Dynamic generation of test cases with metaheuristics

Juan Pablo La Battaglia, Laura Lanzarini

III-LIDI (Institute of Research in Computer Science LIDI)
Faculty of Computer Sciences. National University of La Plata
La Plata, Buenos Aires, Argentina
{juanlb, laural}@lidi.info.unlp.edu.ar

Abstract. The resolution of optimization problems is of great interest nowadays and has encouraged the development of various information technology methods to attempt solving them. There are several problems related to Software Engineering that can be solved by using this approach. In this paper, a new alternative based on the combination of population metaheuristics with a Tabu List to solve the problem of test cases generation when testing software is presented. This problem is of great importance for the development of software with a high computational cost and which is generally hard to solve. The performance of the solution proposed has been tested on a set of varying complexity programs. The results obtained show that the method proposed allows obtaining a reduced test data set in a suitable timeframe and with a greater coverage than conventional methods such as Random Method or Tabu Search.

Keywords: Software Testing, Evolutionary Testing, Particle Swarm Optimization, Evolutionary Algorithms, Metaheuristics.

1 Introduction

The automatic generation of a test dataset that allows measuring the performance of a given program is a highly important task in software development that requires a high computational cost and is generally hard to solve.
The solution to this problem has been widely studied for a long time now. The first paradigm used was the so-called random test data generation, which consisted in creating a test dataset in a random manner until reaching the termination condition or until a maximum number of test datasets had been generated [2].
An alternative method to solve this problem is the symbolic generation of test data [11]. It consists in using symbolic values for the variables, instead of real values, thus allowing a symbolic execution. This execution generates algebraic restrictions that determine test cases.
A third paradigm is the dynamic generation of test data [9]. In this case, the program is modified to provide information to the seed in order to verify whether a

given criterion was reached. Thus, if the criterion was not reached, new data can be built to be used as input to the program. Under this paradigm, data generation becomes an optimization process, since each condition within the program can be analyzed as a function to minimize.
In particular, various metaheuristics have been used to dynamically generate the necessary test cases. There are solutions based on genetic algorithms [12], simulated annealing [16] and immune systems [3]. Some recent solutions use Tabu Search [5] and Scatter Search [13].
The purpose of this paper is to present a new solution to the problem of finding a suitable test dataset for testing the performance of a program by using a PSO-based populational metaheuristics combined with a Tabu list.
A white box testing procedure will be carried out, that is, the test-case seed will use information from the program structure to guide the search for new input data. Usually, the structural information is taken from the flow control graph of the program. The input data that are generated by the structural testing must be subsequently assayed against the program to check if they generate an incorrect behavior.
This article is organized as follows: Section II describes the original PSO method, Section III details the special considerations that should be adapted to solving the problem of a multi-objective, such as the generation of test cases using this structure, Section IV shows the obtained results. Finally, some conclusions are presented.

2 Particle Swarm Optimization

An algorithm based on particle swarms, also called Particle Swarm Optimization (PSO), is a populational metaheuristics where each individual represents a possible solution to the problem and adapts following three factors: its knowledge of the environment (its fitness value), its historical knowledge or previous experiences (its memory), and the historical knowledge or previous experiences of the individuals in its neighborhood [4]. Its purpose is to evolve in its behavior so as to resemble the most successful individuals within its environment. In this type of technique, each individual is in continuous movement within the search space and never dies. On the other hand, the population can be seen as a multi-agent system where each individual or particle moves within the search space storing, and ultimately communicating, the best solution that it has found [10].
There are different versions of PSO; the most widely known are gBest PSO, which uses the entire population as neighborhood criterion, and lBest PSO, which uses a small neighborhood size [6] [14]. Neighborhood size affects algorithm convergence speed as well as the diversity of individuals in the population. As neighborhood size increases, algorithm convergence speed increases and individual diversity decreases.

Each particle p_i is made up by three vectors and two fitness values:

- Vector $x_i = (x_{i1}, x_{i2}, , x_{in})$ stores the current position of the particle in the search space.
- Vector $pBest_i = (p_{i1}, p_{i2}, , p_{in})$ stores the best position of the solution found by the particle up to the moment.
- Speed vector $v_i = (v_{i1}, v_{i2}, , v_{in})$ stores the gradient (direction) according to which the particle will move.
- The fitness $value fitness\_x_i$ stores the current solution capacity value (vector x_i).
- The fitness value $fitness_pBest_i$ stores the capacity value of the best local solution found up to the moment(vector $pBest_i$).

