Generating Software Test Data by Particle Swarm Optimization

Ya-Hui Jia[1,3,4], Wei-Neng Chen[2,3,4,**], Jun Zhang[1,2,3,4], and Jing-Jing Li[5]

[1]School of Information Science and Technology, Sun Yat-sen University, Guangzhou, China
[2]School of Advanced Computing, Sun Yat-sen University, Guangzhou, China
[3]Key Lab. Machine Intelligence and Advanced Computing, Ministry of Education, China
[4]Engineering Research Center of Supercomputing Engineering Software, MOE, China
[5]School of Computer Science, South China Normal University, Guangzhou, China
chenwn3@mail.sysu.edu.cn

Abstract. Search-based method using meta-heuristic algorithms is a hot topic in automatic test data generation. In this paper, we develop an automatic test data generating tool named particle swarm optimization data generation tool (PSODGT). The PSODGT is characterized by the following two features. First, the PSODGT adopts the condition-decision coverage (C/DC) as the criterion of software testing, aiming to build an efficient test data set that covers all conditions. Second, the PSODGT uses a particle swarm optimization (PSO) approach to generate test data set. In addition, a new position initialization technique is developed for PSO. Instead of initializing the test data randomly, the proposed technique uses the previously-found test data that can reach the target condition as the initial positions so that the search speed of PSODGT can be further accelerated. The PSODGT is tested on four practical programs. Experimental results show that the proposed PSO approach is promising.

Keywords: Particle swarm optimization, Automatic software test case generation, Software testing,· Code coverage.

1 Introduction

With the rapid development of software industry, software is becoming bigger and more subtle. In 2002, NIST estimated the loss caused by software failure which reached 0.6 percent of GDP in America [1]. Hence, software testing, as a necessary part during the circle of software development, is more difficult than before. Software testing is also an expensive and labor-intensive work, which sometimes occupies about half of the total workload [2] and brings lots of redundant expenditure both in time and money. Hence, developing automatic test tool has important practical significance.

The basic prerequisite for automatic software testing is generating test data automatically. However, test data generation is a very challenging task, as a good data set should not only fulfill all the requirements defined by test criterion well but also be as

* Corresponding author.

G. Dick et al. (Eds.): SEAL 2014, LNCS 8886, pp. 37–47, 2014.

smaller as possible. As a result, more and more research effort has been attracted in software test data generation in recent years [7], [16]. In general, these studies can be classified into three classes, random, symbolic and dynamic. Random method just generates inputs at random until a useful input is found. In symbolic method, variables are assigned with symbolic values so that test data generation can be turned into a problem of solving algebraic expressions [13], [14]. In dynamic test generation, the source code is instrumented to collect information about the program when it executes. This information can help test generators to modify the program's input to satisfy the requirement heuristically. Then the problem of generating test data converted to function minimization problem. As the dynamic method is efficient and robust for different kinds of programing codes, it has been increasingly considered as a promising software test data generation technique in recent years [17]. The dynamic method is also known as search-based software testing thus several meta-heuristic optimization algorithms have been proposed for this problem, e.g. hill climbing [15], tabu search algorithm [3], genetic algorithm (GA) [5] and particle swarm optimization (PSO) [8].

Meanwhile most existing researches focused on covering paths in a program as many as possible [6]. This strategy is known as path coverage which is a coverage criterion in the field of white-box testing requiring that every path in the target program should be reached. But sometimes it is not enough just covering all paths in a program. Path coverage may also cause some conditions in the target program cannot be fully covered. In order to overcome this problem, there is also another coverage criterion called condition-decision coverage (C/DC). C/DC requires that every condition and every decision should take all possible outcomes at least once. Michael et al. [4] used C/DC as criterion in their test data generation tool GADGET but the approaches they proposed are based on GA. Though GA has strong ability in global searching, the local search capability is not good enough. Hence the convergence speed of GA often cannot satisfy the software testing requirement.

