Using Static Information of Programs to Partition the Input Domain in Search-based Test Data Generation

Atieh Monemi-Bidgoli
Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran
monemiatieh@gmail.com
Hassan Haghighi*
Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran
h_haghighi@sbu.ac.ir

Received: 01/Oct/2020 Revised: 06/Dec/2020 Accepted: 10/Jan/2021

Abstract
The quality of test data has an important effect on the fault-revealing ability of software testing. Search-based test data generation reformulates testing goals as fitness functions, thus, test data generation can be automated by meta-heuristic algorithms. Meta-heuristic algorithms search the domain of input variables in order to find input data that cover the targets. The domain of input variables is very large, even for simple programs, while this size has a major influence on the efficiency and effectiveness of all search-based methods. Despite the large volume of works on search-based test data generation, the literature contains few approaches that concern the impact of search space reduction. In order to partition the input domain, this study defines a relationship between the structure of the program and the input domain. Based on this relationship, we propose a method for partitioning the input domain. Then, to search in the partitioned search space, we select ant colony optimization as one of the important and prosperous meta-heuristic algorithms. To evaluate the performance of the proposed approach in comparison with the previous work, we selected a number of different benchmark programs. The experimental results show that our approach has 14.40% better average coverage versus the competitive approach.

Keywords: search-based software testing; test data generation; ant colony optimization; input space partitioning.

1- Introduction

Software testing is a vital part of the software development life cycle with the aims of revealing failures in a program under test. Besides improving the quality of the testing activity, automation also reduces cost and time [3][4]. Test data generation as the main part of the software testing process is the activity of finding test data for testing programs, effectively.

Symbolic execution and dynamic methods are known as the two main approaches in automatic test data generation [5]. In symbolic execution [6] symbolic values are assigned to the input variables in order to formulate program paths in terms of logical constraints. These constraints must be solved in order to discover input values that trace specific paths in the program. Dependency on the capability of constraint solvers (that are unable to solve complex constraints) is the main issue of this method; pointer references, loop-dependent or array-dependent variables, and calls to functions whose implementations are unknown and external libraries also are the issues related to this approach. In dynamic methods by instrumenting the program and executing it with some input data, the state of the program is observed. Since functions are executed with real argument, pointer values and array subscripts are known at run-time, thus, many of the problems relevant to the symbolic execution are resolved.

The application of optimization algorithms as dynamic methods in test data generation is called Search-Based Software Testing (SBST). In this approach, the input domain of the program is the search space, and a fitness function is determined to evaluates and scores different inputs of the program (as solutions) with respect to the given test criterion. The fitness function aims to guide the search into promising, unevaluated areas of the search space.

The authors of [1] investigated the relationship between the search space size and search effectiveness and efficiency. They proposed a method to reduce the search space by removing irrelevant variables that are recognized based on the slicing approach. This approach has been applied to three categories of meta-heuristic algorithms, Hill Climbing as a local search, Genetic Algorithm as a global technique, and Memetic Algorithm as a hybrid optimization technique, which is based on the combination of global and local searches. The method has shown a

* Corresponding Author

positive effect on three meta-heuristic algorithms but has not outperformed random testing.

In order to not being limited to irrelevant variables, we introduce an approach to partition the input domain of relevant variables. Our approach focuses on the analysis of program predicates, i.e., places where logical expressions of variables are evaluated to select the next branch to continue. In fact, we are going to establish a relationship between the structure of the program and the input domain. We obtain some values per each input variable. Then, we partition the search space with respect to these values.

Furthermore, we customize the basic Ant Colony Optimization (ACO) algorithm, as a well-known optimization algorithm, according to the partitioned space. This way, we propose a new test data generation approach which is based on the static analysis of the code.

To evaluate our proposed approach, we consider average coverage as the evaluation metric. According to the results of the experiments, this approach has better results in comparison to the previous work. We will also show that the suggested static input space partitioning approach implicitly contains irrelevant variable removal capability [1], as well.

The rest of the paper is structured as follows. In the next section, a brief overview of related work is given. In Section 3, our approach to partition the search space with the modified ACO algorithm is presented. The experimental results and analysis are presented in Section 4, followed by the conclusion and an outline of future work in Section 5.

2- Related Work

In this section, we first review some of the important works for test data generation based on different optimization algorithms, such as Genetic Algorithm (GA), Simulated Annealing (SA), ACO, and Particle Swarm Optimization (PSO) with more emphasis on ACO because we customize ACO based on our input space partitioning. Then, the approaches related to input domain reduction in the search-based test data generation [1] are presented.

In the 1990s, GA was tuned to generate test data. Jones [7] and [8] examined the usage of GA in order to automate test data generation with respect to branch coverage. Their experiments on some small programs demonstrate that GA notably works better than the random testing method. An empirical study on GA-based test data generation for large-scale programs performed by Harman and McMinn [9][10]. Their experiments showed the superiority of GA over other optimization algorithms such as Hill Climbing. A tool named EvoSuite was implemented by Fraser et al. [11] to generate test suite for satisfying the determined coverage criterion. In EvoSuite, a list of coverage criteria can be set such as branch coverage, data flow, and mutation testing.

