International Journal of Information Technology and Knowledge Management (ISSN: 0973-4414)
July-December 2012, Volume 5, No. 2, pp. 489-493

# Empirical Evaluation of Metaheuristic Approaches for Symbolic Execution based Automated Test Generation

Surender Singh[1], Parvin Kumar[2]
[1]CMJ University, Shillong, Meghalya, (INDIA)
[2]Meerut Institute of Science & Technology, Meerut, UP (INDIA)
surendahiya@gmail.com, pk223475@yahoo.com

**Abstract:** This paper empirically evaluates four meta-heuristic search techniques namely particle swarm optimization, artificial bee colony algorithm, Genetic Algorithm and Big Bang Big Crunch Algorithm for automatic test data generation for procedure oriented programs using structural symbolic testing method. Test data is generated for each feasible path of the programs. Experiments on ten benchmark programs of varying sizes and complexities are conducted and the subsequent performance results are presented. All the four algorithms have been evaluated on average test cases per path and average percentage coverage per path. It has been observed that the particle swarm optimization based algorithm outperforms the other three algorithms. The result also concludes that predicates solving difficulty (such as constraints having equality operator'&&' as join operator) has a direct relationship with testing efforts rather than program complexity measures such as cyclomatic complexity, number of decision nodes etc.
**Keyword:** Software testing, Symbolic execution, Genetic Algorithm, Swarm Intelligence, Particle Swarm optimization, Artificial Bee Colony algorithm, Big Bang Big Crunch Algorithm.

## 1. Introduction

Although manual generation of test cases is relatively easy but it is a slow and costly process. Automatic generation of test cases can save time and testing resources. At the same time, it is also free from human biases and doesn't require special team of testers other than the developers. Despite having so many benefits, automated test case generation is not so easy because it requires intelligence of human mind to identify the non-linearity and discreteness in test inputs' search space. For improving the quality of automation and fulfilling the requirements of test case generation, many researchers have explored new soft computing based techniques such as genetic algorithm, simulated annealing, tabu search, ant colony optimization, particle swarm optimization, memetic algorithms etc. to fulfill testing requirement and to generate suitable test cases automatically [1,2].

## 2. Software testing using population based approaches

Search techniques are applied for generation of test data by transforming testing objective into search problem. 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. The population based metaheuristic search algorithm where global population represents every possible solutions and global search space, are frequently applied in applications where search space is very large.Each member of population is called an individual or a probable solution which is evaluated for its fitness so that new and better individual(s) may be generated. This process is iterated in algorithm till the search stopping criterion is met. Population based search algorithm-workflowsare comparatively depicted in Table 1.

For software testing purpose, as solution lies in searching inputs, every possible set of inputs represent the global populationin 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 [3]. The all-path coverage criterion has been chosen for experimentation because of its being the hardest one to follow and in true sense, it is the real representative of structural testing. The pathtesting 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. Figure 2 shows the different

building blocks of a path based automatic test data generator. First test object source code is fed to program instrumentation for CFG and node expressions generation. Subsequently CFG is used to generate all possible paths which are filtered manually for feasible path in order to become input to search algorithm. Node expressions include branch node predicates as well as non-branch node statements which are used to evaluate candidate solutions in test object fitness functions.

