c. Each test sequence in the pair $\{\sigma_q, \sigma_r\}$ may then be mutated based on P_m.

After each of these operations has been executed, both σ_q and σ_r are set into R_{g+1}, as seen on line 17. The same transformations are applied to all pairs selected by Select($R_g - \{\sigma_1, \sigma_2\}, F$) until R_{g+1} contains N test sequences. In total, g_{max} sets of N test sequences are iteratively created as described in Algorithm 1 on lines 6 to 20. When the final set $R_{g_{max}}$ has been created, the test sequence with the greatest fitness, σ_{max}, is determined on line 21.

2.2.2 Fitness function

computeFitness(σ_i, M_c, T_S) on line 9 uses $F(\sigma_i, M_c, T_S)$ to calculate the fitness value. The fitness function, represented by $F(\sigma_i, M_c, T_S)$, assigns each test sequence a fitness value based on the program entity coverage (P) of that sequence and the execution time of each test case ($T(\langle S_j \rangle)$).

$F(\sigma_i, M_c, T_S)$ is computed by summing the products of execution time $T(\langle S_j \rangle)$ and the P of the subsequence $\sigma_{i|1,j|} = \langle S_1, S_2, \cdots, S_j \rangle$ for each test case $S_j \in \sigma_i$. The $F(\sigma_i, M_c, T_S)$ gives precedence to test sequences that have more codes covered early in execution. Formally, for some $\sigma_i \in \text{perms}(2^S)$,

$$F(\sigma_i, M_c, T_S) = \sum_{j=1}^{|\sigma_i|} T(\langle S_j \rangle) \times P(\sigma_{i|1,j|}, M_c) \quad (10)$$

2.2.3 Selection operation

The selection operation is roulette wheel selection. First, the fitness values of test sequences are denoted as $\{f_1, f_2, \cdots, f_N\}$, and then the sum of the fitness values is $\sum_{i=1}^{N} f_i$. Secondly, we can obtain N results by computing $f_i / \sum_{i=1}^{N} f_i$, then the test sequences are sorted by descending results and the result of each sequence accumulates the results in front of it. Finally, a random number $r \in [0, 1)$ is generated, and the first sequence whose accumulated result is greater than or equal to r is selected. The selection operator is repeated until enough sequences are selected to fill the set R_{g+1}. The number of sequences is N this moment.

2.2.4 Crossover operation

As explained in section 2.2.1, pairs of test sequences are selected from R_g. Crossover(P_c, σ_j, σ_k) performs crossover operation and creates two new test sequences from $\{\sigma_j, \sigma_k\}$. A random number $r_1 \in [0, 1)$ is generated, and the crossover operator is executed when P_c is greater than r_1. When the crossover begins, another random number $k \in [0, n)$ is generated as the crossover point, and n is the number of test cases in S. The first k numbers in σ_j are copied to the first k positions in σ_q. Let us denote the numbers which are in the σ_k and which is not equal to the first k numbers in the σ_j as n_{w_i}, and then we use each number in n_{w_i} to fill σ_q one by one into the last $(n-k)$ positions in σ_q. Thus, a new test sequence σ_q is generated by the above steps. Analogously, σ_r is also generated as the same steps. Finally, we obtain the two new test sequences $\{\sigma_q, \sigma_r\}$ by crossover operation.

2.2.5 Mutation

Mutation(P_m, σ_q) is used to mutate σ_q. First, a random number $r_3 \in [0, 1)$ is generated. If P_m is greater than r_3, the mutation operation is executed. If mutation is to occur, the two positions will be selected from σ_q and the two test cases which are in the two positions will be swapped. Mutation(P_m, σ_r) is used to mutate σ_r, and it is similar to Mutation(P_m, σ_q).