Tracey et al. introduced a framework for generating test data by using SA as one of the well-known optimization algorithms which works based on the idea of neighborhood search [12]. Their method applies SA to structural test data generation with the hope of overcoming some of the problems raised with the application of local search. In this work, test data can be generated for coverage criteria such as branch and statements coverage. Moreover, Cohen et al. used SA in order to generate test data in combinatorial testing [13].

Sine ACO has shown notable results in solving optimization problems [14][15][16], some scholars have utilized it to resolve software engineering problems in a wide range of sub-fields such as software project scheduling [17], release planning optimization [18], software quality prediction [19], and software testing [20]. Lam et al. [21] and Srivastava et al. [22] utilized ACO to generate test sequences for state-based software testing. In conformance testing of object-oriented software, the problem of state explosion was solved by Bouchachia et al. [23] via presenting Class Finite State Machines (CFSM).

Li et al. [24] also utilized the ACO algorithm for generating test data in respect of the branch coverage criterion. However, this study lacked detailed experimental and comparative analysis. Mao et al. [2] also applied ACO for generating test data and have compared their approach against GA, PSO, and SA for the same purpose. Their findings exhibited that ACO has better performance than GA and SA and is comparable to PSO.

The approach in [25] outperformed the work of Mao et al. [2] by incorporating (1+1)-evolution strategies to enhance search exploitation through improving the movement of ants in the local search. In [25]**,** pheromone values were defined in each branch, and also, were considered as a part of the fitness function to discourage every ant from traversing branches already covered by other ants. Since this viewpoint is only appropriate for branch coverage, authors in [26][27] introduced a method for using ACO to cover prime paths. They applied the idea of adaptive random testing in local search and used the information of program predicates in partitioning the search space [26]. The experimental results confirm the positive effects of the proposed approach, especially for programs with complex predicates.

Regarding the approaches related to the input domain reduction, the authors of [28] reduced the search space by the interval arithmetic method. Their approach is appropriate only for simple predicates such that each side of the clauses contains a single interval variable. Also, some important issues, such as converting local variables to input variables, have not been addressed in their

approach. Therefore, we do not consider this approach in the evaluation section.

An approach in order to reduce the dimension of the input domain introduced by Harman [1], [29] called "irrelevant input variable removal". Irrelevant input variables are input variables that do not affect executing the target structure. Therefore, they can be removed from the input domain without affecting the feasibility of the target. Their approach was empirically evaluated for search-based structural test data generation. The results showed that irrelevant input variable removal has no impact on the random search, but enhance the performance of optimization techniques. The authors of [1] encouraged concentrating on relevant variables to more reduce the search space via utilizing the static analysis stage; an idea which is followed in this study.

3- Proposed Approach

In this section, after describing the overall process of our approach, we explain our static analysis approach to input space partitioning. Then, we explain the customization of the ACO algorithm based on the partitioned space in section 3.3. The proposed algorithm produces test data to satisfy the desired coverage criterion.

3-1- Overall Process

Fig. 1 shows the overall process of our approach, which consists of two main phases: *Partitioning the input domain* and *Customizing the ACO algorithm*.

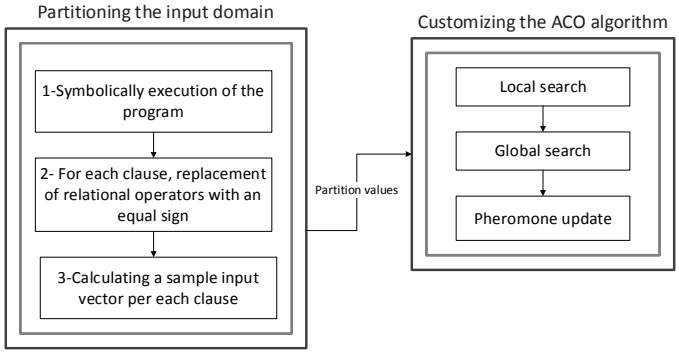

Fig. 1: The overall process of the proposed approach

Partitioning the input domain: Since the search space is constructed by the domain of input variables, we start with symbolic execution [30] to transform each clause of the program to a new clause that only includes input variables. Each resulting clause divides the search space into two partitions; the input vectors in one partition cause the clause to be evaluated to True, while the input vectors in the other partition lead to False value for the clause. Replacing the relational operator (i.e., $<, \leq, >, \geq, \neq$) by the equality operator in clauses, the borders of these partitions are determined. Each input vector on these borders can be considered as partition values.

Customizing the ACO algorithm: In this phase, the ACO algorithm is customized based on the partitioned search space obtained in the previous phase.

3-2- Partitioning the Input Domain

To perform static partitioning, the program clauses should be initially analyzed. A clause is a predicate that does not have any logical operator. For example, the predicate $((a + b > c)\,\&\&\,(b + c > a)\,\&\&\,(c + a > b))$ contains three clauses. The output of analysis is a set of values per each input variable. The partitioning of the input space is done based on these values. In the rest of the paper, these values are called partition values. For obtaining partition values, the following three steps must be done. Step 1 is done for predetermined test paths of the program (We assume that these test paths cover all the branches of the program); step 2 and step 3 are done for each clause.

