COMPLEX SYSTEM MODELING AND SIMULATION
ISSN 2096-9929 02/06 pp 12−31
Volume 3, Number 1, March 2023
DOI: 10.23919/CSMS.2022.0027

Search-Based Software Test Data Generation for Path Coverage Based on a Feedback-Directed Mechanism

Stuart Dereck Semujju, Han Huang, Fangqing Liu*, Yi Xiang, and Zhifeng Hao

Abstract: Automatically generating test cases by evolutionary algorithms to satisfy the path coverage criterion has attracted much research attention in software testing. In the context of generating test cases to cover many target paths, the efficiency of existing methods needs to be further improved when infeasible or difficult paths exist in the program under test. This is because a significant amount of the search budget (i.e., time allocated for the search to run) is consumed when computing fitness evaluations of individuals on infeasible or difficult paths. In this work, we present a feedback-directed mechanism that temporarily removes groups of paths from the target paths when no improvement is observed for these paths in subsequent generations. To fulfill this task, our strategy first organizes paths into groups. Then, in each generation, the objective scores of each individual for all paths in each group are summed up. For each group, the lowest value of the summed up objective scores among all individuals is assigned as the best aggregated score for a group. A group is removed when no improvement is observed in its best aggregated score over the last two generations. The experimental results show that the proposed approach can significantly improve path coverage rates for programs under test with infeasible or difficult paths in case of a limited search budget. In particular, the feedback-directed mechanism reduces wasting the search budget on infeasible paths or on difficult target paths that require many fitness evaluations before getting an improvement.

Key words: automated test case generation; software testing; path coverage; many-objective optimization

1 Introduction

Software testing is widely recognized as important activity for improving software quality. Due to its significance, researchers and software engineers have proposed automated test case generation techniques to improve the process over the years. In the context of white-box unit-testing, proposed techniques have been widely employed to satisfy structural coverage criteria such as statement coverage[1], branch coverage[2], and path coverage[3]. This work concentrates on path coverage, a widely studied structural test adequacy criterion. Ideally, path coverage seeks to maximize coverage of all feasible control flow paths through a program[4−10].

Automated Test Case Generation based on Path Coverage (ATCG-PC) has attracted much research attention in software testing in the past decade. However, a major hurdle that prevents wider application of existing approaches is the enormous number of paths in real-world programs. Moreover, many of these paths can be more difficult to cover than others or infeasible (i.e., impossible to cover). A path is infeasible if there does not exist a program input for

• Stuart Dereck Semujju, Han Huang, Fangqing Liu, and Yi Xiang are with the School of Software Engineering, South China University of Technology, Guangzhou 510006, China. E-mail: stuartsemujju@gmail.com; hhan@scut.edu.cn; peerfog@scut.edu.cn; gzhuxiang_yi@163.com.
• Zhifeng Hao is with the College of Science, Shantou University, Shantou 515063, China. E-mail: zfhaomail@gmail.com.
∗ To whom correspondence should be addressed.
※ This article was recommended by Associate Editor Wenyin Gong.
Manuscript received: 2022-08-31; revised: 2022-12-02; accepted: 2022-12-08

which the path can be traversed. For example, consider the simple program in Fig. 1a, and some of its paths in Fig. 1b. Paths P1, P2, and P4 (highlighted in bold) in Fig. 1b, are infeasible paths since the branch ($x > y$) at Line 6 cannot be triggered because when "then else" branch (i.e., $x \leqslant z$) at Line 3 is triggered, variables y and x contain the same values. A significant amount of the search time would be wasted during the evaluation of individuals on infeasible paths P1, P2, and P4. There can be several factors causing paths to be infeasible, such as defensive programming, dead code, and semantics of the program, that make it impossible to find a test input traversing the desired path. Hence, the presence of infeasible paths wastes the search time devoted to their coverage. Unfortunately, early detection of infeasible paths is a laborious task in programs with enormous number of paths.

To address ATCG-PC, two main techniques are mainly employed: symbolic execution[11–16] and search-based techniques[17–21]. Symbolic execution leverages constraint solvers to generate test cases systematically exploring program paths. Although symbolic execution has been widely studied in literature, its application is limited by several challenges when addressing test case generation in real-world programs such as constraint explosion, paths related to exceptions, and dependencies on external libraries. In contrast to symbolic execution, search-based techniques, such as Genetic Algorithms (GAs), require a fitness function to guide the search toward optimal solutions covering target paths.

There are two main search-based strategies for addressing ATCG-PC: single-target (i.e., one-path-at-a-time) and multi-target approaches. Single-target approaches consider one target path for coverage at a time[4, 6, 7, 22, 23]. Despite their simplicity, in the presence of an infeasible target path, the search budget would be entirely wasted. Furthermore, single-target approaches do not take advantage of collateral coverage, the phenomenon in which test cases generated when covering a target path accidentally cover other paths. Recent research shows that the performance of single-target approaches is equivalent to or worse than random search if collateral coverage is not leveraged[24]. In contrast to single-target approaches, multi-target approaches consider all coverage targets (e.g., paths) as objectives to optimize simultaneously[8, 9]. As such, the search budget is uniformly distributed across all paths. Recent empirical studies by Campos et al.[25] and Panichella et al.[26] show that multi-target strategies are more effective and efficient than single-target approaches when attempting to satisfy many coverage targets (e.g., paths).

```
s  int example1 (int x, int y, int z, int q){
1  int max=0; int i=0; int signal=0;
2  if (x>z) {q=0;}
3  else {y=x;}
4  while (i++<5){
5  if (x<q) {max−−;}
6  else if (x>y) {max=0;}
7  else {max++;}}
8  if (max==4) {signal=1;}
9  else {signal=−1;}
10  return signal;
e   }
```

(a) Example program

P1: **⟨s, 1, 3, 4, 6, 4, 7, 8, 4, 5, 4, 6, 4, 10, e⟩**
P2: **⟨s, 1, 3, 4, 6, 4, 7, 4, 5, 4, 5, 4, 9, e⟩**
P2: ⟨s, 1, 2, 4, 5, 4, 5, 4, 5, 4, 5, 4, 10, e⟩
P4: **⟨s, 1, 3, 4, 6, 4, 7, 4, 5, 4, 6, 4, 9, e⟩**

(b) Example of program's paths for Fig.1a

Fig. 1 Example of a program with infeasible paths.

Despite the superiority of multi-target strategies over single-target approaches, their efficiency needs to be further improved when a program under test has many difficult or infeasible paths and the time allotted for the search is limited. A significant amount of the search budget is consumed during the evaluation of individuals on difficult paths. Furthermore, a large portion of the search budget is wasted during the evaluation of individuals on infeasible paths.

In this work, we present an approach, called Feedback-Directed Algorithm for Path Coverage (FDA-PC) that is tailored for addressing ATCG-PC. In particular, FDA-PC reduces the chances of wasting the search time on infeasible paths or difficult paths. In particular, FDA-PC first organizes paths into groups. Then, the objective scores of each individual for all paths in each group are summed up in each generation. For each group, the minimum of the summed up objective scores among all individuals is assigned as the best aggregated score for a group. A group is removed when no improvement is observed in its best aggregated score over the last two generations. We devise this strategy so that the search time is devoted to evaluating individuals on paths with a higher chance to be covered.

For a clear exposition of the ideas discussed so far, this work is organized as follows. Section 2 presents an overview of search-based software test data generation and previous work is discussed. Section 3 presents the formulation of ATCG-PC as a many-objective optimization problem. Section 4 presents the proposed approach. Section 5 presents the description of the experiments we conducted for the evaluation of the proposed algorithm. Section 6 presents the results and discussions from the experiments conducted while Section 7 concludes this work.

2 Background

In this section, we present an overview of search-based software test data generation, basic concepts, and related work on search-based techniques addressing the test case generation problem.

2.1 Search-based software test data generation

Search-based software test data generation relies on the usage of meta-heuristic optimization techniques, such as Differential Evolution (DE)[27], GA[28], to find program inputs in the input domain of a program under test[29]. Suppose that U is a program under test with n input variables represented by vector $T=(t_1,t_2,\ldots,t_n)$. Assuming that ϕ_{t_i} $(1\leqslant i\leqslant n)$ is domain of the input variable t_i, then ϕ_{t_i} is the set of all values that t_i can hold. The input domain of U is a cross product of each input variable's domain: $\phi=\phi_{t_1}\times\phi_{t_2}\times\cdots\times\phi_{t_n}$. In general, a test input (i.e., test case in this context) is an input vector where each input is a specific element of the function's input domain.

Meta-heuristic optimization techniques, such as GAs evolve candidate, test cases over multiple generations to find test cases, also called individuals, covering structural targets (e.g., paths) in a program under test. The selection of individuals is guided by fitness functions, such that individuals with good fitness values have a higher probability to be selected for reproduction. The individuals selected for reproduction go through a crossover operator to generate offsprings. Then, a mutation operator is applied to introduce small changes to the offsprings. The evolution ends when an optimal solution is found or the search runs out of the allocated search budget.

2.2 Basic concepts

A Control Flow Graph (CFG) of a program under test is a tuple G = (V, E, s, e), where (V, E) is a finite directed graph; V is a set of nodes, with each node being basic code block; $E\subseteq V\times V$ is the set of edges connecting the nodes; $s,e\in V$ are unique entry and exit nodes of the program, respectively. A path in the CFG is a sequence of edge-connected nodes $P=<s,n_1,\ldots,n_m,e>$ starting at unique node s and ending at node e such that for all i, $1\leqslant i<m$, $(n_i,n_{i+1})\in E$.

Suppose that $P=\{\pi_1,\pi_2,\ldots,\pi_m\}$ is the set of paths in the CFG of a program under test. A path π_j $(1\leqslant j\leqslant m)$ is the j-th path in P. The length of π_j, denoted as $|\pi_j|$, refers to the number of nodes in π_j. Suppose that $X=\{x_1,x_2,\ldots,x_n\}$ is a set of individuals (i.e., test cases). The path traversed by test case $x\in X$ is denoted as $p(x)$. A test case x covers path π_j if and only if $p(x)$ and π_j have successively the same nodes from the entry node to the exit node.

2.3 Related work

The application of search-based techniques to address the test case generation problem has been the subject of extensive research efforts. Proposed approaches can be categorized into two strategies: single-target and multi-target approaches.