2.3 Analysis of algorithm

First, we analyze the time complexity of the genetic algorithm as follows. We denote the number of test cases as n_1 and the number of program entities of a program as n_2. The time complexity of reading coverage information from I/O is $O(n_1 n_2)$. The time complexity of the reading execution time of test cases from I/O is $O(n_1)$. The time complexity of initializing population is $O(n_1 N)$. The time complexity of computing fitness values for all test sequences is $O(n_1 n_2 N)$. The time complexity of elite strategy is $O(N^2)$. We use roulette wheel selection technique and bubble sort in selection operation, so the time complexity of selection operation is $O(n_1 n_2 N) + O(N^2)N/2$. The time complexity of crossover selection is $O(n_1 N)$. The time complexity of the mutation operation is $O(N)$. Therefore, the time complexity of the genetic algorithm is $O(n_1 n_2 N) + O(n_1 N) + O(N^2)$. The calculation of fitness values and genetic operators is main time overhead, and the number of calling the genetic algorithm determines the efficiency of the algorithm.

Secondly, if we denote the number of decision variables as n and the number of constraint conditions as m, then the time complexity of ILP is $O(mn)$. Furthermore, $n = n_1 + n_2$ approximately and $m = n_2$ approximately, so the time complexity of ILP is $O(n_2(n_1 + n_2))$.

In summary, the time complexity of ILP-GA is $O(n_1 n_2 N) + O(n_1 N) + O(N^2) + O(n_2(n_1 + n_2))$. The time complexity of ILP-GA is determined by n_1, n_2 and N. The ILP-GA is a polynomial algorithm.

3 Empirical Evaluation

We implemented the approach described in Section 2

and measured its effectiveness and stability.

3.1 Experimental setup

In the experiments, we used five classical open sources Java projects[10] from GitHub, which had been widely used in software testing research, and they are described in Tab. 1. All experiments were performed on Windows 764 bit, a 3.60 GHz Intel Core 3 processor and 8 GB of main memory. The ILP-total and ILP-additional were implemented in the Python language. The genetic algorithm was implemented in the Java language based on JDK 1.7.0. We ran these algorithms in Eclipse 4.3. To implement the ILP-based techniques (i.e., total & additional test prioritization via ILP), we used a mathematical programming solver, GUROBI Optimization (http://www.gurobi.com), which was used to represent and solve the equations formulated by the ILP-based techniques. The number of initial orderings of test cases is 60. We adopt the same parameters in Ref. [6] to compare the experimental results with the classical result. The number of iterations for the GA is 25. The crossover probability and mutation probability are 0.70 and 0.10, respectively.

Tab. 1 Experimental subjects

Subject	The number of branches	The number of methods	The number of test cases	The website of project
Jasmine-maven	95	138	24	https://github.com/searls/jasmine-maven-plugin
Java-apns	82	128	47	https://github.com/notnoop/java-apns
La4j	967	536	245	https://github.com/vkostyukov/la4j
Metrics-core	460	695	129	https://github.com/nablex/metrics-core
Scribe-java	62	173	51	https://github.com/Kailashrb/scribe-java

3.2 Experiments and results

3.2.1 Effectiveness for different techniques

ILP-total, ILP-additional and ILP-GA were used to solve time-aware test case prioritization in our experiments, and we compared them by effectiveness and stability. We used 25% and 75% as the two different time constraints. The branch and method are used as test adequacy criteria. As shown in Tab. 2, each project was applied by three approaches under different time constraints and different test adequacy criteria, so each project will have four different comparison methods by three approaches. For each comparison method, we ran three algorithms 20 times, then calculated the average value of 20 fitness values of each algorithm. The prioritization efficiency of the ILP-GA is improved on average by approximately 54% and 13%, respectively, under 25% time constraints and it is improved on average by, approximately, 42% under 75% time constraints, so we conclude that our technique is effective. It should be noted that the branch coverage of the last two projects have a close average fitness value under 25% time constraint, and we speculate that the two average fitness values are close to optimal solutions.

