The Journal of Systems and Software 161 (2020) 110495

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Augmenting ant colony optimization with adaptive random testing to cover prime paths

Atieh Monemi Bidgoli, Hassan Haghighi*

Faculty of Computer Science and Engineering, Shahid Beheshti University G.C.,Tehran, Iran

ARTICLE INFO

Article history:
Received 11 February 2019
Revised 26 November 2019
Accepted 10 December 2019
Available online 16 December 2019

Keywords:
Search-based test data generation
Ant colony optimization
Prime path coverage
Adaptive random testing
Mutation analysis

ABSTRACT

Test data generation has a notable impact on the performance of software testing. A well-known approach to automate this activity is search-based test data generation. Most studies in this area use branch coverage as the test criterion. Since the prime path coverage criterion includes branch coverage, it has higher probability to detect software failures than the branch coverage criterion. This paper customizes and improves ant colony optimization (ACO) to provide a test data generation approach for covering prime paths. The proposed approach incorporates the notion of input space partitioning to maintain pheromone values in the search space. In addition, it employs the idea of adaptive random testing in the local search. At last, it uses the information of program predicates in order to make a relation between the logic of the program and pheromone values in the search space. The experimental results confirm the positive effects of the mentioned contributions, especially for programs with complex predicates. Furthermore, they represent that, on average, test suites generated by the proposed approach has 9% better mutation score in comparison to test suites produced by EvoSuite, a well-known test data generation tool.

1. Introduction

Testing is still the most popular approach for evaluating the reliability of software systems, and test data generation is a key activity influencing the efficiency and effectiveness of the testing process (Bertolino, 2007). Search-based test data generation has been widely proposed (Ali et al., 2010) to automate this activity. This approach applies meta-heuristic search techniques guided by fitness functions to find those test data capturing the current test goal.

Test data generation methods can be divided into two distinct categories: functional and structural. In the former, functional specification of the software under test (SUT) is used as the source for test data generation, whereas for the latter, test data are generated based on the structural elements of the SUT. Compared with functional testing, structural test data generation is more cost-effective to detect failures of the programs and has been widely applied in practice. In the structural search based test data generation, the notion of control flow graph (CFG) is often used to model the source code of the SUT. The well-known criteria to cover CFGs modeling source code are node coverage (or statement coverage), edge coverage (or branch coverage), edge-pair coverage, and prime path coverage (PPC). The last criterion has essentially been proposed as a practical alternative to the complete path coverage (CPC) criterion (Ammann and Offutt, 2016) because CPC may result in high (or even infinite) number of test requirements for programs having loops.

With respect to PPC, tests must tour every prime path in the given CFG. A prime path is a simple path which is not included in any other simple path. A simple path is a path without repetitive nodes, except the first and the last nodes which may be the same. Since sub-paths of length 0 (i.e., nodes) as well as sub-paths of length 1 (i.e., edges) are prime, PPC subsumes node coverage and branch coverage (BC) (Fig. 1). Moreover, in cases where there are loops, PPC results in executing each loop zero times, once, and more than once. Thus, it supports the zero-one-multiple strategy (Ammann and Offutt, 2016) while it does not suffer from the problem of infinity in CPC.

Most of the researches in search-based test data generation are based on the BC criterion as the test goal and propose fitness functions with respect to this goal. Since PPC subsumes BC, this paper aims at proposing a search-based approach to automatically generate test data regarding the PPC criterion. To the best of our knowledge, this is the first search-based test data generation work which explicitly considers the PPC criterion (Durelli et al., 2018).

To show that applying PPC may expose failures which cannot be detected using BC, consider the sample code given in Fig. 2. The given code comprises a fault in line 10 (i.e., b/c). Since c = 0

* Corresponding author.
E-mail addresses: a_monemi@sbu.ac.ir (A. Monemi Bidgoli), h_haghighi@sbu.ac.ir (H. Haghighi).

https://doi.org/10.1016/j.jss.2019.110495
0164-1212/© 2019 Elsevier Inc. All rights reserved.

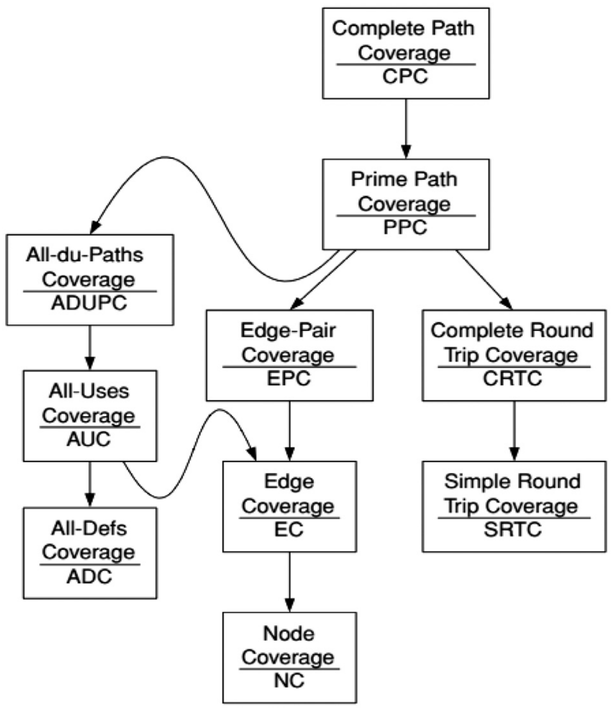
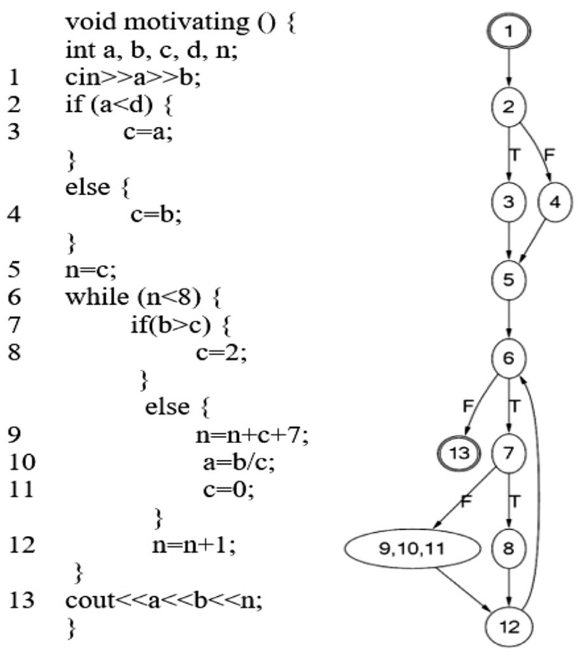


Fig. 1. Subsumption relationship among structural coverage criteria (Ammann and Offutt, 2016).



```
    void motivating () {
    int a, b, c, d, n;
1   cin>>a>>b;
2   if (a<d) {
3           c=a;
    }
    else {
4           c=b;
    }
5   n=c;
6   while (n<8) {
7           if(b>c) {
8                   c=2;
            }
            else {
9                   n=n+c+7;
10                  a=b/c;
11                  c=0;
            }
12          n=n+1;
    }
13  cout<<a<<b<<n;
    }
```

Fig. 2. An example program and its CFG.

in the second iteration of the present loop, this fault provokes the "division by zero" failure. According to the test paths outlined in Table 1, this failure is detected if we execute path 7, i.e., [1,2,4,5,6,7,9,10,6,7,9,10,11,12,6,13]. Unlike the statement coverage and branch coverage criteria, the PPC criterion forces us to traverse this path.

Although PPC subsumes BC, as (Frankl and Weyuker, 1993) showed, if criterion C1 subsumes criterion C2, we cannot necessarily conclude that C1 has a more capability to detect failures. Therefore, the authors in Durelli et al. (2018); Li et al. (2009b), by mutation analysis, experimentally investigated the failure detection capability of different coverage criteria. Mutation analysis has been proposed to experimentally investigate the failure detection capability of test data generation approaches (Andrews et al., 2006; Gupta and Jalote, 2008). The authors in Durelli et al. (2018) conducted an experiment to probe into the fault finding ability of three structural coverage criteria: edge coverage, edge-pair coverage, and prime path coverage. The result of experiments on 189 functions from 39 C programs showed that PPC has higher ability to detect failures with the same cost.

Table 1
Test paths for the given example, according to BC and PPC.

	BC	PPC
1	[1,2,3,5,6,13]	[1,2,3,5,6,7,9,10,11,12,6,13]
2	[1,2,4,5,6,7,9,10,11,12,6,13]	[1,2,3,5,6,7,8,12,6,13]
3	[1,2,3,5,6,7,8,12,6,13]	[1,2,4,5,6,13]
4		[1,2,3,5,6,13]
5		[1,2,4,5,6,7,9,10,6,7,8,12,6,13]
6		[1,2,4,5,6,7,8,10,6,7,9,10,11,12,6,13]
7		[1,2,4,5,6,7,9,10,6,7,9,10,11,12,6,13]
8		[1,2,4,5,6,7,8,10,6,7,8,12,6,13]

In Li et al. (2009b), the authors experimentally compared the fault detection capability of four coverage criteria: PPC, mutation, all-uses, and edge-pair. The results of experiments on 29 java classes showed that the failure detection capability of edge-pair and PPC criteria are almost similar, while PPC subsumes the edge-pair coverage criterion. In this paper, we perform mutation analysis to compare our PPC-based approach with EvoSuite when this tool uses the BC criterion.

To satisfy PPC, the fitness function of our search-based test data generation approach should support the capability of covering paths that contain loops (Ammann and Offutt, 2016). As far as we know, two fitness functions have been introduced in order to consider program loops (Bueno and Jino, 2002; Lin and Yeh, 2001). These fitness functions, that are capable to support PPC, are referred to as NEHD (Normalized Extended Hamming Distance) (Lin and Yeh, 2001) and BP (Branch Predicate) (Bueno and Jino, 2002).

However, the approaches in Bueno and Jino (2002); Lin and Yeh (2001) are based on the genetic algorithm (GA). Since ACO has shown noteworthy results in other optimization problems (Elbeltagi et al., 2005), we use ACO as the basis of our PPC-based test data generation approach. ACO is mainly used in optimization problems with the graph-based structure (details are represented in Section 2.1). But, it is not straightforward to use this algorithm for the test data generation problem because the search space in this problem is n-dimensional (Floreano and Mattiussi, 2008) (Suppose a program under test P with n input variables represented by vector $X = (x_1,\ x_2,\ \dots,\ x_n)$. Assuming that the domain of each input variable $x_i (1 \le i \le n)$ is D_i, the input domain (i.e., search space) of P is $D = D_1 \times\ D_2 \times \cdots \times D_n$. Therefore, in the structural test data generation problem, the search space is the cartesian product of the domain of each input variable); hence, ACO must be customized to apply in the structural test data generation.

There are two approaches that have customized ACO for test data generation (Mao et al., 2015; Sharifipour et al., 2018). However, both approaches consider BC as their coverage criterion. The approach in Sharifipour et al. (2018) defines pheromone values in each branch. The underlying idea is discouraging every ant from traversing branches already covered by other ants. Therefore, this approach is not appropriate for covering loops, and thus, we consider the approach in Mao et al. (2015) as the basis of our work. However, to make this approach usable for PPC, we first use a fitness function that concerns loops.

Next, we improve the performance of ACO in test data generation via three contributions. First, we use the notion of input space partitioning presented by Ammann and Offutt (2016) to maintain pheromone values in the search space. This is done to strengthen the exploration of ants in the search space. Maintaining pheromone values in the search space makes the proposed ACO algorithm applicable for all structural coverage criteria, including the PPC criterion. It must be mentioned that compared to the basic ACO that maintains pheromone values in each edge of the graph-based search space, and unlike the previous methods that define pheromones in ants or branches of the program, we maintain pheromone values in (partitions of) the n-dimensional search space. This approach conforms to the basic functionality of pheromones, which is tracing of good solutions ever found in the search space.

Partitioning the search space gives us the opportunity to use the idea of adaptive random testing (ART). ART improves random testing by considering the fact that inputs causing failure tend to form contiguous regions (Anand et al., 2013; Sabor and Thiel, 2015). As the second contribution, we suggest a new method for local search based on ART to enhance the exploitation of ants.