1. By performing symbolic execution for predestinate test paths of the program, the clauses of the program are converted such that they only involve input variables. This is carried out because the space we are searching through is constructed by input variables.
2. Modifying each clause *C* by using the (=) operator instead of the $(\leq, \geq, \neq, <, >)$ operators. The resulting clause is called C'.
3. Finding a combination of input values that satisfy C'. For example, values that satisfy $a + b = c$ (e.g., a = 100; b = 100; c = 200) can be considered as partition values for clause a + b > c.

For clarifying the elimination of non-input variables in step 1, static partitioning is done on a sample program shown in *Fig. 2*.

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

(a)

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

(b)

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

(c)

Fig. 2: Sample program (a) The first predicate (b) The second predicate with the related assignments (c) The third predicate with the related assignments

As it can be seen in Fig. 2, three conditions exist in the program, in lines 3, 8, 9; the first predicate does not have non-input variables (Fig. 2. (a)), while the second predicate in line 8 has variable n that is a non-input variable; assignments which could be used to calculate the relationship between n and input variables are distinguished by red in Fig. 2. (b) . At last, two functions $"n = a"$ and $"n = d"$ are achieved to show the relationship between n and input variables. The same thing is done for the predicate in line 9 (Fig. 2. (c)).

The predicates that are obtained by eliminating the non-input variables along with the obtained partition values are shown in Table 1.

Table 1: Partition values for each predicate of the sample program

Line number	predicate	Partition values		
		a	b	d
3	$a < d$	100		**100**
8	$a < 8$	8		
8	$d < 8$			**8**
9	$b > a$	100	100	
9	$b > d$		100	**100**

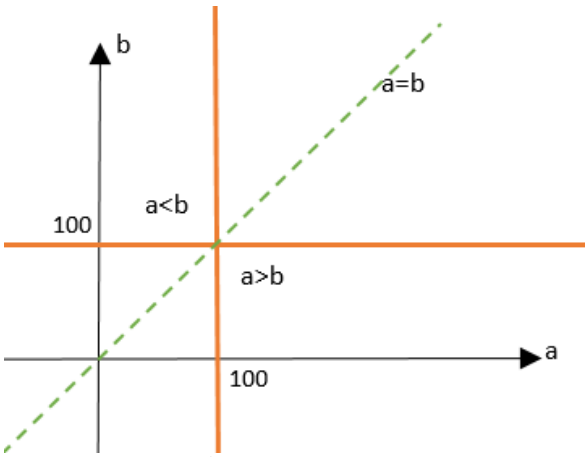

Fig. 3: Partition values for the predicate a<b in the two-dimensional search space

To illustrate why we use these partition values in partitioning the search space, we review some examples. The only input vector for partitioning the clause *a<8* is *a=8*, which causes the input domain of *a* to be divided into two parts; input vectors in one of these parts lead the True value for *a<8* while input vectors in the other part make *a<8* False. Regarding clause *a<b*, with *a=100* and *b=100* as selected partition values, there are (2*2=4) partitions in the whole input domain. As shown in *Fig. 3*, all input

vectors in the top left part force the program execution to traverse the True case, while all input vectors in the bottom right part cause it to traverse the False case of clause $a<b$. As additional example, input vectors that satisfy $a+b=c$ (e.g., $a= 100$, $b=100$ and $c=200$) are suitable to be selected as partition values for clause $a+b>c$. Due to partitioning the input domain of a, b and d into two parts, there are (2*2*2=8) partitions in the whole input domain. Values in at least one of these parts make $a+b>c$ True, and values in at least one of these parts cause $a+b>c$ to be False.

Although we illustrated the partitioned area in the search space only for one clause in the above example, in the next section, we use the partitioned space produced by all the clauses of the program.

3-3- Customizing the ACO Algorithm

ACO as an optimization algorithm is inspired from ants that release pheromone in the environment. ACO algorithms were originally utilized to solve the shortest route in traveling salesman problem [31]. To generate test data, we change the basic ACO with respect to the partitioned search space.

We first formally describe the test data generation for the program under test P. Suppose P has d input variables represented by vector $X = (x_1, x_2, \dots, x_d)$. Vector X is the position vector of an ant in ACO. If each input variable x_i $(1 \le i \le d)$ takes its values from the domain D_i, the corresponding input domain of the program is $D = D_1 \times D_2 \times \dots \times D_d$.

In the problem of generating test data, each ant's position actually is a test data which is shown by a vector in the input domain D. For any ant k $(1 \le k \le n)$, its position is marked as $X_k = (x_1, x_2, \dots, x_d)$.

For generating test data with the ACO algorithm, an important consideration is the form of pheromone. In this paper, pheromones are defined on the partitions established by static partitioning, explained in the previous section. This way, in each partition, there is a pheromone value that is initialized to one.

The pseudo-code of the customized ACO algorithm is presented in Algorithm 1[1]. Table 2 contains the notations and parameters used in this algorithm. The output of Algorithm 1 is a set of test data (or a test suite) that cover the given test targets. The test data generation process in Algorithm 1 is repeated until all the test targets are traversed by the test suite, or the predefined number of iterations is exceeded. The data generation process consists of two stages. In the first stage, all pheromone values are initialized by one (Lines 4-9) and an input vector in the input domain is randomly assigned to each ant as the position vector (Lines 10-12).