In this paper, we intend to introduce a PSO approach to search-based software testing with the C/DC criterion and further develop a PSO Data Generation Tool (PSODGT). The reason of using PSO is that PSO has a fast convergence speed [10], [18]. In addition, the self-cognitive and social-influence learning strategies of PSO make it more reliable in detecting conditions which are difficult to reach. Though PSO has been used in test data generation with the path coverage criterion in a few works like [6], [9], different from these existing approaches, our proposed PSO approach focuses on a different criterion, i.e., the C/DC criterion. In addition, we further improve the performance of PSO for test data generation by introducing a new initialization technique to PSO and adjust its parameter setting. During each optimization procedure, particles should reach the target condition before optimizing the fitness function. In the proposed initialization technique, particles are initialized according to the test data that can reach the current target condition found in previously. This modification saves time for particles to reach the target condition so that particles can early start to optimize fitness function. As for experiment, most researches just tested their approaches by simple programs like triangle classification, bubble sort. These programs are not complicated enough to simulate real situations because they are too simple and the search space is small. In this paper, four programs with different complexities of inputs and conditions are tested. Our PSO approach is compared with a

GA approach [4] which is also proposed for C/DC. Experimental results are evaluated in two aspects, conditions coverage rate and convergence rate. When observing the convergence rate, the number of executions of the target program is used as measurement instead of time consumption.

The rest of this paper is organized as follows. In section 2, we introduce the PSO algorithm and the test adequacy criteria. Section 3 shows some pivotal details about PSODGT. Then experimental results and analysis are shown in section 4. Finally in section 5 the conclusions are drawn.

2 Particle Swarm Optimization and Test Adequacy Criteria

2.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) algorithm was proposed by Russell Eberhart and James Kennedy in 1995 [10]. In PSO algorithm, each particle keeps track of a position which is the best solution it has achieved so far as *pbx* and globally optimal solution is stored as *gbest*. The basic steps of PSO are as follow:

1. Initialize N particles with random positions px_i and velocities v_i on D dimensions. Evaluate every particle's current fitness $f(px_i)$. Initialize $pbx_i = px_i$ and $gbest = i, f(px_i) = min(f(pbx_0), f(pbx_1), ..., f(pbx_N))$;
2. Check whether the criterion is met. If the criterion is met, loop ends else continue;
3. Change velocities according to formula (1):

$$v_i = \omega v_i + c_1 r_1 (pbx_i - px_i) + c_2 r_2 (pbx_{gbest} - px_i); \tag{1}$$

4. Change positions according to formula (2):

$$px_i = px_i + v_i \tag{2}$$

5. Evaluate every particle's fitness $f(px_i)$; if $f(px_i) < f(pbx_i)$ then $pbx_i = px_i$;
6. Update *gbest* and loop to step 2.

Usually particle's position cannot overstep the boundary of the search space and velocity also cannot exceed one particular value which is often set as 20% of the search space's width. In formula (1), the particle velocity updating formula, ω presents inertia factor, generally obtained by formula (3) [18]

$$\omega = \omega_{max} - \frac{(\omega_{max} - \omega_{min})}{maxIt} k . \tag{3}$$

maxIt means the maximum iteration number and k means the k-th iteration; c_1, c_2 are accelerated factors which present cognition and social of the particle; r_1, r_2 are random numbers between 0 and 1.

2.2 Coverage Criteria

The goal of software test is to uncover as many faults as possible with a potent set of tests. But predicting how many faults will be uncovered by a given test set is almost impossible [11]. We need test adequacy criteria to help us judge whether a data set is good enough to accomplish the test. Regardless of whether test adequacy criteria really can represent the quality of a test suite, they do represent the thoroughness of testing.

There are several common coverage criteria in structural test like statement coverage, branch coverage, condition coverage, multiple condition coverage, condition-decision coverage (C/DC) and path coverage [2]. A condition is a leaf-level Boolean expression and cannot be broken down into a simpler Boolean expression. A decision is a Boolean expression composed of conditions and Boolean operators. A decision without any Boolean operators is a condition. In these criteria exists a hierarchy, the top one is multiple condition coverage which requires every permutation of values for the Boolean variables in every condition occurring at least once. On the contrary, function coverage only requires the execution of every function.