The position of a particle is updated as follows

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{1}$$

As previously explained, the speed vector is modified taking into account its experience and the environments. The expression is the following:

$$v_i(t+1) = w.v_i(t) + \varphi_1.rand_1.(pBest_i - x_i(t)) + \varphi_2.rand_2.(g_i - x_i(t)) \tag{2}$$

where w represents the inertia factor [15], φ_1 and φ_2 are acceleration constants, $rand_1$ and $rand_2$ are random values belonging to the interval (0,1), and g_i represents the position of the particle with the best fitness of the environment of p_i (lBest o localbest) or the whole swarm (gBest o globalbest). Values of w, φ_1 and φ_2 are essential to assure the algorithms convergence. For more details on the selection of these values, consult [4] and [17].

Figure 1 shows the basic PSO algorithm.

```
S ← InitializeSwarm()
while termination condition is not reached do
  for all i = 1 to size(S) do
    Assess particle x_i of swarm S
    if fitness(x_i) is better than fitness(pBest_i) then
      pBest_i ← x_i;
      fitness(pBest_i) ← fitness(x_i)
    end if
  end for
  for all i = 1 to size(S) do
    Choose g_i based on the neighborhood criterion used
    v_i(t+1) = w.v_i(t) + φ1.rand1.(pBest_i − x_i(t)) + φ2.rand2.(g_i − x_i(t))
  end for
end while
Output : the best solution found
```

Fig. 1. Basic PSO

3 Description of the proposed solution

The dynamic generation of test cases involves knowing if the coverage criterion is achieved during execution or not. To this end, the original program is modified by inserting instructions that allow the seed to gather the required information. New input data are added to the test dataset until the desired criterion is reached. Thus, this software engineering problem becomes an optimization problem, since the purpose is minimizing a certain distance to a preset coverage criterion. The method used to achieve this minimization is based on the particle swarm optimization algorithm.

3.1 Program coverage

The criterion used in this paper to determine if a program is correctly covered or not is *condition-decision coverage*. This means that every condition in a decision takes all possible outcomes at least once. There are other criteria, such as the *statement coverage*, which require the execution of all of the instructions of the program or *branch coverage* wich requires the execution of every branch of the program. However, the criterion selected, by requiring that all conditions reach both truth values, ensures that all branches are covered, which also means that all the instructions of the program will be executed.
In order to carry out this task, each condition of the program is analyzed independently. For each of them, the strategy described in the previous section is applied. Since the program has to be run to verify the status of each condition, it is possible to check more than one condition in one execution.

3.2 Modifications of the PSO method

The optimization method used is a modified version of the basic PSO algorithm to take into account the specific characteristics of the problem at hand:

- It is a multi-purpose optimization process that uses a different population for each condition. Each population has a different size [7].

- Since PSO is an optimization strategy, it moves population individuals within the solution space in search of the optimum. This occasionally leads to oscillatory movements [8] or results in a loss of diversity [2]. In the case of test case generation, the function to minimize for each condition is an expression that allows inverting the truth value. For this function to work properly, the inertia of each particle has to be conserved; that is, w is not used in the usual way.

- Each population associated to a condition is formed by individuals that allow assessing it. They will all yield the same truth value. The purpose of the proposed method is using them to obtain the opposite truth value. It should

be noted that the execution of the program using one of these individuals in its new position as input may not be enough to assess the desired condition, which would prevent the assignment of a fitness value. This would be the same as using a non-continuous solution space, where individuals leave the interest space when they move. For this reason, the PSO has been modified so as to only allow movements within the solution space; the rest of the individuals keep their current position.

3.3 Fitness function

The proposed solution is only applicable to numeric input variables. The fitness function used in each case is indicated in Table 1. Its goal is returning a positive value, which will gradually approach zero as the individual that represents the input data being used for the execution of the program moves forward in the correct direction to obtain the opposite truth value.

Table 1. Fitness function used for each type of condition

Condition	FitnessFunction
$x = y$, $x \neq y$	abs(x-y)
$x < y$, $x \leq y$	y-x
$x > y$, $x \geq y$	y-x
$x \wedge y$	min(cost(x), cost(y))
x v y	if (x=true and y=true) then min(cost(x), cost(y)) else $\sum_{c_j FALSE}(cost(c_j))$

3.4 Generation method

The selected method for test case generation is of the white-box type; therefore, the values of the variables involved in each condition at the moment of execution must be known. To this end, a method composed by two modules was used:

- an execution wizard which, based on some symbols introduced in the source code (which do not affect execution), generates information on the values of each variable, and
- a process that, taking any given program as input, adds the aforementioned symbols.

