# Code Coverage Optimisation in Genetic Algorithms and Particle Swarm Optimisation for Automatic Software Test Data Generation

Chahine Koleejan, Bing Xue, and Mengjie Zhang
School of Engineering and Computer Science, Victoria University of Wellington
PO Box 600, Wellington 6140, New Zealand
Email: {Soyaume.Koleejan, Bing.Xue, Mengjie.Zhang}@ecs.vuw.ac.nz

***Abstract*—Automatic software test data generation is the process of generating a set of test cases for a given program which can achieve a high code coverage. Genetic algorithms (GAs) and particle swarm optimisation (PSO) can automatically evolve a set of test data, but the traditional representation in GAs and PSO produces solutions with a single set of data cases, which may not achieve good performance on programs with many complex conditions. This paper proposes a multi-vector representation in GAs and PSO, which can generate multiple sets of data cases in a single run, to generate test data for complex test programs. Experiments have been conducted to examine and compare the performance of GAs and PSO on six commonly used benchmark test programs and three newly developed programs with a relatively large number of complex conditions. The experimental results show that the proposed multi-vector representation can improve the performance of GAs and PSO on all the nine tested programs, achieving the optimal 100% code coverage on the relatively easy programs. PSO outperforms GAs in terms of both the code coverage and the computational efficiency, especially on the hard programs.**

## I. Introduction

Software testing is an essential part in the software development process, which aims to ensure that the software produced meets both technical and business requirements. However, software testing is extremely expensive and laborious, and it has been estimated that about half of software development costs are allocated to testing [1]. Test data generation is the process of producing a set of data for testing software based on a given criterion, which is a complex problem. Though a lot of solutions have come forth, most of them are limited to relatively easy programs. Since doing this manually is time-consuming, error-prone and complex [2], automatic test generation can avoid such problems to improve the performance and reduce the time and cost [3].

There have been many different ways for automatically generating software test data. Heuristic search techniques for automatic generation of test data becomes a hot topic in recent years [4], which includes random search, hill climbing, and simulated annealing. However, they have their own advantages and limitations [4]. For example, random search is fast and simple but does not deal well with programs with many different structures. Hill climbing can deal with this problem and is also fast but often fails to find the optimal solution. Simulated annealing is computationally expensive and slow if a good solution is required [4]. Evolutionary computation techniques, such as genetic algorithms (GAs) and particle swarm optimisation (PSO), are powerful heuristic search methods, which have been widely used to solve problems in different areas [5], [6], [7], [8]. GAs and PSO have shown some success in automatic software test data generation [9], [10], [11], [12], [3], [13], but most of them were tested on programs with relatively easy structures. The potential of GAs and PSO for automatic test data generation has not been fully investigated.

978-1-4799-7492-4/15/$31.00 

There are a number of possible objectives that a test data set may seek to achieve [4], [14]. Functional testing is concerned with verifying specific actions or functions of the code and includes bug fixing. Structural testing is used to test the structure of the system in question and how the software executes its code. One of the most important structural properties is code coverage. Code coverage analysis reveals which areas of the code have not been tested and as such enables software developers to release products of higher quality [15]. A basic and fundamental code coverage criterion is statement coverage, which is a measure of how many statements in a program are executed. Traditionally, when using GAs or PSO for data generation to optimise code coverage, a candidate solution is represented by a single input vector of the arguments. However, when dealing with programs with conditions, using a single input vector means half the branches in the program are missed. Therefore, a new representation is needed to improve the performance of GAs and PSO for automatic test data generation.

### A. Goals

The overall goal of this paper is to improve the performance of GAs and PSO in generating test data to optimise the statement coverage in complex programs. To achieve this, a multi-vector representation is proposed for both GAs and PSO to generate multiple sets of test data cases in a single run. Since the current commonly used benchmark test programs are relatively simple, new test programs will be developed with more complex program structures to examine and compare the performance of GAs and PSO. Specifically, we will investigate

- whether GAs and PSO with the multi-vector representation can improve their performance over using the traditional single vector representation,
- whether PSO can achieve better statement coverage performance than GAs, and

- whether PSO can find good or optimal solutions faster than GAs.

### B. Organisation

The rest of this paper is organised as follows: section II provides background information of this paper. Section III describes the proposed multi-vector representation and the GAs and PSO algorithms. Section IV presents the existing test programs, the newly developed test programs, and the parameter settings of the GAs and PSO algorithms. Section V presents the experimental results and discussions. Section VI draws conclusions and discusses future work.

## II. Background

This section presents the background of this work, mainly about genetic algorithms, particle swarm optimisation, and typical work on automatic software test data generation.

### A. Genetic Algorithms

Genetic algorithms (GAs) [16] are evolutionary algorithms that model the Darwinian theory of evolution, which are possibly the first algorithmic models developed to simulate genetic systems [17].