In the second stage (Lines 13-34), the local search and global search are performed for each ant. Then, the pheromone values are updated with respect to Eq. 1 (Section 3.2.3). The fitness values of the ants are calculated at the end of each iteration. The position of any ant k that covers an uncovered test target is added to the test suite, i.e., TS. The methods local search, global search, and pheromone update are explained in the following sub-sections.

Algorithm 1 The proposed ACO algorithm

```
1: Input:
       1. SUT and ACO input parameters.
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:     τ[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 iteration ≤ maxIteration 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 path then
27:       Add ant[k].position to TS;
28:     end if
29:   end for
30:   if all path are covered then
31:     Add ant[k].position to TS;
32:     Flag = true;
33:   end if
34: end while
35: Return TS;
```

[1] This algorithm along with Table 2 is similar to Algorithm 1 and Table 3 in [26], respectively. This is because both are based on ACO. However, the way of development was different in both papers.

Table 2: The parameters and notations used in the proposed ACO algorithm

Parameter/Notation	Description
maxIteration	Maximum iteration
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*
b[i]	The number of parts for the ith input variable
TS	The output test suite

3-3-1- Local Search

In the local transfer of ants, for an ant k $(1 \leq k \leq n)$ in partition b, the aim is to investigate whether there is a partition in the neighborhood of b that has a better fitness function. If there is such a partition, the ant k will transfer to the partition with the best fitness function value. This transfer increases the value of pheromone in the destination partition.

The neighboring partitions of an ant k $(1 \leq k \leq n)$ in partition *b* are the partitions that have at least one common partition value. In our implementation, a random location per neighboring partition is selected, and these locations are the representatives of their partitions.

In the local search ant k transfer from Xk to Xk if the fitness of Xk is better than that of Xk and Xk is in the neighborhood's partition of Xk which has the best fitness value amongst neighborhoods of Xk; otherwise, the ant's location does not change in the local search and remains in the previous location. It must be noted that less fitness is considered better fitness, and the best fitness value is 0. This process is done for all ants in the partitioned space.

3-3-2- Global Search

The global search is used to solve two problems related to local search. First, there may be some partitions with acceptable fitness that are not visited by any ant in a reasonable time (or iteration limit) and second, there may be ants with the local optima trap [32][32] (that could not find a neighboring place with superior fitness value). To resolve these issues, when each ant's fitness value is worse than the average fitness value of all ants, a random value q is generated. If q is less than a predefined probability $q0$, the ant will randomly be moved to a new partition; otherwise, the partition with the highest pheromone value will be the destination of the ant.

3-3-3- Pheromone Update

Eq.1 is used to update the pheromone value in each partition of the input domain.

$$\tau(j) \leftarrow \alpha \times number\ of\ ants\ in\ partition\ j + (1-\alpha) \times \tau(j) \quad (1)$$

Where $\alpha \in (0, 1)$ represent pheromone evaporation rate, $\tau(j)$ is the amount of pheromone in the *jth* partition, and *j* stands for partition index.

4- Experiment

In this section, we compare our approach with the approach presented in [1]. Although the competitive approach has been applied to the genetic algorithm, hill climbing, and memetic algorithm, we select its implementation with the genetic algorithm because the experimental results in [1] showed that removing irrelevant input variables has the greatest effect on the genetic algorithm.

Evaluation Metrics

Average Coverage (AC), Average Time (AT), and Mutation Score (MS) are used as the evaluation metrics. Average coverage is the average percentage of the covered branches and is calculated while the two competitive approaches run with the same iterations.

Average time is the average of elapsed time that has been taken to run the algorithms and is calculated to compare the efficiency of the two competitive approaches.

Mutation score is a testing metric provided by the mutation analysis as a fault-based testing technique. To perform mutation analysis, PIT [33] is used as a state-of-the-art tool for this purpose.

Benchmark Programs

To conduct experiments, several benchmark programs have been selected (see Table 3): the first eleven programs from the Numerical Case Study (NCS) of EvoSuite[1]. Tcas and Totinfo from the Software-artifact Infrastructure Repository (SIR)[2], and the others from various related work. Table 3 displays the number of lines of code (LoC), and the description of each benchmark program.

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

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

The Parameters of the Algorithms

The parameters of algorithms have been set to the values presented in Table 4 before performing the experiments. Parameter selection for our algorithm was done based on the sensitivity analysis which had been done in [2]. Although we can use any coverage criteria, in this paper, we consider branch coverage with the fitness function proposed in [1].