2.3.1 Single-target strategies

In single-target strategies, search-based algorithms, such as GAs, are run multiple times, once for every target path (i.e., one-path-at-a-time). Huang et al.[5] proposed a mathematical model to address ATCG-PC in fog computing systems as a single-objective problem. In their work, a relationship matrix is incorporated into DE algorithm to collect the correlation coefficients between test cases and paths from the used test cases. Bouchachia[22] improved test case generation based on path coverage by incorporating immune operators in a GA. Mala et al.[7] addressed ATCG-PC by using artificial bee colony optimization-based approach. Their approach combines both global search and local search to improve the efficiency of finding optimal solutions. Lin and Yeh[8] proposed an approach that extends hamming distance to calculate the fitness of individuals and applied a GA to search for optimal solutions.

Although single-target approaches have been widely applied, they are not well-suited for addressing ATCG-PC in programs under test with many paths, among which some are infeasible. When target paths are infeasible, the entire search budget would be wasted in attempting to cover them.

2.3.2 Multi-target strategies

Multi-target approaches consider all paths as objectives to optimize simultaneously. Ahmed and Hermadi[10] were the first to propose a multi-target approach. Their approach attempts to cover all target paths simultaneously in order to overcome the disadvantages of targeting one path at a time. The final fitness value of an individual is based on the path it performs best. For each path, the individual that has the best objective score with regard to that path in comparison to other test cases in the population is given a higher probability to be selected for reproduction. They employed a standard GA to search for test cases covering the paths.

Gong et al.[9] proposed an approach which generates test data to cover many target paths based on grouping. In their work, ATCG-PC is formulated as a many-objective optimization problem. They reduced the complexity of considering all coverage targets by dividing paths into groups as sub-optimization problems to optimize. The formulated sub-optimization problems are optimized in parallel by evolving sub-populations using GA. More specifically, each sub-population optimizes paths (i.e., objectives) in a particular group. The final fitness value of an individual in a sub-population is taken as the lowest value among all its fitness values for the paths in the particular group.

Fraser and Arcuri[2] proposed a multi-target approach, called Whole Test Suite (WTS) which optimizes all coverage targets simultaneously using GA. In their approach, an individual is a set of test cases (i.e., a test suite). As a fitness function, the sum of all intermediate distances by all test cases in the test suite from all the coverage targets is used. The computed sum of the intermediate distances is the fitness value of the test suite for the coverage targets in the program under test. At the end of the search, the best solution in the population is given as the output test suite.

Panichella et al.[1] proposed a generic many-objective optimization formulation of the test case generation problem. The overall fitness of a test case is measured based on an m-dimensional vector of m objectives, where each dimension corresponds to the fitness value for a particular test target. In their work, a many-objective algorithm called Dynamic Many-Objective Sorting Algorithm (DynaMOSA) is proposed to generate test cases satisfying the coverage targets. DynaMOSA narrows the search on coverage targets free of control dependencies. New coverage targets are iteratively considered when their dominators are covered. The authors argue that the control dependency graph concept employed in DynaMOSA can be extended to arbitrary coverage targets (e.g., paths).

Recent large scale studies by Campos et al.[25] and Panichella et al.[26] show that: (1) multi-target approaches are more effective than single-target approaches when dealing with many coverage targets, and (2) approaches employing an entirely many-objective search (i.e., using many-objective algorithms) to address the test case generation problem are superior to alternative multi-target approaches using standard GAs.

Although multi-target strategies are generally superior, their efficiency needs to be improved further when addressing ATCG-PC in programs under test with many paths, among which some are infeasible or difficult to cover. Unlike previous works, the multi-target strategy employed in our work attempts to focus on paths with higher chance to be covered rather than on infeasible or difficult paths that would consume a large portion of the search time during the evaluation of individuals.

3 Problem Formulation

Test case generation can be modeled as an optimization problem. As such, meta-heuristic algorithms can be used to search in the input domain of a program under test for test cases satisfying the structural targets. In this work, we formulate ATCG-PC as many-objective optimization problem in order to consider all paths for coverage simultaneously.

Suppose that there are m target paths for coverage, and $P=\{\pi_1, \pi_2, \ldots, \pi_m\}$ is the set of paths in the program under test. Find a set of test cases $X=\{x_1, x_2, \ldots, x_n\}$, which minimize the fitness functions for all paths $\pi_1, \pi_2, \ldots, \pi_m$, i.e., minimizing the following m objectives:

$$\begin{cases} \min f_1(x) = \sum\limits_{i=1}^{w} d(\pi_1^i, x); \\ \vdots \\ \min f_m(x) = \sum\limits_{i=1}^{w} d(\pi_m^i, x) \end{cases} \tag{1}$$

where π_j $(1 \leqslant j \leqslant m)$ is the j-th path, w is the number of branching nodes in the target path π_j; $d(\pi_j^i, x)$ denotes the branch distance. The branch distance quantifies how far a test case is from solving the conditional expression at the i-th branching node of path π_j, where the path $p(x)$ traversed by a test case diverges from π_j. The branch distance is computed using Korel's distance function[30]. For example, in Fig. 2b, suppose that path $p(x) = <s, 1, 2, 4, 6, e>$ (highlighted in bold) is the path traversed by test input x and path π_1: $<s, 1, 3, 4, 7, e>$ is one of the uncovered paths. In Fig. 2b,

```
s void example2 (int a, int b, int c){
1 intres=0;
2 if (a<=0) {a=a+10;}
3 else {a=a−10;}
4 if (a<=b) {res=a−b;}
5 else {res=a+b;}
6 if (res>c) {res=1;}
7 else {res=0;}
e }
```

(a) Example program

```
coveredpath p(x):    <s, 1, 2, 4, 6, e>
uncoveredpath π1:  <s, 1, 3, 4, 7, e>
uncoveredpath π2:  <s, 1, 3, 5, 6, e>
```

(b) Example of paths for Fig.2a

Fig. 2 Example of a program and some of its paths.

path $p(x)$ diverges from π_1 by not triggering the branching nodes at Line 3 ($a>0$) and at Line 7 (res $<= c$). The branch distance at Line 3 is computed as $dist_1=(0-a)+K$. The branch distance at Line 7 is computed as $dist_2=(\text{res}-c)+K$. K is a constant. The branch distances are normalized to a value between [0, 1]. For example $dist_1$ can be normalized as follows: $(1/(1+dist_1))$[31]. The sum of the accumulated normalized branch distances is the objective score of a test case for a particular path π_j. The fitness vector of a test case x is denoted as $\langle f_1,f_2,\ldots,f_m\rangle$, where each dimension corresponds to the objective score for an uncovered path.

4 Feedback-Directed Algorithm for Addressing ATCG-PC

This section presents the proposed method, called FDA-PC for addressing ATCG-PC. FDA-PC is based on NSGA-II[32], a widely known multi-objective genetic algorithm and the corner sort algorithm proposed in Ref. [33] to obtain non-dominated solutions. We highlight in bold the main modification over NSGA-II in Algorithm 1. In a nutshell, FDA-PC is designed to temporarily remove groups of paths from the target paths when no improvement is observed in two generations.

Algorithm 1 provides a high-level pseudo-code of FDA-PC. At the beginning, FDA-PC organizes paths into m/n groups, where m is the number of paths in the program under test and n is the population size (Line 2 of Algorithm 1). Each path is assigned to only one group. Suppose that $G=\{group_1,group_2,\ldots,group_k\}$ is the set of groups of paths. A group $group_i$ ($1\leqslant i\leqslant k$) is the i-th group of paths. The number of paths in $group_i$ is denoted as $|group_i|$. Figure 3 depicts an example of a program and some of its groups.

Algorithm 1 FDA-PC

```
Require: Set of paths $P=\{p_1,p_2,\ldots,p_m\}$ in the program under test and population size $n$
Ensure: Test suite $X$ covering the paths
1:  $H \longleftarrow \varnothing$ // Set of removed groups of paths
2:  $G \longleftarrow$ Create $m/n$ groups of paths
3:  $X \longleftarrow \varnothing$ // Set of generated test cases (test suite)
4:  $r \longleftarrow 0$ // Current generation
5:  $T_r \leftarrow$ Randomly generate initial population of $n$ individuals
6:  for $x \in T_r$ do
7:      Evaluate individual $x$ with respect to all paths in all the groups
8:      if $x$ covers a path in any group in $G$ then
9:          Remove covered path from the group
10:         $X \leftarrow X \bigcup \{x\}$
11:     end if
12: end for
13: while search budget not consumed do
14:     $C_r \leftarrow$ Apply reproduction operations on $T_r$ // Crossover and mutation
15:     for $x \in C_r$ do
16:         Evaluate individual $x$ with respect to all paths in all the groups
17:         if $x$ covers a path in any group in $G$ then
18:             Remove covered path from the group
19:             $X \leftarrow X \bigcup \{x\}$
20:         end if
21:         Compute sum of objective values for paths in each group $group_i \in G$
22:     end for
23:     if $r > 0$ then
24:         if $G \neq \varnothing$ then
25:             $H,G \longleftarrow$ REMOVE-PATHS $(G,C_r,r,H)$
26:             Update fitness vector for all individuals in $T_r$ and $C_r$
27:         end if
28:     end if
29:     $Pop \leftarrow T_r \bigcup C_r$
30:     $\mathcal{F} \longleftarrow$ CORNER-SORT $(Pop, G, n)$
31:     $T_{r+1} \longleftarrow \varnothing$
32:     $z \longleftarrow 0$
33:     while $\{|T_{r+1}|+|\mathcal{F}_z| \leqslant n$ do
34:         Perform crowding distance assignment on $\mathcal{F}_z$
35:         $T_{r+1} \longleftarrow T_{r+1} \bigcup \mathcal{F}_z$
36:         $z \longleftarrow z+1$
37:     end while
38:     Sort $(\mathcal{F}_z)$ // Based on crowding distance
39:     $T_{r+1} \longleftarrow$ Add individuals from all fronts to new starting from $\mathcal{F}_0$ to form the new population of size $n$
40:     if $r > 0$ and $G = \varnothing$ then
41:         $G \longleftarrow$ ADD-PATHS $(G,r,H)$
42:     end if
43:     $r \longleftarrow r+1$
44: end while
45: return $X$
```

```
s  int example3 (int x, int y, int z, int q){
1  int max=0;
2  int i=0;
3  int signal;
4  if (x>z) {q=0;}
5  else {y=x;}
6  while (i++<5){
7  if (max>q) {max−−;}
8  else if (max<q&&x>y) {q++;}
9  else {max++;}}
10 if (max==3) {signal=1;}
11 else {signal=0;}
e  return signal;}
```

(a) Example program

Group 1

P1: ⟨*s*, 1, 2, 3, 5, 6, 8, 6, 9, 6, 7, 6, 8, 6, 11, *s*⟩
P2: ⟨*s*, 1, 2, 3, 5, 6, 8, 6, 9, 6, 7, 6, 8, 6, 10, *e*⟩
P3: ⟨*s*, 1, 2, 3, 4, 6, 8, 6, 9, 6, 7, 6, 8, 6, 11, *e*⟩
P4: ⟨*s*, 1, 2, 3, 4, 6, 7, 6, 9, 6, 7, 6, 8, 6, 10, *s*⟩

Group 2

P1: ⟨*s*, 1, 2, 3, 5, 6, 9, 6, 8, 6, 8, 6, 9, 6, 12, *e*⟩
P2: ⟨*s*, 1, 2, 3, 4, 6, 9, 6, 9, 6, 7, 6, 8, 6, 11, *e*⟩
P3: ⟨*s*, 1, 2, 3, 5, 6, 8, 6, 8, 6, 8, 6, 8, 6, 11, *e*⟩
P4: ⟨*s*, 1, 2, 3, 4, 6, 7, 6, 7, 6, 7, 6, 7, 6, 11, *e*⟩

(b) Example of groups of paths

Fig. 3 Example of a program and some of its groups.