In GAs, candidate solutions of the target problem are encoded as a population of chromosomes. A standard representation of each chromosome is a fixed-length array of bits (bit-strings). The population is evolved to search for the optimal solution by applying genetic operators, such as selection, crossover, and mutation. The evolutionary process usually starts from a population of randomly generated individuals/chromosomes, and is an iterative process, with the population in each iteration associated with a generation. In each generation, the goodness (i.e. fitness) of every individual in the population is evaluated by a fitness function, which is predefined according to the problem to be solved. The more fit individuals are stochastically selected from the current population, and new solutions are produced by using mainly two different genetic operators, i.e. crossover and mutation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been conducted, or a satisfactory fitness level has been reached for the population.

### B. Particle Swarm Optimisation

Particle swarm optimisation (PSO) [18], [19] is an optimisation method based on the behaviour of swarms of organisms such as flocks of birds. It works by initialising a population of particles randomly, each of which represents a candidate solution. Each particle has a position vector and a velocity vector. Each particle keeps track of its best position achieved so far called personal best, as well as the entire population's best achieved position so far called global best. At each iteration each particle's velocity and position are updated according to the following equations. This is repeated until the termination criterion is met.

$$v_{id}^{t+1} = w * v_{id}^{t} + c_1 * r_{i1} * (p_{id} - x_{id}^{t}) + c_2 * r_{i2} * (p_{gd} - x_{id}^{t}) \quad (1)$$

$$x_{id}^{t+1} = x_{id}^{t} + v_{id}^{t+1} \quad (2)$$

where $v_{id}^{t+1}$ shows the velocity of particle $i$ in the $d$th dimension in the $(t+1)$th iteration. $w$ is the inertia weight, which indicates the influence of the previous velocity. $c_1$ and $c_2$ are acceleration constants. $r_{i1}$, $r_{i2}$ and $rand()$ are random values, which are uniformly distributed in [0, 1]. $p_{id}$ and $p_{gd}$ shows the values of the personal best and the global best in the $d$th dimension. A predefined maximum velocity, $v_{max}$, is used to limit $v_{id}^{t+1}$ to $[-v_{max}, v_{max}]$.

### C. Related Work on Software Test Data Generation

Traditional methods, such as random search and gradient descent, were considered viable for the problem of generating test data [20]. Generating test data through random search is probably the simplest way, which can be used to generate input data for any type of program. However, it is hard to ensure a good quality of the generated data since the data is generated solely based on probability, it cannot accomplish high coverage as the chances of finding semantically small faults is quite low [20].

Evolutionary computation techniques have attracted much attention for automatic test data generation. Michael et al. [21] formed the task as a minimisation problem, and applied GAs to this task by optimising the code coverage. The performance of GAs was compared with that of random search and the results showed that GAs outperformed random search, but the test benchmark programs are quite simple. Pargas et al. [11] used GAs to solve test data generation problems, where the fitness function is to optimise the control-dependence graph of the program. Michael et al. [9] compared the GAs with random search and gradient search for test data generation on a set of benchmark programs, which showed that GAs outperformed both random search and gradient search. The results also suggested that the performance of random search deteriorated quickly as the increase of the program size.

The use of PSO for software test data generation is much later than that of GAs. In 2007, Windisch et al. [13] suggested that since PSO is simpler and has fewer parameters than GAs, it would be worth investigating its ability to generate software test data. They compared the performance of GAs and PSO on a set of simple test programs and the results showed that in most cases PSO was competitive with GAs. Jia et al. [3] proposed a new position initialising technique in PSO for software testing, which aimed to optimise the condition-decision coverage of benchmark programs. The results were compared to GAs and the standard PSO algorithm, which showed that the proposed PSO algorithm achieved better performance in terms of condition-decision coverage.

Besides GAs and PSO, other evolutionary computation techniques have also been used for automatic software test data generation [22], [23], [24]. Arcuri and Yao [22] proposed algorithms to automatically generate data for testing Java containers, where five different search algorithms were used: random search as a natural baseline, hill climbing as a local search method, simulated annealing and GAs as global search methods, and memetic algorithms combining together local and global searches. The experimental results show that the five

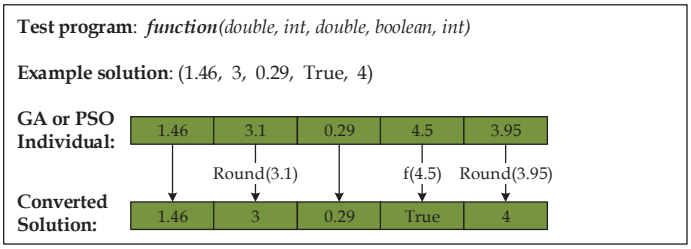


Fig. 1: Single Vector Representation. An example of a test program with five input variables.

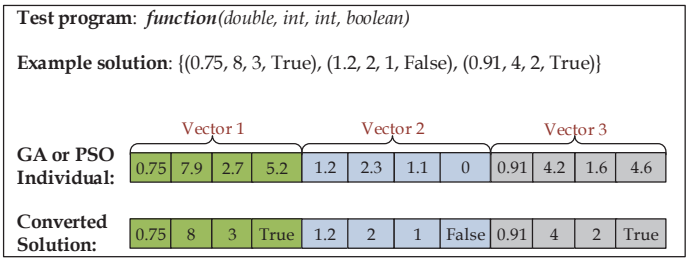


Fig. 2: Multi-Vector Representation. An example of an individual with three vectors for a test program with four input variables.

search algorithms achieved better performance than traditional techniques. Genetic programing (GP) has also been applied to automatic software testing [24], [25] due to its flexible tree-based representation.

