Using Augmented Genetic Algorithm for Search-Based Software Testing

Zahir Hasheminasab[11], Zaniar Sharifi[1], Khabat Soltanian[1], Mohsen Afsharchi[1]

[1] Zanjan University, Zanjan 45371-38791, Iran
Zahir.hashemi,Zaniar.sharifi,K.soltanian,Afsharchim(@znu.ac.ir)

Abstract. Automatic test case generation has been received great attention by researchers. Evolutionary algorithms have increasingly gained special places as means of automating the test data generation for software testing. Genetic algorithm is the most commonplace algorithm in search-based software testing. One of the key issues of search-based testing is the inefficient and inadequate informed fitness function due to the rigidness of fitness landscape. To deal with this problem, in this paper we improved a recently published fundamental approach where a new criterion, branch hardness factor is used to calculate fitness. However, the existing methods are unable to cover the whole of the targets. Herein, we added a local search strategy to the standard genetic algorithm for faster convergence and providing more intensification. In addition, different selection and mutation operators are examined and appropriate choices selected. Our approach gained remarkable efficiencies on 7 standard benchmarks. The results showed that adding local search is likely to boost another search-based algorithm for path coverage even.

Keywords: genetic algorithm; path coverage testing; automatic test data generation.

1 Introduction

Nowadays, using software production is becoming more and more indispensable in daily life, therefore role of software testing is being highlighted for verifying quality of software. Approximately 50 percent of software development process cost is being consumed on Software testing [1]. Moreover, this process is a time consuming and tedious process, since it is done manually. Therefore, automated software testing is being evaluated as the indispensable method to decline time and cost.

There are different types of testing criteria, which are classified into two testing strategies, such as black-box testing and white-box testing [2]. Black box testing, is a software testing method which the code block being tested is not known, whereas white-box testing is respected to only the implementation of items that can be tested.

---

[1] Corresponding author

In the other word, in white box testing the internal structure of the program under test is known for tester.

Generally Speaking, the main goal of software testing is to generate test cases satisfying test criteria. A test case is a set of conditions or variables under which the tester will determine whether a system under test satisfied requirements or works correctly.

Test case generation approaches based on the algorithm can be classified to static methods and dynamic methods.

Static methods are software testing techniques in which the software is tested without executing the code. They comprise symbolic execution [4] and domain reduction [5, 6]. Although these methods have had important successes, they still face challenges in managing handles indefinite loops, array, procedure calls and pointer references in any tested program [7].

In symbolic execution method, instead of using actual value, symbolic value is being used, i.e., variable of x and y are considered with x_1 and x_2 respectively. In this method at every point of implementation, symbolic value of program variable and path constraint are presented as a rational formula on the symbolic values of the program variables. For access to that point, the path constraints must be "true". In addition, the path constraints are determined by the logical expressions used in the branches, which are updated with each branch. Any combination of real inputs, for which the value of the path constraint is "true", it could be considered as a program input that guarantees the execution of the desired path. In this method, we must use constraint solvers to find the actual values in order to produce the test case. These approaches can determine infeasible paths simply. In these methods, constraints solvers have been used to find the actual values in order to produce the test case. Therefore, the efficiency of the method is strongly dependent on the efficiency of solver and the calculation of host hardware. Moreover, in case of non-linear branch conditions, static methods have significant overhead cost.

Dynamic methods involve in testing the software for the input values and analyze the output values according to the generated input. In fact, dynamic methods generate input values for program under test and the comprise random testing, local search approach [8], goal-oriented approach [5], chaining approach [9] and evolutionary approach [9, 10-13]. In these methods, the software is tested by inserting inputs and measuring the number of target paths covered by the software. Moreover, due to predefined of input variables determined during the execution of the program, the production of dynamic test data can prevent those problems encountered by static methods.

Hybrid methods combine the advantages of static methods (like reducing domain of problem) with the benefits that can be obtained from the dynamic methods (such as reducing the costs), combination methods have been developed [17].

All of method evaluations are based on different criteria. There are different test criteria, such as instruction coverage, branch coverage and path coverage.

Instructions Coverage: In this case, it is necessary to select input data from the problem space that all instructions are executed at least once.

branch coverage: The input data is selected from the problem space that all the branches are executed at least once [3].

Path coverage: the input data is selected from the problem domain that all the paths are traversed at least once.

This paper addresses path coverage. In particular, consider the most difficult paths. It used the hybrid method that the symbolic execution as static method and evolutionary algorithm as dynamic method selected to generate test data generation.