In ART, partitioning is a method to simulate the distribution of test data. Partitioning provides the best performance when the resulting partitions are homogeneous (Chen et al., 2015). A partition is homogeneous if either all of its elements yield a failure or none of these elements cause failure. There is no partitioning strategy in practice, which guarantees this behavior. In this paper, as the third contribution, we use the program predicates' information for partitioning the search space in order to approximate the ideal homogeneous behavior. This is done to make a relation between the logic of the program and the pheromone values in the search space.

The experimental results indicate that our proposed changes in ACO yield better performance in test data generation, especially for programs with complex predicates. In addition, the results of mutation analysis demonstrate that test data generated by our approach (which regards the PPC criterion), on average, has more ability to detect failures compared to test data generated by EvoSuite (when this tool considers the BC criterion).

The organization of the paper is as follows. In the next section, we describe background concepts: the basic ACO, and ART. Related work is reviewed in Section 3. Section 4 introduces the proposed approach. Then, experimental results are presented in Section 5. Section 6 discusses the threats to the validity of our work, which is followed by the conclusion and the outline of future works in Section 7.

2. Background

2.1. Basic ant colony optimization

ACO is one of the swarm intelligence algorithms (Blum and Li, 2008), with well-known application in various optimization problems. Like many other meta-heuristic algorithms, the main idea of ACO is inspired by observing the natural behavior of living organisms. In ACO, different behaviors of ants in their community have been the source of inspiration.

ACO was originally conceived to find the shortest route in the traveling salesman problem with graph-based structure. In ACO, several ants travel across the edges that connect the nodes of a graph while depositing virtual pheromones. Ants that travel the shortest path will be able to make more return trips and deposit more pheromones in a given amount of time. Consequently, that path will attract more ants in a positive feedback loop.

Several types of ACO algorithms have been developed with variations to address the specificities of the problems to be solved. Most ACO algorithms consist of three procedures: local search, global search and update pheromone.

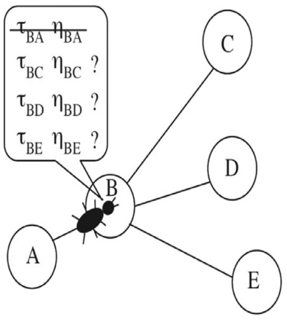

~~τ_{BA} η_{BA}~~
τ_{BC} η_{BC} ?
τ_{BD} η_{BD} ?
τ_{BE} η_{BE} ?
C
B
D
A
E

Fig. 3. Choosing an edge with a probabilistic rule.

Local search: The aim of local search is selecting the next destination among the neighboring nodes of each ant. Each ant chooses an edge from its location with a probabilistic rule that takes into account the length of the edge and the value of pheromones on that edge. As shown in Fig. 3, a virtual ant arriving from node *A* considers which edge to be chosen next on the basis of pheromone level τ_{ij} and visibility η_{ij} (inverse of distance). The edge to node *A* is not considered because that node has already been visited.

Global search: In the local search, there might be an ant that could not find a neighboring position with better fitness value. This situation with no movement is known as the local optima trap. The aim of global search is probing the whole search space in order to resolve the local optima trap situation. Global search refers to searching among all possible solutions whereas local search probes the neighboring set of candidate solutions.

Update pheromone: Once all ants have completed a full tour of the graph, each of them retraces its own route while depositing on the traveled edges a value of pheromones inversely proportional to the length of the route. Before restarting the ants from random locations for another search, the pheromones on all edges evaporate by a small quantity. The pheromone evaporation, combined with the probable choice of the edge, ensures that ants eventually converge on one of the shortest paths.

2.2. Adaptive random testing

Adaptive random testing is an enhancement to random testing inspired by the empirical observation that failure-causing inputs tend to form contiguous failure regions, hence non-failure-causing inputs should also form contiguous non-failure regions (Anand et al., 2013; Sabor and Thiel, 2015). Therefore, if previously executed test data have not revealed a failure, new test data should be far away from the already executed test data.

The failure regions, as described by Anand et al. (2013), include the block pattern, the strip pattern, and the point pattern. Fig. 4 shows these typical failure patterns in a two-dimensional search space. The block pattern describes the most common case, where failures are clustered in one or a few contiguous areas. For example, in a function that takes as input two integers *a* and *b*, if there is a fault in the body of a predicate such as $if(a \geq 0 \;\&\&\; b \geq 0 \;\&\&\; a \geq b)$, then we could have a contiguous region of failing test data in the positive sub-area of $a \geq b$. Strip patterns involve a contiguous area but stretched greatly in one dimension while considerably narrow in the other. Finally, the point pattern failures are spread throughout the input domain in very small groups.

3. Related work

In Jones et al. (1996) and Pargas et al. (1999), GA was applied to generate test data automatically. The experiments revealed that GA considerably shows better performance than the random

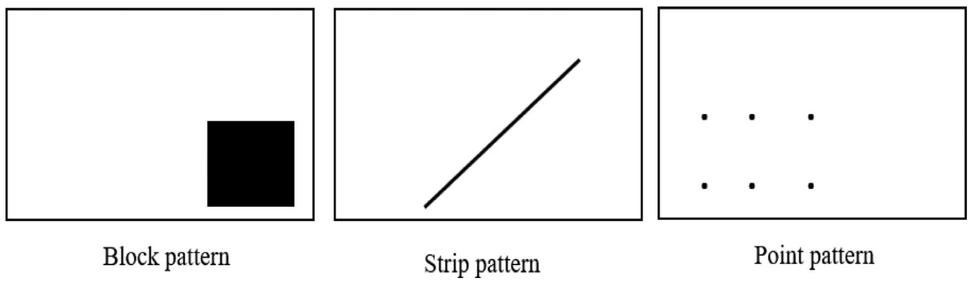


Fig. 4. Block, Strip, and Point failure patterns in a two-dimensional input domain.

method regarding the BC criterion. The authors in Harman and McMinn (2010) empirically confirmed the effectiveness of the hybrid global-local search for testing large-scale programs. In their approach, hill climbing was used for the local search, while GA was applied for the global search. A tool named EvoSuite was introduced in Fraser and Arcuri (2013), which uses evolutionary algorithms like GA in order to generate test suites for Java classes. Several test criteria, such as line coverage, branch coverage, top-level method coverage, weak mutation, and exception coverage are supported by EvoSuite.

Cohen et al. applied simulated annealing (SA) for test data generation (Cohen et al., 2003). SA leverages the idea of neighborhood search to solve complex optimization problems. In Windisch et al. (2007), particle swarm optimization (PSO) was used in order to produce test data. The experimental results in Windisch et al. (2007) show that PSO outperforms GA in most cases. In Mao (2014a), Mao has built a method based on PSO and has demonstrated that the proposed method outperforms both SA and GA.

In comparison to various meta-heuristic algorithms, ACO has revealed promising results when applied on optimization problems (Dorigo et al., 2006; Elbeltagi et al., 2005; Simons and Smith, 2012). ACO was adopted in Li and Lam (2005) and Srivastava and Baby (2010) to produce test sequences (not test data) for state-based software testing. Li et al. (2009a) used ACO to generate test data regarding the branch coverage criterion. Because the basic ACO can only be applied to graph-based problems, while the test data generation problem has an n-dimensional search space, the work in Li et al. (2009a) transforms the program input domain to a graph structure. Since the authors in Li et al. (2009a) have not provided any implementation and evaluation of their proposed method, the performance of this method cannot be judged.

ACO was also used by Mao et al. (2015) in order to generate test data with respect to the BC criterion. The findings in Mao et al. (2015) show that ACO is better than GA and SA while it is comparable with PSO in terms of the average coverage of branches. In this study, pheromones were dedicated to each ant. Authors in Sharifipour et al. (2018) improved the work of Mao et al. (2015) by incorporating (1+1)-evolution strategies to enhance search exploitation through improving the movement of ants in the local search. In Sharifipour et al. (2018), pheromone values were defined in each branch, and also, were considered as a part of fitness function to discourage every ant from traversing branches already covered by other ants. This viewpoint is only appropriate for branch coverage, and cannot be applied for the prime path coverage criterion. This is because when we consider loops, we cannot discourage ants from revisiting branches that have previously been covered.

Lin and Yeh (2001), and Bueno and Jino (2002) introduced GA-based approaches that consider loops in path testing; therefore, these approaches are appropriate for covering prime paths. These works are based on GA. In Bidgoli et al. (2017), a basic method for using ACO to cover prime paths was presented, and the preliminary results were reported. This paper addresses the following extensions in comparison to Bidgoli et al. (2017):

- Applying the idea of adaptive random testing in local search.
- Using the information of program predicates in partitioning the search space.
- Using a better structure for maintaining pheromones.

In addition, mutation analysis is carried out to experimentally compare test data generated by the proposed approach based on the PPC criterion against test data produced by EvoSuite based on the BC criterion.

4. ACO-based test data generation

This study focuses on unit testing. Suppose a unit under test F with n input variables represented by vector $X = (x_1,\ x_2,\ \ldots,\ x_n)$. As mentioned before, if the domain of each input variable $x_i\,(1 \le i \le n)$ be D_i, the input domain of F is $D = D_1 \times\ D_2 \times \cdots \times D_n$ (i.e., this search space has n-dimensional structure). In our approach, the position of an ant in the search space is a representation of an input value of F in the form of a vector in D; hence, the position of ant $k(1 \le k \le m)$ is represented as $X_k = (x_{1k},\ x_{2k},\ \ldots,\ x_{nk})$.

In order to cover prime paths of the unit under test F, we repeat the following steps, respectively, until all the prime paths are covered or a predetermined timeout is reached:

1. A population of ants is randomly distributed in the search space.
2. F is executed by considering the position of each ant in the search space as an input value; according to the traversed paths, the fitness function is computed for each input value.
3. The procedures “local search”, “global search”, and “updating pheromone values” are executed in turn, and then, the next iteration of the algorithm is performed.

4.1. The proposed ACO

The basic ACO was introduced for problems with search space that could be modeled as a graph; pheromone values are maintained on the edges of such graph. However, in the n-dimensional search space of the test data generation problem, there is not any notion like “edge” for maintaining pheromone values. To tackle this problem, in the work of Mao et al. (2015), the pheromone values were defined for each ant. But this definition does not conform to the basic functionality of pheromones released in the search space, which are traces of the good solutions ever found. In addition, the definition in Mao et al. (2015) strengthens exploitation rather than exploration. So, ants with higher pheromone values will get a higher probability to absorb other ants; see Sharifipour et al. (2018).

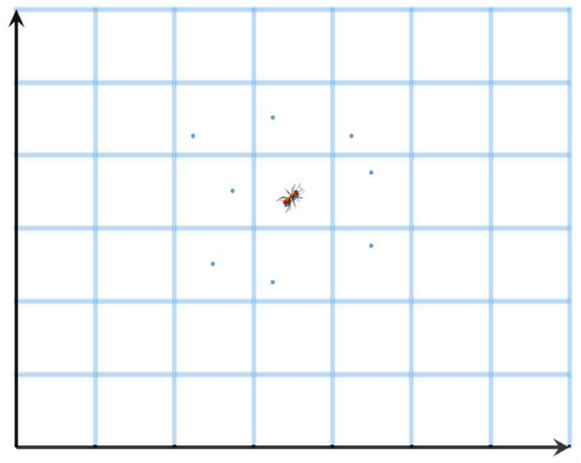


Fig. 5. Local search based on adaptive random testing in a 2-dimensional search space.

4.1.1. The modifications applied to the basic ACO

The main modification is related to the maintenance of pheromones in the search space. To solve this problem, we divide the domain of each input variable into parts in order to maintain pheromone values in the resulting parts. For example, if the unit under test has two input variables, we divide the input domain of these variables into b_1 and b_2 parts, resulting in totally $\varphi = b_1 \times b_2$ partitions in the search space (Throughout the paper, the word "part" is used to refer to the sections in a dimension, while the word "partition" refers to the sections obtained in the d-dimensional space after performing division in all dimensions). The division in each part can be done based on either an arbitrary number or more effectively, the logic of the program under test. In Section 4.1.5, we propose an approach that uses predicates' information of the program in partitioning.