Test data generation is a highly complex task particularly with large programs and complex program structures. Most of the work done in this area is based on programs that either are very short in length or very low in complexity. Building new test programs to examine the performance of the test data generation algorithms is also an open issue.

## III. The Approach

This section proposes a multi-vector representation in GAs and PSO for automatic software test data generation. The representation, the fitness function, and the pseudo-code of the algorithms are described here.

### A. Representation

Traditionally, in the context of optimising code coverage, a candidate solution is a single input vector of the input arguments [13], which is called single vector representation here. For example, if a function has three integer arguments/variables, a possible solution would be (1,2,3). The code coverage would be obtained by running the program in question with arguments (1,2,3). However, when dealing with programs with conditional structures, using a single input vector means half the branches in the program are missed. Multiple vectors are needed in order to obtain optimal code coverage. Therefore, a multi-vector representation is proposed here to encode a chromosome in GAs or a particle's position in PSO for optimising the code coverage in software test data generation tasks.

*1) Single Vector Representation:* In this representation, the length/dimensionality of a chromosome in GAs or a particle in PSO equals to the number of input arguments/variables that the test program has. A chromosome or a particle's position represents a vector as a solution of the test program, where each bit corresponds to one argument/variable.

Fig. 1 takes a test program with five input arguments as an example and shows how an individual in GAs or PSO is converted to a solution vector. To convert a GA and PSO individual to input arguments, one of the following decoding processes is used:

- If the input argument type is float or double, we can simply use the corresponding entry value in the chromosome or position;
- If the input argument type is integer, the corresponding entry value is rounded to the nearest integer;
- If it is a boolean, use the logistic function, i.e. Equation (3) to convert the corresponding entry value to 0 or 1 (False or True).

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}} \tag{3}$$

where $x$ is the value of an element in the chromosome or the particle's position, $x_0 = 0$, $L = 1$, and $k = 1$.

*2) Multi-Vector Representation:* To avoid/reduce the limitation of the traditional single vector representation, the proposed multi-vector representation includes a set of vectors in each chromosome or a particle. The length/dimensionality of GAs or PSO equals to the number of vectors multiplying the number of arguments/variables in the test program. When the number of vectors is one, the multi-vector representation becomes the same as the single vector representation. By using multiple vectors in a single solution, it is expected to avoid the situation in the single vector representation, where half the branches in the program are missed when dealing with programs containing conditional structures.

Fig. 2 shows an example of the multi-vector representation, where the test program has four input variables and each individual in GAs or PSO includes three vectors. The decoding processes described in the single vector representation are still applied here to convert an individual to a desired solution.

### B. Fitness Function

There are many commonly used code coverage performance criteria [26], such as statement coverage, condition coverage, multiple condition coverage, path coverage, branch coverage, and condition-decision coverage. In this paper, we mainly use the basic and fundamental statement coverage as the code coverage criterion.

The fitness function here is to maximise the statement coverage rate, which is calculated by converting an individual in GAs or PSO to a solution and then running the test program with the solution. Of course, the best (maximum) fitness value is the 100%, which is achieved if every line of the program is executed.

**Algorithm 1** Pseudo-code of the PSO-based Algorithm

randomly initialise particles in the swarm;
convert each particle's position to a solution with multiple input vectors;
evaluate fitness value of each particle */*the code coverage of the converted solution*/*;
**while** termination condition is not reached **do**
  evaluate fitness value of particle $i$;
  **for** each particle $i$ **do**
    update $pbest$ of $i$;
    update $gbest$ of $i$;
  **end for**
  **for** each particle $i$ **do**
    **for** each dimension $d$ **do**
      update $v_{id}$ according to Equation (1) ;
      update $x_{id}$ according to Equation (2);
    **end for**
  **end for**
**end while**
return the $gbest$ and the code coverage rate.

**Algorithm 2** Pseudo-code of the GA-based Algorithm

initialise the population;
convert each chromosome to a solution with multiple input vectors;
evaluate fitness value of chromosome */*the code coverage of the converted solution*/*;
**while** termination condition is not reached **do**
  evaluate fitness value of each chromosome;
  perform selection operator;
  perform crossover operator;
  perform mutation operator;
**end while**
return the best chromosome and the code coverage rate.

### C. Pseudo-code

The pseudo-codes of the PSO and GAs with the proposed multi-vector representation are shown in Algorithms 1 and 2, respectively. Both GAs and PSO are population-based search techniques, but they follow different mechanisms. PSO employs velocity and position updating equations to gradually increase or decrease the values in each dimension of candidate solutions while GAs use genetic operators to create new solutions by updating a block of dimensions in candidate solutions. So, when the input arguments have more numerical types (float, double or integer), PSO may achieve better performance than GAs, but when there are more nominal arguments, GAs may achieve better performance than PSO.