In many automatic testing researches, path coverage was used as their criterion. Though path coverage is applicable to a number of program's testing, it is not perfect. For example, assume there is a decision consisted by disjunction of two conditions like ($a \parallel b$) in the program. For the true path, according to the short circuit evaluation, most programming language will check condition a first and if condition a is true then ignore the condition b; for the false path, both condition a and b need to be false. Finally we find that condition b may never be true even both paths were already covered.

In PSODGT, we use condition-decision coverage as the coverage criterion. Condition-decision coverage requires that every decision in the program has taken all possible outcomes at least once, and every condition in a decision in the program has taken all possible outcomes at least once. Although in the hierarchy, the level of C/DC is lower than multiple condition coverage, C/DC already can make sure that every piece of the program can be executed if the requirement of C/DC is fully fulfilled.

3 The PSO Data Generation Tool (PSODGT)

In this section, the PSO data generation tool is introduced. First of all, we give a brief overview of the PSODGT. Then we discuss three issues of the PSODGT in detail, including the main data structure, implementation of the fitness function and the improved PSO algorithm for this tool.

3.1 Overview of PSODGT

PSODGT is designed to work on programs written in C or C++ programing language and the architecture of PSODGT is shown in Fig. 1. There are two parts in PSODGT, automatic instrumentation and test data generation. Original source code is automatically instrumented and compiled in the automatic instrumentation part. After compiling, an instrumented executable program is generated for data generation part to work on. The

data generation part also consists of two classes. Class controller maintains a condition table and a test data set, taking charge of choosing target condition and branch, storing useful inputs found by class optimizer and updating the condition table all by using a key function named runOnce. The optimizer class only focuses itself on reaching the target branch chosen by controller.

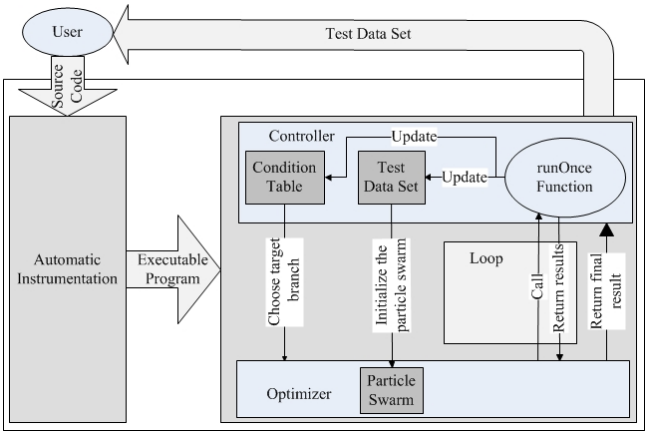

Fig. 1. Architecture of the PSODGT

3.2 Condition Table

The condition table is a vitally important data structure for the PSODGT. It is derived from the decision table proposed in [12] and modified in [4] by replacing decision with condition. Different from the condition table in [4], each branch in PSODGT's condition table has three statuses not two. A sample condition table is shown in Table 1.

Table 1. Condition table

	Branch	
Condition	TRUE	FALSE
1	1	2
2	0	0
3	1	1
4	1	0

Status 0 means this branch is not covered yet, 1 means this branch has been covered and 2 means that the algorithm has failed in optimizing this branch of the condition. Status 2 is used to avoid endless loop. When a goal branch is needed, PSODGT always chooses the condition the state of which is 1/0 or 0/1, i.e., this condition has been reached but its branches are not fully covered yet. To satisfy C/DC requirement, we take advantage of the short circuit evaluation. If every condition is fully covered

(both TRUE and FALSE branches have been achieved), all decisions can be fully covered. So we can only focus on covering conditions as many as possible.

3.3 Fitness Calculation

When a condition's branch is chosen, things we need to do next is to get its fitness under test data. If a strip of input data can reach the chosen condition, variables in the chosen condition have relationship with the input data. Because different inputs should cause different value for variables in condition so that conditions may take different outcomes. For example, suppose that a hypothetical program contains the condition

```
if(temp > 10){…} else {…}
```

on line 50 and the goal is to reach its FALSE branch. Denoting x as input, according to the relationship between input data and variables in condition, `temp` can be indicated as $line_{50}(x)$. Then the fitness function of this condition can be express as $f(x) = 10 - line_{50}(x) + 1$. When $f(x)$ <= 0, the goal branch will be achieved. Using the value of $f(x)$, the problem of generating test data turns into function minimization problem. If the input data cannot reach the chosen condition, we set a very large value as fitness to represent that this condition is not related to the input data.