Tab. 2 Experimental results of three algorithms

Subject	Algorithms	Average fitness value			
		Time constraint 25%		Time constraint 75%	
		Branch coverage	Method coverage	Branch coverage	Method coverage
Jasmine-maven	ILP-GA	2 203.779	1 788.725	8 504.132	7 494.476
	ILP-additional	2 094.011	1 736.797	7 560.532	7 491.445
	ILP-total	1 407.663	1 171.757	6 128.897	5 519.373
Java-apns	ILP-GA	187.693	214.997	187.912	213.958
	ILP-additional	180.316	175.434	180.246	175.090
	ILP-total	106.415	135.280	106.357	135.179
La4j	ILP-GA	553.364	551.660	1 647.211	1 663.788
	ILP-additional	322.437	498.952	609.098	785.844
	ILP-total	270.352	313.703	1 001.389	1 184.559
Metrics-core	ILP-GA	197.274	173.841	718.520	650.585
	ILP-additional	176.254	172.447	542.923	513.442
	ILP-total	145.690	137.439	565.031	460.181
Scribe-java	ILP-GA	55.645	44.184	209.748	176.717
	ILP-additional	55.490	44.071	152.235	176.164
	ILP-total	45.844	33.850	179.586	151.966

3.2.2 Stability for different techniques

The boxplots in Fig. 3 and Fig. 4 depict the results of the three approaches under two different time constraints. We have the following observations. First, whether the

time constraint is 25% or 75%, the results of ILP-GA are overall superior to the ILP-total and ILP-additional for Java-apns and La4j. Secondly, we consider the distribution of results which is generated by each approach. As can be seen from Fig. 3 and Fig. 4, the height of each box is small. The fitness values in the box are close and concentrated. Thus, all the approaches perform stably. In short, no matter what the time constraint is and what the test adequacy criterion is, our approach always achieves the expected effectiveness and stability.

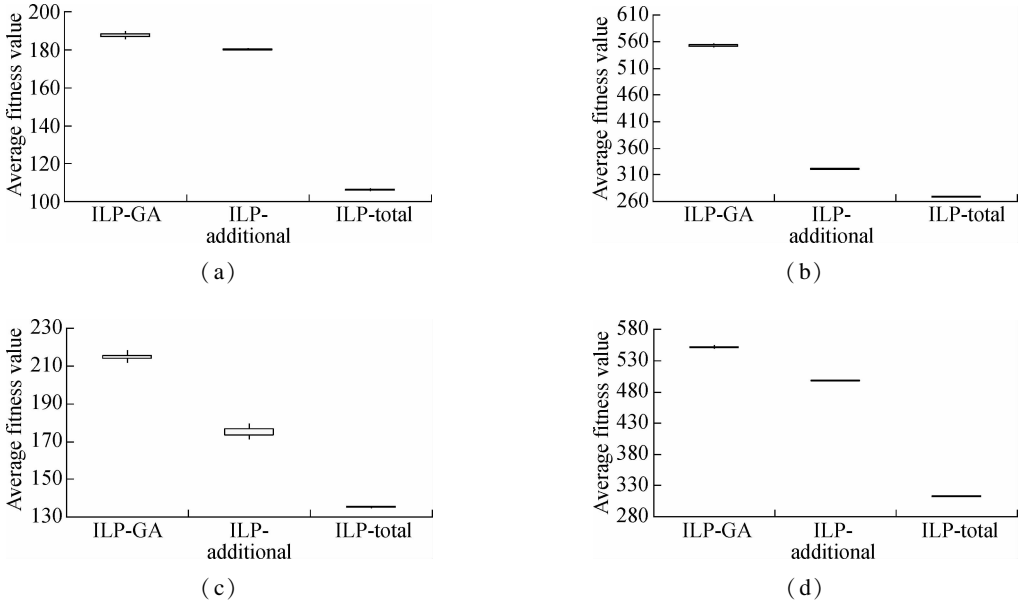

Fig. 3 Box-plots of result contrast of ILP-GA, ILP-additional and ILP-total under 25% time constraint. (a) Branch coverage of Java-apns; (b) Branch coverage of La4j; (c) Method coverage of Java-apns; (d) Method coverage of La4j