In this paper, one of the most recent works in the field of static and dynamic methods for test data generation has been improved. In [17], by combining the previous fitness functions and improving them, they developed a new fitness function for the genetic algorithm. In this paper, by using the proposed silent function, as well as changing in the main architecture of the genetic algorithm, a new approach is developed in the field of automatic test data generation. The proposed method has been experimented on the 7 standard benchmarks introduced in [21]. The results and performance demonstrated a significant improvement in the efficiency and effectiveness of the software testing.

The remainder of this paper is organized as follows: The second section and the third section introduce background and related work in this area respectively. Genetic algorithm and our approach in detail are presented in the fourth section. In the five section, the proposed method is applied to 7 standard benchmarks and provided the illustrative experiments that compared with recent papers and the last section gives the conclusion and future work to the paper.

2 Background

Most of fitness functions in software testing research area are based on approach level [15] and branch distance [16] which are two approaches to calculate generated test cases fitness functions. Approach level was proposed [15] and calculate test cases fitness function by enumerating remained branches to execute to gain the target branch. Branch distance factor is the test case's distance from satisfying a branch's condition. In other word, a number must be added or subtracted from the test case to satisfy the condition. Consequently, this two-fitness factor combine together to improve fitness functions accuracy which calculate by following equation:

$$f_{AL}(i) = level(b) + \eta(i, b)$$

In above equation $level(b)$ is approach level and $\eta(i, b)$ is the branch distance.

Discussed approaches did not consider executed branches, therefore Symbolic Enhanced Fitness Function was proposed by harmen et al at 2011 [17]. They add a simple static analysis. i.e., symbolic executor to evolutionary algorithms for software testing. It calculates the cost of that a test case can satisfy all branch conditions with a normalized branch distance. By mean that this approach attends all executed and non-executed branches. This approaches equation is calculating as follow equation:

$$f_{SE}(i) = \sum_{b \epsilon P} \eta(i, b)$$

In [14] by portion of Symbolic Enhanced Fitness Function, proposed a factor for determining branches hardness level and calculate test cases finesses according to the

branches harnesses. formulated the hardness considering two main factors, first one is number of variables in the branch condition(α(c)) which extracted by Symbolic analyzer and second one is the branch conditions tightness(β(c)) Which is ratio of number of solutions in the problem's domain to the size of domain. It also used a reinforcement coefficient to tunes effect of these two discussed factors in calculation of branches hardness. their hardness factor is calculated as follow:

DC(c)=B^2×α(c)+B×β(c)+1

This hardness is as a punishment to test cases who cannot satisfy the branch. And its related fitness function calculation is as the following equation:

$$f_{DC}(i, C) = \sum_{c \epsilon C} DC(c) \times \eta(i, b)$$

For example consider $i_1 = (10, -30, 60)$, $i_2 = (30, -20, -20)$ as two test cases and figure 1 as our source code.

```
sample(int x,int y,int z){        1
  if(y==z)                         2
    if(y>0)                        3
      if(x==10)                    4
        ...//Target                5
}                                  6
```

Figure. 1 Example source code [14]

There are three branches in this source code in lines: 2, 3 and 4. This program branches hardness's has been calculated as:

DC("y==z")=$10^2 \times 0.5 + 10 \times 0.995 + 1 = 60.95$
DC("y>0")=$10^2 \times 1 + 10 \times 0.5 + 1 = 106$
DC("x=10")=$10^2 \times 1 + 10 \times 0.995 + 1 = 110.95$

Therefore i_1 and i_2 finesses would be:

$$f_{DC}(i_1, C) = 60.95 \times \frac{90}{91} + 106 \times \frac{31}{32} + 110.95 \times \frac{0}{1} = 162.9677$$

$$f_{DC}(i_2, C) = 60.95 \times \frac{0}{1} + 106 \times \frac{21}{22} + 110.95 \times \frac{20}{21} = 206.8485$$

According to their fitness values i_1 had been preferred than i_2.

3 Related work

In this section, we review the most important methods that centered around different meta-heuristic algorithms.

In [14] benefited from both static and dynamic approaches advantages. it extracts some information from path conditions using static analyzing. the information had been used for defining more exact population instead of random initialization of the first population for GA.

After 2014 most of researchers concentrate on guiding Genetic algorithm to faster converge which that leaded to decrease in calculation costs. Accordingly, to that designing an appropriate fitness function considered by researchers. In [14] proved that branches have no equivalent values according to their hardness. It means that satisfying a harder branch is more valuable, therefore a test case who satisfies harder branches is more valuable. So they had been defining hardness factor to determining each branch harnesses, which has been used in fitness function equation [18].