All this information is automatically assessed by the test case generator.

Figure 2 summarizes the proposed method.

Conditions are ordered based on their occurrence within the code. After the first execution, at least one condition has been tested.
For each iteration, the first condition that is tested but is not covered is identified,

```
P ← CreateInitialStructure {all populations are empty}
TestData ← {Generate one random solution }
RunProgram(TestData)
the conditions that were reached have now one individual in their population
AnswList = TestData
while termination condition is not reached do
  idNoC ← {identify the 1st.condition that was assessed but not covered}
  TestData ← Apply_Modified_PSO(P(idNoC) )
  for all i = 1 to size(TestData) do
    if TestData(i) was not tested then
      changes = RunProgram(TestData(i))
      if changes > 0 then
        Add TestData(i) to AnswList set;
      end if
      Add TestData(i) to the list of already tested data.
    end if
  end for
end while
```

Fig. 2. Proposed method

and its population used to generate new input datasets with a modified version of PSO (*Apply_Modified_PSO* procedure). When there is only one individual in the population, it is used to generate a predefined number of variations, half of them within the 10% of the range allowed and the other half a bit further away, within the 50% of this range.

If the population has more than one individual, a variation of the global PSO is applied. The value gBest is obtained by averaging the position vectors of the two best individuals. All the individuals of the population, with the exception of the two best ones, calculate their velocity vector as follows:

$$v_{i+1} = 0,75.rand_1.v_i + 0,75.rand_2.(gBest - x_i) \tag{3}$$

whereas the two best individuals use less pressure to remain in their place and change their velocity vector update as follows:

$$v_{i+1} = 0,75.rand_3.v_i + 0,25.rand_4.(gBest - x_i) \tag{4}$$

As already mentioned, the concept of inertia is not used in the usual way, since the expected effect is that the particle pass through the optimum for the condition to invert its truth value therefore, in (3) and (4), the value used as inertia factor is a random number between 0 and 0,75. As in equation (2), $rand_1$, $rand_2$, $rand_3$ and $rand_4$ are random values belonging to the interval (0,1).
The new input data to be considered will be the positions of the individuals after their corresponding velocity vectors are added.
The RunProgram process is in charge of applying the input data and identifying which conditions have changed their status, since with every execution, new fulfilled or tested conditions may appear. During this process, the conditions that

Table 2. Results obtained with the method proposed and how it compares to two existing solutions

	Tabu Search		Random		Modified PSO Proposed Method	
Method	Coverage	Testing	Coverage	Testing	Coverage	Testing
Triangles	73.83	60.09	95.25	35.64	99	55.17
Calday	81.93	1440.23	98.45	202.56	99.11	504.06
Select	99.04	145.87	100	16.67	100	63.96
QuickSort	100	5.78	100	1.63	100	2.05
Bessel	96.03	2235.96	99.16	294.13	100	620.32

have been tested incorporate the used input data to their populations, replacing the original individual.
Unlike the conventional PSO algorithm, those individuals that generated new input data when moving but which did not allow testing the condition when running the program, will not be recorded in the population, leaving the original individual in the same position.
Each input dataset used to run the program is recorded on a list in order to reduce computation time. All input data that modified the status of any condition are incorporated to the output test dataset, AnswList.

4 Results

The solution was implemented in Ruby, an interpreted, reflexive, object-oriented programming language which is highly flexible and allows not only the quick modification of the solution, but also the implementation of the execution wizard that informs the value of the variables of each condition to the test case generator. The performance of the proposed method was tested in the generation of test data for some typical programs of the data testing field:

- Triangles: it receives the length of the three sides of a triangle and indicates the type of triangle.
- Calday: it receives a date and indicates the corresponding day of the week.
- Select: it receives an array with a disordered list and a k index and returns the kth lower element.
- QuickSort: list sorting method.
- Bessel: algorithm that solves Bessel functions Jn and Yn.

Table 2 shows the average of the results obtained after 100 independent executions of the proposed method considering a maximum number of 150 iterations. The results obtained with the Tabu Search method [5] and a completely Random generation under the same conditions are also included.