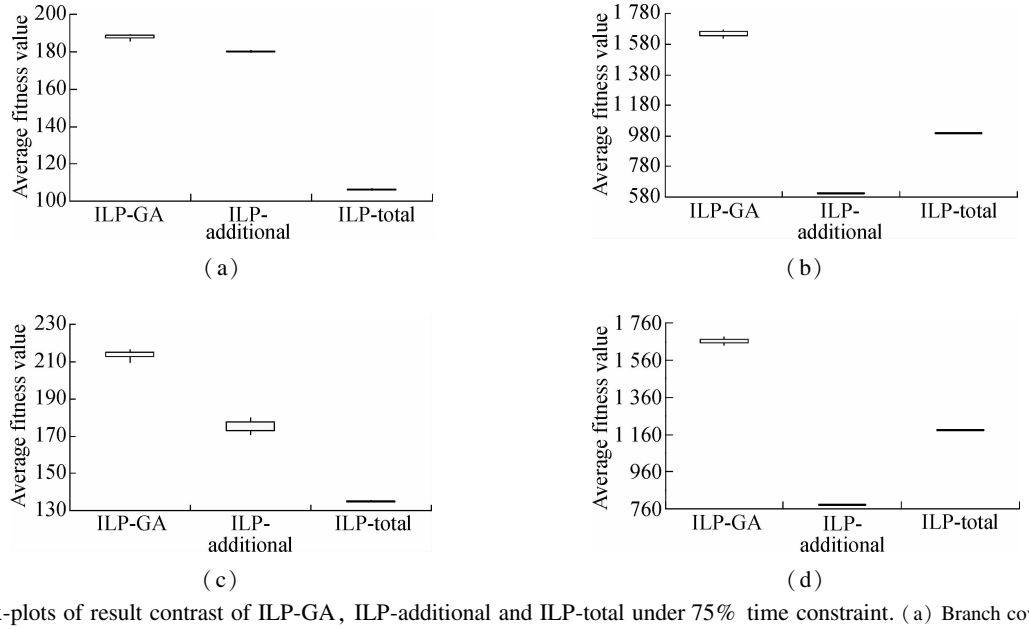

Fig. 4 Box-plots of result contrast of ILP-GA, ILP-additional and ILP-total under 75% time constraint. (a) Branch coverage of Java-apns; (b) Branch coverage of La4j; (c) Method coverage of Java-apns; (d) Method coverage of La4j

3.2.3 Convergence for ILP-GA

Fig. 5 and Fig. 6 describe the change of the average fitness value along with iterations for three projects under two time constraints. As can be seen from Fig. 5 and Fig. 6, the average fitness values improve along with iterations. It also can be seen that the rise of curves is a near-linear trend in the former iterations and curves become less steep in the last several iterations. Therefore, the convergence rate of ILP-GA is fast, and our experimental results are close to the global optimal solutions.

4 Conclusions

1) Considering the rate of coverage, our approach outperforms the ILP-additional and ILP-total under different time constraints.

2) Considering stability and convergence, our approach has good stability and a fast convergence rate.