Table 2. Computation of the fitness function

Condition Type	Goal Branch	Fitness Calculation
c > d	T	d – c + minConst
	F	c – d
c < d	T	c – d + minConst
	F	d – c
c >= d	T	d – c
	F	c – d + minConst
c <= d	T	c – d
	F	d – c + minConst
c == d	T	|c – d|
	F	minConst – |c – d|
c	T	1000
	F	1000

Table 2 shows how fitness is calculated for all condition types if the condition is reached.We add a constant named minConst to some fitness so that all fitness calculations can be evaluated as a positive number no matter what the goal branch is. The goal is to reduce the fitness down to zero or negative numbers. For integer problem, minConst is set to 1; for float problem, minConst can be set to a very small float number according to the precision needed in the real situation.

3.4 Particle Swarm Optimization for PSODGT

According to the actual situation in test data generation, some improvements are applied to the PSO algorithm in initialization step and parameters setting. Initialization step in PSO can be divided into position initialization and velocity initialization.

Plenty of existing researches use a strategy in initial population generating that add the test data which can successfully reach the target condition to the population firstly; if additional inputs are needed, generate some random inputs to fill the population. However for PSO, these randomly generated inputs take a lot of time in reaching the target condition. Considering this problem, this paper proposes a new method for the initialization step of PSO using the idea of crossover operation in GA and stratified sampling for reference. Suppose there are N particles needed in the swarm and the number of the target program's input is D. The position of the i-th particle is presented as an array $P_i[1…D]$. The positions initialization steps are as follow:

1. If the number of the test data which can successfully reach the condition is bigger or equal to N, randomly add N of them into swarm and end the initialization process; else add all of them into swarm and calculate how many additional particles are needed.
2. Assume M particles are needed. If $M<=N–M$, use random M particles in the swarm as seeds and each seed generates one additional particle, else use all the particles in the swarm as seeds and each seed generates $M/(N–M)$ particles.
3. For every generation of every seed, it begins with copying the position of i-th seed to the new particle, $NewP[1…D]=P_i[1…D]$.
4. After copy, generate a number d smaller than D randomly. Then construct an d-length array $changePos[1…d]$ filling up with different numbers which are generated randomly and smaller than D.
5. Finally substitute the values in $NewP$ with random numbers according to $changePos$, $NewP[\ changePos[1…d]\ ]=randomNumber$.

Research in [4] found that there are lots of serendipitous coverages during test data generation. This means some test data do cover new condition branches but these conditions are not the one the optimizer is currently working on. Serendipitous coverage requires degree of randomness in optimizer. However the directional character makes PSO perform badly in gaining serendipitous coverage. Considering this problem, particles' velocities are initialized within the same boundary as positions to obtain more randomness in early stages of iteration. This setting causes a consequence that convergence speed becomes slower than basic PSO. To make up the losses on convergence rate, ω the inertia factor is set as 0.4 down to 0.3 with the iteration growing. A lot of experiments have been done to verify this setting about inertia factor. When it is set much higher, convergence speed is too slow to meet the requirement. While, if it is smaller than 0.3, the algorithm usually fails.

4 Experimental Studies

4.1 Experimental Settings

In the experiments, we test the proposed method on four programs: triangle classification program, week calculation program, student grade judgment program and blood glucose judgment program. (Denoted as P1, P2, P3, and P4) Different from the simple programs tested in [4], these programs are practical and full of various conditions. When we test these programs, two main aspects are taken into consideration. One is the dimension (number of inputs) and size of search space; the other is the number of conditions. Specific figure about the dimension of search space and the number of conditions are shown in Table 3. And each program is tested with two different size of search space measured by bit shown in Table 4.