Since each partition should have its own pheromone value, in both the pheromone update step and the global search step, when we try to find a partition with the most pheromone value, all partitions must be investigated. In a d-dimensional search space, if b_i be the number of parts in domain i, there are $b_1 \times b_2 \times \cdots \times b_d$ different partitions to investigate. To improve efficiency, we can maintain the pheromone values in each part instead of each partition. In this way, the pheromone value in a partition is a vector of pheromone values in its corresponding parts. With the new structure, to find a partition with the most pheromone value, we search for the best part in each dimension, and, instead of investigating $b_1 \times b_2 \times \cdots \times b_d$ pheromone values, $b_1 + b_2 + \cdots + b_d$ different pheromone values must be examined. Thus, updating pheromone values and finding a partition with the most pheromone value in the global search are done more efficiently.

4.1.2. Local search based on adaptive random testing

As mentioned before, the idea behind adaptive random testing is inputs causing a failure in a program are usually in continuous failure areas and vice versa. So, in adaptive random testing, when a random value is selected, its neighboring area is discarded from the search space anymore. Thus, the generated random values are uniformly distributed in the whole search space (Anand et al., 2013).

If d be the number of input variables of the unit under test (i.e., the number of dimensions in the search space), the number of neighboring partitions will be $3^d - 1$; see Fig. 5 for a 2-dimensional search space. Since, in a search space with many dimensions, this number might have a negative effect on the performance of the ACO algorithm, we only select some of the neighboring partitions to be investigated to find a better place to be. In our approach, to transfer locally based on the idea of ART, we randomly select l (out of $3^d - 1$) neighboring partitions. Finally, a random location is chosen in each selected partition. Finally, all of these random locations will be investigated to find a better place to transfer.

In our local search procedure, ant k is transferred to a new position $X_{k'}$ (of one of its neighbors), if the position of k' has a better fitness value, and also, this fitness value is the best among the neighbors of X_k. Otherwise, the ant k does not have any movement in the local search.

4.1.3. Global search

As usual, we consider the global search to perform exploration as well as mitigating the problem of local optima. Suppose $Fitness_{avg}$ is the average of all ant's fitness values in the colony with size m, and q_0 is a preset parameter between 0 and 1. The global transfer is done for any ant k, provided that at least one of the following constraints is satisfied:

1. Ant k does not have any movement in the local search.
2. $Fitness(X_k) > Fitness_{avg}$ (i.e., $Fitness(X_k)$ is worse than $Fitness_{avg}$); the fitness function is introduced in Section 4.2.

To do the global transfer, a random number $0 < q < 1$ is chosen. Ant k is moved to a random position in the whole search space, if $q < q_0$; otherwise, a random position in a partition with the highest pheromone value is selected as the new location for ant k.

4.1.4. Update pheromone

Updating pheromone values for each part is done based on formula (1):

$$\tau(j) \leftarrow \ (1 - F_j^*) \times count(j) + \tau(j) \times (1 - \alpha) \tag{1}$$

where j stands for the part index, $\tau(j)$ and $count(j)$ represent the pheromone value and the number of ants in the j_{th} part, respectively, F_j^* is the best fitness already achieved in the j_{th} part, and $(0 < \alpha < 1)$ is the pheromone evaporation rate.

4.1.5. Using the predicates' information in partitioning

As explained in Anand et al. (2013), partitioning as a method to simulate the distribution of test data works in the best way if all partitions are homogeneous. A partition is homogeneous if either all of its elements yield a failure or none of these elements cause failure. There is no partitioning strategy in practice which guarantees this behavior. Therefore, in this section, a strategy which approximates this ideal is proposed by defining the notion of *near homogeneous* partition. A near homogeneous partition is a partition which either all of its values cause a program clause evaluated to *True* or all of its values cause a program clause evaluated to *False*. By clause, we mean predicates with no logical connective. To produce *near homogeneous* partitions, a partitioning strategy based on the program predicates' information is proposed.

The process of using predicate's information in partitioning is shown in Fig. 6. Since the search space in the test data generation problem is equivalent to the cartesian product of the domains of *input variables*, we must re-express the program predicates to involve just input variables. To achieve this goal, we first execute the program symbolically. Having predicates constructed by only input variables, we modify each clause C by using the $(=)$ operator instead of the $(\leq , \geq , \neq , < , >)$ operators, and then, solve the resulting clause, called C'.

Suppose C' contains n_c input variables $(x_1, x_2, \ldots, x_{n_c})$ and let $(x_1 = a_1, x_2 = a_2, \ldots, x_{n_c} = a_{n_c})$ be a combination of input values that satisfy C'. Each of the hyper plains $x_i = a_i$ $\{1 \leq i \leq n_c\}$ divides the n_c-dimensional search space into 2 parts. From the intersection of all such parts, 2^{n_c} partitions are made in the search space. There are two near-homogeneous partitions in the search space based on

```
Execute the program symbolically to rewrite the conditional statements in terms of just input variables
For each clause C of the given program,
    Generate clause C' for clause C by the replacement of its relational operator with the equality operator
    Find values for input variables such that C' is satisfied
End For
Use all the values found per input variable to partition its corresponding domain
```

Fig. 6. Using the predicates' information in partitioning.

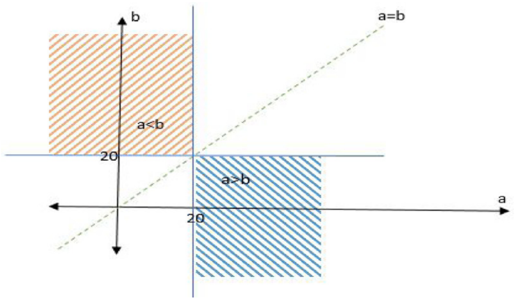


Fig. 7. Partitioning for the clause $a < b$.

Table 2
The branch distance for all kinds of branch predicates (Tracey et al., 1998).

No	Predicate type	Branch distance
1	Boolean	if *true* then 0 else δ
2	$\neg a$	Negation is propagated over a
3	$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + \delta$
4	$a \neq b$	if $abs(a-b) \neq 0$ then 0 else δ
5	$a < b$	if $(a-b) < 0$ then 0 else $abs(a-b) + \delta$
6	$a \leq b$	if $(a-b) \leq 0$ then 0 else $abs(a-b) + \delta$
7	$a > b$	if $(a-b) > 0$ then 0 else $abs(a-b) + \delta$
8	$a \geq b$	if $(a-b) \geq 0$ then 0 else $abs(a-b) + \delta$
9	a and b	$f(a) + f(b)$
10	a or b	$Min(f(a), f(b))$

this type of partitioning. It is worth to mention that although too many combinations of input values could satisfy C', only one of these combinations suffices for our purpose.

To illustrate this approach, we use some examples. Considering the clause $x > 200$, the only point for partitioning is $x = 200$, which divides the domain of a, into two parts; values in one of these parts make $x > 200$ as *True*, while using values in the other part make $x > 200$ as *False*. Regarding clause $x < y$, with $x = 20$ and $y = 20$ as partition points, there are 2×2 partitions in the whole search space as a result of dividing the input domain of x and y into two parts. As shown in Fig. 7, all values in the red partition traverse the *True* case and all values in the blue partition traverse the *False* case of clause $x < y$. As another example, considering clause $x + y > z$, values that satisfy $x + y = z$ (e.g., $x = 50, y = 50, z = 100$) are appropriate to be considered as partition values.

After symbolic execution of the program, due to loop-dependent and array-dependent variables, pointer references and calls to the external libraries whose implementations are not available, there may be clauses with non-input variables. No knowledge can be obtained from such clauses, and thus, these clauses cannot be used in this step. If no knowledge (i.e., partition value) can be extracted for an input variable a, we use our first approach to divide the domain of a into equal parts.

4.2. Fitness function

In the previous studies on path testing, every test path is considered separately as a target (There are some works regarding multiple path coverage but not all test paths simultaneously; for more details, see Fraser and Arcuri (2013)). Based on rigorous experiments done in Fraser and Arcuri (2013), considering just one target at a time has some drawbacks. For example, in the case of targeting an infeasible test path individually, any resources invested are wasted. An infeasible path is a path that cannot be covered by any test data. In addition, we may wrongly decide that a test requirement (e.g., a prime path) is infeasible, while it is in fact feasible, but we have tried to cover it by traversing a specific infeasible test path. Furthermore, many test paths might be inadvertently traversed by individuals (e.g., ants in ACO) while the target path is another one. As a solution, we should simultaneously consider all the test requirements as the target. Correspondingly, we must define a new fitness function for this purpose.

The fitness function for evaluating each individual is presented in formula (2). It consists of two distinct parts. One part is related to the branch predicate distance notion which measures the closeness to cover the appropriate branch. The other is related to the approximation level, which indicates the similarity of the traversed path with the targeted prime paths. Each part results in a normalized value in range [0,1]. Lower values mean better fitness.

$$FT = \left(1 - \frac{NC}{APP}\right) + \frac{SEP}{SEP + \beta} \quad (2)$$

- *NC* is equal to the number of targeted prime paths existing in the traversed path.
- APP is equal to the number of all the prime paths to be covered.
- *SEP* is equal to the sum of all the branch distances for the traversed path. The branch distance for a given branch predicate is the deviation from its trueness (with respect to the opposite branch) when the input values are assigned to the variables. The branch distance is calculated based on Table 2. For example, consider the CFG presented in Fig. 2. If an ant traverses the path [1,2,3,5,6,13], SEP will be equal to the sum of the branch distances of edges 2–4 and 6–7.
- β is a parameter for normalization. For example, if $EP = 10$ and $\beta = 1$, the calculated value for the part related to the branch predicate (i.e., $\frac{EP}{EP+\beta}$) is $\frac{10}{11}$ or 0.9, a normalized value between 0 and 1. In experiments, we set the value of β to 1, based on the results presented in Arcuri (2013).

4.3. The proposed algorithm

Algorithm 1 presents the pseudo code of the proposed ACO algorithm. The list of parameters and notations used in this algorithm has been given in Table 3. The result of the algorithm is a test suite (i.e., a set of test data) covering the given prime paths. Algorithm 1 simultaneously takes all the prime paths as the target, and repeats the test data generation process until all the prime

Table 3
The parameters and notations used in the proposed ACO algorithm.

Parameter/Notation	Description
$maxTime$	Timeout (maximum time)
$time$	The elapsed time set in a separate thread
n	The number of input variables
m	The number of ants
$ant[k]$	The ant k
$ant[k].position$	The position of ant k that is a vector in the input domain (i.e., a test data)
$ant[k].fitness$	The fitness of ant k
$ant[k].part[i]$	The part corresponding to the ith input variable of ant k
$\tau[i][j]$	The pheromone corresponding to the jth part of the ith input variable
$count[i][j]$	The number of ants in the jth part of the ith input variable
l	The number of neighboring partitions that must be investigated in the local search
α	Pheromone evaporation rate
q_0	Control parameter used in the global search
$b[i]$	The number of parts for the ith input variable
TS	The output test suite

Algorithm 1 The proposed ACO algorithm.

```
1: Inputs:
     1. SUT.
     2. ACO input parameters according to Table 3.
2: Output: Test suite $TS$.
3: $Flag = false$;
4: Stage1: Initialization
5: for $i = 1 : n$ do
6:     for $j = 1 : b[i]$ do
7:         $\tau[i][j] = 1$;
8:     end for
9: end for
10: for $k = 1 : m$ do
11:     randomly initialize $ant[k]$;
12: end for
13: Stage2: Optimum Solution Searching
14: while $time \le maxTime$ and $Flag == false$ do
15:     for $k = 1 : m$ do
16:         LocalSearch($k$);
17:     end for
18:     calculate the average fitness $f_{avg}$ of the ant colony;
19:     for $k = 1 : m$ do
20:         if $ant[k].fitness > f_{avg}$ then
21:             GlobalSearch($k$);
22:         end if
23:     end for
24:     UpdatePheromone();
25:     for $k = 1 : m$ do
26:         if $ant[k]$ covers an uncovered prime path then
27:             Add $ant[k].position$ to $TS$;
28:         end if
29:         if all prime paths are covered then
30:             $Flag = true$;
31:         end if
32:     end for
33: end while
34: return $TS$;
```

paths are covered or the predetermined timeout is elapsed. The data generation process consists of two stages. In the initialization stage, all pheromone values are first initialized by one (Lines 5–9). Then, a position vector in the search space is randomly assigned to each ant (Lines 10–12).