## IV. Experiment Design

To examine the performance of the GAs and PSO algorithms, different numbers of vectors are used in the multi-vector representation, which are 2, 3, 4, 5, and 6. Therefore, five different GAs and five different PSO algorithms are developed, which are GA-2, GA-3, GA-4, GA-5 and GA-6, and PSO-2, PSO-3, PSO-4, PSO-5 and PSO-6 (e.g. PSO-3 means PSO with 3 vectors in the representation). GAs and PSO with the single vector representation are used as baseline algorithms in the experiments, which are denoted as GA-1 and PSO-1. They will be examined and compared with each other on a number of benchmark test programs in the experiments.

TABLE I: Benchmark Test Programs

| Program | Parameters | Lines | Conditionals | Loops | Recursive |
|---|---|---|---|---|---|
| TC | 3 | 41 | 8 | 0 | No |
| MaxMin | 3 | 47 | 5 | 0 | No |
| BubbleSort | 5 | 33 | 1 | 2 | No |
| MergeSort | 5 | 59 | 2 | 4 | Yes |
| GCD | 2 | 24 | 1 | 0 | Yes |
| Month | 1 | 60 | 12 | 0 | No |
| $f_1$ | 2 | 65 | 14 | 0 | No |
| $f_2$ | 3 | 71 | 17 | 0 | No |
| $f_3$ | 4 | 113 | 32 | 0 | No |

```
Input: int a, int b, int c
  if a < b + c then
    if b < a + c then
      if c < a + b then
        if a! = b then
          if a! = b then
            if a! = b then
              return 2
        if a = b then
          if b = c then
            return 0
          else
            return 1
  return -1
```

Fig. 3: Pseudo-code of TriangleClassification.

### A. Benchmark Test Programs

Six test programs which are commonly used in recent papers are used as benchmark programs in the experiments. They are TriangleClassification [27], [3], [28], [9], [13], MaxMin [27], BubbleSort [27], [9], [29], MergeSort [29], GCD [9], and Month. They are described in detail below.

The complexity of a test program in the context of code coverage is primarily determined by how many conditional statements it contains. Our initial experiments show that these six programs are relatively easy to solve. We further design three programs, which contain a large number of conditions. Since specific conditions are difficult to satisfy, the three programs are harder to achieve good coverage rate on. The three programs are called $f_1$, $f_2$, and $f_3$.

The detailed information of the nine benchmark programs are summarised in Table I. These nine benchmark programs differ in terms of the length, the number of input arguments, the number of conditional statements, whether containing loops, and whether containing recursive structures.

*1) TriangleClassification (TC):* This is a commonly used test program which takes the lengths of the three sides in a triangle as input and determines whether it is scalene, isosceles, equilateral or not a triangle. The pseudo-code is given in Fig. 3.

*2) MaxMin (MM):* This is a program which takes three numbers and returns the largest and smallest numbers. Since this program is very simple, the pseudo-code is not given here, and so as BubbleSort, Recursive MergeSort, Greatest Common Divisor, and Month.

*3) BubbleSort (BS):* This is the standard version of the BubbleSort sorting algorithm.

*4) Recursive MergeSort (RM):* This is the recursive version of the MergeSort sorting algorithm.

```
Input: int a, int b
  if a + b > 100 then
    if a < 50 then
      if b = 5 then
        print 1
      else if b = 80 then
        print 2
      else
        print 3
    else if a > 75 then
      if b = 10 then
        print 4
      else if b = 60 then
        print 5
      else
        print 6
    else
      if b = 55 then
        print 7
      else if b = 45 then
        print 8
      else
        print 9
  else
    if b > 50 then
      if a = 1 then
        print 10
      else
        print 11
    else if b < 25 then
      if a = 15 then
        print 12
      else
        print 13
    else
      if a = 90 then
        print 14
      else
        print 15
```

(a) Pseudo-code of $f_1$.

```
Input: int a, int b, int c
  if a + b + c > 200 then
    if a + b < 10 then
      if c = 100 then
        print 1
      else if c = 80 then
        print 2
      else
        print 3
    else if a + b > 150 then
      if c = 30 then
        print 4
      else if c = 10 then
        print 5
      else
        print 6
    else
      if c = 20 then
        print 7
      else if c = 1 then
        print 8
      else
        print 9
  else
    if c + b > 100 then
      if a = 1 then
        print 10
      else if a = 11 then
        print 11
      else
        print 12
    else if b + c < 50 then
      if a = 15 then
        print 13
      else if a = 95 then
        print 14
      else
        print 15
    else
      if a = 90 then
        print 16
      else if a = 40 then
        print 17
      else
        print 18
```

(b) Pseudo-code of $f_2$.

Fig. 4: Pseudo-code of $f_1$ and $f_2$.

*5) Greatest Common Divisor (GCD):* This is a program that returns the greatest integer that divides two given numbers.

*6) Month (M):* This is a program that returns the name of the month (e.g. "February") by giving the corresponding nominal number (e.g. 2). Let $m$ be a variable, *if m=1, return "January"; if m=2, return "February"; ... if m=12, return "December";* . Note that although the 12 possible numbers are integer/ordinal numbers, there is not direct relationship between any two numbers, they are actually nominal argument.