3) When the time constraint is not tight, our approach

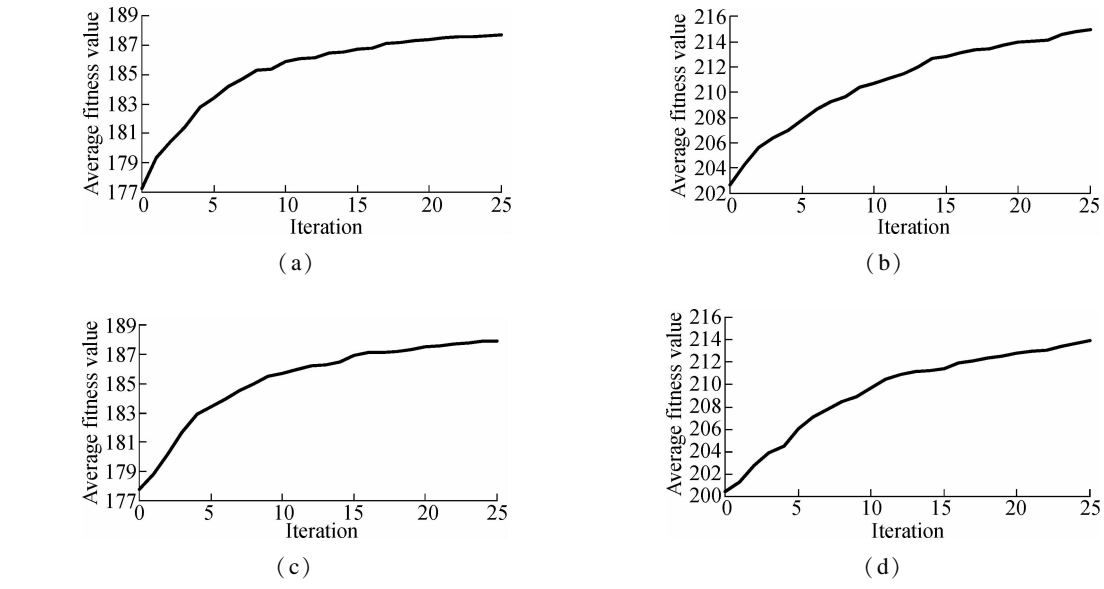

Fig. 5 The convergence of ILP-GA of Java-apns. (a) Branch coverage under 25% time constraint; (b) Method coverage under 25% time constraint; (c) Branch coverage under 75% time constraint; (d) Method coverage under 75% time constraint

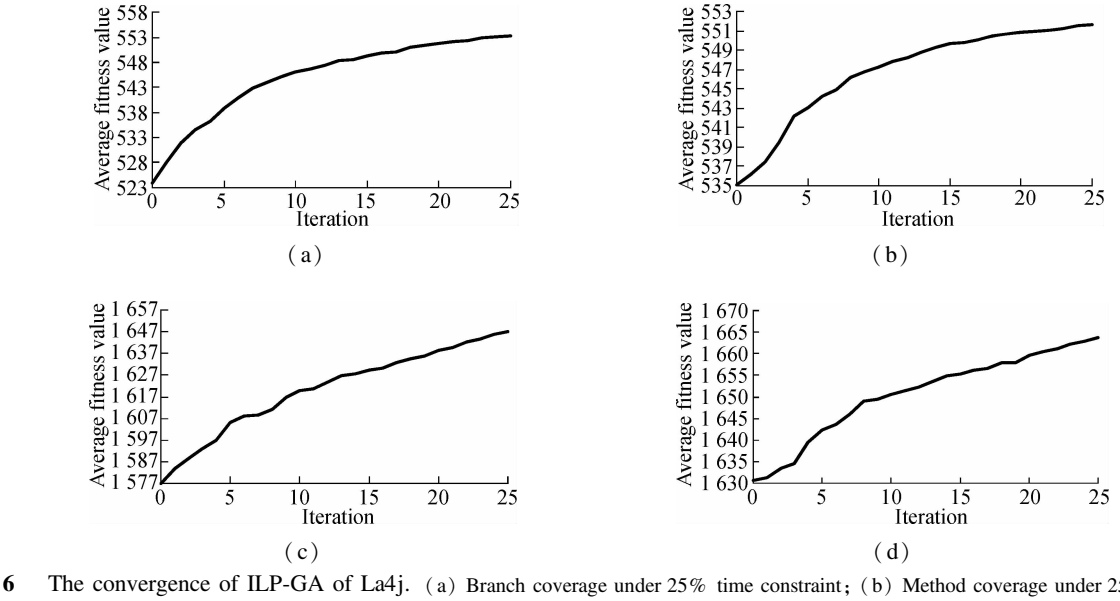

Fig. 6 The convergence of ILP-GA of La4j. (a) Branch coverage under 25% time constraint; (b) Method coverage under 25% time constraint; (c) Branch coverage under 75% time constraint; (d) Method coverage under 75% time constraint

is superior to other techniques, but ILP-additional can perform competitively when time constraint is tight.

In future work, we intend to explore the following several aspects. First, we intend to study new algorithms and new evaluation indices to improve the effectiveness of prioritization. Secondly, we plan to implement a prototype tool to complete the whole test case optimization process from the information collection to the result visualization.