Next, FDA-PC randomly generates an initial population (i.e., test cases) of size n (Line 5 of Algorithm 1). Then, each test case is executed against the program under test and evaluated on all the paths in all the groups in G (Lines 6−12 of Algorithm 1). The "while" loop at Line 13 evolves the test cases until the search budget is consumed. Next, new offspring test cases are created using crossover and mutation (Line 14 of Algorithm 1). Similarly, the offspring test cases are evaluated on all paths in all the groups in G. In addition, for each group's paths, the corresponding objective scores of each individual for the paths is summed up. For example, the objective scores of a test case $x \in C_r$ for paths in $group_i \in G$ is computed as follows:

$$group_i(x) = \sum_{\pi_j \in group_i} f_j(x) \tag{2}$$

where π_j $(1 \leqslant j \leqslant |group_i|)$ is the j-th path in $group_i$.

Next, FDA-PC executes the function REMOVE-PATHS in Algorithm 2. Note that in the first generation, this function is not executed. It is executed in the subsequent generations. The rationale behind this function is to ensure when no improvement is observed for some groups of paths in two generations, they are temporarily removed from the target paths to cover in the next generation. More specifically, for each group, the lowest value of the summed up objective scores among all individuals is assigned as the best aggregated score for a group $group_i \in G$ (Lines 1−4 of Algorithm 2) in each generation r. A group is removed from G if its best aggregated score in the current generation is not better than its best aggregated score in the previous generation (Lines 5−10 of Algorithm 2). Next, the fitness vector for both the parent and offspring test cases is updated by removing objective scores corresponding to the paths in the groups that have been removed (Line 26 of Algorithm 1).

Algorithm 2 REMOVE-PATHS

```
Require: $G$: remaining groups of paths; $C_r$: new offsprings; $r$: current iteration; $H$: temporarily removed groups
Ensure: Updated groups of targets paths in $G$ and updated group of removed paths in $H$
1: for $group_i \in G$ do
2:     $x_{\min}$ ← Find test case in $C_r$ with the lowest sum of objectives score for $group_i$
3:     $group_i^{best}$ ← Get the sum of objective scores of $x_{\min}$ for $group_i$
4: end for
5: for $group_i \in G$ do
6:     if $group_i^{best}$ in $C_r$ is not better than $group_i^{best}$ in $C_{r-1}$
7:         $G$ ← Update $G$ by removing $group_i$ from $G$
8:         $H \longleftarrow H \bigcup group_i$ // Add $group_i$ to set of removed groups
9:     end if
10: end for
11: return $G$
```

Next, FDA-PC selects candidate test cases (Lines 29−39 of Algorithm 1). First, the parent and offspring test cases are combined to form an intermediate population Pop. Unlike NSGA-II that uses the traditional non-dominated sorting algorithm to rank individuals, we use the corner sort based algorithm[33]. Corner sort is applied to save comparisons when obtaining the non-dominated test cases (Algorithm 3). For example, suppose the size of population is n, corner sort only requires $(n-1)$ objective comparisons for each objective. In traditional non-dominated sorting 2 to m objective comparisons are required for the comparison of two individuals (i.e., test cases). Only $(n-1)$ comparisons are required for one single objective. The number of objective comparison times is fewer than $m(n-1)$. The function CORNER-SORT ranks the new population Pop using a preference procedure. More formally, an test case x is preferred to

Algorithm 3 CORNER-SORT

Require: Pop: current population; G: remaining groups of paths; and population size n

Ensure: $\mathcal{F}$: Ranking assignment of Pop

```
1: $j \longleftarrow 0$
2: while exists test case in $Pop$ not ranked do
3:   $\mathcal{F}_j \longleftarrow \varnothing$
4:   for each remaining group $group_i \in G$ do
5:     for each uncovered path in each group $\pi \in group_i$ do
6:       $x_{lowest} \longleftarrow$ Find test case $x$ in $Pop$ with lowest objective score for uncovered target $\pi$
7:       $\mathcal{F}_j \longleftarrow \mathcal{F}_j \bigcup \{x_{lowest}\}$
8:     end for
9:   end for
10:  $Pop \longleftarrow Pop - \mathcal{F}_j$
11:  $\mathcal{F} \longleftarrow \mathcal{F} \bigcup \mathcal{F}_j$
12:  if $|\mathcal{F}| \leqslant n$ then
13:    $j \longleftarrow j+1$
14:  end if
15: end while
16: return $\mathcal{F}$
```

Note: $\mathcal{F}_j$ is a subset of $\mathcal{F}$ which contains the test cases with the j-th ranking assignment of Pop.

a test case y for path π_j if and only if $f_j(x) < f_j(y)$. As such, a test case with lowest objective score among all test cases for a path π_j is assigned to the first non-dominated front $\mathcal{F}_0$ (Lines 4−9 of Algorithm 3). Similarly, to establish other fronts, the remaining test cases in Pop are ranked using the same procedure applied when assigning test cases to $\mathcal{F}_0$. After executing the CORNER-SORT function, FDA-PC proceeds to implement the crowding distance procedure to give more diverse test cases in the same front a higher chance of being selected in the next population.

In the subsequent generations, if G is empty after removing all groups of paths, the function ADD-PATHS (see Algorithm 4) adds the removed groups in the previous generations to the target groups of paths (i.e., added to G).

5 Empirical Evaluation

This section presents the empirical study we conducted to evaluate the proposed approach (FDA-PC). The empirical evaluation seeks to answer the following research questions:

● RQ1: How does FDA-PC perform compared to alternative search-based approaches addressing ATCG-PC in programs under test with many target paths?

Algorithm 4 ADD-PATHS

Require: G: remaining groups of paths; r: current generation; and H: temporarily removed groups

Ensure: Updated groups of paths in G

```
1: for $group_i \in H$ do
2:   if $group_i$ not removed in current generation $r$ then
3:     $H \leftarrow$ Update $H$ by removing $group_i$ from $H$
4:     $G \longleftarrow G \bigcup group_i$ //Add $group_i$ to $G$
5:   end if
6: end for
7: return $G$
```

This research question investigates to what extent FDA-PC is able to cover more paths when compared to alternative approaches addressing ATCG-PC.

● RQ2: What is the effectiveness of FDA-PC without the feedback-directed mechanism?

This research question aims at assessing the internal functioning of our approach. FDA-PC incorporates a feedback-directed mechanism that temporarily removes groups of paths when no improvement is observed in subsequent generations. Particularly, we investigate whether or not temporarily removing groups of paths when no improvement observed is enough to attain higher path coverage.

5.1 Baseline comparison

To answer the first research question, we compared FDA-PC with a single-target approach and four multi-target approaches tailored for addressing ATCG-PC:

● RP-DE[4]. It is a single-target approach that generates test cases to cover target paths. RP-DE uses a relationship matrix to empower Differential Evolution to address ATCG-PC.

● Method in Ref. [9]. It is a multi-target approach that evolves sub-populations to generate test cases covering target paths. First, the approach organizes paths into groups. Then, the paths in each group are transformed into objectives to optimize simultaneously. Each group is assigned a sub-population to optimize its objectives by using a standard GA. The final fitness of an individual (i.e., a test case) in a sub-population is computed as the minimum objective score among all its objective scores in its fitness vector obtained from a group.

● Method in Ref. [10]. It is a multi-target approach that generates test cases to simultaneously cover many target paths. The approach considers all paths as objectives to optimize simultaneously in all generations. The method employs a standard GA to find the optimal solutions. The final fitness value of a test case is based on the path it performs best. That is, the lowest objective score among all the objective scores of a test case for all paths is assigned as its final fitness value. Finally, for each path, the test case with the lowest objective score in comparison to other individuals in the population is given higher probability to be selected for reproduction.

● DynaMOSA is a many-objective test case generation algorithm that considers a subset of coverage targets at a time based on a control dependency hierarchy. DynaMOSA is an improvement of MOSA algorithm. MOSA[34] is many-objective test case generation algorithm that considers all coverage targets for optimization simultaneously. In DynaMOSA, new coverage targets are considered as test targets (e.g., branches) only when their dominators are covered based on a control dependency hierarchy. The control dependency graph concept employed in DynaMOSA can be extended to arbitrary coverage targets such as paths. We have extended the implementation of DynaMOSA and adapted it to path coverage.

● Whole Test Suite (WSA) with archive[35] is an improvement of the whole test suite generation[2]. In WTS a GA is used where an individual is a set of test cases (i.e., a test suite). As a fitness function, the sum of all accumulated distances from the coverage targets (i.e., paths in this context) in the program under test is used. Mutation and crossover operators are applied on the combined set of test suites. At the end of the search, the best solution in the population is given as output test suite. In addition, in the whole test suite generation, all coverage targets including those already covered during the search, are considered as part of the optimization until the search ends. WSA keeps a test archive for the already satisfied coverage targets, and focuses the search only on those coverage targets not yet satisfied. We have extended the implementation of WSA and adapted it to path coverage.

To answer the second research question (RQ2), we have built a variant of FDA-PC called No-FDA. No-FDA is a variant of FDA-PC without the feedback-directed mechanism.