**Table 1.** Comparative Chart of algorithm showing population based algorithms

| | | | |
|---|---|---|---|
| 1. Generate the *M* number of solution string known as parent population<br>2. Evaluate fitness to each of the solution<br>3. Select some of the best fit chromosomes from parent population according some selection criteria<br>4. Crossover partial solution between pair of selected solution with some probability value to generate child population.<br>5. Change the value of an allele of child with some small probability value<br>6. Evaluate child population and replace parent population<br>Go to step 3 and repeat the process until termination criteria satisfies | 1:Initialize the random population of solutions $X_{ij}$ (flower patch positions)<br>2: Evaluate the population<br>3: Produce new solutions $Y_{ij}$ in the neighborhood of $X_{ij}$ for the employed bees by using following equation.<br>$Y_{ij} = X_{ij} + \varphi_{ij} * (X_{ij} - X_{kj})$--------(1)<br>Where $\varphi_{ij}$ is a random number between 0 to 1 and $X_{kj}$ is a randomly selected solution.<br>4: Apply the greedy selection process between $Y_i$ and $X_i$.<br>5: Calculate the probability values $p_i$ for the solutions $X_i$ by means of their fitness values,<br>$p_i = \frac{fitness_i}{\sum_{i=1}^{n} fitness_i}$----------(2)<br>6: Produce new solutions (new positions) for the onlookers from the solutions $X_i$ depending on probability $p_i$ and evaluate them.<br>7: Apply the greedy selection process between new and old solution<br>8: Determine the abandoned solution (source), if exists, and replace it with a new randomly produced solution for the scout.<br>9: Memorize the best food source position (solution) achieved so far<br>10: Repeat step 3 to 9 until stopping criterion is reached | 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)<br>2. Evaluate the fitness of the individual particle and record the best fitness $P_{best}$ for each particle till now and update P-vector related to each $P_{best.}$<br>3. Also find out the individuals' highest fitness $G_{best}$ and record corresponding position $p_g$.<br>4. Modify velocities based on $P_{best}$ and $G_{best}$ position using eq3.<br>5. Update the particles position using eq4.<br>6. Terminate if the condition is met<br>7. Go to Step 2 | 1. Create random population of solution.<br>2. Evaluate Solutions.<br>3. The fittest individual can be selected as the center of mass.<br>4. Calculate new candidates around the center of mass by adding or subtracting a normal random number whose value decreases as the iterations elapse.<br>5. The algorithm continues until predefined stopping criteria has been met |
| **GA Algorithm** | **ABC Algorithm** | **PSO Algorithm** | **BBBC Algorithm** |

## 3. Fitness Function Design for symbolic path 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.

**Table 2.** Fitness function of a branch predicate

| **Violated individual predicate** | **Penalty to be imposed in case predicate is not satisfied** | **Violated individual predicate** | **Penalty to be imposed in case predicate is not satisfied** |
|---|---|---|---|
| A < B | A – B + ζ | A >= B | B – A |
| A <= B | A – B | A = B | Abs(A – B) |
| A > B | B – A + ζ | 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. | | | |

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 upin 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 1. This method avoids premature convergence problem in GA optimization and has been already used by several researchers [4,5]. Ahmed *et al* [6] have proposed several techniques such as normalization, weighting and rewarding schemes for making fitness function effective and useful. Watkins *et al* [7] compared many fitness function construction techniques and concluded through their experiments that branch-distance based function is best performer in the static structural testing category. Algorithm for fitness function is given in Table 3.

**Table 3.** Fitness function used for symbolic testing

```
For each individual in population
        Assign input variables values from individual.
        For each node in target path of CFG
                If node is non-branch node
                        Execute all the statements related to that node;
                Else
                        Find the predicate of branch node.
                        Find the traversal link to next node in target path from CFG matrix.
                        If traversal link is for false execution of branch predicate
                                Then simplify predicate for negation.
                        End
                        If simplified predicate is evaluated true by the individual
                                Then continue without any penalty to individual solution
                        Else
                                Extract distinct predicates from combined branch predicate.
                                For each distinct predicate
                                        If distinct predicate is evaluated true by individual
                                                Then assign zero penalty corresponding to that distinct
predicate
                                        Else
                                                Determine Penalty w.r.t. distinct predicate by following
                                                the concept for branch distance function given in table1.
                                        End
                                End
                                Replace each distinct predicate with its corresponding penalty in composite
                        predicate.
                        End
                        Replace each '&&' symbol with plus (+) sign and 'll' symbol with comma (,) sign
                        and each bracket '(' with min([ in branch predicate to determine fitness related to
                        the combined node predicate.
                End
        Add fitness of each branch node predicate to determine the fitness for whole individual.
        End
End
```

## 4. Experimental Setup and Results

The algorithms are implemented using MATLAB programming environment. The performance of the algorithms is measured using average test cases generated per path (ATCPP) and average percentage coverage (APC) metrics. Experiment is conducted 10 times for averaging results. In each attempt, BBBC is iterated for 100 generations for each of 10 runs. In each run, except for the first run, first-generation population is seeded with the best solution from the previous run. This is done to check premature convergence of population. Total number of real encoded individuals in each population is 30. 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 has failed for that particular attempt. This value has been obtained by multiplying total number of runs, generations and number of individual in each population. An invalid test case is a solution, which does not qualify to become a test case.
Ten real world programs for test data generation activityhave been selected. Some of these are frequently used by researchers. These are called test objects here and brief explanation for each test object is given below.Detail characteristics of these test objects are given in table 4.

