Performance Evaluation of GSA and PSO based Algorithms for Automated Test Data Generation for Software

Ankur Goel[1] Dr. Ashok Kumar[2]
[1,2]Department of Computer Science Engineering
[1,2]MMEC, MM University, Mullana, India

Abstract— In this Dissertation, we have implemented Meta-heuristic based search algorithms namely PSO and GSA algorithms for automatic test case generation using path testing criterion. For generation of test cases, symbolic execution method has been used in which first, target path is selected from Control Flow Graph (CFG) of Software under Test (SUT) and then inputs are generated using search algorithms which can evaluate composite predicate corresponding to the target path true. We have experimented on two real world programs showing the applicability of these techniques in genuine testing environment. The algorithm is implemented using MATLAB programming environment. The performance of the algorithms is measured using average test cases generated per path (ATCP) and average percentage coverage (APC) metrics. Number of individual or population size doesn't affect performance much but it should be between 20 and 40 but various parameters have to be tuned inorder to get good results.

***Key words*:** Fitness Function, testing, population.

I. INTRODUCTION

Software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation. Even after intensive research in software testing, the success stories are limited and hence there is tremendous scope of research/improvement. Problem of infeasible paths, unpredictability of test cases from specification domain of program, automatic test cases generation of complex programs, problems of testing of program containing loops, nested loops, pointers and global variables, applicability of test coverage criteria over large and complex programs, optimization of test cases are some research directions, which need to be addressed for. A latest technique of optimization is gravitational search algorithm. It is being implemented by many researchers in different fields where optimization is required. Even after employing metaheuristic techniques from last two decades in testing, no significant progress has been done in actual software testing environment. In research there are several papers which shows that metaheuristic techniques are employable but program specific behavior has to be adopted which restrict their use in general sense. In this paper has work to evaluate and compare the working of Gravitational Search Algorithm (GSA) and PSO based search algorithm for software test data generation. Applicability of BBBC (Big Bang Big Crunch) algorithm on software testing is demonstrated by [1] where authors have compared it with PSO algorithm. PSO performance is better than BBBC algorithm. But before declaring it as best algorithm for automated test case generation it should be tested against other algorithm such as GSA. In this synthesis, we are planning to evaluate the Gravitational Search Algorithm (GSA) algorithm and Particle Swarm Optimization (PSO) algorithm for automatic test case generation using symbolic execution based path testing criterion.

II. RELATED WORKS

Different types of search algorithms employed in testing research are divided into three classes. First class belongs to random testing based search algorithms. In this type of testing, random values are generated from domains of inputs and program is executed using these values. If these inputs are able to satisfy the testing criterion then they form a test case. Several tools combine random testing with other strategies. DART [7] combines random testing with symbolic execution. Agitator [9] combines several strategies: static analysis, random input generation, and heuristics to find data likely to expose bugs. In a recently published report [3], Andrews et al. state that the main reasons behind the so far poor adoption of random generation for object-oriented unit tests is the lack of tools and of a set of recognized best practices. The authors provide such a set of best practices and also compare the performance of random testing to that of model checking. They show that random testing produces good results both when used on its own and when used as preparation for model checking.

Another class of testing is algorithmic such as numerical maximization techniques by Korel [11] and domain reduction procedure used by Demillo et al for its fault based test data generator named as GODZILA [19]. Korel's technique is based upon the dynamic execution of software under test (SUT). On the other hand, Demillo et al used symbolic execution for testing purpose.

Last decades experienced a number of research attempts in automation of test cases generation [20, 21, 13 10, 6, 8, and 18], specifically heuristic approaches were used to fulfill the requirements of this activity.

Software testing objective-functions are very complex and usually non-linear for which heuristic approaches are the fit case to be used. In the past decades several soft computing based methods such as genetic algorithm (GA) [15, 18, 23, 22 and 16], simulated annealing (SA) [24], tabu search [2], ant colony optimization (ACO) [12, 5] were used to fulfill this requirement. Out of these techniques, GA has been the most successful and frequently used technique by researchers. Xanthakis [16] first time applied GA for automatic test case generation. He constructed an objective function for GA program using the concept of branch distance.