5.2 Case study subjects

A key factor of evaluating test case generation approaches is the selection of programs under test. Our subjects consist of 27 programs. All these programs are sampled from the open source repositories, such as Software-artifact Infrastructure Repository[36], SF110[37], and Apache software foundation. Some of the programs are also considered in related literature[8, 9]. Details of the programs are shown in Table 1.

5.3 Parameter setting

We considered a number of parameters to control the performance of the approaches under evaluation. Table 2 shows the parameter settings used in our implementation of FDA-PC and its variant No-FDA.

For RP-DE we followed the same DE parameter settings in the original work[4] and assigned a maximum search time of 4 minutes. For the method in Ref. [9], the method in Ref. [10], WSA[2], and DynaMOSA[1], we set a maximum search time of 4 minutes and followed the same parameter settings used in their work.

The input variables were sampled from a range of [1, 10 000] for all approaches (i.e., FDA-PC, RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, DynaMOSA, and No-FDA).

5.4 Experimental procedure

We run FDA-PC, RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, DynaMOSA, and No-FDA for each program, collecting the resulting coverage. We set a maximum search time of 4 minutes. Hence, the search stops when the maximum time allocated for the search is consumed. Due to randomness of search algorithms, different results can be produced in different runs. Therefore, we repeated the experiments 30 times. Thus, we performed a total of 7 (i.e., FDA-PC, RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, DynaMOSA, and No-FDA) $\times$ 27 (programs) $\times$ 30 (repetitions) = 5670 experiments.

To answer research questions RQ1 and RQ2, we measure the percentage of covered paths as

$$\text{path\_coverage} = \frac{\text{Number of covered paths}}{\text{total paths to be covered}}.$$

We also conduct statistical analysis of the results. Statistical significance is measured by using the non-parametric Wilcoxon test[38] with a p-value threshold of 0.05. This is done to check whether the difference between any two approaches under comparison is statistically significant or not. In addition, we conduct the Vargha-Delaney ($\hat{A}_{12}$) statistical test[39] to measure the effect size. The Vargha-Delaney ($\hat{A}_{12}$) statistic also categorizes the obtained effect size into four different magnitude levels (i.e., negligible, small, medium, and large).

Table 1 Details of the subjects under study.

Package	Repository	Function	Number of paths	Number of inputs
org.apache.commons.math3	—	min	1024	10
		max	1024	10
		checkPositive	1024	10
		checkNonNegative	1024	10
coreNLP-master	—	entropy	1024	10
sglib-1.0.4	—	compare_counts	2187	14
		compare_unique_counts	1024	10
		check_that_int_array_is_sorted	1024	11
org.apache.commons.lang	—	normalizeSpace	2187	7
org.apache.commons.jcs.utils.config	—	convertSpecialChars	1000	12
org.apache.commons.lang3	—	containsAny	1024	20
		convertRemainingAccentCharacters	2187	7
flex_1.1	Software-artifact Infrastructure Repository	bubble	1024	10
grep_1.2		equal	1024	20
make_1.4		find_next_argument	625	12
sed_2.0		get_space	2187	7
org.apache.commons.cli	—	parsePattern	1024	5
corina	SF110 Corpus	stringDistance	1024	10
caloriecount		containsCharacters	1024	20
liferay		_hasNonASCIICode	1024	10
a4j		stripString	1024	20
caloriecount		toSlashClassName	1024	10
battlecry		metricMatch	2187	14
water-simulator		parseString	2187	7
lagoon		encodePath	1000	3
jaw_br		toAttributo	2187	7
liferay		getRGB	2401	4

Table 2 Parameter settings.

Parameter	Value
Population size	50 individuals
Crossover probability	0.75
Crossover operator	One-point crossover
Mutation operator	One-point mutation
Mutation probability	$1/h$
Maximum search time	4 minutes

Note: h is the number of inputs to the program under test.

6 Result and Discussion

This section presents the results and discussions to answer the research questions formulated in Section 5.

6.1 What is the performance of FDA-PC compared to alternative search-based approaches addressing ATCG-PC?

Table 3 summarizes the results of the average path_coverage and standard deviation for each averaged coverage value achieved by FDA-PC, RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, and DynaMOSA for each program. To better understand Table 3, the indicators used in the experiments are presented as follows:

- Path_coverage (%) (standard deviation σ): The average path_coverage achieved for each program over 30 independent runs and the standard deviation for each averaged coverage value.
- Mean over programs: The mean of the average path_coverage over all the programs.

We highlight in bold the programs where FDA-PC achieves higher average path_coverage than RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, and DynaMOSA. It can be observed that FDA-PC outperforms RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, and DynaMOSA in majority of the programs. Furthermore, FDA-PC achieved the highest overall mean coverage (61.29%).

Table 3 Average path_coverage and standard deviation achieved for each program by FDA-PC and alternative approaches.

Function	Path_coverage (%) (standard deviation σ)					
	FDA-PC	RP-DE	Method in Ref. [9]	Method in Ref. [10]	WSA	DynaMOSA
min	**98.00 (1.48)**	70.00 (5.89)	96.00 (0.82)	66.26 (6.62)	65.33 (12.02)	97.80(3.25)
max	**97.00 (0.72)**	72.00 (4.55)	95.00 (0.61)	93.00 (10.03)	94.26 (2.73)	94.00 (0.17)
checkPositive	100.00 (0.00)	99.00 (0.85)	99.00 (1.03)	100.00 (0.00)	99.76 (0.42)	96.00 (3.48)
checkNonNegative	100.00 (0.00)	98.00 (0.87)	100.00 (0.00)	98.00 (1.03)	99.73 (0.44)	92.00 (2.84)
equal	**78.00 (13.62)**	3.00 (0.58)	5.00 (0.68)	2.00 (0.41)	43.70 (4.96)	62.80 (3.03)
get_space	**35.00 (2.86)**	1.00 (0.37)	4.00 (0.86)	1.00 (0.68)	1.40 (0.80)	9.13 (2.04)
entropy	**44.00 (9.44)**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	30.46 (0.95)	36.63 (4.76)
compare_counts	**89.00 (7.27)**	22.00 (0.84)	26.00 (0.06)	18.00 (4.10)	31.80 (0.89)	93.00 (9.34)
compare_unique_counts	**48.00 (9.24)**	1.00 (0.00)	0.00 (0.00)	1.00 (0.00)	1.00 (0.00)	36.00 (12.42)
check_that_int_array_is_sorted	99.00 (0.00)	95.00 (1.32)	99.00 (0.00)	99.00 (1.31)	95.30(1.83)	99.00 (0.76)
normalizeSpace	**4.00 (0.82)**	0.00 (0.00)	1.00 (0.00)	0.00 (0.00)	0.20 (0.4)	0.00 (0.00)
convertSpecialChars	**15.00 (4.58)**	0.00 (0.51)	0.00 (0.10)	0.00 (0.10)	0.93(0.62)	7.00 (3.68)
bubble	80.00 (3.58)	89.00 (9.91)	91.00 (0.41)	97.00 (10.82)	66.26(6.64)	79.00 (10.37)
find_next_argument	**19.00 (1.82)**	5.00 (0.72)	8.00 (0.93)	2.00 (0.34)	14.63 (1.04)	9.83 (2.57)
containsAny	**100.00 (0.00)**	13.00 (2.10)	36.00 (2.17)	11.00 (2.44)	11.36 (2.77)	100.00 (0.00)
convertRemainingAccentCharacters	**82.00 (14.10)**	0.00 (0.10)	1.00 (0.72)	0.00 (0.10)	0.66 (0.64)	48.00 (14.65)
parsePattern	**32.00 (5.03)**	7.00 (1.58)	12.00 (0.89)	9.00 (1.89)	24.40 (2.93)	13.73 (0.92)
stringDistance	**54.00 (12.00)**	2.00 (0.37)	1.00 (0.72)	2.00 (0.86)	12.80 (0.54)	33.85 (0.95)
containsCharacters	**91.00 (2.03)**	15.00 (1.31)	34.00 (2.37)	29.00 (7.27)	16.90 (1.22)	83.00 (6.93)
_hasNonASCIICode	100.00 (0.00)	98.00 (0.96)	100.00 (0.00)	99.00 (0.75)	99.80 (0.40)	100.00 (0.00)
stripString	**93.00 (1.51)**	13.08 (1.34)	34.00 (2.06)	26.00 (6.96)	14.76 (1.33)	85.00 (9.00)
toSlashClassName	**28.00 (1.34)**	1.00 (0.72)	5.00 (0.37)	1.00 (0.55)	1.60 (0.75)	21.00 (1.45)
metricMatch	**8.00 (5.62)**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	2.26 (0.44)	3.80 (0.47)
parseString	**5.00 (0.00)**	0.00 (0.40)	1.00 (0.24)	0.00 (0.62)	0.50 (0.50)	1.00 (0.00)
encodePath	**97.00 (0.06)**	9.00 (2.72)	11.00 (2.34)	20.00 (4.00)	11.13 (2.61)	69.00 (0.42)
toAttributo	**3.00 (0.82)**	0.00 (0.27)	1.00 (0.00)	0.00 (0.58)	1.13 (1.04)	0.93 (0.24)
getRGB	**56.00 (5.03)**	2.80 (0.79)	5.00 (0.89)	33.86 (1.44)	3.00 (2.23)	41.48 (5.93)
Mean over programs (%)	61.29	26.48	32.03	29.88	31.29	52.30

Besides capturing the mean coverage and standard deviation, we report the effect size values obtained from Vargha-Delaney ($\hat{A}_{12}$) statistic, the magnitude of the difference, and p-values in Table 4. To better understand Table 4, the indicators used in the experiments are presented as follows:

● $\hat{A}_{12}$ statistics (magnitude) (p-value): The effect size, the magnitude of the difference, and p-value.

● vs.: It stands for versus. It indicates the comparison between FDA-PC against an alternative approach.

● +/=/−: These signs indicate the number of programs where FDA-PC performs better than, equivalently to, and worse than the compared approach, respectively, according to the Wilcoxon test.

In Table 4, we highlight in bold the effect size, the magnitude of the difference, and p-value for the programs, where FDA-PC is significantly better than another approach according to the Wilcoxon test. FDA-PC is significantly better than RP-DE in 25 programs. Among these programs, the magnitude of the difference is large in 23 program instances, small in 1 program instance, and negligible in 1 program instance. There is one program _hasNonASCIICode where statistically significant difference is not observed. In this program instance the magnitude of the difference is negligible. FDA-PC is significantly worse than RP-DE in one program instance, called bubble. In this program instance the magnitude of the difference is large.