In [13] an approach to improve genetic algorithm efficiency proposed. They defined their exclusive branch distance and fitness function. In addition [1] reinforced genetic algorithm by considering a preprocessing step before performing the algorithm. They extracted hard path conditions and used them to made a kind of adjustment for genetic algorithm which tunes individuals for faster converging. [19] combined static and dynamic approaches to generating test cases, they developed their static analyzer (JDBC) to extract path conditions, and used a search problem converter that converts extracted path conditions to optimization problems and finally they use genetic algorithm to solve these optimization problems. In [20] a branch hardness factor defined using probability of visits, hence branches with fewer Expected number of visits are harder than other.

4 Proposed Approach

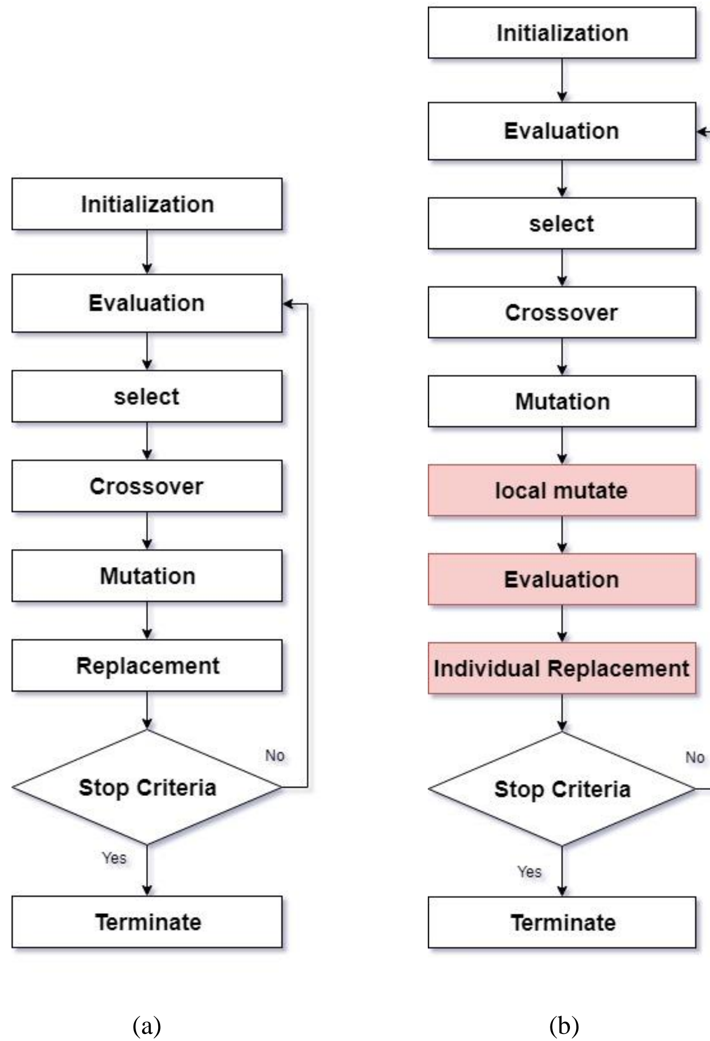
This section depicts details of our proposed approach, to generate test cases for path coverage using augmented GA. By using the proposed silent function in [14], and changing in the main architecture of the genetic algorithm, a new approach is developed in the field of automatic test data generation.

Generally speaking, evolutionary algorithms search for a general optimal point in the solution space, and usually cannot search locally around specific responses [22]. They could be trapped in an optimal point. In addition, sample space of software testing problem is very extensive. Therefore, this problem would be obvious. Have the feature of evolutionary algorithms (general search) is combined with a local search algorithm, the results will be improved. In other words, the evolutionary algorithm first finds good answers. Then, this area could be accurately searched by a local search algorithm to find the optimal point. Details of our approach is described below.

Genetic algorithm is a search heuristic that is inspired from Charles Darwin's theory of natural evolution. This algorithm models the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring for the next generation. The process of natural selection starts with the selection of fittest individuals from a population. They generate individuals that almost keep the characteristics of their parents and will be added to the next generation. If parents are fitter, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found. Genetic algorithm has a wide application in optimization problems [23].

Based on the figure 2. (a), the genetic algorithm architecture consists of six phases:

1. Initial population to start the algorithm.
2. Population Fitness functions evaluation and assign a fitness number to each individual.
3. Selection: select a pair of individuals as parent to make offspring.
4. Crossover: is evolution operator which exchange parents' bits with together to generates better individuals.
5. Mutation: mutate some bits to avoiding trapping in local optimums.
6. Replacement: replace new generated population with old one.