In the stage of searching (Lines 14–33), local search using ART is conducted for the current location of each ant. Next, global search is done. After that, the pheromone trail is updated according to Eq. (1). At the end of each iteration, the fitness values of the new population are calculated, and if there is any ant k that covers at least an uncovered prime path, its corresponding position is added to TS.

```
1: Procedure1 : LocalSearch($k : int$)
2: local-optima-trap=$false$;
3: randomly select $l$ neighboring partitions
4: for each chosen neighboring partition do
5:     select a random position in the partition;
6:     maintain the best ant with the lowest fitness value in the
       temporary variable $bestAnt$;
7: end for
8: if $bestAnt.fitness > ant[k].fitness$ then
9:     local-optima-trap=$true$;
10: else
11:     $ant[k]$ moves to the new position, i.e., $bestAnt.position$;
12:     for $i = 1 : n$ do
13:         $count[i][ant[k].part[i]] + +$;
14:     end for
15: end if
16: if local-optima-trap=$true$ then
17:     GlobalSearch(k);
18: end if
```

```
1: Procedure2 : GlobalSearch($k : int$)
2: if $U(0, 1) < q_0$ then
3:     randomly select a position $p$ in $D$;
4: else
5:     randomly select a position $p$ in a partition with the best
       pheromone value;
6: end if
7: $ant[k]$ moves to the new position, i.e., $p$;
8: for $i = 1 : n$ do
9:     $count[i][ant[k].part[i]] + +$;
10: end for
```

```
1: Procedure3 : UpdatePheromone()
2: for $i = 1 : n$ do
3:     for $j = 1 : b[i]$ do
4:         $\tau[i][j] = \tau[i][j] \times (1 - \alpha) + count[i][j] \times (1 - F_j^*)$;
5:         $count[i][j] = 0$;
6:     end for
7: end for
```

Procedure 1 describes the local search process in pseudo code. In case a neighbor solution with a better fitness is found, the ant moves to the new position and the number of ants in the corresponding parts are increased by one (Lines 12–14); otherwise, it stays in its current position.

Procedure 2 is the pseudo code of the global search process. A random move (Line 7), either in the whole search space or in the partition with the best pheromone value (subject to the probability q_0; Lines 2–6), is applied to any ant whose fitness value is greater than the average value (Algorithm 1, Lines 20–22) or for any ant with the local optima trap situation (Procedure 1, Lines 16–18).

Procedure 3 details the process of updating pheromone values. After updating pheromone value in each part, the number of ants in that part is re-assigned with 0 ($count[i][j] = 0$).

5. Evaluation

Our evaluation aims to answer the following research questions:

- **RQ1**: Does using the idea of adaptive random testing improve the performance of ACO in test data generation?
- **RQ2**: What is the effect of using the program predicates' information in partitioning the search space to trail the pheromone values?
- **RQ3**: Though PPC subsumes BC, does our approach, which is based on PPC, have a better failure detection capability in comparison to the approaches relying on BC?

To answer these questions, we perform two categories of experiments:

1. To answer RQ1 and RQ2, we ran five different algorithms, GA, PSO, TDG-ACO, ACO-AR and ACO-AR-SP (AR stands for Adaptive Random and SP stands for Static Partitioning). We refer to the ACO algorithm presented in Mao et al. (2015) by "TDG-ACO" (that is the name selected by the authors in the respective paper). In ACO-AR, local search is performed based on ART but partitioning is still conducted by dividing each input domain to a number of equal parts. In addition to performing the local search based on ART, ACO-AR-SP partitions the search space based on the program predicates' information. GA, PSO, and TDG-ACO were implemented the same as what presented in Bueno and Jino (2002), Mao (2014a), and Mao et al. (2015), respectively.
2. To answer RQ3, we performed mutation analysis to experimentally investigate the failure detection capability of test suites generated by the proposed approach against test suites produced by Lin and Yeh (2001) based on the BC criterion. EvoSuite is one of the main reference tools for test data generation. It achieved the overall highest score of all competing tools at the SBST 2017 unit testing tool competition (Fraser et al., 2017). Mutation analysis is a fault-based testing technique which provides a testing metric called the mutation score. The mutation score can be used to measure the effectiveness of a test suite in terms of its ability to detect failures. The idea of using mutation analysis to measure the test suite adequacy was originally proposed in Hamlet (1977), and explored extensively by Offutt and Untch (2001). The general principle underlying mutation analysis is that the faults used by mutation analysis represent the mistakes that programmers often make. Such faults are deliberately seeded into the original program, by simple syntactic changes, to create a set of faulty programs called mutants. To assess the quality of a given test suite, these mutants are executed against the test suite. If the result of running a mutant is different from the result of running the original program, the seeded fault in the mutant is detected. Such mutants are called killed mutants. The mutation score is defined as the ratio of the number of killed mutants on the total number of mutants. To perform mutation analysis, PIT (Coles et al., 2016) is used as a state-of-the-art tool for this purpose.

5.1. Experiment design

Evaluation metrics

To analyze the results of the first and the second category of experiments, average coverage (AC) and mutation score are respectively used as the evaluation metrics. Average coverage denotes the average percentage of covered test paths in repeated runs. It is calculated while all the competitive algorithms run in the same timeout.

In the previous studies (Bidgoli et al., 2017; Mao et al., 2015; Sharifipour et al., 2018), average coverage was calculated while all the competitive algorithms ran in the same number of iterations; this approach causes having two different metrics for performance evaluation: average coverage for effectiveness and average time for efficiency. Now, consider the results in Bidgoli et al. (2017); Mao et al. (2015); Sharifipour et al. (2018) that show their ACO method has higher average coverage, while it takes more time than PSO in test data generation. This raises the question of whether PSO will result in a better average coverage if it has the same time as the ACO's time. Due to this issue, the competitive algorithms should be run in the same time rather than in the same number of iterations. Therefore, in this paper, we run all algorithms in the same timeout and use just average coverage as our cost-effectiveness metric. Our approach causes the comparison to be done fairly. It must be mentioned that the computed time includes the time spent in each benchmark to calculate fitness values, as well.

Benchmark programs

To conduct experiments, several benchmark programs have been selected (see Table 4): the first eleven programs from the Numerical Case Study (NCS) of EvoSuite,[1] Tcas and Totinfo from the Software-artifact Infrastructure Repository (SIR),[2] FastMath and Complex from apache commons math3, and the others from various related works. All of these benchmark programs have numerical input parameters. It must be noted that the existing customizations of ACO for test data generation can only be applied to numerical data types. In TDG-ACO, this limitation is because of using the method of Euclidean distance calculation in the local search (Mao et al., 2015). Making ACO-AR and ACO-AR-SP applicable for non-numerical data types will be considered as our future work. The required changes in the proposed approach in order to support this capability will be discussed in Section 7.

Table 4 displays the number of units, the number of lines of code (LoC), clause complexity (CC), and the description of each benchmark program. CC of a program is defined based on the notion of clause complexity presented in Mao (2014b). The difficulty degree of satisfying a clause depends on its operator. In this regard, the author of Mao (2014b) classified clause operators into four groups: equation, boolean, inequality and non-equation. In Table 5, the complexity degree of all kinds of operators is presented. Clause complexity of a program is the sum of the complexity degree of all clauses in the program. This metric will be used in the discussions of the experimental results.

[1] https://github.com/EvoSuite/evosuite/tree/master/removed/examples/ncs/src/ncs.

[2] http://sir.unl.edu/php/showfiles.php.

Table 4
The selected programs for the experiments.

#P	Program name	LoC	Units	CC	Description
1	Bessj	245	3	7.5	Bessel J_n function
2	BubbleSort	38	8	1.8	Bubble sort algorithm
3	Encoder	86	1	2.4	Encoding the input
4	Expint	109	1	12	Exponential integral function
5	Fisher	157	1	6	Fisher statistical function
6	Gammq	112	4	8.3	Gamma Function
7	Median	40	1	2.4	Calculate the median value of array input
8	Remainder	30	1	7.8	The remainder of the first argument divided by the second argument
9	TT1	43	1	12.9	Find the type of triangle
10	TT2	50	1	8.8	Find the type of triangle
11	Variance	43	1	1.2	Calculate the variance of array input
12	GCD	24	1	1.1	Find greatest common divisor
13	MinMax	41	1	1.8	Find minimum and maximum value in an array
14	BinarySearch	55	1	3.3	Binary search algorithm
15	ComputeTax	164	1	18.0	Compute tax amount
16	PrimeBetween	42	1	5.8	Calculate the prime number between two numbers
17	Synthesis-1	48	1	3.6	Synthesis of while, for and if
18	Synthesis-2	101	3	12.6	Synthesis of while, for and if
19	PrintCalender	187	9	22.8	Print calendar according to the input of year and month
20	Number	265	1	42.7	Calculate the number of days between two dates
21	Tcas	173	9	20.3	Air craft avoid colision system
22	Totinfo	416	6	22.6	Information measure
23	Mcknap	1620	42	225.9	Solve the knapsack problem
24	FastMath	4297	59	496.2	Mathematical operations
25	Complex	1301	57	179	Mathematical operations

Table 5
The reference complexity degree of clause operators (Mao, 2014b).

.No	Operator type	Weight
1	$=$	0.9
2	$\leq, \geq, <, >$	0.6
3	*Boolean*	0.5
4	$\neq$	0.2

The parameters of the algorithms

The values of the parameters in the compared algorithms are presented in Table 6. These values must be initialized before conducting the experiments. Most values were taken from Mao et al. (2015) in which the authors performed a parameter sensitivity analysis to find the most proper parameter settings for GA, PSO, and TDG-ACO. Like *LS-itr* in TDG-ACO, the parameter l in our algorithm was set as 10, because both parameters similarly specify the number of alternative places could be investigated in the local search. The number of population for all algorithms was set as 30.

We performed sensitivity analysis to select an appropriate value for parameter “b” (i.e. the number of parts) in our proposed ACO algorithm. To do this, by taking a specific timeout, we calculated average coverage. We randomly selected TT2, Bessj, Tcas, and PrintCalender as the benchmarks to do sensitivity analysis. Each experiment was repeated 10 times with different random seed to take into account the random behavior of the algorithm. The trends of the results for each benchmark are shown in 8. As can be seen, the average coverage increases with the increasing of the number of parts, but, when we reach a specific value for the number of parts, we have a decrease in the value of average coverage. Considering the results presented in Fig. 8, it seems that $b = 10$ might be an appropriate choice for this parameter. So, we select this value for parameter b in our experiments.

A rigorous experiment was done in Arcuri and Fraser (2013), in order to do parameter tuning for EvoSuite. The results were reported in Arcuri and Fraser (2013) as follows: population size 100, crossover rate 0.8, rank selection with 1.7 bias, 10% of elitism rate and no parent replacement check. Since these parameter settings are the best for EvoSuite, we used the same settings for EvoSuite in our experiment.

Table 6
The parameter settings of the compared algorithms.