III. METHODOLOGY OF TESTING

Two components are essential for a problem which is to be modeled as search target. First a mechanism should be derived through which the problem is encoded in search

algorithm and second component is assessment of the suitability of solutions produced by search technique to guide the individuals for exploring search space.

For software testing purpose, as solution lies in searching inputs, every possible set of inputs represent the global population in search algorithm and selected inputs from this global set are represented by individuals in the population. Suitability of the individuals can be assessed by following a testing criterion for which a unique fitness function has to be defined. In structural testing, these criteria can be anything from all-statement-execution to all-path-coverage. We will use the all-path coverage criterion for our experimentation because one, it is the hardest to follow and second, in true sense, it is the real representative of structural testing. Very few test data generators have followed this criterion. The path testing method involves generation of test data for a target feasible path in such a way that on executing program, it covers all branches on that path. To cover a particular branch, the condition(s) at branch node must be satisfied by the test data, which directs the control flow of program to the next branch of the path. A path may contain several branches and in order to execute that path, all these branch-conditions must be evaluated true by the test data. Consequently, problem of path testing can be formulated simply as constraint satisfaction problem which should be analyzed and solved with the help of some search method by generating inputs in such a way that can satisfy all the branch constraints on the path. A valid test case is generated, which should execute the particular path by satisfying all of the boolean expressions included in that path.

A. Design of Fitness Function for Testing

In path testing approach a candidate solution (also called an individual) is used to evaluate constraint system of the target path. This evaluation can be dynamic as well as static. In dynamic analysis, a program is actually executed with values of the inputs and then fitness function determines the extent up to which it has satisfied the testing criterion, which becomes the fitness of the individual. On the other hand, static testing does not require the actual execution of program, but it symbolically executes a testing path as identified from CFG of program by using symbols instead of actual values. Symbols are replaced for variables in predicates or constraints of the entire target path and then this resultant constraint system is evaluated for fitness.

The extent up to which this constraint system is satisfied by the individual determines its fitness. This constraint system is also called composite predicate (CP). If CP is not evaluated to be true by an individual then all the constraints of a particular path are broken up in distinct predicates (DP). A distinct predicate is the one, which contains only one operator (a constraint with modulus operator is exception). Each DP is evaluated by taking values of its operands from candidate solution. If it is evaluated to be true then no penalty is imposed to candidate solution, otherwise candidate solution is penalized on the basis of branch distance concept rules as shown in table 4.1. If there is a branch predicate A < B which needs to be satisfied in order to follow the true branch of predicate node, where A and B are operands which may be atomic or hybrid and two or more candidate solutions (inputs' alternates) are contending to satisfy this predicate in an optimization technique then solution which is more close to satisfy the branch predicate is selected for subsequent solution generations.

Violated individual predicate	**Penalty to be imposed in case predicate is not satisfied**
A < B	A – B + ζ
A <= B	A – B
A > B	B – A + ζ
A >= B	B – A
A = B	Abs(A – B)
A ≠ B	ζ – abs(A – B)
A and B are operands and ζ is a smallest constant of operands' universal domains. In case integer it is 1 and in case real values it can be 0.1 or 0.01 depending on the accuracy we need in solution.	

Table 1: Fitness function of a branch predicate

To calculate the fitness of an individual in population, first, values from the individual are assigned to the symbols related to each input of the program. In CFG of a program, a node is either branch node if it can deviate the control flow of program from target path otherwise it is a non-branch node. Non branch nodes' statements are evaluated whenever they are encountered in the course of path traversal so that any future predicate dependency on internal variables can be addressed properly. The CP of each branch node corresponding to the target path is evaluated using the current values of inputs and any internal variable. If it is not evaluated true then DP(s) are extracted from CP one by one and fitness for each DP is determined by means of the branch distance concept as explained above. After this integrated fitness for whole of CP is determined by adding fitness values of two DPs, if they are connected by a conditional 'and' (&&) operator. If two DPs are connected by a conditional 'or' (||) operator then minimum fitness of two DPs is considered for the evaluation of whole CP fitness. Negation is resolved by propagating inward over variables and changing 'and' into 'or' and vice versa. If integrated fitness is zero or negative then CP is called evaluated or satisfied by the individual and search process for particular path is terminated otherwise search is allowed to proceed further. Algorithm for fitness function is given in figure 4.1

IV. TESTING SEARCH ALGORITHMS

Two of the metaheuristic search techniques which we have employed for our experimentation are described below with their respective algorithm workflow.