Table 3. Programs selected for the empirical studies

#p	Program name	LoC	Description
1	Bessj	245	Bessel *J*n function
2	BubbleSort	38	Bubble sort algorithm
3	Encoder	86	Encode the input
4	Expint	109	Exponential integral function
5	Fisher	157	Fisher statistical function
6	Gammq	112	Gamma function
7	Median	40	Calculate the median value of the input array
8	Remainder	30	Calculate the remainder of the first argument divided by the second argument
9	TT1	43	Find the type of triangle
10	TT2	50	Find the type of triangle
11	Variance	43	Calculate the variance of the input array
12	GCD	24	Find the greatest common divisor
13	MinMax	41	Find the minimum and maximum values in an array
14	BinarySearch	55	Binary search algorithm
15	ComputeTax	164	Compute tax amount
16	PrimeBetween	42	Calculate the prime number between two numbers
17	Synthesis-1	48	Synthesis of while, for and if
18	Synthesis-2	101	Synthesis of while, for and if
19	PrintCalender	187	Print calendar according to two inputs "year" and "month"
20	Number	265	Calculate the number of days between two dates
21	Tcas	173	Avoid air craft collision
22	Totinfo	416	Calculate some statistical information
23	Mcknap	1301	Solve the knapsack problem

Table 4. Parameter setup

	Parameter	Value
Genetic Algorithm	Selection method	Roulette wheel
	Crossover method	Single point
	Crossover probability	80%
	Mutation probability	0.05%
	Chromosome-type	Binary string
ACO	α	0.3
	q_0	0.5
Both Algorithms	Population size	30
	No of iteration	50

4-1- Experiment Results

Experiments were repeated 50 times with various initial population to consider the accidental nature of optimization algorithms. The average coverage resulted per each algorithm for all benchmarks are displayed in Fig. 4. The results demonstrate that the proposed approach has better average coverage for most benchmarks, except three, i.e., BubbleSort, Median, and Variance. The two approaches reached 100% average coverage for these three benchmark programs because satisfying the conditions of these programs is very easy.

The Wilcoxon test in R [34] are conducted to statistically evaluate our experimental results. Table 5 presents the average coverage and average time along with resulted P-values and effect size. The effect sizes of the comparisons are quantified with the Vargha-Delaney $\hat{A}$ statistics. In case of average coverage, $\hat{A}_{xy}$ is an estimation of the probability that, if we run the approach *x*, we will obtain better coverage than running it with the approach *y*. When two approaches are equivalent, then $\hat{A}_{xy} = 0.5$. A high-value $\hat{A}_{xy} = 1$ means that, in all of the runs of *x*, we obtained higher coverage than the coverage obtained in all of the runs with *y*.

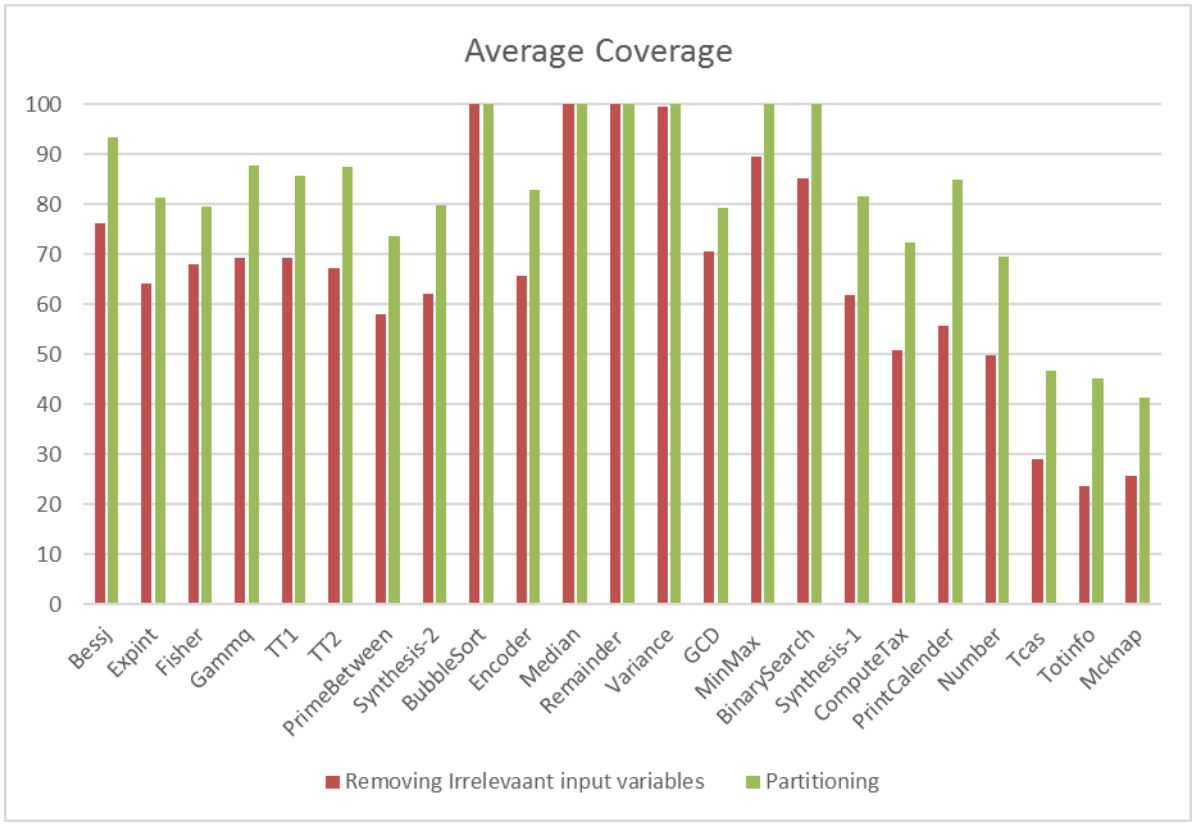

Fig. 4: The resulted average coverage by the proposed approach and the approach presented in [1]