**Table 4.** Test Object characteristics

| Name of Program | Lines of Code | Cyclomatic Complexity | Number of Decision Nodes | Highest Nesting Level | Total Paths in CFG | Feasible Paths |
|---|---|---|---|---|---|---|
| **TC** | 35 | 07 | 06 | 05 | 07 | 07 |
| **LRC** | 56 | 19 | 18 | 12 | 17 | 17 |
| **DBTD** | 123 | 26 | 22 | 05 | 1643 | 566 |
| **A2F** | 48 | 15 | 14 | 07 | 910 | 568 |
| **BS** | 23 | 05 | 04 | 03 | 124 | 62 |
| **REM** | 35 | 10 | 8 | 04 | 22 | 22 |
| **BUB** | 21 | 04 | 03 | 03 | 121 | 31 |
| **QUAD** | 24 | 06 | 05 | 03 | 06 | 06 |
| **MINMAX** | 27 | 04 | 03 | 03 | 121 | 121 |
| **ISPRIME** | 16 | 03 | 02 | 02 | 10 | 08 |

## 5. Results and Discussions

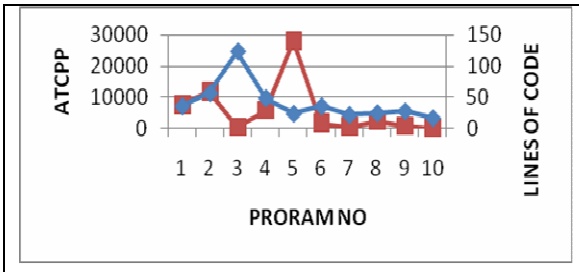
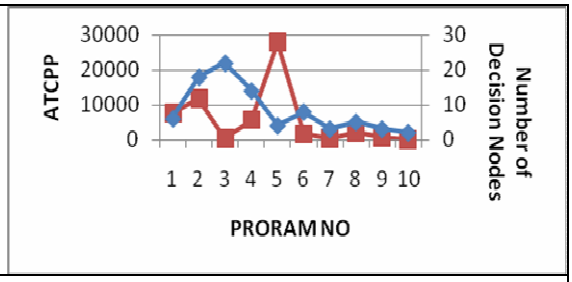
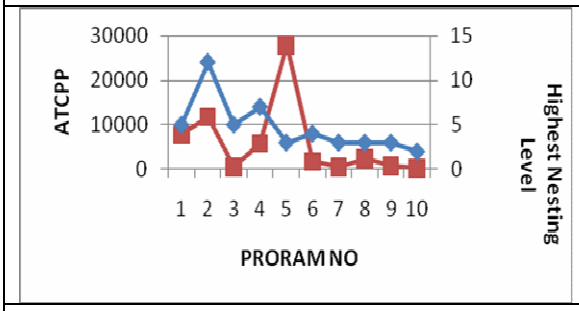
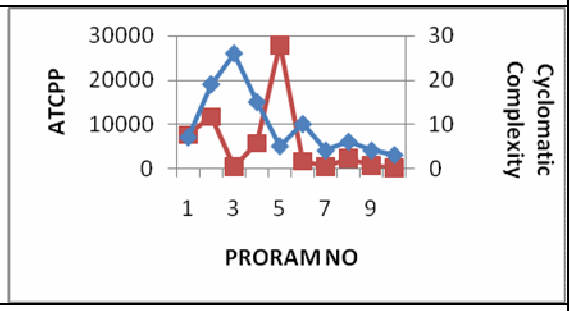
Table 5 presents the experimental results. From the table, it can be easily deduced that PSO has performed consistently better than other two search algorithms. It has generated less ATCPP as compared to GA and ABC. It has covered all paths in every attempt for each test object barring binary search program. Manual analysis by us has found that in this case problem is not with search algorithm but inability of handling of index value zero by MATLAB. Due to this those paths which require such test cases where middle index calculation should result into having a value zero could not be covered by experiments. If we remove such paths from the category of feasible path then APC in PSO is more than 90 percent. Hence we can say that PSO is a good search algorithm for fulfilling the testing requirement.