A. Gravitational Search Algorithm

Gravitational Search Algorithm (GSA) is another latest search algorithm developed by Rashediet.al in 2009. The GSA is based on the Newtonian law of gravity and the law of motion [17]. Here agents are considered as objects and their performance are based on its masses. All the objects attach to each other by a force called gravitational force and this force causes the global movement off all objects to object with heavier masses. The gravitational force between two particles is directly proportional to the product of their masses and inversely proportional to the square of the distance between them. The heavy masses will correspond to good solutions and move more slowly and conversely

light masses correspond to poor solutions and move towards heavy masses much faster. This guarantees the exploitation step of the algorithm.

In the GSA, consider a system with n masses in which position of the ith mass is defined as follows:

$$X_i = \left(x_i^1, \ldots x_i^d, \ldots x_i^n\right), i = 1, 2, \ldots, n \quad (4.1)$$

where x_i^d is position of the ith mass in the dth dimension and n is dimension of the search space. At the specific time 't' a gravitational force from mass 'j' acts on mass 'i', and is defined as follows: (4.1)

$$F_{ij}^d(t) = G(t)\frac{M_{pi}(t) \times M_{aj}(t)}{R_{ij}(t)+\epsilon}\left(x_i^d(t) - x_j^d(t)\right) \quad (4.2)$$

Where Mi is the mass of the object i, Mj is the mass of the object j, G(t) is the gravitational constant at time t, Rij (t) is the Euclidian distance between the two objects i and j, and is a small constant. The total force acting on agent i in the dimension d is calculated as follows:

Furthermore, the next velocity of an agent is a function of its current velocity added to its current acceleration. Therefore, the next position and the next velocity of an agent can be calculated as follows:

$$v_i^d(t+1) = rand_i \times v_i^d(t) + a_i^d(t) \quad (4.3)$$

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1) \quad (4.4)$$

The general steps of the gravitational search algorithm are given in Fig. 4.2

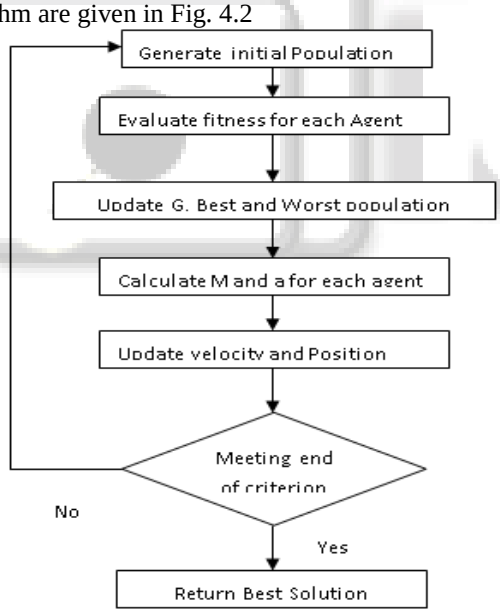

Fig. 4.2: Gravitational search algorithm work flow

B. Particle Swarm Optimization (PSO)

PSO is a population-based biologically inspired algorithm which applies to concept of social interaction to problem solving where each individual is referred to as particle and represents a candidate solution. Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995 [14], inspired by social behavior of bird flocking or fish schooling.