References

[1] Chittimalli P K, Harrold M J. Recomputing cover age information to assist regression testing [J]. *IEEE Transactions on Software Engineering*, 2009, **35**(4):452 - 469. DOI:10.1109/TSE.2009.4.

[2] Li Z, Harman M, Hierons R M. Search algorithms for regression test case prioritization [J]. *IEEE Transactions on Software Engineering*, 2007, **33**(4):225 - 237. DOI: 10.1109/TSE.2007.38.

[3] Hao D, Zhang L M, Zhang L, et al. A unified test case prioritization approach [J]. *ACM Transactions on Software Engineering and Methodology*, 2014, **24**(2):1 - 31. DOI:10.1145/2685614.

[4] Luo Q, Moran K, Poshyvanyk D. A large-scale empirical comparison of static and dynamic test case prioritization techniques[J]. *Proceedings of the* 2016 24*th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle, WA, USA, 2016: 559 - 570. DOI:10.1145/2950290.2950344.

[5] Mei H, Hao D, Zhang L M, et al. A static approach to prioritizing junit test cases [J]. *IEEE Transactions on Software Engineering*, 2012, **38**(6):1258 - 1275. DOI: 10.1109/TSE.2011.106.

[6] Walcott K R, Soffa M L, Kapfhammer G M, et al.

Time aware test suite prioritization [C]//*Proceedings of the* 2006 *International Symposium on Software Testing and Analysis*. Portland, Maine, USA, 2006: 1 - 11. DOI:10.1145/1146238.1146240.

[7] Zhang L, Hou S S, Guo C, et al. Time-aware test case prioritization using integer linear programming[C]//*Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. Chicago, IL, USA, 2009: 213 - 224. DOI:10.1145/1572272.1572297.

[8] You D J, Chen Z Y, Xu B W, et al. An empirical study on the effectiveness of time-aware test case prioritization techniques [C]//*Proceedings of the* 26*th ACM Symposium on Applied Computing*. Taichung, China, 2011: 1451 - 1456. DOI:10.1145/1982185.1982497.

[9] Do H, Mirarab S, Tahvildari L, et al. The effects of time constraints on test case prioritization: A series of controlled experiments [J]. *IEEE Transactions on Software Engineering*, 2010, **36**(5): 593 - 617. DOI:10.1109/TSE.2010.58.

[10] Lu Y F, Lou Y L, Cheng S Y, et al. How does regression test prioritization perform in real-world software evolution [C]//*Proceedings of the* 38*th International Conference on Software Engineering*. Austin, TX, USA, 2016: 535 - 546. DOI:10.1145/2884781.2884874.

[11] Yoo S, Harman M. Regression testing minimization, selection and prioritization: A survey [J]. *Software Testing, Verification and Reliability*, 2012, **22**(2): 67 - 120. DOI:10.1002/stvr.430.

[12] Alipour M A, Shi A, Gopinath R, et al. Evaluating non-adequate test case reduction [C]//*Proceedings of the* 31*st IEEE/ACM International Conference on Automated Software Engineering*. Singapore, 2016: 16 - 26. DOI: 10.1145/2970276.2970361.

基于 ILP 和 GA 的时间感知测试用例优先排序混合算法

孙家泽[1,2] 王 刚[1]

([1] 西安邮电大学计算机学院, 西安 710121)

([2] 西安邮电大学陕西省网络数据智能处理重点实验室, 西安 710121)

摘要:针对基于时间感知的测试用例优先排序问题,提出了整数线性规划与遗传算法相结合的混合求解算法. 首先,基于整数线性规划,选择待测程序实体的覆盖量最大且满足时间约束的测试用例集;然后,根据程序实体覆盖矩阵对个体进行编码,以程序实体覆盖速率作为适应度函数,利用遗传算法对测试用例集进行排序. 选择 5 个经典的基准开源项目进行实验,以分支和方法作为程序实体,时间约束分别为 25% 和 75%. 实验结果表明,混合算法收敛速度快、稳定性好,优于传统整数线性规划方法. 该算法有助于尽早发现软件缺陷,降低回归测试成本.

关键词:测试用例优先排序;整数线性规划;遗传算法;时间约束

中图分类号:TP311.53