Regarding the comparison between FDA-PC and the method in Ref. [9], FDA-PC is significantly better than

Table 4 Effect size and statistical significance achieved for each program following a comparison of FDA-PC with alternative approaches.

Function	$\hat{A}_{12}$ statistics (magnitude) (p-value)				
	FDA-PC vs. RP-DE	FDA-PC vs. method in Ref. [9]	FDA-PC vs. method in Ref. [10]	FDA-PC vs. WSA	FDA-PC vs. DynaMOSA
min	**1.00 (large) (< 0.001)**	**0.78 (large) (<0.001)**	0.47 (negligible) (0.809)	**1.00 (large) (< 0.001)**	**0.62 (small) (0.017)**
max	**1.00 (large) (< 0.001)**	**0.99 (large) (< 0.001)**	**0.62 (large) (< 0.001)**	**0.98 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
checkPositive	**0.58 (small) (0.025)**	0.50 (negligible) (1.000)	1.00 (large) (0.008)	**0.58 (small) (< 0.025)**	**0.77 (large) (< 0.001)**
checkNonNegative	**0.56 (negligible) (0.025)**	0.50 (negligible) (1.000)	0.55 (negligible) (0.102)	**0.61 (small) (< 0.001)**	**1.00 (large) (< 0.001)**
equal	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**0.96 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
get_space	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
entropy	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (0.006)**
compare_counts	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	0.72 (medium) (0.053)
compare_unique_counts	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**0.74 (large) (< 0.001)**
check_that_int_array_is_sorted	**1.00 (large) (< 0.001)**	0.50 (negligible) (1.000)	0.50 (negligible) (1.000)	**0.96 (large) (< 0.001)**	**0.50 (negligible) (0.025)**
normalizeSpace	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
convertSpecialChars	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
bubble	1.00 (large) (< 0.001)	0.15 (large) (< 0.001)}	1.00 (large) (< 0.001)	**1.00 (large) (< 0.001)**	**0.64 (small) (0.026)**
find_next_argument	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (<0.001)**	**1.00 (large) (0.005)**
containsAny	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (1.000)**
convertRemainingAccentCharacters	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
parsePattern	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
stringDistance	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
containsCharacters	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	0.50 (negligible) (1.000)
_hasNonASCIICode	0.53 (negligible) (0.157)	0.5 (negligible) (1.000)	**0.63 (small) (0.005)**	**0.61 (small) (0.005)**	0.50 (negligible) (1.000)
stripString	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
toSlashClassName	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
metricMatch	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
parseString	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
encodePath	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
toAttributo	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
getRGB	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**	**1.00 (large) (< 0.001)**
+/=/–	25/1/1	22/4/1	23/3/1	27/0/0	23/3/1

Note: “+” indicates the number of programs where FDA-PC performs better than the compared approach according to the Wilcoxon test. “=” indicates the number of programs where FDA-PC performs equivalently to the compared approach according to the Wilcoxon test. “–” indicates the number of programs where FDA-PC performs better than the compared approach according to the Wilcoxon test.

the method in Ref. [9] in 22 programs. The magnitude of the difference is large in the 22 programs. There are 4 programs where statistically significant difference is not observed. In all these programs, the magnitude of the difference is negligible. FDA-PC is significantly worse than the method in Ref. [9] in one program instance, bubble. In this program instance the magnitude of the difference is large.

FDA-PC is significantly better than the method in Ref. [10] in 23 programs. Among these programs, the magnitude of the difference is large in 22 programs instances and small in 1 instance. There are 3 programs where statistically significant difference is not observed. In these 3 programs, the magnitude of the difference is negligible in 2 instances and small in 1 instance. FDA-PC is significantly worse than the method in Ref. [10] in only 1 program, named bubble. In this program, the magnitude of the difference is large.

FDA-PC is significantly better than WSA in all the programs. Among these programs, the magnitude of the difference is large in 24 programs instances and small in 3 program instances.

Regarding the comparison between FDA-PC and DynaMOSA, FDA-PC is significantly better than DynaMOSA in 23 programs. The magnitude of the difference is large in 21 programs, small in 2 programs. There are 3 programs where statistically significant difference is not observed. The magnitude of the difference is negligible in all the 3 programs. FDA-PC is significantly worse than DynaMOSA in 1 program instance, named compare_counts. In this program instance, the magnitude of the difference is medium.

To provide more insight into the distribution of the path coverage scores, Figs. 4−6 highlight that FDA-PC leads to larger path coverage scores in a majority of the programs, which is verified by the mean, maximum, minimum, and median of the average path coverage.

We notice that FDA-PC outperforms RP-DE, the method in Ref. [9], the method in Ref. [10], WSA, and DynaMOSA on a large number of programs. We can conclude that in case of limited search budget, FDA-PC improves average path coverage scores in programs under test with many paths, among which some are infeasible. More specifically, the feedback-directed mechanism employed in FDA-PC reduces the probability that a significant portion of the search budget is consumed in attempting to cover infeasible paths or difficult paths. Hence, the search effort is always focused on paths that are more likely to covered.

6.2 What is the impact of the feedback-directed mechanism used in FDA-PC?

Table 5 summarizes the results of the average path_coverage and standard deviation for each averaged coverage value achieved by FDA-PC, and its variant No-FDA for each program. To better understand Table 5, the indicators used in the experiments are presented as follows:

- Path_coverage (%) (standard deviation σ): The average path_coverage achieved for each program over 30 independent runs and the standard deviation for each averaged coverage value.
- Mean over programs: The mean of the average path_coverage over all the programs.
- $\hat{A}_{12}$ statistics (magnitude) (p-value): The effect size, the magnitude of the difference, and p-value.
- vs.: It stands for versus. It indicates the comparison between FDA-PC against No-FDA.
- +/=/−: These signs indicate the number of programs where FDA-PC performs better than, equivalently to, and worse than No-FDA, respectively, according to the Wilcoxon test.

We highlight in bold the programs where FDA-PC achieved higher average path_coverage than its derivative No-FDA. It can be observed that FDA-PC outperformed its variant No-FDA in all the programs. Besides capturing the mean coverage and standard deviation, we report the effect size values obtained from Vargha-Delaney $\hat{A}_{12}$ statistic, the magnitude of the difference, and p-values for the programs where FDA-PC is significantly better than its variant No-FDA according to the Wilcoxon test. FDA-PC achieved significantly better coverage than No-FDA in all the 27 programs. The magnitude of the difference is large in all the 27 program instances. To provide more insight into the distribution of the path_coverage scores, Figs. 7−9 highlight that FDA-PC achieved higher path_coverage scores in a majority of the cases, which is verified by the mean, maximum, minimum, and median of the average path_coverage. Hence, we can conclude that the feedback-directed mechanism is a key component in the overall performance of FDA-PC.

7 Conclusion

We have presented a feedback-directed algorithm for addressing ATCG-PC (FDA-PC) in programs under

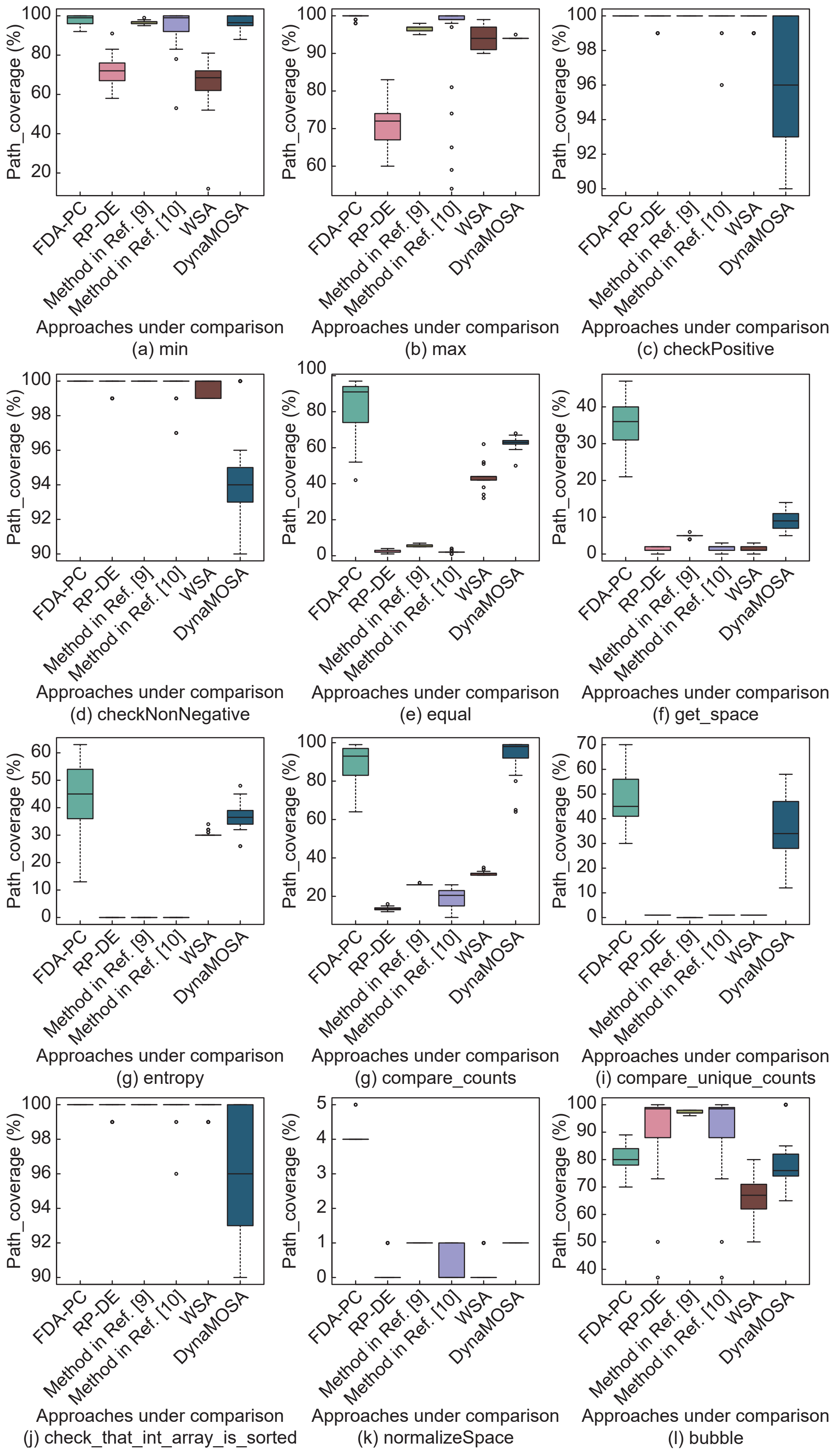

Fig. 4 Box plots for min, max, checkPositive, checkNonNegative, equal, get_space, entropy, compare_counts, compare_unique_counts, check_that_int_array_is_sorted, normalizeSpace, and bubble.