PSO shares many similarities with evolutionary computation techniques such as Genetic Algorithms (GA). The system is initialized with a population of random solutions and searches for optima by updating generations. However, unlike GA, PSO has no evolution operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles. The particle swarm optimization concept consists of, at each time step, changing the velocity of (accelerating) each particle toward its Pbest and Lbest locations (local version of PSO). Acceleration is weighted by a random term, with separate random numbers being generated for acceleration toward Pbest and Lbest locations.

$$v_{id}^{t+1} = w \times v_{id}^t + r_1 \times c_1 \times \left(p_{gd} - x_{id}^t\right) + r_2 \times c_2 \times \left(p_{id} - x_{id}^t\right) \quad (4.5)$$

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \quad (4.6)$$

where

- v_{id}^{t+1} is a velocity vector at t+1 time for i particle in dth dimension
- x_{id}^{t+1} position vector at t+1 time for ith particle in d dimension
- r_1, r_2 are random number generators.
- C_1 and C_2 are are learning rates governing the cognition and social components.
- p_{gd} represents the global best position in d dimension
- p_{id} represents the ith particle best position so far in d dimension.
- W is the inertia factor that dynamically adjust the velocities of particles gradually focusing the PSO into a local search.

C. PSO Algorithm Workflow

Following steps illustrate the overall optimization scheme of Global PSO.

1) Initialize the particle population by randomly assigning locations (X-vector for each particle) and velocities (V-vector with random or zero velocities- in our case it is initialized with zero vector)
2) Evaluate the fitness of the individual particle and record the best fitness Pbest for each particle till now and update P-vector related to each Pbest.
3) Also find out the individuals' highest fitness Gbest and record corresponding position Pg.
4) Modify velocities based on Pbest and Gbest position using eq3.
5) Update the particles position using eq4.
6) Terminate if the condition is met
7) Go to Step 2

In equation (4.5) above, new velocity at t+1 is generated with the help of global fitness which all the particles have achieved till iteration t. In this equation, (given in bold) is position given by the global best fitness in dimension d. Usually, global best fitness concept is expected to give a global search exploration possibilities in the search space.

V. EXPERIMENTAL SETUP

Testing experiments are done on two small but on real world benchmark programs showing the applicability and scalability of proposed technique in genuine testing environment. The algorithms have been implemented using MATLAB programming environment. The performance of the algorithms has been measured using average test cases generated per path and average percentage coverage metrics. All experiments are performed in the MATLAB framework. GATBX toolbox is used for the GSA and PSO algorithm implementation. In these experimentations, main aim is to fine tune the parameters of GSA and PSO algorithm to prove the usefulness and utility of algorithm for test case generation concept. We have taken two test objects; triangle classifier and line-rectangle classifier programs which are benchmark programs used frequently in testing literature. For triangle classifier and line-rectangle classifier programs, test data is generated from input variables by taking different domain; one very large of the size of the order of 105 and one small with a size of order of 103 for each path and experiment is conducted 100 times for averaging results. In each attempt, the GSA and PSO are iterated for 100 generations for each of 10 runs. In each run except of the first run, 1st generation population is seeded with the best solution from the previous run. This is done to check premature convergence of the population. Total number of real encoded individuals in each population is 30. Twenty percent of individuals in each population are taken from previous population. If a solution is not found within all runs that generates total 30,000 invalid test cases then it is declared that the test case generation process has failed for that particular attempt.

VI. RESULT AND DISCUSSION

Table 5.2 shows the experiments results. First and second column of each category programs register invalid average test cases per path (ATCP) and average percentage coverage (APC) of all paths for both programs in 100 attempts of test case generation for each path. Least value of ATCP and higher value of APC are desirable. If test case is not found for a particular path in one attempt then 30000 unsuccessful ATCP is registered for that path. This amount is calculated as multiplication of number of run, number of generation and total number of individual in one generation. 100 value of APC signifies that we are able to generate test cases for each of 100 attempts. Average is done for all paths and all attempts. So average value 1 in APC shows that program is able to generate test cases for each path in each attempt. There are several types of paths involving different types of path constraints with different complexity. This is the reason, why they are taking different amount of efforts for different path.