(a) (b)

Figure. 2 (a), (b) show the architecture of traditional GA and augmented GA, respectively

In our proposed architecture showed in figure 2. (b), in addition to the above steps, a new step is added in which selection and mutation operators are re-evaluated and appropriate operators selected. The basis of this algorithm is inspired by the hill climbing algorithm, therefore, it could be defined as a local search algorithm.

Local search. the algorithm, among the neighbors of each individual, probs the fittest point. To calculate neighborhood of Individual k, D-dimensional space is considered. The neighbors of Individual k with position vector $IND_k = (x_{k1}, x_{k2}, \dots, x_{kd})$ have a new position vector of $I\acute{N}D_k = (x'_{k1}, x'_{k2}, \dots, x'_{kd})$ where $x'_{k1} = x_{k1} + p$, -500 <p<+500 and $x'_{k1} \neq x_{k1}$ that p based on a gaussian distribution is selected.

The rule for local search or local transfer of Individual position can be represented as follows: Individual k transfers from x_k to a new position x'_k if the fitness of x'_k is better than that of x_k (i.e., fitness(x'_k)>fitness(x_k)), and x'_k has the best fitness value among x'_k neighbors. Otherwise, the Individual k must stay at its current position (i.e., x_k).

5 Experimental results

We implemented [14] as a base and improve this approach. ran our proposed algorithm on 7 standard benchmarks which had used them. it is Noteworthy that we had 30 runs on each benchmark and all of presented data is averaged out 30 times of run. We compared our approach with three others according to 2 factors, coverage percentage of targets in the benchmarks and average time cost of running of each benchmark which has been calculated using this formula:

$$\text{ATC} = \frac{1}{|S|}\sum_{i\epsilon S} TC_i$$

Which in the above equation S is the set of successful runs of the algorithm. And TC is the time cost of each run individually. ATC determines the fair time cost for the algorithm.

Table. 1 a comparison between this paper approach and others [20]

Benchmarks	Fitness function approaches							
	proposed approach		Sakti [14]		Symbolic EXE		Approach level	
	coverage	ATC(s)	coverage	ATC(s)	Coverage	ATC(s)	coverage	ATC(s)
Gammaq	100%	0	100%	0	66%	0.370	59%	0.309
Expint	100%	2.133	75%	2.158	31%	2.180	1%	1.495
Ei	100%	0.133	75%	0.597	77%	0.947	77%	0.685
Bessj	100%	0.541	60%	2.103	31%	2.240	6%	1.059
Bessi	100%	0.539	85.5%	2.001	51%	1.978	11%	1.406
Plgndr	100%	0	-	-	0%	-	0%	-
Betai	100%	0	100%	1.259	70%	1.115	13%	0.938

Our results clearly prove this approach's superiority than former approaches.
In the following diagram we can see the speed of convergence of proposed approach against other former approaches. Figure 3 shows the percentage of coverage in the number of generations produced in five different approaches. As we can see, number of generation that our proposed approach needed to completely cover all targets is far less than other approaches. While other approaches in the number of generations more than our attitude have reached 80% coverage, none of them have been able to fully cover 54 goals.

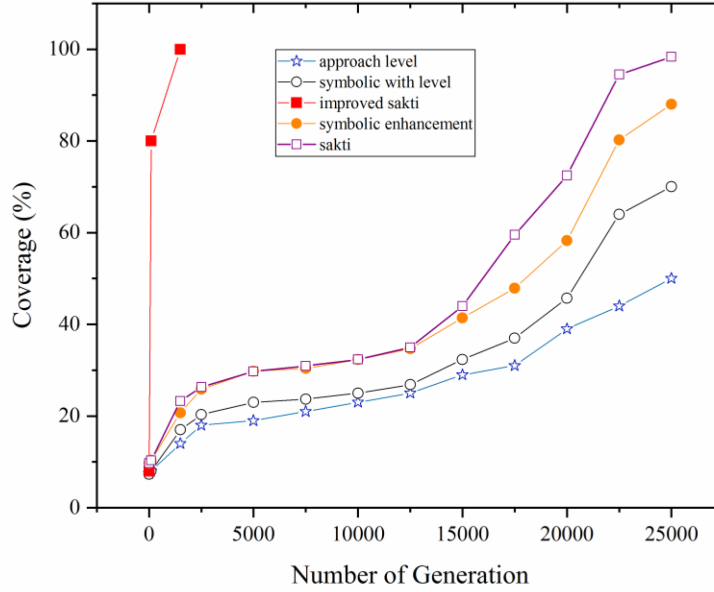


Fig. 3 show coverage rate of different approaches vs our approach

Tuning parameters of these papers are according to the following table:

Table.2. implementation details

Bounds	[-1000,1000]
Population	100
Mutation rate	0.5
Number of comparisons in local search for each individual	2 per each gene

6 Conclusion and Future Work

In this paper, we proposed a search-based test data generation approach to cover Paths coverage of the program under test. By using the proposed silent function in [14], as well as improving in the main architecture of the genetic algorithm. The experimental results of some programs under test demonstrated that augmented GA generated test data can cover all feasible paths having path conditions which cannot be covered by

test data generated from regular GA. The main reason for this superiority is due to the local search.