*7) Three New Test Programs:* $f_1$, $f_2$, and $f_3$. Since the other programs are relatively simple, three more complicated programs $f_1$,$f_2$ and $f_3$ were created for this research. In terms of complexity, they are more representative of the kind of programs used in practice. They take 2, 3 and 4 arguments, respectively. They are of increasing the program length and the number of conditions to be much larger than those of the simpler programs. Moreover, a large number of these conditions are hard to satisfy because multiple variables need to be optimised concurrently in order to do so. The pseudo-codes of $f_1$ and $f_2$ are given in Fig. 4, and the pseudo-code of $f_3$ is given in Fig. 5.

### B. Parameter Settings

Both the GAs and PSO algorithms have been conducted for 50 independent runs on each test benchmark program. The GAs and PSO settings that are commonly used in literature in recent years were used in the experiments. Both the GAs and PSO share the same termination condition. The algorithm stops

```
Input: int a, int b, int c, int d
  if a + b > 100 then
    if c < 50 then
      if d = 5 then
        print 1
      else if d = 80 then
        print 2
      else
        print 3
    else if c > 75 then
      if d = 10 then
        print 4
      else if d = 60 then
        print 5
      else
        print 6
    else
      if b = 55 then
        print 7
      else if b = 45 then
        print 8
      else
        print 9
  else
    if c > 50 then
      if d = 1 then
        print 10
      else
        print 11
    else if c < 25 then
      if d = 15 then
        print 12
      else
        print 13
    else
      if d = 90 then
        print 14
      else
        print 15
```

```
if c − d > 100 then
  if a < 50 then
    if b = 5 then
      print 16
    else if b = 80 then
      print 17
    else
      print 18
  else if a > 75 then
    if b = 10 then
      print 19
    else if b = 60 then
      print 20
    else
      print 21
  else
    if b = 55 then
      print 22
    else if b = 45 then
      print 23
    else
      print 24
else
  if b > 50 then
    if a = 1 then
      print 25
    else
      print 26
  else if b < 25 then
    if a = 15 then
      print 27
    else
      print 28
  else
    if a = 90 then
      print 29
    else
      print 30
```

Fig. 5: Pseudo-code of $f_3$.

TABLE II: Parameter Settings in GAs

| Parameter | value |
|---|---|
| Population Size | 100 |
| Termination | Global optimum found or 60 generations |
| Crossover type | Two-point |
| Mutation type | Reset |
| Mutation probability | 0.05 |
| Elitism rate | 0.05 |

either when the maximum number of iterations or generations have been reached or when the algorithm have found the best solution.

To show the difference in the obtained code coverage rates and the computational time used between the GAs and PSO, a non-parametric statistical significance test, Wilcoxon test, is used in the experiments. The significance level is chosen to be 0.05 (or equivalently, 5%).

*1) GAs Settings:* The settings in GAs are shown in Table II. The breeding pipeline consists of two individuals selected by tournament selection of size 4 being crossed-over and the resultant invididual mutated with probability 0.05.

*2) PSO Settings:* The parameters used in PSO are summarised in Table III, which follow the common settings suggested in [30].

## V. Results and Discussions

The results of the statement coverage obtained by the GAs and PSO algorithms are shown in Tables IV, V and VI and Fig.6. The computational cost are presented in Fig. 7 and Table VII.

TABLE III: Parameter Settings in PSO

| Parameter | Value |
|---|---|
| No. of particles | 30 |
| Inertia weight | 0.7298 |
| Acceleration coefficients | 1.49618 |
| Termination | Global optimum found or 200 iterations |

TABLE IV: Statement coverage(%) achieved by each GA algorithm on the test programs

| | GA-1 vector Mean±Std | GA-2 Mean±Std | GA-3 Mean±Std | GA-4 Mean±Std | GA-5 Mean±Std | GA-6 Mean±Std |
|---|---|---|---|---|---|---|
| TC | 83.0±0.0 | 91.8±0.8 | 96.0±0.0 | 100.0±0.0 | 100.0±0.0 | 99.9±0.6 |
| MaxMin | 60.0±0.0 | 77.0±0.0 | 87±0.0 | 93.0±0.0 | 97.0±0.0 | 100.0±0.0 |
| BubbleSort | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| MergeSort | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| GCD | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| Month | 36.0±0.0 | 42.0±0.0 | 47.0±0.0 | 53.0±0.0 | 58.0±0.0 | 64.0±0.0 |
| $f_1$ | 35.0±0.0 | 44.0±0.0 | 52.0±0.0 | 58.5±0.5 | 64.5±0.6 | 68.6±0.5 |
| $f_2$ | 33.0±0.0 | 41.8±0.4 | 49.0±0.3 | 54.7±0.5 | 60.0±0.8 | 63.1±0.9 |
| $f_3$ | 28.0±0.0 | 36.0±0.0 | 41.7±0.5 | 47.2±1.1 | 51.3±1.2 | 55.0±1.8 |