Table 5. Statistical analysis of the results of experiments

Program name	The approach in [1]		Our approach		AC		AT	
	AC	AT	AC	AT	Effect size	P-value	Effect size	P-value
Bessj	76.11	135.34	93.44	138.98	0.73	**0.01**	0.43	0.05
BubbleSort	64.20	35.23	81.22	39.78	0.53	1.00	0.41	1.00
Encoder	67.89	65.39	79.44	57.67	0.59	1.00	0.58	1.00
Expint	69.22	76.23	87.66	70.36	0.71	**<0.001**	0.60	0.06
Fisher	69.20	110.67	85.83	67.89	0.64	**0.03**	0.61	0.09
Gammq	67.23	104.78	87.39	99.67	0.69	**<0.001**	0.53	**<0.001**
Median	57.89	3.65	73.55	6.30	0.52	1.00	0.42	1.00
Remainder	62.22	4.6	79.78	4.99	0.58	0.76	0.49	0.76
TT1	79.67	43.2	100	41.49	0.50	**<0.001**	0.5	**0.01**
TT2	65.74	36.2	82.79	30.5	0.71	**<0.001**	0.58	**<0.001**
Variance	100	23.7	100	31.89	0.5	0.15	0.34	0.95
GCD	100	25.4	100	20.5	0.5	0.94	0.59	0.74
MinMax	99.5	3.6	100	4.50	0.56	0.28	0.42	0.98
BinarySearch	70.6	6.5	79.33	7.78	0.58	0.90	0.49	0.40
ComputeTax	89.6	48.9	100	40.5	0.69	**<0.001**	0.52	**<0.001**
PrimeBetween	85.2	250.7	100	200.31	0.59	**0.01**	0.69	**0.03**
Synthesis-1	61.78	154.7	81.56	143.64	0.67	**0.02**	0.77	**0.04**
Synthesis-2	50.76	120.7	72.44	119.43	1	**<0.001**	0.51	**0.03**
PrintCalender	55.78	67.98	85	50.32	1	**<0.001**	0.69	**<0.001**
Number	49.67	301.67	69.55	278.43	0.98	**<0.001**	0.76	**<0.001**
Tcas	28.9	404.7	46.8	306.43	0.87	**<0.001**	0.80	**<0.001**
Totinfo	23.67	505.6	45.17	398.54	0.82	**<0.001**	0.76	**<0.001**
Mcknap	25.78	408.5	41.33	375.89	0.78	**<0.001**	0.61	**<0.001**

The results reveal significant improvements in the average coverage for 15 out of 23 benchmarks and significant improvement in the average time for 7 out of 23 benchmarks in comparison to the approach presented in [1]. The main cause for this outperformance is partitioning the input domain based on the information that exists in the conditional statement. In other words, we created a relationship between the input domain and the structure of the program. This causes performing searches more intelligently. Therefore, individuals converge to the test goal with higher speed. Utilizing the logic of the program to trace pheromone values results in having better exploitation. Furthermore, this causes having better exploration in the partition which has the highest pheromone value.

To explain more precisely, consider the following clauses and calculated partition values that are selected from program Synthesis-1[1]:

- $x > 200 : \{x = 200\}$,
- $x < y : \{x = 20; y = 20\}$,
- $x + y > z : \{x = 50; y = 50; z = 100\}$,
- $y \times y - 4 \times x \times z > 0 : \{x = 4; y = 4; z = 1\}$
- $y \neq x : \{y = 150; x = 150\}$.

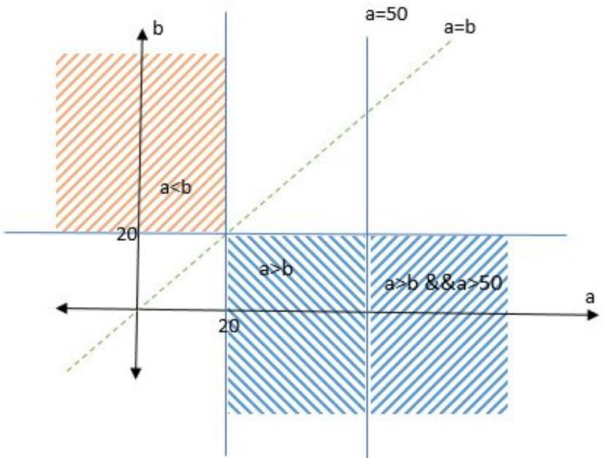

Fig. 5: The partitioned input domain based on the clauses a > 50 and a > b [26]

The obtained partition values for each input variable are: x={4, 20, 50, 150, 200}, y={4, 20, 50, 150}, and z={1, 100}. As the result, the input domain of x, y, and z respectively divided into 6, 5, and 3 parts. The composition of them creates $3 \times 5 \times 6 = 90$ partitions in the whole input domain. If we assume each predicate involves only one clause, choosing one input vector from each of these partitions will lead to one of the branches being traversed. Partitioning based on the program structure causes individuals to converge to the targets sooner than when we just consider irrelevant input variables.