Algorithm	Parameter	Value
GA	Crossover probability p_c	0.80
	Mutation probability p_m	0.03
PSO	Inertia weight w	Monotonically reduces from 1 to 0.2
	Acceleration constants c_1 and c_2	$c_1 = c_2 = 2.05$
TDG-ACO	Pre-set slope T	1.0
	Pheromone decay parameter α	0.3
	Maximum radius $r_{\max}$	Monotonically reduces from 20 to 6
	Threshold of global random search control q_0	0.5
	Adjustment coef. φ	0.5
	Threshold of neighborhood transfer control ρ_0	0.3
	Maximum local search iteration *LS-itr*	10
ACO-AR	Number of parts b	10
	Pheromone evaporation rate α	0.3
ACO-AR-SP	Control parameter for applying the global transfer q_0	0.5
	Number of neighboring partitions that must be investigated in the local search l	10

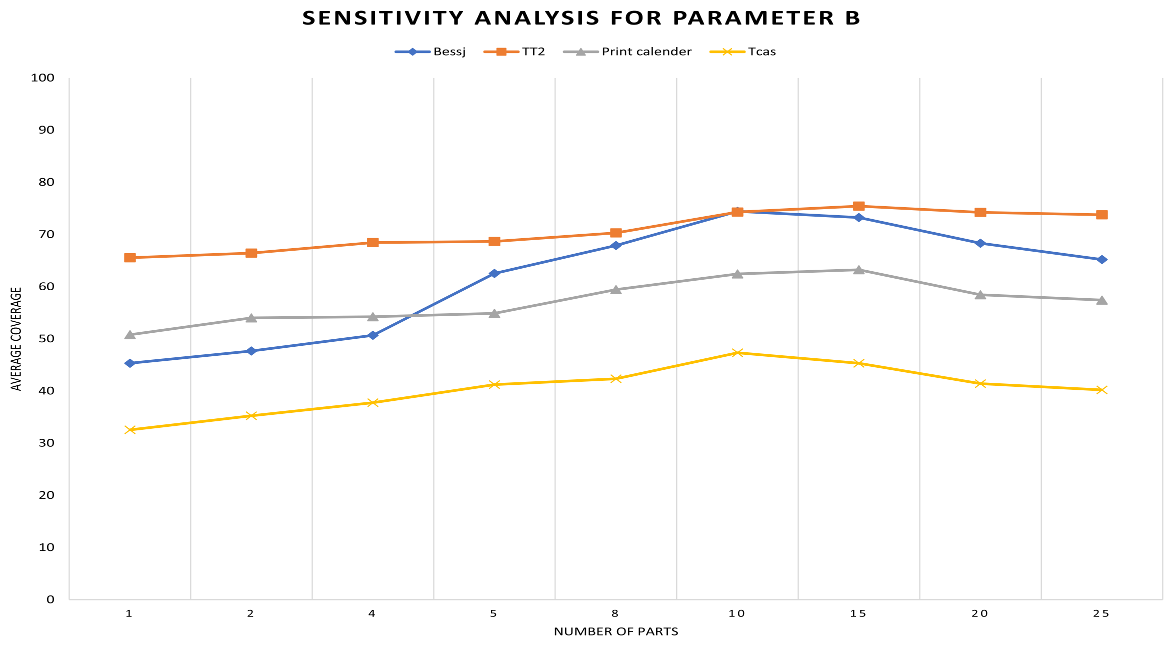


Fig. 8. The sensitivity analysis for parameter b.

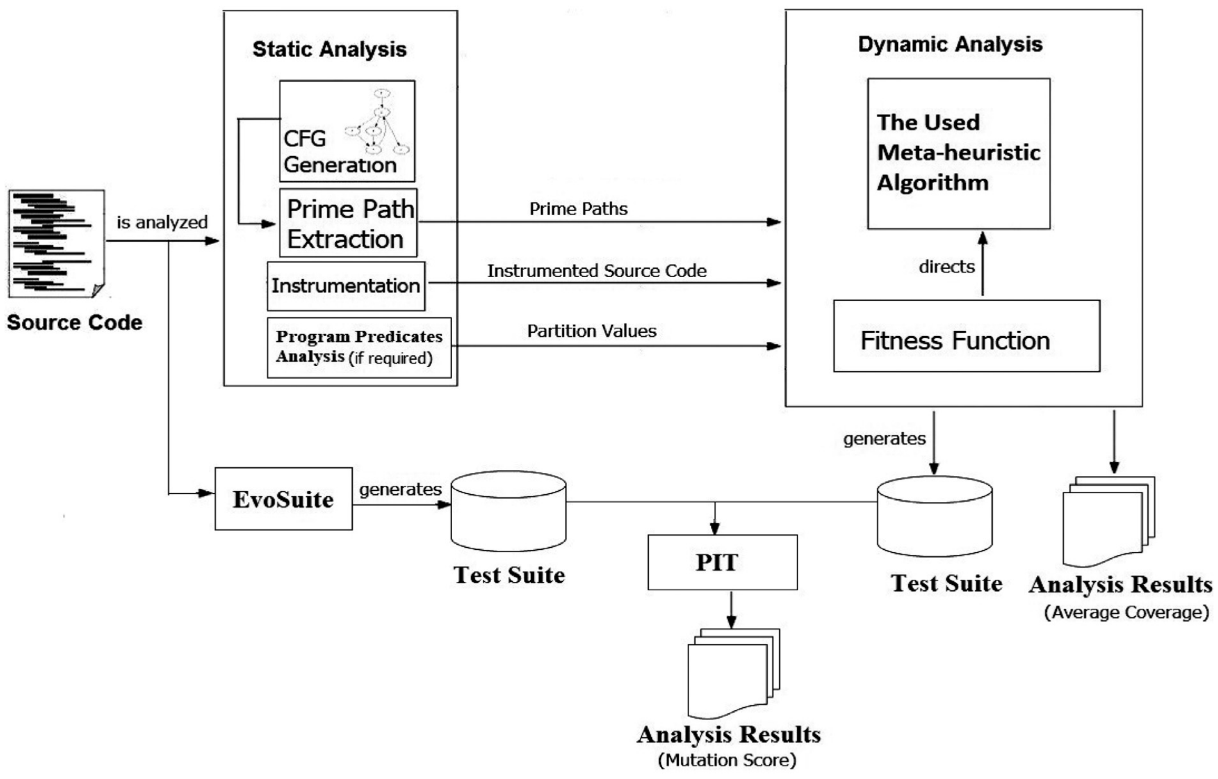


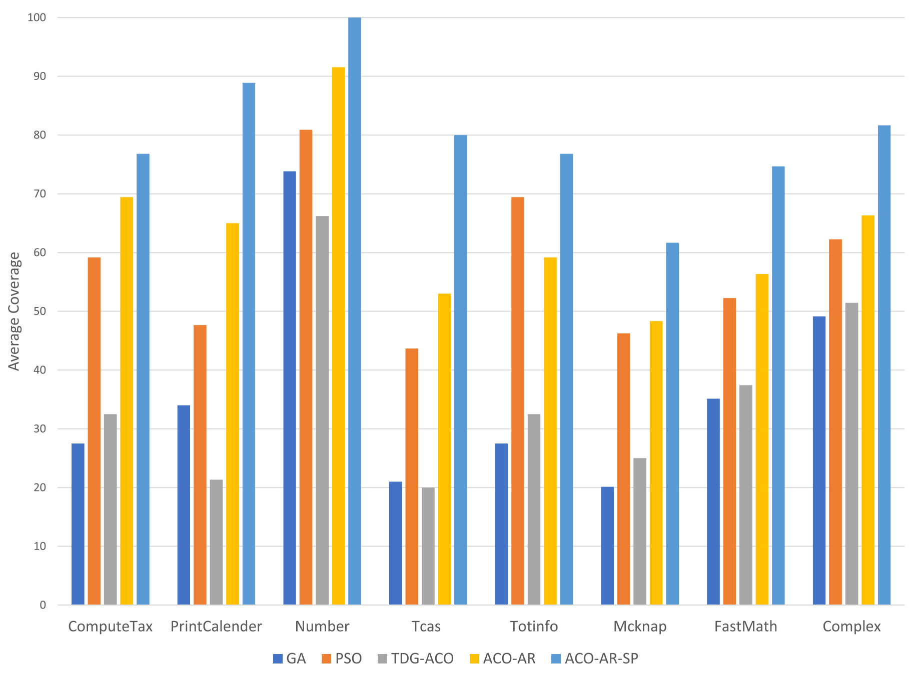
Fig. 9. The overall process of evaluation.

Overal process

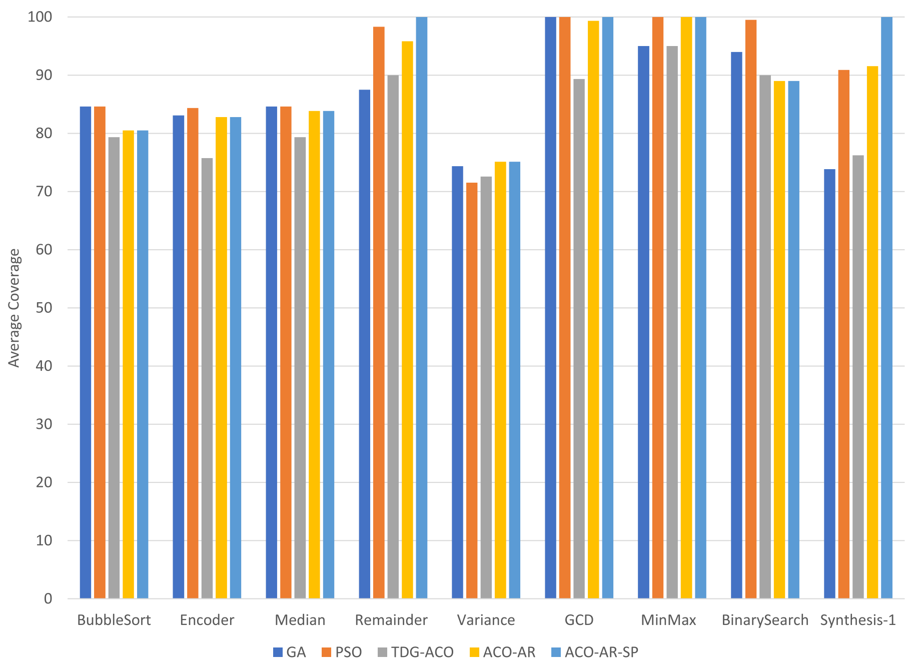
The overall process of evaluation is shown in Fig. 9. In the "Static Analysis" part, prime paths were extracted from the CFG of the given program. In addition, each program was instrumented as explained in Lin and Yeh (2001). For partitioning in ACO-AR-SP, according to what presented in Section 4.1.5, the analysis of the predicates was done statically. In the part of *Program Predicates Analysis* in Fig. 9, Java Pathfinder[3] was utilized to extract clauses of the benchmark programs just in terms of input variables. In addition, we used a python script provided by the authors of Durelli et al. (2018) to extract the prime paths.

The term "The Used Meta-heuristic Algorithm" in the "Dynamic Analysis" part could be replaced by each of GA, PSO, TDG-ACO, ACO-AR and ACO-AR-SP. In this part, the fitness function presented in Section 4.2 guides the used meta-heuristic algorithm. Mutation analysis was done by PIT. Finally, since optimization algorithms have random behavior, based on the guidelines presented in Arcuri and Briand (2014), we used the Wilcoxon test in R (TEAM et al., 2014) to analyze the results.

[3] https://github.com/javapathfinder.



(a)



(b)

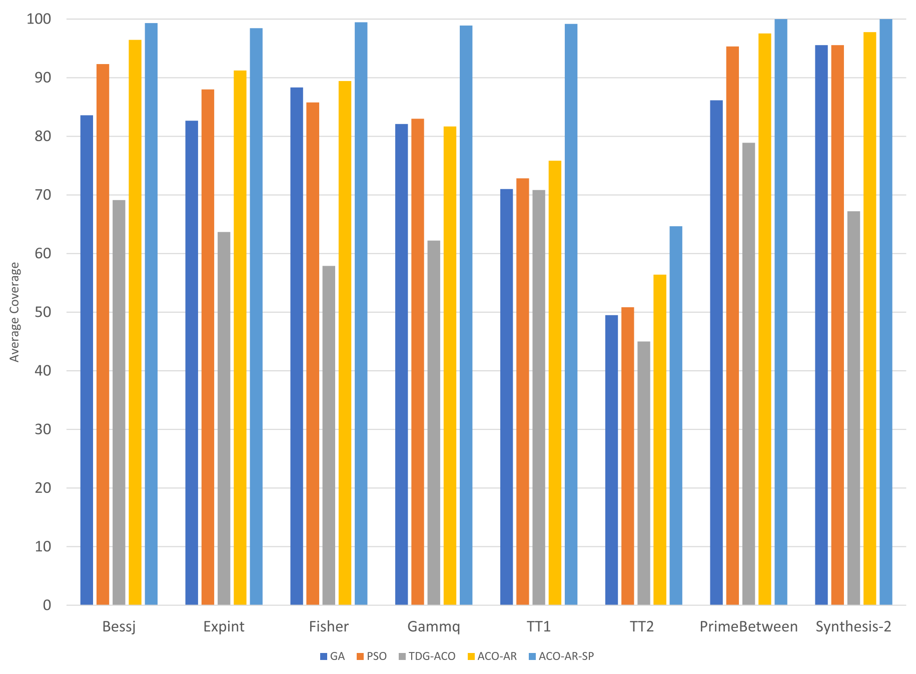
Fig. 10. The average coverage achieved by each algorithm per benchmark.