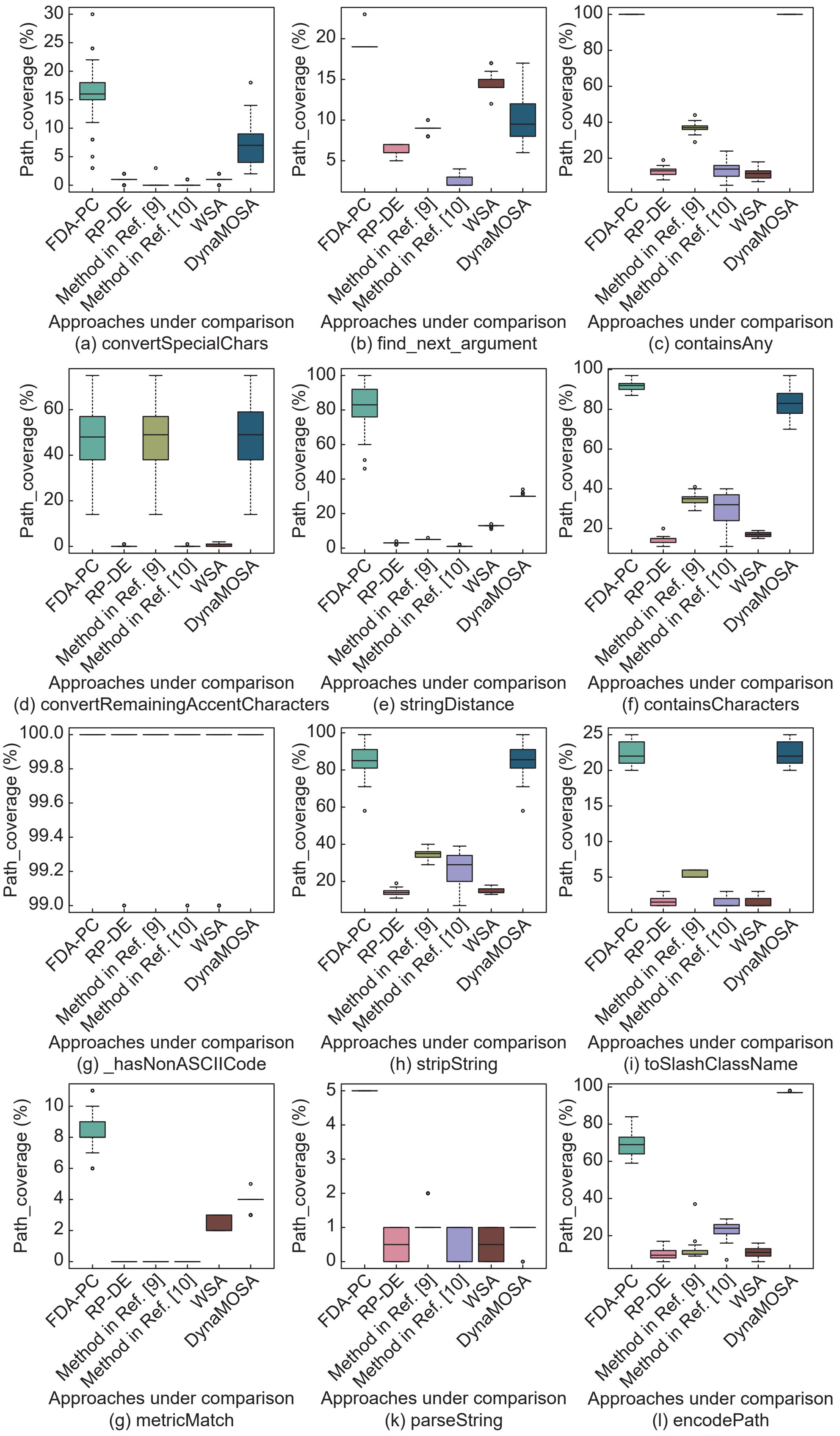

Fig. 5 Box plots for convertSpecialChars, find_next_argument, containsAny, convertRemainingAccentCharacters, stringDistance, containsCharacters, _hasNonASCIICode, stripString, toSlashClassName, metricMatch, parseString, and encodePath.

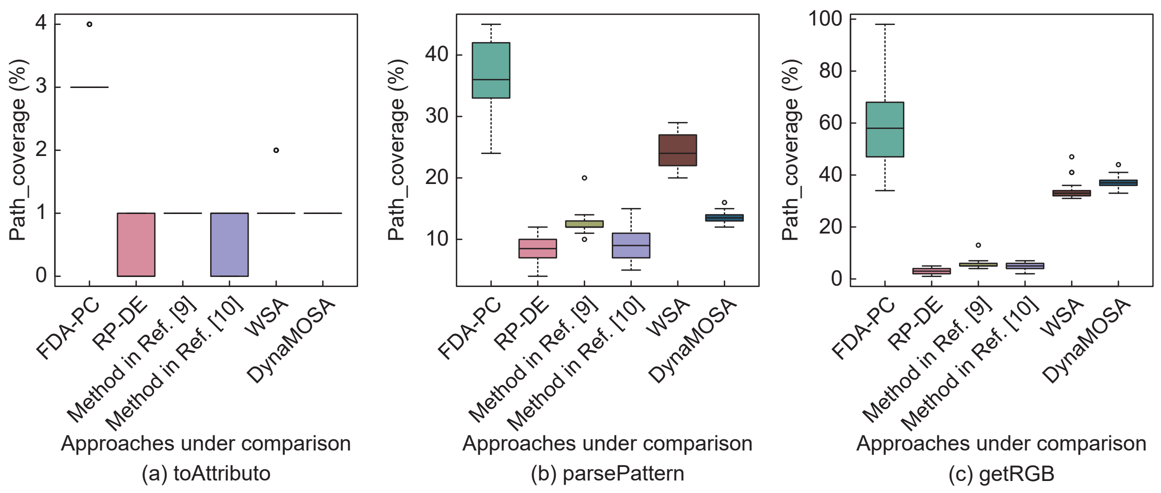

Fig. 6 Box plots for toAttributo, parsePattern, and getRGB.

Table 5 Mean path_coverage, standard deviation, effect size, and statistical significance achieved by FDA-PC and No-FDA.

Function	Path_coverage (%) (standard deviation σ)		$\hat{A}_{12}$ statistics (magnitude) (p-value) for FDA-PC vs. No-FDA
	FDA-PC	No-FDA	
min	**98.00 (1.48)**	48.00 (15.09)	**1.00 (large) (< 0.001)**
max	**97.00 (0.72)**	72.00 (17.47)	**1.00 (large) (< 0.001)**
checkPositive	**100.00 (0.00)**	51.00 (6.06)	**1.00 (large) (< 0.001)**
checkNonNegative	**100.00 (0.00)**	56.00 (9.17)	**1.00 (large) (< 0.001)**
equal	**78.00 (13.62)**	12.00 (3.75)	**1.00 (large) (< 0.001)**
get_space	**35.00 (2.86)**	5.00 (0.66)	**1.00 (large) (< 0.001)**
entropy	**44.00 (9.44)**	9.00 (1.69)	**1.00 (large) (< 0.001)**
compare_counts	**89.00 (7.27)**	15.00 (4.72)	**1.00 (large) (< 0.001)**
compare_unique_counts	**48.00 (9.24)**	8.00 (1.28)	**1.00 (large) (< 0.001)**
check_that_int_array_is_sorted	**99.00 (0.00)**	57.00 (12.00)	**1.00 (large) (< 0.001)**
normalizeSpace	**4.00 (0.82)**	0.00 (0.52)	**1.00 (large) (< 0.001)**
convertSpecialChars	**15.00 (4.58)**	3.00 (0.44)	**1.00 (large) (< 0.001)**
bubble	**80.00 (3.58)**	38.00 (10.06)	**1.00 (large) (< 0.001)**
find_next_argument	**19.00 (1.82)**	14.00 (0.55)	**0.98 (large) (< 0.001)**
containsAny	**100.00 (0.00)**	37.00 (11.17)	**1.00 (large) (< 0.001)**
convertRemainingAccentCharacters	**82.00 (14.1)**	5.00 (1.32)	**1.00 (large) (< 0.001)**
parsePattern	**32.00 (5.03)**	6.00 (1.62)	**1.00 (large) (< 0.001)**
stringDistance	**54.00 (12.00)**	5.00 (2.58)	**1.00 (large) (< 0.001)**
containsCharacters	**91.00 (2.03)**	18.00 (4.32)	**1.00 (large) (< 0.001)**
_hasNonASCIICode	**100.00 (0.00)**	99.00 (2.00)	**0.9 (small) (< 0.001)**
stripString	**93.00 (1.51)**	19.00 (4.20)	**1.00 (large) (< 0.001)**
toSlashClassName	**28.00 (1.34)**	11.00 (1.58)	**1.00 (large) (< 0.001)**
metricMatch	**8.00 (5.62)**	8.00 (3.83)	**1.00 (large) (< 0.001)**
parseString	**5.00 (0.00)**	2.00 (0.42)	**1.00 (large) (< 0.001)**
encodePath	**97.00 (0.06)**	9.00 (3.78)	**1.00 (large) (< 0.001)**
toAttributo	**3.00 (0.82)**	1.00 (0.52)	**1.00 (large) (< 0.001)**
getRGB	**56.00 (5.03)**	22.00 (6.28)	**1.00 (large) (< 0.001)**
Mean over programs	61.29	23.33	–
+/=/–	–	–	27/0/0

Note: "+" indicates the number of programs where FDA-PC performs better than No-FDA according to the Wilcoxon test. "=" indicates the number of programs where FDA-PC performs equivalently to No-FDA according to the Wilcoxon test. "–" indicates the number of programs where FDA-PC performs better than No-FDA according to the Wilcoxon test.