	PSO				**GSA**			
Triangle Classifier								
Path No	**Small Domain (10^2)**		**Large Domain (10^5)**		**Small Domain (10^2)**		**Large Domain (10^5)**	
	Test Coverage	**Percent Coverage**	**Test Coverage**	**Percent Coverage**	**Test Coverage**	**Percent Coverage**	**Test Coverage**	**Percent Coverage**
1	100	100	100	100	100	100	100	100
2	100	100	100	100	100	100	100	100
3	39182	100	147112	100	2843492	17	3000000	0
4	15161	100	93108	100	563200	89	2995234	3
5	12176	100	102020	100	497645	92	2974327	4
6	18143	100	98143	100	487542	90	2998991	3
7	100	100	100	100	100	100	100	100
	ATCP	APC	ATCP	APC	ATCP	APC	ATCP	APC
Avg.	122	100	630	100	6274	84%	17098	44%
Line-Rectangle Classifier								
Path No	**Small Domain (10^2)**		**Large Domain (10^5)**		**Small Domain (10^2)**		**Large Domain (10^5)**	
	Test Coverage	**Percent Coverage**	**Test Coverage**	**Percent Coverage**	**Test Coverage**	**Percent Coverage**	**Test Coverage**	**Percent Coverage**
1	100	100	100	100	100	100	100	100
2	18128	100	186195	100	3139	100	480167	100
3	279131	100	3000000	0	162120	100	3000000	0
4	318135	100	1695105	100	3000000	0	3000000	0
5	27150	100	732123	100	3000000	0	3000000	0
6	51104	100	1784110	100	3000000	0	3000000	0
7	18126	100	435156	100	81140	100	3000000	0
8	9134	100	6103	100	99102	100	3000000	0
9	42123	100	66144	100	765139	100	3000000	0
10	639187	100	447161	100	546111	100	3000000	0
11	9198	100	100	100	3132	100	100	100
12	100	100	100	100	12112	100	100	100
13	9123	100	3131	100	100	100	90162	100

14	100	100		6165	100		6140	100		100	100
15	24110	100		6132	100		100	100		60165	100
16	100	100		100	100		100	100		62341	100
17	100	100		100	100		8108	100		150100	100
	ATCP	APC		ATCP	APC		ATCP	APC		ATCP	APC
Avg.	850	100%		4922	94%		6286	82%		14613	52%

Table 5.1: Average Test cases generation per path (ATCP) and Average percentage coverage with both algorithms PSO and GSA

Following results can be interfered from the table.

- Algorithms' performance depends on input domain size. If domain is small the algorithm is able to find test case each and every time. But in large domain it sometime fails to generate test case and it is also taking more effort. The search algorithm is taking much time for those path constraints which involve equality constraints. But before making general statement, a further study needs to be done to find out the cause and effect relationship between nature of predicates and efforts made toward generation of test data.
- From table 5.2, it can be clearly deduced that Local PSO is performing better than Global PSO for test case generation. for different value of Number of Individual NIND, combination of inertia weight W, social learning rate C1 and cognitive learning rate C2 with value 0.5, 2.1 and 1.2 respectively give the excellent results.

ATCP for GSA is nearly five times that of PSO. It seems that GSA converges very fast to sub optimal targets and this restrict it to explore new areas making it unsuitable for those problems where we need exact solution. Another problem with GSA is its few parameters which do not control or tune algorithm properly. In small domain it found solution but with much effort as compared to PSO. Same treatment is for LRC program. In LRC program for those paths where equality constraints are involved GSA fails to find any solution or test cases for such paths especially in large domains. From these statistics it is clearly established that GSA is not a good algorithm for test data generation and PSO is really a very good algorithm as compared to GSA.

VII. Conclusion

From experiments it has been also observed that PSO performs better than GSA algorithm. Although APC coverage is more or less same in both of the algorithms in smaller domain but efforts taken are less in case of PSO thus it may save time in testing. During experimentation we observed that Meta-heuristic techniques exhibits domain dependency in test case generation, as these have to generate huge number of ATCP In larger domain. PSO is also taking more effort to generate test cases for larger domain especially when path constraint involves equality predicates. PSO is also taking more effort to generate test cases for larger domain especially when path constraint involves equality predicates. It has also been found that the performance of the PSO method is also not good for path constraints which are having large number of equality predicates. Hence, it has been observed to be ideally suited for test case generation problem where less equality predicates has to be solved. We need to work for equality constraints by finding new fitness functions to make metaheuristic solutions effective and efficient.