5.2. Experimental results and discussion

5.2.1. The first category of experiments

In the first category of experiments, to take into account the stochastic nature of meta-heuristic algorithms, experiments were repeated 50 times with different random initial population. For each benchmark, we ran all five competitive algorithms with a specific timeout; we selected 1500 milliseconds (ms) for Mcknap, 150 ms for FastMath, Complex, Tcas, Number and Totinfo, and 20 ms for the other benchmarks. The results in terms of the average percentage of covered prime paths are shown in Fig. 10 per benchmark. To increase the accuracy of our analysis, we did not count infeasible paths when computing the average coverage metric. In experiments, we considered those paths, that had not been covered by any of the five algorithms as infeasible paths.

It is worth mentioning that, if there is no information from static analysis, ACO-AR-SP behaves as ACO-AR; so, for programs with no predicates' information, the results of average coverage for ACO-AR are the same as the results for ACO-AR-SP. This happens for BubbleSort, MinMax, and BinarySearch, Encoder, Median and



(c)

Fig. 10. Continued

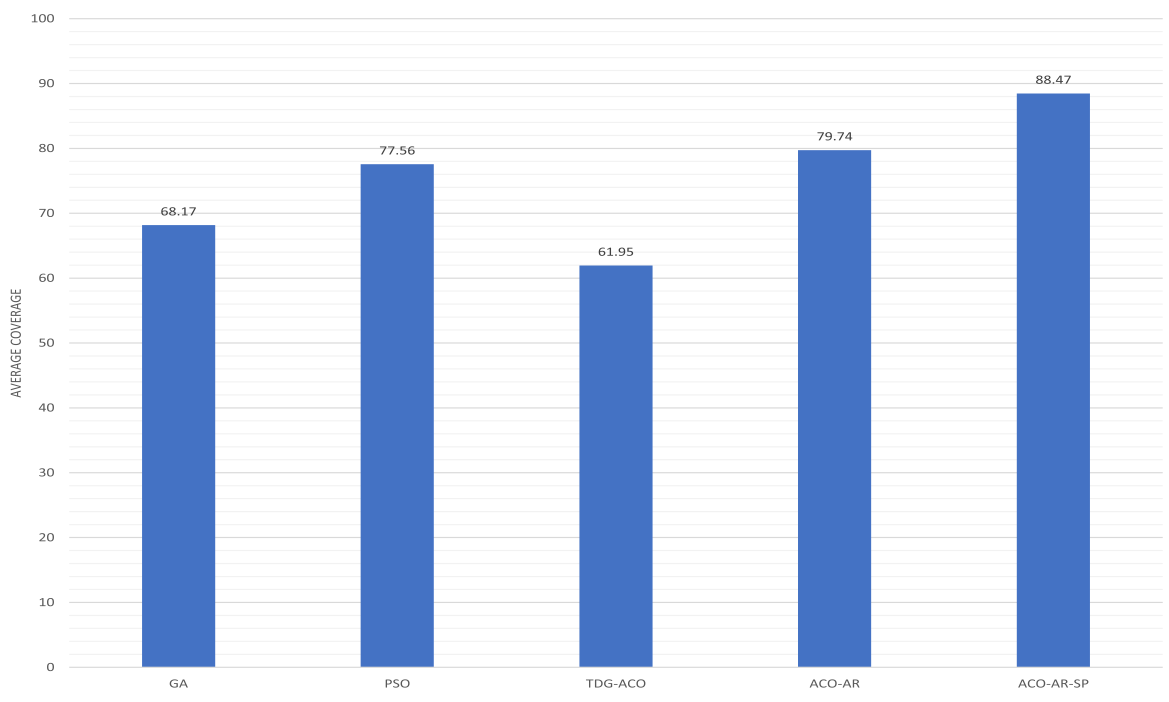


Fig. 11. Overview of the average coverage achieved by each algorithm for all benchmarks.

Variance (see Fig. 10b). In these programs, executing the program symbolically, and thus, representing clauses just by input parameters was not succeeded because of array-index variables that exist in the conditional statements. Static information was found for only one input variable of Mcknap because there are calls to the library functions that their source code is unavailable.

Fig. 11 displays the average coverage achieved by each algorithm for all benchmarks. The results show that, on average, ACO-AR-SP has better average coverage in comparison to the other algorithms. This is due to incorporating the ART idea as well as partitioning the search space based on the predicates' information. This leads to doing the local search in a smarter way. In addition, tracing and maintaining pheromone values are done based on the logic of the program in each partition. Thus, individuals converge to the target with more speed. Partitioning based on the logic of the program leads to having better exploitation in the local search and better exploration in the partition with the highest pheromone value. It is worth noting that covering paths in programs with higher CC is likely more time-consuming and more complicated because less data from the input space are desired. Therefore, it is

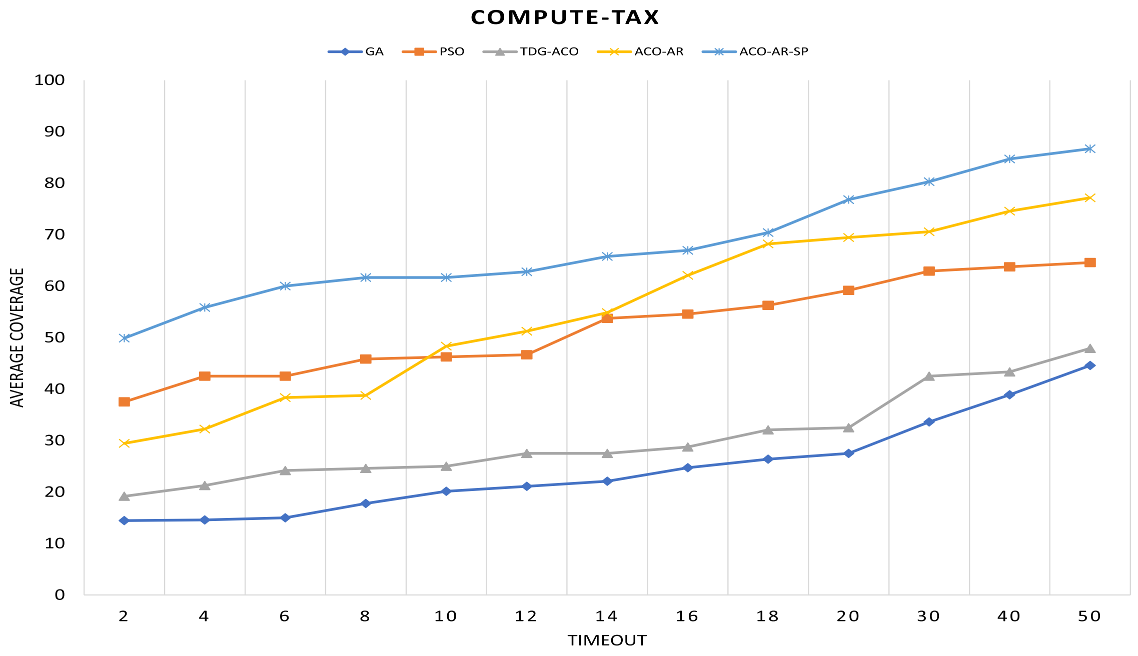


Fig. 12. Average coverage for ComputeTax in different timeouts.

expected that the improvements we applied to the ACO algorithm are usually better shown for programs with higher CC; see Fig. 10a which shows the results for programs with higher CC.

In the second place, ACO-AR and PSO showed better performance than GA and TDG-ACO, while they were comparable with each other (see Fig. 11). The superiority of ACO-AR compared with GA is due to the fact that ACO-AR is a hybrid approach consisting of local and global search, but GA is purely a global search method. In addition, ACO-AR uses the idea of ART in the local search. Using this idea resulted in better performance for ACO-AR compared with TDG-ACO as well. Search space in the test data generation problem is often large and doing the local search based on the idea behind ART leads individuals to have better exploitation. In other words, local search based on this idea causes the local optima trap situation happens less than once the local search is done randomly as presented in Mao et al. (2015). Furthermore, the pheromone definition in Mao et al. (2015) reinforces exploitation instead of exploration in the global search. Partitioning the search space solves this problem and causes ACO-AR to have better performance in comparison to TDG-ACO. The superiority of PSO over TDG-ACO is because of the overhead resulted from using or manipulating the pheromone structure and calculating transition probability in TDG-ACO (see Section 4 of Mao et al., 2015, formula 7).

PSO reaches higher average coverage than ACO-AR for some programs with low CC (for example, see the results for BubbleSort, Median, GCD, and BinarySearch in Fig. 10b). This is due to the overhead resulted from using or manipulating the pheromone structure by the procedures of ACO-AR (For some programs like MinMax and Variance, despite they have low CC, the results of PSO and ACO-AR have been almost similar. This is because it might be possible that ACO-AR reaches the targets by chance in the initialization stage of the algorithm). In contrast, due to the simplicity of PSO, it covers paths with simpler clauses promptly. For example, as shown in Fig. 12 for ComputeTax, PSO has better average coverage at low timeouts, but, as time increases, ACO-AR gains a better performance. This causes the trends of ACO-AR to have a more slope compared with the trends of PSO because in low timeouts the overhead resulted from the pheromone structure is effective, while as time increases, this overhead is ignorable. Consequently, ACO-AR works better than PSO for the prime paths that need more time to cover and vice versa. Accordingly, for programs with low CC like BubbleSort, PSO can reach high coverage at low timeouts while it takes more time for TDG-ACO and ACO-AR to reach such coverage.

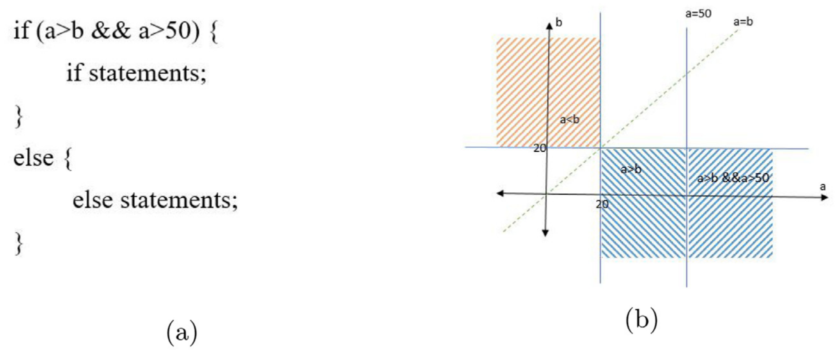
To statistically evaluate our experimental results, we conducted the Wilcoxon test in R (TEAM et al., 2014) on the average coverage values obtained from the five compared algorithms. The Wilcoxon test was done for GA versus ACO-AR-SP, ACO-AR versus TDG-ACO and ACO-AR-SP, and PSO versus ACO-AR-SP. The resulted P-values are shown in Table 7. The p-value is a number between 0 and 1 and a small value for it (typically ≤ 0.05) indicates strong evidence against the difference between results. For example, consider the resulted p-value for TDG-ACO versus ACO-AR with respect to Bessj. This p-value indicates ACO-AR significantly performs better than TDG-ACO for Bessj. As another example, based on the p-value resulted for Bubble-Sort, there is no strong evidence to show the superiority of ACO-AR compared to TDG-ACO.