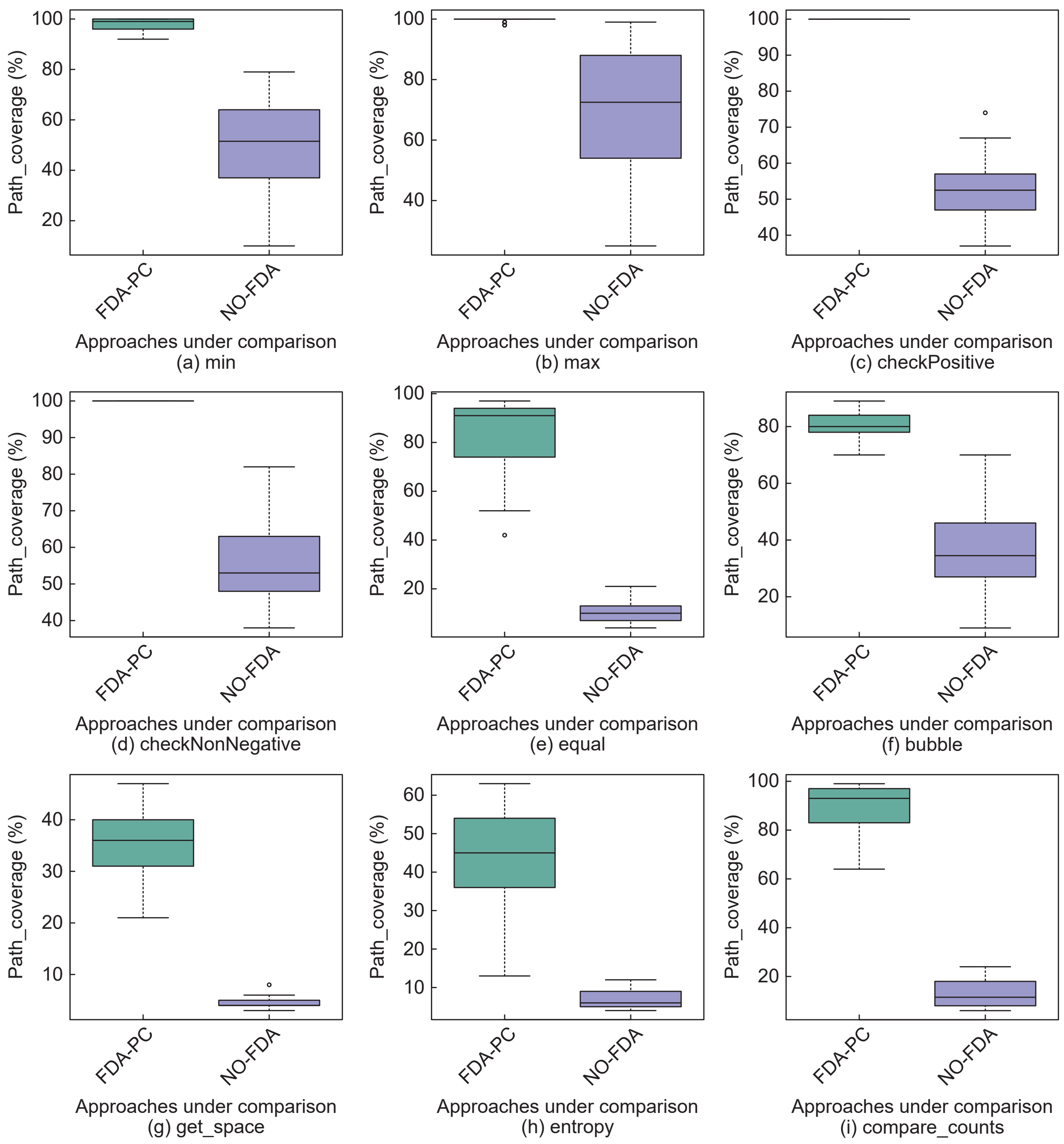

Fig. 7 Box plots for min, max, checkPositive, checkNonNegative, equal, bubble, get_space, entropy, and compare_counts.

test with many paths, among which some are infeasible or difficult to cover. The feedback-directed mechanism temporarily removes groups of paths from the target paths when no improvement is observed in subsequent generations. As such, the search concentrates on the paths that are more likely to be covered. We have carried out an empirical study to compare FDA-PC with other test case generation approaches. Results show that FDA-PC achieves higher path coverage on average than alternative approaches RP-DE[4], the method in Ref. [9], the method in Ref. [10], WSA[2], and DynaMOSA[1] when many infeasible or difficult paths exist in a program under test and a limited time is given to the search. In addition, FDA-PC also achieves higher coverage levels than its variant without the feedback-directed mechanism. Therefore, our method is well suited for generating test cases of programs under test with many target paths.

However, there are some shortcomings in our work. The instrumentation of the program under test is done manually. For future work we plan to build an automated tool to instrument the program under test. In addition, while this work focuses only on ATCG-PC, we plan to include other non-coverage aspects such as

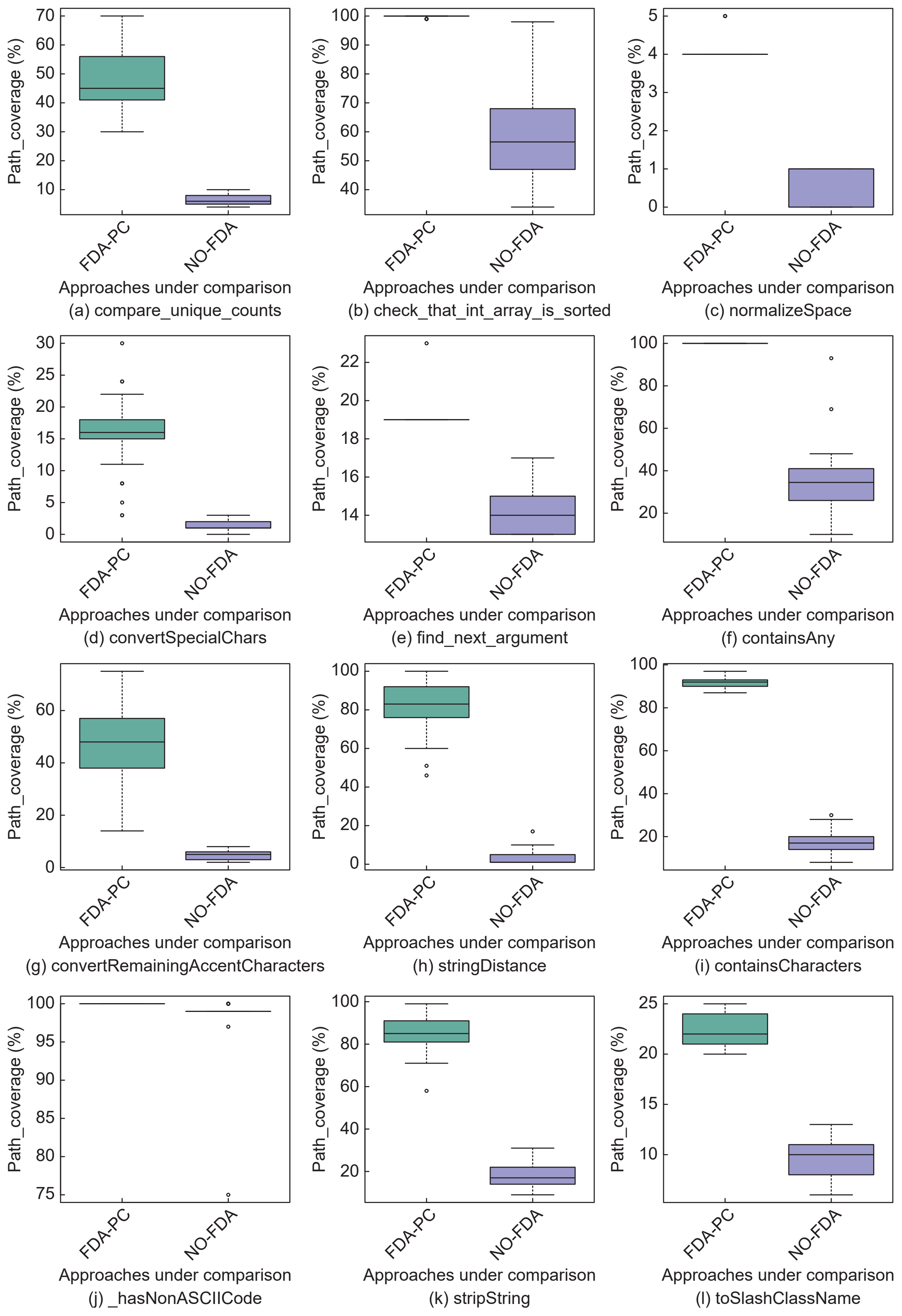

Fig. 8 Box plots for compare_unique_counts, check_that_int_array_is_sorted, normalizeSpace, convertSpecialChars, find_next_argument, containsAny, convertRemainingAccentCharacters, stringDistance, containsCharacters, _hasNonASCIICode, stripString, and toSlashCl-assName.