TABLE V: Statement coverage(%) achieved by each PSO algorithm on the test programs

| | PSO-1 vector Mean±Std | PSO-2 Mean±Std | PSO-3 Mean±Std | PSO-4 Mean±Std | PSO-5 Mean±Std | PSO-6 Mean±Std |
|---|---|---|---|---|---|---|
| TC | 83.0±0.0 | 92.0±0.0 | 96.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| MaxMin | 60.0±0.0 | 77.0±0.0 | 87.0±0.0 | 93.0±0.0 | 97.0±0.0 | 100.0±0.0 |
| BubbleSort | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| MergeSort | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| GCD | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 | 100.0±0.0 |
| Month | 36.0±0.0 | 42.0±0.0 | 47.0±0.0 | 53.0±0.0 | 58.0±0.0 | 63.8±0.8 |
| $f_1$ | 35.0±0.0 | 44.0±0.0 | 52.0±0.0 | 58.9±0.3 | 64.9±0.3 | 68.9±0.3 |
| $f_2$ | 33.0±0.0 | 42.0±0.0 | 49.0±0.0 | 55.0±0.0 | 60.5±0.5 | 63.6±0.5 |
| $f_3$ | 28.0±0.0 | 36.0±0.0 | 42.0±0.2 | 48.5±0.6 | 53.0±0.8 | 56.5±1.1 |

### A. *Code Coverage*

Tables IV and V show the statement coverage achieved by each algorithm on each benchmark test program, and Table VI shows the results of statistical significance tests between the PSO and GA algorithms, where $\approx$ means there is no statistical significant difference between PSO and the GA algorithm. Fig. 6 compares the performance of using different numbers of vectors in the multi-vector representation in GAs and PSO.

From Table IV, it can be seen that on the simple test programs, such as BubbleSort, MergeSort, and GCD, GAs with the traditional single vector representation are able to generate data with 100% statement coverage. On the four programs with more conditions, i.e. Month, $f_1$, $f_2$, and $f_3$, the performance of the GA with single vector representation is poor, with less than 40% of the code coverage. By using the proposed multi-vector representation, the performance of GAs can reach 100% on four or five of the six commonly used test programs. The performance on the four hards programs are also substantially increased. The results also suggest that the loop and recursive structures in the BubbleSort and GCD programs do not cause much difficulties for GAs, but the increase in the number of conditional structures brings difficulties.

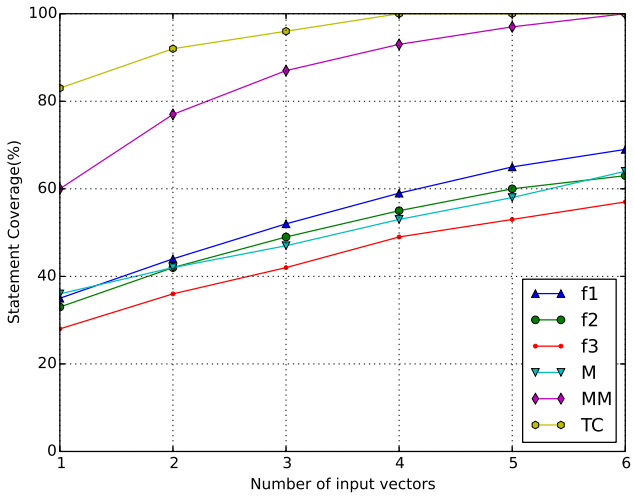


Fig. 6: Statement Coverage(%) vs No. of Input vectors.

According to Table V, PSO with the single vector representation can address simple programs like BubbleSort, MergeSort, and GCD to achieve a coverage of 100%, but not on the four hard programs, i.e. Month, $f_1$, $f_2$, and $f_3$. By using the proposed multi-vector representation, PSO are able to achieve 100% statement coverage on five of the six commonly used benchmark test programs. On the four hard programs, PSO-6 improves the coverage performance to 63.8%, 68.9%, 63.6%, and 56.5% from 36%, 35%, 33%, and 28% in PSO with single vector representation, respectively.

Fig. 6 shows the performance of GAs and PSO with different number of vectors in the representation on the six benchmark programs, i.e. $f_1$, $f_2$, $f_3$, Month (M), MaxMin (MM), and TriangleClassification (TC). BubbleSort, MergeSort, and GCD are not included since both GAs and PSO can achieve 100% coverage with a single vector representation. The pattern in Fig. 6 is very clear that using more vectors in the representation can improve the statement coverage on all these six programs in both GAs and PSO. This is because more vectors being used means more conditions are able to be satisfied. The programs with a relatively large number of conditions require a considerable number of vectors to obtain a good statement coverage. Conversely, the test programs which have fewer conditions (BubbleSort, MergeSort and GCD) can be easily optimised with only one vector. A perfect statement coverage was unable to be achieved on the programs with many conditional structures (Month, $f_1$, $f_2$ ,$f_3$), even using six input vectors. It can be generalised that the more conditions a program has, the more input vectors are required to achieve a high coverage rate.