The results indicate that ACO-AR-SP considerably operates better than the other algorithms for most benchmarks (17 out of 25 compared to PSO, 18 out of 25 compared to GA, and 16 out of 25 compared to ACO-AR). The main reason for this better performance is that, in the static partitioning in ACO-AR-SP, we try to build near-homogeneous partitions. We build near-homogeneous partitions in order to maintain pheromone values in the search space based on the structure of the program. The division of the search space based on the conditional statements in the program and having near-homogenous partitions may help to satisfy branching conditions earlier (i.e., ants converge to the target with higher speed).

To understand why near-homogenous partitions help to enhance the performance, we need to know why using homogeneous partitions is an effective approach for ART. Given two inputs in the same homogeneous partition, their execution patterns will be similar. Therefore, in the case of having homogenous partitions, there is no need to test the program with more than one value from each partition (Chen et al., 2015). Since there is no partitioning strategy in practice that guarantees this behavior (Chen et al., 2015), we try to address near similarity between execution patterns by the definition of near-homogeneous partitions.

Table 7
The statistical analysis of the results in the first category of experiments.

Program name	TDG-ACO vs ACO-AR	ACO-AR vs ACO-AR-SP	GA vs ACO-AR-SP	PSO vs ACO-AR-SP
Bessj	**<0.001**	0.1	**0.04**	**0.04**
BubbleSort	0.12	1.00	0.34	0.12
Encoder	0.33	1.00	0.88	0.86
Expint	**<0.001**	**0.03**	**<0.001**	**<0.001**
Fisher	**<0.001**	**0.02**	**<0.001**	**0.03**
Gammq	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Median	1.43	1.00	0.63	0.67
Remainder	0.05	0.66	**0.03**	0.33
TT1	**0.04**	**<0.001**	**<0.001**	**<0.001**
TT2	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Variance	0.34	1.00	1.2	0.91
GCD	0.23	1.00	1.00	1.00
MinMax	0.32	1.00	0.94	0.89
BinarySearch	0.24	1.00	0.12	0.14
ComputeTax	**<0.001**	**<0.001**	**<0.001**	**<0.001**
PrimeBetween	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Synthesis-1	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Synthesis-2	**<0.001**	**0.03**	**0.05**	**0.04**
PrintCalender	**0.02**	**<0.001**	**0.04**	**<0.001**
Number	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Tcas	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Totinfo	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Mcknap	**<0.001**	**0.04**	**<0.001**	**<0.001**
FastMath	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Complex	**<0.001**	**<0.001**	**<0.001**	**<0.001**



```
if (a>b && a>50) {
        if statements;
}
else {
        else statements;
}
```

Fig. 13. (a) A sample code with logical operator (b) The corresponding partitioned search space based on the clauses $a > b$ and $a > 50$.

For example, consider a program with the following clauses and obtained partition points. $a > 200$: $\{a = 200\}$, $a < b$: $\{a = 20, b = 20\}$, $a + b > c$:$\{a = 50, b = 50, c = 100\}$, $b \times b - 4 \times a \times c > 0$: $\{a = 4, b = 4, c = 1\}$ and $b \neq c$: $\{a = 150, b = 150\}$. The partition values obtained per input variable are: a={4, 20, 50, 150, 200}, b={4, 20, 50, 150}, and c={1, 100}. Consequently, the input domain of each input variable is respectively divided into 6, 5 and 3 parts. The combination of these parts builds $3 \times 5 \times 6 = 90$ partitions in the search space. If we suppose each conditional statement consists of only one clause, there are near-homogeneous partitions for each conditional statement. In this way, selecting one point from each of the near-homogeneous partitions will result in one of the branches being covered.

Even though we consider the notion of near-homogeneous partitions based on the evaluation of just one clause (i.e., we do not consider logical operators in predicates), sometimes, the partitions that cause a combination of clauses evaluated to *True* or *False* are made automatically. This causes more improvements in the performance of ACO-AR-SP. For example, consider the sample code in Fig. 13a. As shown in Fig. 13b, the partition that causes the predicate ($a > b$ && $a > 50$) evaluated to *True* is made automatically only by partitioning the search space based on clauses "$a > b$" and "$a > 50$", separately.

About the reasons for improvements caused by ART, we can refer to the results obtained in the respective papers that investigated the performance of ART compared to the pure random testing (Anand et al., 2013).

5.2.2. The second category of experiments

Experiments in the second category were repeated 30 times for each benchmark with different random seeds to take into account the stochastic nature of EvoSuite and our approach. Since EvoSuite generates test data for programs written in Java, we used 21 benchmarks of Table 4, implemented in Java. The other four benchmarks, i.e., PrintCalender, Number, Totinfo, and Mcknap, had been implemented in C.

The results of the second category of experiments are shown in Figs. 14 and 15. Fig. 14 generally displays the mutation scores of two different test suites: one produced based on BC (by EvoSuite), and the other one produced based on PPC (by our approach). The mutation scores of two different test suites per benchmark are shown in Fig. 15. The statistical analysis of the results is shown in Table 8. In this table, the significant level for the p-value is considered as p-value ≤ 0.05. The results show that, with high statistical confidence, in most programs, the generated test suites by our approach has a more mutation score, and thus, has a better ability to detect failures.

EvoSuite outperforms our approach for only two programs TT1 and Synthesis-2. In TT1, since the test paths needed for covering prime paths are the same as the test paths to be covered for

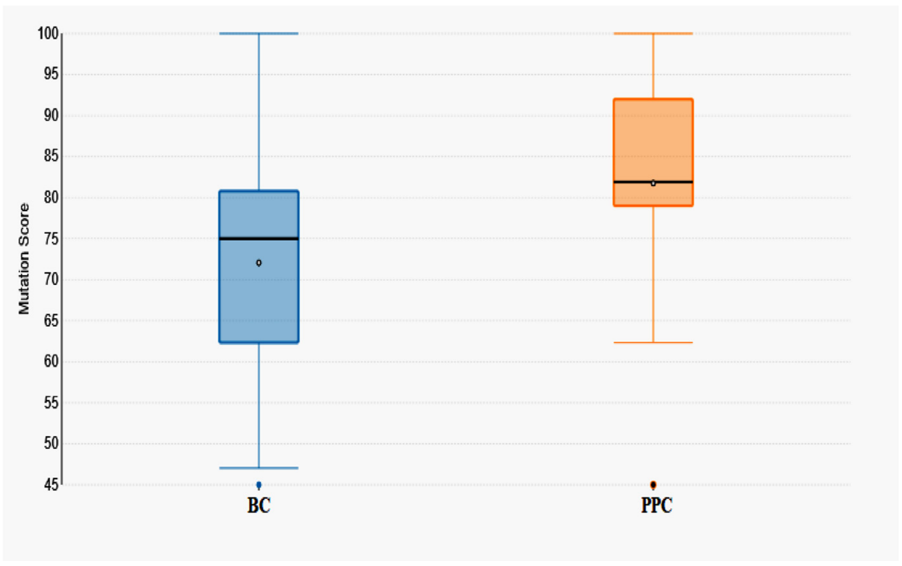


Fig. 14. Overview of the mutation score of two different test suites: one produced based on BC (by EvoSuite), and one produced based on PPC (by our approach). The mutation score is defined as the ratio of the number of killed mutants on the total number of mutants. The scale of the mutation score is from 0 to 100.

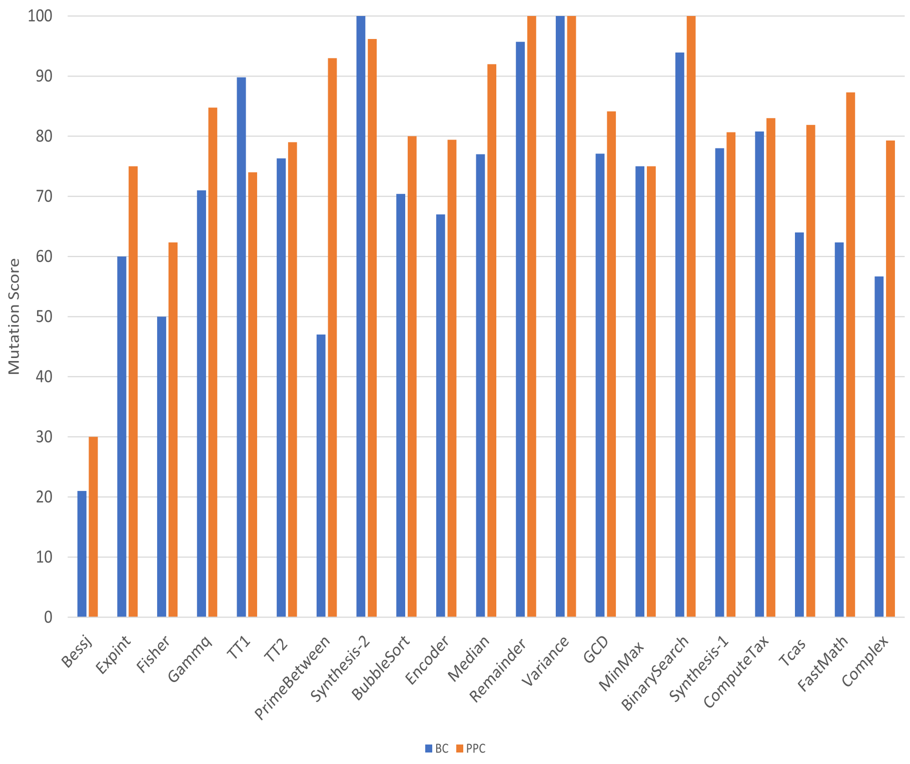


Fig. 15. The mutation score of two different test suites per benchmark.

branch coverage, our approach does not result in more effective test data in comparison to branch coverage based methods. Moreover, the strength of EvoSuite to support more than one coverage criterion at the same time (Rojas et al., 2015) may yield more test paths comparing with the methods which only consider branch coverage. This might result in more mutation score for EvoSuite in TT1.

For Synthesis-2, EvoSuite has more mutation score due to the in-feasibility of many prime paths in this program. The reason is that, by mutation analysis, a syntactic change might be applied on an infeasible path while this path has a feasible branch. In this condition, the resulting mutant can be killed by EvoSuite while it cannot be killed by test suites generated based on the prime path coverage.

In the following, we answer research questions based on the results of experiments.

- **RQ1**: Does using the idea of adaptive random testing improve the performance of ACO in test data generation?
  The results of the first category of experiments show that, with high statistical confidence, ACO-AR that applies the ART idea in the local search has a better performance compared with TDG-ACO. This is expected because the search space in the test data generation problem is often large and local search based on ART leads individuals to have better exploitation.

Table 8
Statistical analysis of the resulting mutation scores.

Program	EvoSuite	Our approach	P-value
Bessj	21.00	30.00	**<0.001**
BubbleSort	70.40	80.00	**<0.001**
Encoder	67.00	79.40	**<0.001**
Expint	60.00	75.00	**<0.001**
Fisher	50.33	62.33	**<0.001**
Gammq	71.00	84.76	**<0.001**
Median	77.00	92.00	**<0.001**
Remainder	95.70	100.00	0.081
TT1	89.80	74.00	**<0.001**
TT2	76.33	79.00	**<0.001**
Variance	100.00	100.00	1.000
GCD	77.10	84.13	**0.001**
MinMax	75.00	75.00	1.000
BinarySearch	93.93	100.00	**<0.001**
ComputeTax	80.80	83.00	**<0.001**
PrimeBetween	47.03	93.00	**<0.001**
Synthesis-1	78.00	80.67	0.254
Synthesis-2	100.00	96.17	**<0.001**
Tcas	64.00	81.90	**<0.001**
FastMAth	62.34	87.30	**<0.001**
Complex	56.67	79.30	**<0.001**
Average	72.05%	81.76%	

- **RQ2**: What is the effect of using the program predicates' information in partitioning the search space to trace the pheromone values?
  The results show that, in most programs, ACO-AR-SP that applies partitioning based on the program predicates' information has better coverage in comparison to ACO-AR. Since ACO-AR-SP traces pheromone values based on the logic of the program, individuals converge to the target with a higher speed. Partitioning based on the logic of the program leads to having better exploitation in the local search and better exploration in the global search.
- **RQ3**: Though PPC subsumes BC, does our approach, which is based on PPC, have a better failure detection capability in comparison to the approaches relying on BC?
  The results of the second category of experiments show that, with high statistical confidence, in most programs, the generated test suites to cover PPC has a more mutation score and has a better ability to detect failures.