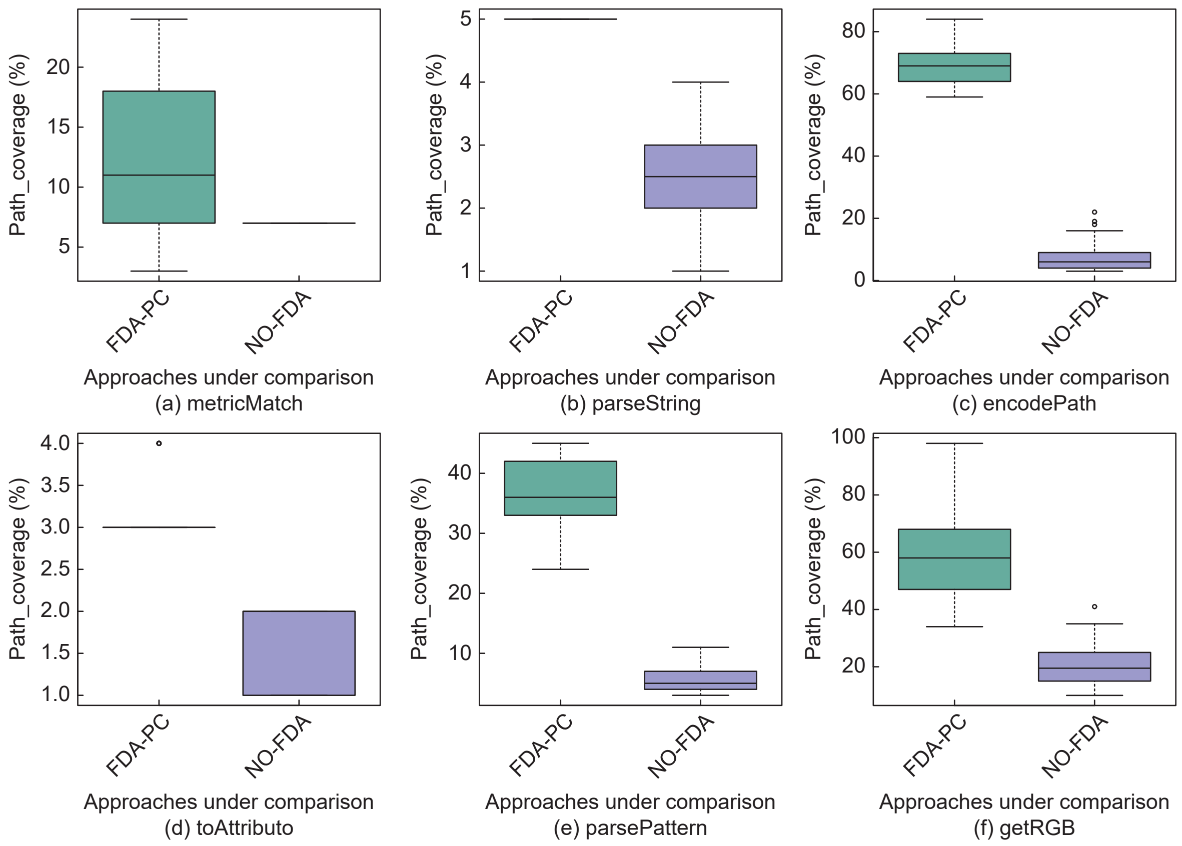

Fig. 9 Box plots for metricMatch, parseString, encodePath, toAttributo, parsePattern, and getRGB.

run time and memory usage as objectives to optimize in future works.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61876207), the Natural Science Foundation of Guangdong Province (No. 2022A1515011491), and the Fundamental Research Funds for the Central Universities (No. 2020ZYGXZR014).

References

[1] A. Panichella, F. M. Kifetew, and P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 122–158, 2018.

[2] G. Fraser and A. Arcuri, Whole test suite generation, *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[3] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, and A. De Lucia, Search-based testing of procedural programs: Iterative single-target or multi-target approach? in *Proc. 8th Int. Symp. Search Based Software Engineering*, Raleigh, NC, USA, 2016, pp. 64–79.

[4] J. R. Horgan, S. London, and M. R. Lyu, Achieving software quality with testing coverage measures, *Computer*, vol. 27, no. 9, pp. 60–69, 1994.

[5] H. Huang, F. Liu, Z. Yang, and Z. Hao, Automated test case generation based on differential evolution with relationship matrix for iFogSim toolkit, *IEEE Trans. Ind. Inf.*, vol. 14, no. 11, pp. 5005–5016, 2018.

[6] J. Wegener, A. Baresel, and H. Sthamer, Evolutionary test environment for automatic structural testing, *Inf. Software Technol.*, vol. 43, no. 14, pp. 841–854, 2001.

[7] D. J. Mala, V. Mohan, and M. Kamalapriya, Automated software test optimisation framework—an artificial bee colony optimisation-based approach, *IET Software*, vol. 4, no. 5, pp. 334–348, 2010.

[8] J. C. Lin and P. L. Yeh, Automatic test data generation for path testing using GAs, *Inf. Sci.*, vol. 131, nos. 1–4, pp. 47–64, 2001.

[9] D. Gong, W. Zhang, and X. Yao, Evolutionary generation of test data for many paths coverage based on grouping, *J. Syst. Software*, vol. 84, no. 12, pp. 2222–2233, 2011.

[10] M. A. Ahmed and I. Hermadi, GA-based multiple paths test data generator, *Comput. Oper. Res.*, vol. 35, no. 10, pp. 3107–3124, 2008.

[11] R. E. Prather and J. P. Myers, The path prefix software testing strategy, *IEEE Trans. Software Eng.*, vol. SE-13, no. 7, pp. 761–766, 1987.

[12] L. A. Clarke, A system to generate test data and symbolically execute programs, *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, pp. 215–222, 1976.

[13] N. Tillmann and J. De Halleux, Pex-white box test generation for. NET, in *Proc. Second Int. Conf. Tests and Proofs*, Prato, Italy, 2008, pp. 134–153.

[14] K. Sen, D. Marinov, and G. Agha, CUTE: A concolic unit testing engine for C, *ACM SIGSOFT Software Eng. Notes*, vol. 30, no. 5, pp. 263–272, 2005.
[15] P. Godefroid, N. Klarlund, and K. Sen, DART: Directed automated random testing, *ACM SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.
[16] M. M. Eler, A. T. Endo, and V. H. S. Durelli, Quantifying the characteristics of Java programs that may influence symbolic execution from a test data generation perspective, in *Proc. IEEE 38th Annu. Computer Software and Applications Conf.*, Vasteras, Sweden, 2014, pp. 181–190.
[17] M. Harman, Software engineering meets evolutionary computation, *Computer*, vol. 44, no. 10, pp. 31–39, 2011.
[18] R. Malhotra and M. Khari, Heuristic search-based approach for automated test data generation: A survey, *Int. J. Bio-Inspired Comput.*, vol. 5, no. 1, pp. 1–18, 2013.
[19] M. Khari and P. Kumar, An extensive evaluation of search-based software testing: A review, *Soft Comput.*, vol. 23, no. 6, pp. 1933–1946, 2019.
[20] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, andom or genetic algorithm search for object-oriented test suite generation, in *Proc. Annu. Conf. Genetic Evol. Comput.*, Madrid, Spain, 2015, pp. 1367–1374
[21] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, Combining multiple coverage criteria in search-based unit test generation, in *Proc. 7th Int. Symp. Search Based Software Engineering*, Bergamo, Italy, 2015, pp. 93–108.
[22] A. Bouchachia, An immune genetic algorithm for software test data generation, in *Proc. 7th Int. Conf. Hybrid Intelligent Systems*, Kaiserslautern, Germany, 2007, pp. 84–89.
[23] N. Zhang, B. Wu, and X. Bao, Automatic generation of test cases based on multi-population genetic algorithm, *Int. J. Multimedia Ubiquitous Eng.*, vol. 10, no. 6, pp. 113–122, 2015.
[24] A. Arcuri, M. Z. Iqbal, and L. Briand, Formal analysis of the effectiveness and predictability of random testing, in *Proc. 19th Int. Symp. Software Testing and Analysis*, Trento, Italy, 2010, pp. 219–230.
[25] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, An empirical evaluation of evolutionary algorithms for unit test suite generation, *Inf. Software Technol.*, vol. 104, pp. 207–235, 2018.
[26] A. Panichella, F. M. Kifetew, and P. Tonella, A large scale empirical comparison of state-of-the-art search-based test case generators, *Inf. Software Technol.*, vol. 104, pp. 236–256, 2018.
[27] R. Storn and K. Price, Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces, *J. Global Optim.*, vol. 11, no. 4, pp. 341–359, 1997.
[28] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: University of Michigan Press, 1975.
[29] C. C. Michael, G. McGraw, and M. A. Schatz, Generating software test data by evolution, *IEEE Trans. Software Eng.*, vol. 27, no. 12, pp. 1085–1110, 2001.
[30] B. Korel, Automated software test data generation, *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990.
[31] A. Arcuri, It does matter how you normalise the branch distance in search based software testing, in *Proc. 3rd Int. Conf. Software Testing, Verification and Validation*, Paris, France, 2010, pp. 205–214.
[32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.
[33] H. Wang and X. Yao, Corner sort for Pareto-based many-objective optimization, *IEEE Trans. Cybern.*, vol. 44, no. 1, pp. 92–102, 2014.
[34] A. Panichella, F. M. Kifetew, and P. Tonella, Reformulating branch coverage as a many-objective optimization problem, in *Proc. IEEE 8th Int. Conf. Software Testing, Verification and Validation*, Graz, Austria, 2015, pp. 1–10.
[35] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, A detailed investigation of the effectiveness of whole test suite generation, *Empirical Software Eng.*, vol. 22, no. 2, pp. 852–893, 2017.
[36] H. Do, S. Elbaum, and G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, *Empirical Software Eng.*, vol. 10, no. 4, pp. 405–435, 2005.
[37] G Fraser and A Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, *ACM Trans. Software Eng. Methodol.*, vol. 24, no. 2, p. 8, 2014.
[38] W. J. Conover, *Practical Nonparametric Statistics*., 3rd ed. Hoboken, NJ, USA: Wiley, 1998.
[39] A. Vargha and H. D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong, *J. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000.

Stuart Dereck Semujju received the BEng degree in software engineering from Makerere University, Kampala Uganda in 2014, and the MEng degree in computer applications technology from Shenyang Aerospace University, China in 2017. He is currently a PhD candidate in software engineering at the School of Software Engineering, South China University of Technology, Guangzhou, China. His research interests include routing in wireless sensor networks, automated software test case generation, and evolutionary computation for software testing.

Han Huang received the BEng degree in information management and information system from South China University of Technology (SCUT), Guangzhou, China in 2003, and the PhD degree in computer science from SCUT in 2008. He is currently a full professor at the School of Software Engineering, SCUT. His research interests include theoretical foundation and application of evolutionary computation and microcomputation. He is a distinguished member of CCF, and a senior member of IEEE.

Fangqing Liu received the PhD degree in software engineering from South China University of Technology, Guangzhou, China in 2021. He is currently a postdoctoral researcher in software engineering at School of Software Engineering, South China University of Technology, Guangzhou, China. His current research interests include automated software test case generation and evolutionary computation for software testing.

Yi Xiang received the BS and MS degrees in mathematics from Guangzhou University, Guangzhou, China in 2010 and 2013, respectively, and the PhD degree in computer science from Sun Yat-sen University, Guangzhou, China in 2018. He is currently an associate professor at the School of Software Engineering, South China University of Technology, Guangzhou. His current research interests include many-objective optimization and search-based software engineering.

Zhifeng Hao received the BS degree in mathematics from Sun Yat-sen University, Guangzhou, China in 1990, and the PhD degree in mathematics from Nanjing University, Nanjing, China in 1995. He is currently a professor at the College of Science, Shantou University, Shantou, Guangdong, China. His current research interests include various aspects of algebra, machine learning, data mining, and evolutionary algorithms. He is a senior member of IEEE.