Table 3. Number of conditions and inputs

Program	Condition number	Inputs Number (dimension)
P1	20	3
P2	76	3
P3	16	5
P4	33	12

Our proposed PSO approach is compared with two genetic algorithms with different coding schemes which are gray code (GAG) and binary code (GAB). Also a PSO method using the same initialization way with GA is tested to verify the necessity of the proposed initialization technique, denoted as iPSO. We use 100 individuals, allow 30 generations to elapse before two GAs give up, the same as in [4], and the mutation probability of every bit is 0.01. For PSO, 20 individuals and 100 generations are allowed. Values of accelerated factors c_1 and c_2 are 2.

4.2 Experimental Results and Analysis

In this paper, experimental results are estimated in two aspects, efficiency (convergence rate) and effectiveness (coverage rate). In the course of experiment, we take five serial attempts as a group of tests. After six groups of tests for each method, the best coverage rate in each group is selected and the best, worst, average coverage rate of these six numbers are shown in Table 4, displayed in percentage.

Comparing the experimental data on different search space size for the same program, we can find that when the number of input is relative small, increasing on the search space size doesn't affect the coverage rate greatly. However when the number of input grows larger, the contrary is the case. Though both GA and PSO suffer the increasing on search space size, the PSO approach is much more stable. And the poor performance made by iPSO also demonstrates the importance of our proposed initialization tech in PSO method for software test data generation. In general, except the

data of best coverage rate for P1 in 16-bit search space which is underlined in Table 4, all the rest data show the advantage of our PSO approach in effectiveness.

Table 4. Experimental results on coverage rate

	Prog.	P1		P2		P3		P4	
	Bit	32	16	16	12	12	8	12	8
PSO	mean	95	95	99.45	100	100	100	88.38	100
	worst	95	95	98.68	100	100	100	65.15	100
	best	95	95	100	100	100	100	100	100
GAG	mean	88.75	93.33	96.49	96.49	82.29	100	45.20	100
	worst	82.50	92.50	94.74	93.42	43.75	100	33.33	100
	best	95	97.50	98.03	99.34	100	100	57.58	100
GAB	mean	72.50	85.42	96.71	96.93	59.90	100	36.36	100
	worst	50	72.50	95.39	96.05	43.75	100	28.79	100
	best	85	92.50	98.03	98.68	90.63	100	40.91	100
iPSO	mean	85.42	87.50	82.46	86.95	41.67	100	26.27	94.70
	worst	80	87.50	79.61	85.53	40.63	100	24.24	71.21
	best	87.50	87.50	84.21	87.50	43.75	100	30.30	100

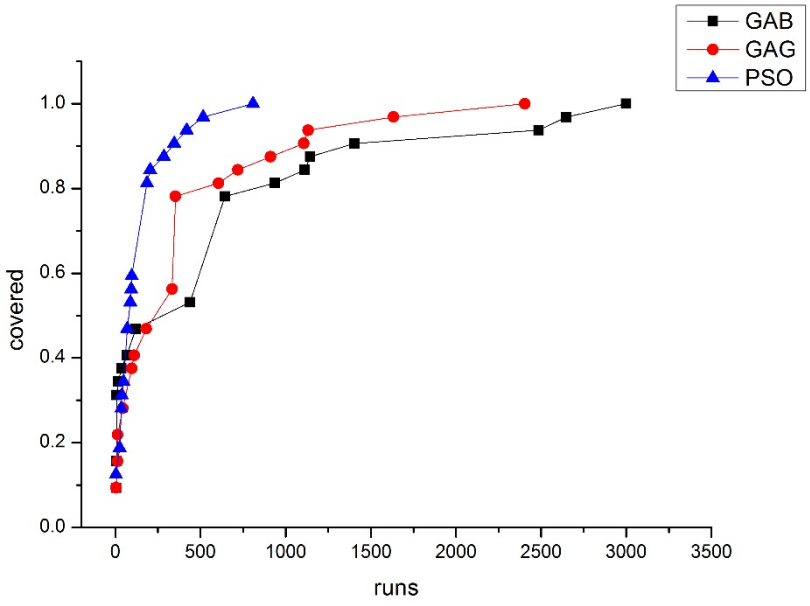

Fig. 2. Converge rate of three methods on P3

As to the convergence rate, P3 with 8-bit search space is used in comparing the efficiency because all three approaches performed well on this program and iPSO is not shown because it is too slow comparing with others. The relationship between runs

and coverage rate is illustrated in Fig. 2. This plot shows that PSO begins to demonstrate its advantages after reach 50% coverage and covers all conditions first. Clearly the convergence rate of PSO is much faster than the two GA approaches. The same fact also can be verified by the other programs.