VIII. References

[1] Surender Singh and Parvin Kumar "Application of Big Bang Big Crunch Algorithm to Software Testing" International Journal of Computer Science and Communication Vol. 3, No. 1, January-June 2012, pp. 259-262(Impact Factor 0.48)

[2] Díaz E, Tuya J, Blanco R. "Automated software testing using a metaheuristic technique based on tabu search". In: The 18th IEEE international conference on automated software engineering, p. 310–3, 2003

[3] Srivastava P. R and Kim T, "Application of Genetic Algorithm in Software Testing," International Journal of software Engineering and Its Applications, vol. 3, no. 4, pp. 87-96, 2009.

[4] Mahanti PK, Banerjee S. Automated Testing in Software Engineering: Using Ant Colony and Self-Regulated Swarms. In Proceeding of Modeling and Simulation 2006.

[5] S. Kuppuraj and S. Priya, "Search Based Optimization for Test Data Generation Using Genetic Algorithms," in Proc of the 2nd International Conference on Computer Applications, 2012, pp. 201-205.

[6] Lin JC, Yeh PL. Automatic test data generation for path testing using GAs. Information Sciences 2001; 131:47–64.

[7] Berndt D., Fisher J., Johnson L., Pinglikar J., and Watkins A., "Breeding Software Test Cases with Genetic Algorithms", IEEE Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03).

[8] McMinn P. Search-based Software Test Data Generation: A Survey. Software Testing, Verification and Reliability June 2004; 14(2):105-156.

[9] Boshernitsan M., Doong R., and SavoiaA.. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pages 169–180, New York, NY, USA, 2006. ACM Press.

[10] Korel B, Automated software test data generation, IEEE transaction on software engineering, 1990; 16(8):870-879.

[11] Korel B., "TESTGEN-A structural test data generation system," Department of Computer

Science, Wayne State University, Detroit, MI, Tech. Rep. CSC-89-001, 1989.

[12] Huaizhong LI, LAM Peng C. An Ant Colony Optimization Approach to Test Sequence Generation for State based Software Testing, Proceedings of the Fifth International IEEE Conference on Quality Software (QSIC'05) 2005.

[13] Harman M and Jones BF, Search-based Software Engineering, Information & Software Technology, 43(14) 2001, pp. 833-839

[14] R.C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In Proceedings of the sixth international symposium on micro machine and human science, volume 43. New York, NY, USA: IEEE, 1995.

[15] Pargas RP, Harrold MJ, Peck R. Test-data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability 1999; 9(4):263–82..

[16] Xanthakis S, Ellis C, Skourlas C, Gall AL, Katsikas S and Karapoulios K. Application of genetic algorithms to software testing. In The fifth international conference on software engineering 1992; 625–36.

[17] EsmatRashedi, HosseinNezamabadi-pour, SaeidSaryazdi, GSA: A Gravitational Search Algorithm, Information Sciences 179 (2009) 2232–2248

[18] Roper M. Computer aided software testing using genetic algorithms. In 10th International Software Quality Week, San Francisco, USA, 1997.

[19] Demillo R. A., and Offutt A. J., Constraint-based automatic test data generation, IEEE transaction on Software engineering. Vol.17, No.9, September, 1991 pp. 900-910

[20] Díaz E, Javier T, Raquel B, José JD. A tabu search algorithm for structural software testing.Computers and Operations Research, 2007

[21] Duran JW, Ntafos AS Report On Random Testing, international Conference on Software engineering Proceedings of the 5th international conference on Software engineering 1981, San Diego, California, United States March 09 - 12, 1981

[22] Wegener J, Baresel A, Sthamer H., "Evolutionary test environment for automatic structural testing". Information and Software Technology 2001; 43:841–854,

[23] Watkins A, Hufnagel E. M. Evolutionary test data generation: a comparison of fitness functions. Software Practice & Experience 2006; 36:95–116

[24] Ferguson R, Korel B. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology, 1996; 5(1):63-86.