**Table 5.** Experimental results

| Name of Program | GA | | ABC | | PSO | | BBBC | |
|---|---|---|---|---|---|---|---|---|
| | **ATCPP** | **APC** | **ATCPP** | **APC** | **ATCPP** | **APC** | **ATCPP** | **APC** |
| **TC (small Domain)** | 1696 | 100 | 6197 | 85.71 | 187 | 100 | 829 | 100% |
| **TC (Large Domain)** | 5382 | 87.14 | 17156 | 42.86 | 2337 | 100 | 7564 | 83.32% |
| **LRC (small Domain)** | 1171 | 100 | 1255 | 100 | 275 | 100 | 2543 | 97.23% |
| **LRC (Large Domain)** | 11495 | 62.53 | 3924 | 89.06 | 607 | 100 | 11765 | 74.42% |
| **DBTD** | 2564 | 87.70 | 206 | 100 | 202 | 100 | 378 | 100 |
| **A2F** | 2341 | 100 | 3195 | 100 | 898 | 100 | 5743 | 100 |
| **BS** | 14289 | 56.61 | 15545 | 51.94 | 9343 | 75.32 | 28021 | 47.83 |
| **REM** | 524 | 100 | 970 | 100 | 518 | 100 | 1652 | 100 |
| **BUB** | 154 | 100 | 258 | 100 | 54 | 100 | 413 | 100 |
| **QUAD** | 15211 | 75 | 1930 | 100 | 1136 | 100 | 2247 | 100 |
| **MINMAX** | 973 | 100 | 619 | 100 | 225 | 100 | 721 | 100 |
| **ISPRIME** | 477 | 100 | 52 | 100 | 30 | 100 | 47 | 100 |

If we analyze the performance of ABC and BBBC then we can say that both compete with each other for performance but these are not anywhere near to GA and PSO for test data generation activity. BBBC performs worst in satisfying equality constraints based branch predicates especiallywhen inputs' domains are large.

**Table 6.** Correlation between testing efforts(ATCPP) and various parameters of programs

| | |
|---|---|
|  |  |
| **Correlation=**-0.1455 | **Correlation=**-0.02143 |
|  |  |
| **Correlation=**0.203183 | **Correlation=**-0.05386 |

Another interesting observation can be made by comparing test objects' characteristics and testing efforts made by search techniques. The figures in table 4, 5 and 6 give the impression that predicates solving difficulty(such as constraints having equality operatoror'&&' as join operator) has a direct relationship with testing efforts rather than program complexity measures such as cyclomatic complexity, number of decision nodes etc. but the statement needs more experimentation before generalization.

## 6. Conclusion

We have compared four heuristic based techniques BBBC, ABC, PSO and GA 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 CFG of SUT and then inputs are generated using search algorithms which can evaluate composite predicate corresponding to the target path true. We have experimented on ten real world programs showing the applicability of swarm intelligence techniques in genuine testing environment. PSO method has given excellent results for each test object except binary search method however it still outperforms other two methods. PSO's performance in small as well as large domains shows that it has both type of search capabilities; local as well as global for fulfilling testing requirements. Hence, it has been observed to be ideally suited for test case generation problem. A direct relationship between number of equality constraints and testing efforts has been identified.

## Reference

[1] Edvardsson J. A survey on automatic test data generation In *Proceedings of the second conference on computer science and engineering*, Linkoping: ESCEL; October 1999; 21–28.

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

[3] Amoui M, Mirarab S, Ansari A and Lucas C, A Genetic Algorithm Approach to Design Evolution using Design Pattern Transformation, International Journal of Information Technology and Intelligent Computing 1(2, 2006 pp. 235-244.

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

[5] Wegener J, Baresel A, Sthamer H., "Evolutionary test environment for automatic structural testing". Information and Software Technology 43, 841–54, 2001;

[6] Ahmed MA, Hermadi I. GA-based multiple paths test data generator. *Computers and Operations Research* (2007)

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