Even though the search space is partitioned by considering only one clause (i.e., we do not consider a predicate), occasionally, the partitions that lead to True or False for a predicate are created spontaneously. This leads to more improvements in the efficiency of the proposed approach. To more explain, consider predicate (a > b && a > 50). As shown in Fig. 5, the partition that leads True for this predicate is made spontaneously just via dividing the input domain by "a > 50" and "a > b", separately.

Most importantly, our approach implicitly benefits from the strength point of the previous work [1]. In the case of having an irrelevant input variable, by definition, this variable is not used in any predicate of the target test paths. Since only the involved input variables are used for obtaining partition points in our approach, no partition value is found for irrelevant input variables. Consequently, only one part exists with respect to the domain of an irrelevant input variable; hence, it does not matter which value is selected for irrelevant input variables.

Table 6: Statistical analysis of the resulting mutation scores.

Program	The approach in [1]	Our approach	Effect size	P-value
Bessj	21.34	29.43	0.98	0.67
BubbleSort	71.4	80.98	0.97	0.06
Encoder	67.56	76.54	0.89	**< 0.001**
Expint	59.43	73.76	0.99	**< 0.001**
Fisher	45.43	56.33	0.86	**< 0.001**
Gammq	68.87	81.96	0.9	0.56
Median	87.22	91.89	0.84	0.34
Remainder	95.7	98.12	0.82	0.08
TT1	92.8	95.1	0.5	0.05
TT2	78.33	79.32	0.87	0.10
Variance	100	100	0.5	1
GCD	77.1	84.13	0.5	0.09
MinMax	75	75	0.51	1
BinarySearch	93.93	100	0.68	0.78
ComputeTax	80.8	83	0.69	**< 0.001**
PrimeBetween	64	81.9	0.1	**< 0.001**
Synthesis-1	78	80.67	0.95	**< 0.001**
Synthesis-2	100	96.17	0.99	0.25
Tcas	47.03	93	0.89	**< 0.001**

We performed mutation analysis to experimentally investigate the failure detection capability of test suites generated by the proposed approach against test suites produced by the previous approach [1]. In order to conduct this analysis, we used 19 of 23 benchmarks presented in Table 4. Four benchmarks PrintCalender, Number, Totinfo, and Mcknap, had been implemented in C, and therefore, could not be used in PIT, which is a java-based tool. The statistical analysis of the results is shown in Table 6. In this table, the significant level for the p-value is considered as p-value ≤ 0.05. The results show that, with

[1] This example is the same as the one presented in [26].

high statistical confidence, in 7 out of 19 programs, the generated test suites by our approach have a more mutation score, and thus, have a better ability to detect failures.

In some benchmarks, such as Remainder, there is no significant difference between mutation score achieved by the two approaches, while the improvement of mutation score on a program like Tcas is noticeable. Test data generation for more complicated programs such as Tcas with 12 input variables is likely more time-consuming. In these programs, fewer data from the input domain are desired, and therefore, using static information to generate test data enhance the mutation score of the generated test suites.

5- 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 experiments 50 times. Then, we followed 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.

6- Conclusions and Future Work

In this paper, we have presented an approach to input space partitioning based on the program's conditional statements. We also customized the ACO algorithm with respect to the partitioned space. In the evaluation section, we have compared our approach with the irrelevant input variable removal method. The results revealed that our approach leads to better results in respect of average coverage. The following research areas will be considered as future work:

- Customizing other meta-heuristic algorithms based on predicate's information
- Considering the combination of clauses to select better partition values
- Presenting a more comprehensive way to reduce the input domain so that it can be applied to all optimization algorithms
-

References

[1] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener, "Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation," IEEE Trans. Softw. Eng., vol. 38, no. 2, pp. 453–477, 2012, doi: 10.1109/TSE.2011.18.

[2] C. Mao, L. Xiao, X. Yu, and J. Chen, "Adapting ant colony optimization to generate test data for software structural testing $," Swarm Evol. Comput., vol. 20, pp. 23–36, 2014, doi: 10.1016/j.swevo.2014.10.003.

[3] S. Anand et al., "An Orchestrated Survey on Automated Software Test Case Generation," J. Syst. Softw., vol. 86, no. 8, pp. 1978–2001, 2013, doi: 10.1016/j.jss.2013.02.061.

[4] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Evolutionary Testing," IEEE Trans. Softw. Eng., vol. 36, no. 6, pp. 742–762, 2010, doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2009.52.

[5] P. Ammann and J. Offutt, Introduction to software testing. Cambridge University Press, 2016.

[6] M. Weiser, "Program Slicing," IEEE Trans. Softw. Eng., vol. SE-10, no. 4, pp. 352–357, 1984, doi: 10.1109/TSE.1984.5010248.

[7] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," Softw. Eng. J., vol. 11, pp. 299–306, 1996.