All these experimental results show that the proposed PSO approach is effective and efficient in automatic software test data generation.

5 Conclusion

In this paper, an improved PSO approach is proposed to apply to search-based test data generation. The main contributions are in two aspects. First, the PSODGT is developed by combining the PSO algorithm and C/DC. Second, a new position initialization technique is developed for PSO to adapt accommodate software testing. Experimental results show that the proposed PSO approach is very promising.

In the future research, it will be interesting to find out which method is suitable to which kind of condition so that more hybrid methods can be proposed to apply to different conditions. And how to use much higher-level coverage criterion is also a promising research topic.

Acknowledgement. This work was supported in part by the National High-Technology Research and Development Program (863 Program) of China No. 2013AA01A212, in part by the NSFC for Distinguished Young Scholars 61125205, in part by the NSFC Nos. 61379061, 61070004, in part by Natural Science Foundation of Guangdong No. S2013040014949, and in part by Programs Foundation of Ministry of Education of China No. 20130171120016.

References

1. National Institute of Standards and Technology, “Then Economic Impacts of Inadequate Infrastructure for Software Testing,” Planning Report 02-3 (May 2002)
2. Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. John Wiley & Sons (2011)
3. Díaz, E., Tuya, J., Blanco, R.: Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search. In: Proc. 18th IEEE Int’l Conf. Automated Software Eng., pp. 310–313 (2003)
4. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Trans. Software Eng. 27(12), 1085–1110 (2001)
5. Bottaci, L.: Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm. In: Proc. Genetic and Evolutionary Computation Conf., pp. 1337–1342 (2002)
6. Li, A., Zhang, Y.-L.: Automatic Generating All-Path Test Data of a Program Based on PSO. In: WRI World Congress on Software Eng., pp. 189-193 (2009)
7. Windisch, A., Wappler, S., Wegener, J.: Applying Particle Swarm Optimization to Software Testing. In: Proc. 9th Ann. Genetic and Evolutionary Computation Conf., pp. 1121–1128 (2007)

8. Cui, H.-H., Chen, L., Zhu, B., Kuang, H.-L.: An Efficient Automated Test Data Generation Method. In: Int'l Conf. Measuring Technology and Mechatronics Automation, vol. 1, pp. 453–456 (2010)
9. Zhang, S., Zhang, Y., Zhou, H., He, Q.-Q.: Automatic Path Test Data Generation Based on GA-PSO. In: Proc. IEEE Int'l Conf. Intelligent Computing and Intelligent Systems, pp. 142–146 (2010)
10. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proc. IEEE Int'l Conf. Neural Networks, vol. 4, pp. 1942–1948 (1995)
11. Frankl, P., Hamlet, D., Littlewood, B., Strigini, L.: Choosing a Testing Method to Deliver Reliability. In: Proc. Int'l Conf. Software Eng., pp. 68–78 (1997)
12. Chang, K.H., Cross II, J.H., Carlisle, W.H., Liao, S.-S.: A Performance Evaluation of Heuristics-Based Test Case Generation Methods for Software Branch Coverage. Int'l J. Software Eng. and Knowledge Eng. 6(4), 585–608 (1996)
13. Clarke, L.A.: A System to Generate Test Data Symbolically and Execute Programs. IEEE Trans. Software Eng. 2(3), 215–222 (1976)
14. Offutt, A.J.: An Integrated Automatic Test Data Generation System. J. Systems Integration 1, 391–409 (1991)
15. Korel, B.: Automated Software Test Data Generation. IEEE Trans. Software Eng. 16(8), 870–879 (1990)
16. Sofokleous, A.A., Andreou, A.S.: Automatic, evolutionary test data generation for dynamic software testing. J. Systems and Software 81(11), 1883–1898 (2008)
17. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. Software Eng. 36(2), 226–247 (2010)
18. Shi, Y., Eberhart, R.: A modified particle swarm optimizer. In: Proc. IEEE Int'l Conf. Evolutionary Computation, pp. 69–73 (1998)