As it can be seen, the final coverage reached by the method herein proposed based on a modification of the PSO is higher for the tested programs. Based on the average number of tests of each method, the application of search strategies to solve problems requiring a small number of iterations should be considered. It can be seen that the Random method allows determining a test dataset that is suitable for the Select program and performs very few tests. In this particular case, the solution is easy to find, and the application of a search strategy only limits the exploratory capacity of the method, which does not occur with the random generation method. Nonetheless, even though the number of tests is higher, the fulfillment of the proposed method is still suitable.
In order to check that the results were really significant, they were subjected to a variance comparison statistical analysis.
Each sample was assessed with the Kolmogorov-Smirnov test to verify if they had a normal distribution. If they were normal (this only happened with the samples corresponding to number of tests in the "triangles" program), comparison was made by means of the Student test. For the remaining cases (whose distribution was not normal), the non-parametric test of Kruskal-Wallis was applied. The p-value obtained was below 0.05, which allowed verifying that the differences are statistically significant.

5 Conclusion

A new method for the generation of test cases has been presented. This method is based on a modified version of the PSO algorithm and uses specific populations associated to each condition of the program.
A testing and assisted execution system has been implemented for programs written in Ruby, which was used to measure the performance of this proposal.
The results obtained for each of the programs with the different methods indicate that the proposed method is robust, increasing fulfillment in all cases and slightly decreasing the number of total executions, thus proving that a significant contribution was made to the field.

References

1. Bird S., Li X. Adaptively Choosing Niching Parameters in a PSO. Proceeding of Genetic and Evolutionary Computation Conference 2006 (GECCO'06), eds. M. Keijzer, et al., p.3 - 9, ACM Press. 2006.
2. Bird D., Muoz C. Automatic generation of random self-checking test cases. IBM Systems Journal, 22(3):229-245, 1983.
3. Bouchachia A. An Immune Genetic Algorithm for Software Test Data Generation. Seventh International Conference on Hybrid Intelligent Systems. 2007. pp.84-89
4. Clerc M., Kennedy J. The particle swarm explosion, stability and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation. Vol 6,nro. 1, pp. 58-73. Feb.2002

5. Daz E., Tuya J., Blanco R. Automated Software Testing using a Metaheuristic Technique based on Tabu Search. 18 th IEEE International Conference on Automated Software Engineering. pp.310- 313. ISBN: 0-7695-2035-9. 2003
6. Kenedy J. and Eberhart R. Particle Swarm Optimization. Proceedings of IEEE International Conference on Neural Networks. Vol IV, pp.1942-1948. Australia 1995
7. Lanzarini L., Leza V., De Giusti A. Particle Swarm Optimization with Variable Population Size. Lecture Notes in Computer Science. Vol 5097/2008. Artificial Intelligence and Soft Computing ICAISC 2008. ISBN 978-3-540-69572-1. pp.438-449. June 2008
8. Lopez J., Lanzarini L., De Giusti A. Particle Swarm Optimization with Oscillation Control. Genetic and Evolutionary Computation Conference. ACM GEECCO Proceeding. Montral, Canada. July 2009.
9. Michael C., McGraw G., and M. A. Schatz. Generating software test data by evolution. IEEE Transactions on Software Engineering, 27(12):1085-1110, 2001.
10. Nieto J. Algorithms based on swarms of particles for solving complex problems. University Málaga. (In Spanish). 2006.
11. Offutt J. An integrated automatic test data generation system. Journal of Systems Integration, 1(3):391-409, November 1991.
12. Pargas R., Harrold M., Peck R. Test-Data generation using Genetic Algorithms. Journal of Software Testing, Verification and Reliability. Vol 9. pp.263-282.1999
13. Sagarna R., Lozano J. Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms. European Journal of Operational Research 169 (2006) 392412
14. Shi Y., Eberhart R. An empirical study of particle swarm optimization. Proceeding on IEEE Congress Evolutionary Computation. pp.1945-1949. Washington DC, 1999.
15. Shi Y., Eberhart R. Parameter Selection in Particle Swarm Optimization. Proceedings of the 7th International Conference on Evolutionary Programming. pp. 591-600. Springer Verlag 1998. ISBN 3-540-64891-7
16. Tracey N., Clark J., Mander K. Automated program flaw finding using simulated annealing. International Symposium on Software Testing and Analysis. 1998. pp. 73-81. ACM/SIGSOFT
17. Van den Bergh F. An analysis of particle swarm optimizers. Ph.D. dissertation. Department Computer Science. University Pretoria. South Africa. 2002