[8] R. Pargas, M. J. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms," J. Softw. Testing, Verif. Reliab., vol. 9, pp. 263–282, 1999.

[9] P. Mcminn, "Search-Based Software Testing : Past , Present and Future," pp. 153–163, 2011, doi: 10.1109/ICSTW.2011.100.

[10] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," IEEE Trans. Softw. Eng., vol. 36, no. 2, pp. 226–247, 2010, doi: 10.1109/TSE.2009.71.

[11] G. Fraser and A. Arcuri, "Whole test suite generation," IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, 2013, doi: 10.1109/TSE.2012.14.

[12] N. Tracey, "An Automated Framework for Structural Test-Data Generation 2 Optimisation-Based Struc- tural Test-Data Generation 1 Introduction," 1904.

[13] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE, vol. 2003-Janua, pp. 394–405, 2003, doi: 10.1109/ISSRE.2003.1251061.

[14] E. Elbeltagi, T. Hegazy, and D. Grierson, "Comparison among five evolutionary-based optimization algorithms," Adv. Eng. Informatics, vol. 19, no. 1, pp. 43–53, 2005, doi: 10.1016/j.aei.2005.01.004.

[15] K. Socha and M. Dorigo, "Ant colony optimization for continuous domains," Eur. J. Oper. Res., vol. 185, no. 3, pp. 1155–1173, 2008, doi: 10.1016/j.ejor.2006.06.046.

[16] C. Simons and J. Smith, "A comparison of evolutionary algorithms and ant colony optimization for interactive software design," 2012.

[17] B. Suri and P. Jajoria, "Using ant colony optimization in software development project scheduling," in 2013 International Conference on Advances in Computing,

Communications and Informatics (ICACCI), 2013, pp. 2101–2106.
[18] J. T. de Souza, C. L. B. Maia, T. do Nascimento Ferreira, R. A. F. Do Carmo, and M. M. A. Brasil, "An ant colony optimization approach to the software release planning with dependent requirements," in International symposium on search based software engineering, 2011, pp. 142–157.
[19] D. Azar and J. Vybihal, "An ant colony optimization algorithm to improve software quality prediction models: Case of class stability," Inf. Softw. Technol., vol. 53, no. 4, pp. 388–393, 2011.
[20] B. Suri, "Literature Survey of Ant Colony Optimization in Software Testing," 2010.
[21] H. Li, "An Ant Colony Optimization Approach to Test Sequence Generation for State-Based Software Testing," no. 1, pp. 255–262, 2005.
[22] P. R. Srivastava and K. Baby, "Automated Software Testing Using Metahurestic Technique Based on an Ant Colony Optimization," Electron. Syst. Des. (ISED), 2010 Int. Symp., 2010, doi: 10.1109/ISED.2010.52.
[23] A. Bouchachia, R. Mittermeir, P. Sielecky, S. Stafiej, and M. Zieminski, "Nature-inspired techniques for conformance testing of object-oriented software," Appl. Soft Comput. J., vol. 10, no. 3, pp. 730–745, 2010, doi: 10.1016/j.asoc.2009.09.003.
[24] K. Li, Z. Zhang, and W. Liu, "Automatic Test Data Generation Based on Ant Colony Optimization," 2009 Fifth Int. Conf. Nat. Comput., vol. 6, pp. 216–220, 2009, doi: 10.1109/ICNC.2009.239.
[25] H. Sharifipour, M. Shakeri, and H. Haghighi, "Structural test data generation using a memetic ant colony optimization based on evolution strategies," Swarm Evol. Comput., vol. 40, pp. 76–91, 2018, doi: 10.1016/j.swevo.2017.12.009.
[26] A. M. Bidgoli and H. Haghighi, "Augmenting ant colony optimization with adaptive random testing to cover prime paths," J. Syst. Softw., vol. 161, p. 110495, 2020.
[27] A. Monemi Bidgoli, H. Haghighi, T. Zohdi Nasab, and H. Sabouri, Using Swarm Intelligence to Generate Test Data for Covering Prime Paths, vol. 10522 LNCS. 2017.
[28] X. Lv, S. Huang, and H. Ji, "Input Domain Reduction of Search-based Structural Test Data Generation using Interval Arithmetic.," Int. J. Performability Eng., vol. 14, no. 6, 2018.
[29] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," Proc. 6th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng., pp. 155–164, 2007, doi: 10.1145/1287624.1287647.
[30] C. Cadar and M. Nowack, "KLEE symbolic execution engine in 2019."
[31] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," BioSystems, vol. 43, no. 2, pp. 73–81, 1997, doi: 10.1016/S0303-2647(97)01708-5.
[32] V. Maniezzo, "Ant System: Optimization by a Colony of Cooperating Agents," vol. 26, no. 1, pp. 1–13, 1996.
[33] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (Demo)," ISSTA 2016 - Proc. 25th Int. Symp. Softw. Test. Anal., pp. 449–452, 2016, doi: 10.1145/2931037.2948707.
[34] R Developement Core Team, "R: A Language and Environment for Statistical Computing," R Found. Stat. Comput., vol. 1, p. 409, 2015, doi: 10.1007/978-3-540-74686-7.

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.