According to Table VI, when using the traditional single vector representation, there is no statistical significant difference between GAs and PSO, i.e. both of them obtain perfect results for the three easy programs (BubbleSort, MergeSort, and GCD), reasonably good solutions on two programs (TC and MM), and equally very poor results on the four hard programs (Month, $f_1$, $f_2$, and $f_3$). By using the multi-vector representation, there are still no statistical significant difference

TABLE VI: Which algorithm is better according to the statistical significance test

| | 1 vector | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| TC | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ |
| MaxMin | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ |
| BubbleSort | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ |
| MergeSort | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ |
| GCD | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ |
| Month | ≈ | ≈ | ≈ | ≈ | ≈ | GA |
| $f_1$ | ≈ | ≈ | ≈ | PSO | PSO | PSO |
| $f_2$ | ≈ | PSO | PSO | PSO | PSO | PSO |
| $f_3$ | ≈ | ≈ | PSO | PSO | PSO | PSO |

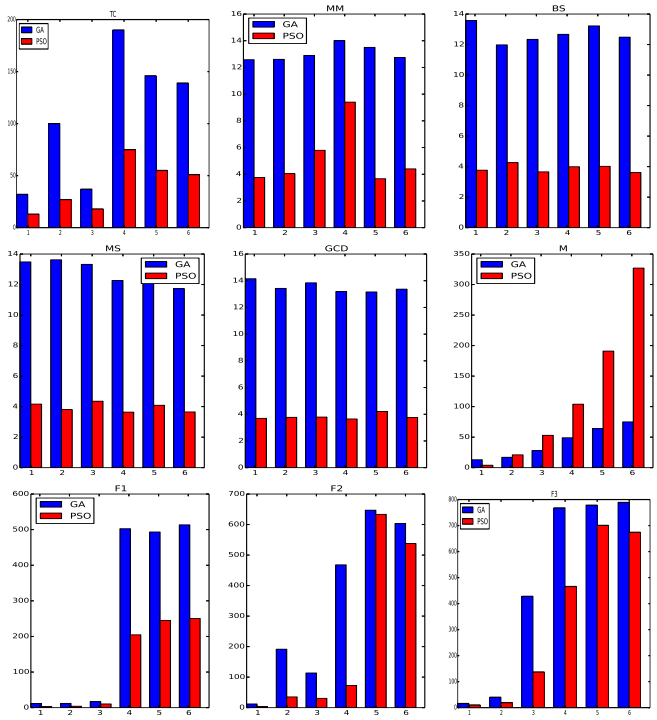


Fig. 7: Time(s) vs No. of Input vectors.

between GAs and PSO on the five relatively simple programs, since both of them can obtain the optimal solution of 100% statement coverage. However, for the four hard programs, twelve cases, PSO achieves significantly better performance than GAs in twelve cases. GAs only achieve significantly better performance than PSO in one case, using 6 vectors on the Month program, where the 12 possible numbers are actually nominal numbers, where GAs is expected to achieve better performance than PSO. The results show that PSO can better utilise the multi-vector representation than GAs to obtain a set of vectors (test data) to cover the code in the hard programs. The main reason is that the search mechanism in PSO is more suitable than GAs for generating numerical data in the tested programs.

### B. Computational Cost

Fig. 7 shows the computational time used by GAs and PSO with different numbers of input vectors on the nine benchmark programs, where the horizontal axis shows the number of input vectors used in the multi-vector representation while the vertical axis shows how many seconds that GAs or PSO took on average for a single run. Table VII shows the results of significance test between the time used by the GAs and PSO algorithms.

TABLE VII: Which algorithm is faster according to the statistical significance test

| | 1 vector | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| TC | PSO | PSO | PSO | PSO | PSO | PSO |
| MaxMin | PSO | PSO | PSO | PSO | PSO | PSO |
| BubbleSort | PSO | PSO | PSO | PSO | PSO | PSO |
| MergeSort | PSO | PSO | PSO | PSO | PSO | PSO |
| GCD | PSO | PSO | PSO | PSO | PSO | PSO |
| Month | PSO | - | GA | GA | GA | GA |
| $f_1$ | PSO | PSO | PSO | PSO | PSO | PSO |
| $f_2$ | PSO | PSO | PSO | PSO | - | - |
| $f_3$ | PSO | PSO | PSO | PSO | PSO | PSO |

In general, the algorithms take longer to run on programs with more conditions and complex conditions. The programs that take the longest time were the most complicated $f_1$, $f_2$ and $f_3$ programs, which contain much more conditions than other programs and some of the conditions are hard to satisfy. On the hard programs, increasing the number of vectors increases the average computational time, as the global optimum or optima become harder to find. In contrast, on the simple programs, increasing the number of vectors does not significantly increase the computational cost, but may even reduce the computational time. The reason is that the algorithms with more vectors can easily find the optimal solution to achieve 100% code coverage and then stop the search process.

When comparing the efficiency of GAs with PSO, PSO is clearly faster than the GAs regardless the number of input vectors, which can be found in Table VII. PSO is statistically significantly faster than GAs in 47 cases and is slower in only 4, all of which are on the same program, Month, since the Month program involves nominal data which are more suitable for GAs than PSO. In two cases, there is no statistical significant difference between the time taken by the GAs and PSO algorithms. PSO is by far the superior algorithm on eight out of the nine test programs, including the three hardest programs, $f_1$, $f_2$ and $f_3$.