6. Threats to validity

Threats to internal validity might come from the way the empirical study was carried out. To reduce the probability of having faults in our implementation, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, optimization algorithms have random behavior, and thus, are affected by chance. To cope with this problem, we repeated each experiment 50 and 30 for the first and the second category of experiments, respectively. Then, we followed rigorous statistical procedures to analyze the results. As a threat to the external validity of our results, it should be noted that a different selection of the benchmark programs might result in different conclusions. However, to the best of our knowledge, there is no standard set of benchmarks in test data generation, that would rather be used in our experiments. Nevertheless, we tried to use 24 different kinds of programs all of which have been used in the related work.

Since our experimental study involves unit codes with LOC$<$100, we have conducted a statistical survey to estimate how much percent of real-world unit codes have more than 100 LOC. To do this, we have randomly selected a sample of 150 unit codes from repositories of Github (as a desirable statistical population) and studied them as a sample of the statistical population. We removed two types of unit codes from the statistical sample; first, the unit codes which do not have any input parameter, and second, the unit codes which do not have any conditional statement. The reason for this exclusion is that generating test data for these unit codes does not have any specific challenge. After removing these unit codes, only 70 unit codes remained in the statistical sample. The results show that the LOC of 92.8% of the resultant unit codes is less than 100.

7. Conclusion and future work

In this paper, we customized and improved the ACO algorithm to provide an approach for covering prime paths. The proposed approach incorporates the notion of input space partitioning to maintain pheromone values in the search space. In addition, it uses a new local search based on the idea of adaptive random testing. Furthermore, the information about program predicates is used in partitioning the search space. The experimental results exhibit that the new local search based on ART and partitioning the search space based on the program predicate's information improve the performance of the ACO algorithm. The results of mutation analysis demonstrate that in the most of the programs, the test suites that cover prime paths have more capability to detect failures than those generated in order to satisfy the branch coverage criterion.

Based on the results presented in Shamshiri et al. (2015), using the information of the program under test in the initialization of individuals improves the performance of the evolutionary algorithms. Regarding this idea, the partition values obtained from predicates' information could be used in the initialization of individuals. Using these values might improve the convergence speed and the average coverage of structural test data generation techniques. This is due to the traits of these values, such that, a little change in these values modifies the value of the corresponding clause from *True* to *False* or vice versa. As our future work, we are going to use these values in the initialization of individuals to improve the performance of structural test data generation approaches.

Like TDG-ACO (Mao et al., 2015), our method can only be applied to numerical inputs. Extending our proposed ACO for programs with non-numerical inputs could be considered as a direction for future work. To customize the proposed approach for non-numerical inputs, the following works need to be performed:

1. Maintaining pheromone values in the domain of non-numerical inputs: For the numerical data types, as presented in this paper, maintaining pheromone values is done by partitioning the input domain. Since there is no direct, mathematical ordering mechanism for the non-numerical data types, we should first define and implement suitable ordering mechanisms for them, and then, provide a way to divide each sorted domain into equal parts (as mentioned for the numerical data type). Recently, to make ART applicable for non-numerical data types, a new method for partitioning non-numerical inputs has been expressed. Extensive experiments have been conducted to investigate the effectiveness of this method (Barus et al., 2016). The results show that ART can be used for all types of inputs via partitioning the search space. We can directly use the same partitioning approach to maintain pheromone values in the search space. In this way, the second and the third contributions of this paper (i.e., local search based on ART and using static information to build near-homogeneous partitions) can be applied to non-numerical inputs, as well.
2. Neighborhood definition for partitions: Because of the lack of mathematical ordering in non-numerical data types, we must provide a definition for the neighborhood concept for these

types. This is a prerequisite of the local search, and is raised in all swarm intelligence algorithms when used to generate test data for non-numerical input parameters. Having a definition for the neighborhood concept, there will be no need to change our procedures, i.e., local search, global search, and update pheromone.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Atieh Monemi Bidgoli: Conceptualization, Software, Formal analysis, Investigation, Writing - original draft, Visualization. **Hassan Haghighi:** Conceptualization, Validation, Writing - review & editing, Supervision, Project administration.

Acknowledgement

The authors would like to thank Dr. Andrea Arcuri for valuable advice regarding the way of doing the evaluation and extremely useful comments on the earlier drafts of this paper. We are also grateful to the authors of Durelli et al. (2018) to provide us the scripts to extract the prime paths of the program under test.

References

Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K., 2010. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Software Eng. 36 (6), 742–762.
Ammann, P., Offutt, J., 2016. Introduction to Software Testing. Cambridge University Press.
Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P., et al., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Software 86 (8), 1978–2001.
Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S., 2006. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Software Eng. 32 (8), 608–624.
Arcuri, A., 2013. It really does matter how you normalize the branch distance in search-based software testing. Software Test. Verif. Reliab. 23 (2), 119–147.
Arcuri, A., Briand, L., 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Test. Verif. Reliab. 24 (3), 219–250.
Arcuri, A., Fraser, G., 2013. Parameter tuning or default values? an empirical investigation in search-based software engineering. Empir. Software Eng. 18 (3), 594–623.
Barus, A.C., Chen, T.Y., Kuo, F.-C., Liu, H., Merkel, R., Rothermel, G., 2016. A cost-effective random testing method for programs with non-numeric inputs. IEEE Trans. Comput. 65 (12), 3509–3523.
Bertolino, A., 2007. Software testing research: achievements, challenges, dreams. In: 2007 Future of Software Engineering. IEEE Computer Society, pp. 85–103.
Bidgoli, A.M., Haghighi, H., Nasab, T.Z., Sabouri, H., 2017. Using swarm intelligence to generate test data for covering prime paths. In: International Conference on Fundamentals of Software Engineering. Springer, pp. 132–147.
Blum, C., Li, X., 2008. Swarm intelligence in optimization. In: Swarm Intelligence. Springer, pp. 43–85.
Bueno, P.M.S., Jino, M., 2002. Automatic test data generation for program paths using genetic algorithms. Int. J. Software Eng. Knowl. Eng. 12 (6), 691–709.
Chen, T.Y., Kuo, F.-C., Towey, D., Zhou, Z.Q., 2015. A revisit of three studies related to random testing. Sci. China Inf. Sci. 58 (5), 1–9.
Cohen, M.B., Colbourn, C.J., Ling, A.C., 2003. Augmenting simulated annealing to build interaction test suites. In: Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on. IEEE, pp. 394–405.
Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. Pit: a practical mutation testing tool for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 449–452.
Dorigo, M., Birattari, M., Stutzle, T., 2006. Ant colony optimization. IEEE Comput. Intell. Mag. 1 (4), 28–39.
Durelli, V.H., Delamaro, M.E., Offutt, J., 2018. An experimental comparison of edge, edge-pair, and prime path criteria. Sci. Comput. Program. 152, 99–115.
Elbeltagi, E., Hegazy, T., Grierson, D., 2005. Comparison among five evolutionary-based optimization algorithms. Adv. Eng. Inf. 19 (1), 43–53.
Floreano, D., Mattiussi, C., 2008. Bio-inspired artificial intelligence: Theories, methods, and technologies. MIT press.
Frankl, P.G., Weyuker, E.J., 1993. A formal analysis of the fault-detecting ability of testing methods. IEEE Trans. Software Eng. 19 (3), 202–213.
Fraser, G., Arcuri, A., 2013. Whole test suite generation. IEEE Trans. Software Eng. 39 (2), 276–291.
Fraser, G., Rojas, J.M., Campos, J., Arcuri, A., 2017. Evosuite at the sbst 2017 tool competition. In: Proceedings of the 10th International Workshop on Search-Based Software Testing. IEEE Press, pp. 39–41.
Gupta, A., Jalote, P., 2008. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. Int. J. Software Tools Technol. Trans. 10 (2), 145–160.
Hamlet, R.G., 1977. Testing programs with the aid of a compiler. IEEE Trans. Software Eng. (4) 279–290.
Harman, M., McMinn, P., 2010. A theoretical and empirical study of search-based testing: local, global, and hybrid search. IEEE Trans. Software Eng. 36 (2), 226–247.
Jones, B.F., Sthamer, H.-H., Eyres, D.E., 1996. Automatic structural testing using genetic algorithms. Software Eng. J. 11 (5), 299–306.
Li, H., Lam, C.P., 2005. An ant colony optimization approach to test sequence generation for state based software testing. In: Quality Software, 2005.(QSIC 2005). Fifth International Conference on. IEEE, pp. 255–262.
Li, K., Zhang, Z., Liu, W., 2009a. Automatic test data generation based on ant colony optimization. In: Natural Computation, 2009. ICNC'09. Fifth International Conference on, 6. IEEE, pp. 216–220.
Li, N., Praphamontripong, U., Offutt, J., 2009b. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on. IEEE, pp. 220–229.
Lin, J.-C., Yeh, P.-L., 2001. Automatic test data generation for path testing using gas. Inf. Sci. 131 (1), 47–64.
Mao, C., 2014a. Generating test data for software structural testing based on particle swarm optimization. Arab. J. Sci. Eng. 39 (6), 4593–4607.
Mao, C., 2014b. Harmony search-based test data generation for branch coverage in software structural testing. Neural Comput. Appl. 25 (1), 199–216.
Mao, C., Xiao, L., Yu, X., Chen, J., 2015. Adapting ant colony optimization to generate test data for software structural testing. Swarm Evol. Comput. 20, 23–36.
Offutt, A.J., Untch, R.H., 2001. Mutation 2000: Uniting the orthogonal. In: Mutation testing for the new century. Springer, pp. 34–44.
Pargas, R.P., Harrold, M.J., Peck, R.R., 1999. Test-data generation using genetic algorithms. Software Test. Verif. Reliab. 9 (4), 263–282.
Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A., 2015. Combining multiple coverage criteria in search-based unit test generation. In: International Symposium on Search Based Software Engineering. Springer, pp. 93–108.
Sabor, K.K., Thiel, S., 2015. Adaptive random testing by static partitioning. In: Proceedings of the 10th International Workshop on Automation of Software Test. IEEE Press, pp. 28–32.
Shamshiri, S., Rojas, J.M., Fraser, G., McMinn, P., 2015. Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1367–1374.
Sharifipour, H., Shakeri, M., Haghighi, H., 2018. Structural test data generation using a memetic ant colony optimization based on evolution strategies. Swarm Evol. Comput. 40, 76–91.
Simons, C., Smith, J., 2012. A comparison of evolutionary algorithms and ant colony optimization for interactive software design. In: Proceedings of the 4 th Symposium on Search Based-Software Engineering, p. 37.
Srivastava, P.R., Baby, K., 2010. Automated software testing using metahurestic technique based on an ant colony optimization. In: Electronic System Design (ISED), 2010 International Symposium on. IEEE, pp. 235–240.
TEAM, R.C., et al., 2014. R: a language and environment for statistical. R Foundation for Statistical Computing, Viena, Austria, Vienna, Austria.
Tracey, N., Clark, J., Mander, K., McDermid, J., 1998. An automated framework for structural test-data generation. In: Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on. IEEE, pp. 285–288.
Windisch, A., Wappler, S., Wegener, J., 2007. Applying particle swarm optimization to software testing. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1121–1128.

Atieh Monemi Bidgoli received her B.S. degree in Computer Science from the Faculty of Computer Science, University of Kashan, Kashan, Iran, in 2010. She received her M.Sc. degree in Software Engineering from the Faculty of Computer Science at Sharif University of Technology, Tehran, Iran in 2012. She is currently a Ph.D. candidate in the Faculty of Computer Science and Engineering at Shahid Beheshti University. Her research interests are in the area of software testing.

Hassan Haghighi is Associate Professor at the Faculty of Computer Science and Engineering, Shahid Beheshti University, Iran. He received his Ph.D. degree in software engineering from Sharif University of Technology, Iran, in 2009. His main research interest includes software testing and using formal methods in the software development life cycle.