Since these issues are inherently different from optimization issues, and in most cases the level of response space is discrete, the combination of search optimization algorithms such as linear programming with this algorithm can be very useful. There have been some studies performed in this area that, definitely, should be used as a function of this combination. (i.e., in the initialization step, some parts of the answer can be obtained with precise methods).

7 References

1. Dinh., Ngoc, Thi., Hieu, Dinh Vo., Thi, Dao Vu., and Viet, Ha Nguyen.: Generation of Test Data Using Genetic Algorithm and Constraint Solver. In: *Asian Conference on Intelligent Information and Database Systems*, 499-513. Springer, Cham (2017).
2. Myers, Glenford J. "77ie Art of Software Testing." (1979).
3. W, Xibo., S, Na.: Automatic Test Data Generation for Path Testing Using Genetic Algorithms. In: Third Int. Conf. pp. 596–599, Meas (2011).
4. C, K, James., A new approach to program testing, in Proceedings of the international conference on Reliable software. ACM, Los Angeles, California (1975).
5. T, Y, Chen., T, H, Tse., Z, Zhiquan., Semiproving.: an integrated method based on global symbolic evaluation and metamorphic testing, international symposium on Software testing and analysis. ACM, Roma (2002).
6. S, Nguyen Tran., D, Yves.: Consistency techniques for interprocedural test data generation, Software Engineering Notes, vol. 28, 108-117, ACM SIGSOFT, (2003).
7. G, M, C C Michael., M, Schatz.: Generating software test data by evolution, IEEE Transactions on Software Engineering, vol. 27, 1085-1110, (2001).
8. B, Korel.: Automated software test data generation, IEEE Transactions on Software Engineering, vol. 16, 870-879, (1990).
9. B, Korel.: Automated test data generation for programs with procedures, in Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis , ACM, San Diego, California, United States: (1996).
10. S, Xanthakis., C, Ellis., C, Skourlas., A, Le Gall., S, Katsikas., K, Karapoulios.: Application of genetic algorithms to software testing, in Proceedings of 5th International Conference on Software Engineering and its Applications, 625-636, Toulouse, France (1992).
11. J, Wegener., A, Baresel., H, Sthamer.: Evolutionary test environment for automatic structural testing, Information and Software Technology, vol. 43, 841-854, (2001).
12. J, Wegener., B, Kerstin., P, Hartmut.: Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing, in Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers Inc., (2002).

13. Thi., D.N., Hieu, V.D., Ha, N.,V.: A technique for generating test data using genetic algorithms. International Conference on Advanced Computing and Applications. IEEE Press, Can Tho (2016).
14. Sakti, Abdelilah., Yann-Gaël Guéhéneuc., Gilles Pesant.: Constraint-based fitness function for search-based software testing. *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer, Berlin, Heidelberg, (2013).
15. Tracey, N., Clark, J.A., Mander, K., McDermid, J.A.: An automated framework for structural test-data generation. In: ASE, pp. 285–288 (1998).
16. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: ICST, pp. 205–214. IEEE Computer Society (2010).
17. Baars, A.I., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.E.J.: Symbolic search-based testing. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (eds.) ASE, pp. 53–62. IEEE (2011).
18. Sakti, Abdelilah.: Automatic Test Data Generation Using Constraint Programming and Search Based Software Engineering Techniques. École Polytechnique de Montréal, (2014).
19. Braione, Pietro., et al.: Combining symbolic execution and searchbased testing for programs with complex heap inputs. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, (2017).
20. Xu, Xiong., Ziming Zhu., and Li Jiao.: An adaptive fitness function based on branch hardness for search based testing. In Proceedings of the Genetic and Evolutionary Computation Conference. ACM, (2017).
21. http://www.crt.umontreal.ca/~quosseca/fichiers/23benchsCPAOR13.zip
22. Yao, X.: Evolving artificial neural networks. In *Proceedings of the IEEE*, vol. 87(9). pp. 1423–1447, (1999).
23. https://towardsdatascience.com/introduction-to-geneticalgorithms-including-example-code-e396e98d8bf3