## VI. Conclusions and Future Work

This paper developed a multi-vector representation in GAs and PSO for automatic software test data generation with the expectation of increasing the code coverage performance on complex programs. Since the recent commonly used benchmark test programs are relatively simple, we also developed three hard test programs with a relatively large number and hard conditions. Experiments on nine test programs (including six commonly used ones and three newly developed ones) show that with the proposed multi-vector representation, both GAs and PSO significantly increased the code coverage performance over the traditional single vector representation, and automatically generated software test data to achieve the optimal, 100%, code coverage on five out of the six simple

programs. On the four hard programs, using more vectors substantially improved the code coverage performance in both the GAs and PSO algorithms. PSO and GAs achieved similar performance on the six simple programs, but on the hard programs and using more vectors, PSO achieved significantly better performance than GAs. In terms of the efficiency, using more vectors increased the cost on hard programs, but not always on the simple programs since the algorithms can easily find the optimal solution and stop searching. PSO generally outperformed GAs in terms of both the code coverage and the computational cost.

The experimental results illustrated the effectiveness and efficiency of the proposed algorithms, but unless a theoretical analysis is done, it is hard to determine the optimal number of vectors prior. Therefore, in future, we will further investigate the relationship between the number of conditions and the number of vectors, and develop novel approaches which are capable of generating test sets of variable sizes. We will also compare the proposed algorithms with existing GAs, PSO and other algorithms in the literature.

Automatic software testing tools are still far from ideal for real-world software. In the future, developing more hard benchmark test programs and further testing on industrial-size programs may give more insight into the differences between algorithms and how the properties of the test programs affect their performances. The criterion of statement coverage was used for the purpose of this investigation. Other important criteria, such as branch and path coverage, could be used to compare the algorithms. Furthermore, GP, which has a flexible representation scheme, can handle different types of input variables and the tree-based representation is more directly comparable and suitable for being mapped instantly to abstract syntax trees commonly used in computer languages and compilers [24], [25]. However, GP is often computationally more expensive than GAs and PSO, so how to utilise the flexible representation in GP and improving the efficiency in automatic software test generation is also an interesting topic.

## REFERENCES


[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[2] J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," in *Artificial Neural Nets and Genetic Algorithms*. Springer, 1998, pp. 325–328.

[3] Y.-H. Jia, W.-N. Chen, J. Zhang, and J.-J. Li, "Generating software test data by particle swarm optimization," in *Simulated Evolution and Learning*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8886, pp. 37–47.

[4] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[5] B. Xue, M. Zhang, and W. N. Browne, "Particle swarm optimization for feature selection in classification: A multi-objective approach," *IEEE Transactions on Cybernetics*, vol. 43, no. 6, pp. 1656–1671, 2013.

[6] A. P. Engelbrecht, *Computational intelligence: an introduction (2. ed.)*. Wiley, 2007.

[7] L. Cervante, B. Xue, L. Shang, and M. Zhang, "A dimension reduction approach to classification based on particle swarm optimisation and rough set theory," in *25nd Australasian Joint Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science, vol. 7691. Springer, 2012, pp. 313–325.

[8] B. Xue, M. Zhang, and W. Browne, "Novel initialisation and updating mechanisms in PSO for feature selection in classification," in *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7835, pp. 428–438.

[9] C. Michael, G. McGraw, and M. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, 2001.

[10] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, and A. Watkins, "Breeding software test cases with genetic algorithms," in *36th Annual Hawaii International Conference on System Sciences*, 2003, pp. 1–10.

[11] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[12] H.-H. Sthamer, "The automatic generation of software test data using genetic algorithms," Ph.D. dissertation, University of Glamorgan, 1995.

[13] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07, 2007, pp. 1121–1128.

[14] P. McMinn, "Search-based software testing: Past, present and future," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2011, pp. 153–163.

[15] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 291–301.

[16] J. H. Holland, *Adaptation in natural and artificial systems*. MIT Press, 1992.

[17] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[18] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948.

[19] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *IEEE International Conference on Evolutionary Computation (CEC'98)*, 1998, pp. 69–73.

[20] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 3, pp. 195–200, 1996.

[21] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *12th IEEE International Conference Automated Software Engineering*. IEEE, 1997, pp. 307–308.

[22] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008.

[23] C. Mao, X. Yu, J. Chen, and J. Chen, "Generating test data for structural testing based on ant colony optimization," in *12th International Conference on Quality Software (QSIC)*, 2012, pp. 98–101.

[24] A. S. H.-G. Gross and A. Seesing, "A genetic programming approach to automated test generation for object-oriented software," *Science*, 2006.

[25] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, 2014.

[26] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[27] A. Panchapakesan, R. Abielmona, and E. Petriu, "Dynamic white-box software testing using a recursive hybrid evolutionary strategy/genetic algorithm," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, 2013, pp. 2525–2532.

[28] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[29] H. Tahbildar and B. Kalita, "Heuristic approach of automated test data generation for program having array of different dimensions and loops with variable number of iteration," *arXiv preprint arXiv:1011.0594*, 2010.

[30] M. Clerc and J. Kennedy, "The particle swarm– explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.