Use of Clonal Selection Algorithm as Software Test Data Generation Technique

Ankur Pachauri
Department Of Mathematics
Dayalbagh Educational Institute
Agra, India
ankurpachauri@gmail.com

Gursaran
Department Of Mathematics
Dayalbagh Educational Institute
Agra, India

Abstract— Clonal selection algorithm is an algorithm that belongs to the class of immune algorithm inspired form clonal selection principle of biological immune system. Initially clonal selection algorithm was designed for machine learning approach and was used in pattern recognition process of artificial intelligence. The other implementation of clonal selection algorithm is in the field of function optimization, which had gained a tremendous attention of the researchers. We too had used the clonal selection algorithm for the software test data generation technique for branch coverage with some modification according to the requirement.

Keywords- Software test data generation, Clonal Selection Algorithm, Path-prefix strategy

I. INTRODUCTION

Software testing is a very labor intensive work, and it consumes almost around 50% of the total cost of the software. Being costly, amongst all phases of development of that software, it is inevitable, as thorough testing give the reliability and confidence about the software. In 70’s, the task of software testing was done manually and statically by the process termed as symbolic execution [1, 2]. In 90’s korals suggested the technique of dynamic test data generation [3]. From then onward many techniques had been developed for test data generation and automated test data generation (this is not the topic of discussion for this paper). Many algorithms were developed while other was inspired by some natural phenomenon. Amongst the inspired algorithms, clonal selection algorithm appeared during the starting of twentieth century, and is explained in the next section.

The paper is further organized as follows in section 2 a general introduction to clonal selection algorithm and its application is described. In section 3 the method of test data generation with search based technique is explained along with the representation and fitness function. In section 4 test data generation technique using clonal selection algorithm is explained. Experimental setup and results are discussed in section 5 and section 6 presents the conclusion and future prospect of the proposed method.

II. CLONAL SELECTION ALGORITHM

Clonal selection algorithm is an algorithm, belongs to the class of immune algorithm inspired from clonal selection principle of biological immune system. It is a population based stochastic method. The variables are represented in binary string format. Clonal selection algorithm has the capability of optimizing multimodal functions and it maintains local solutions. Clonal selection algorithm can exploits and explore the solution space in parallel and in effective manner, but it lacks with poor convergence rate and difficulties in achieving high quality solutions in less time bound. Initially clonal selection algorithm was designed for machine learning approach and was used in pattern recognition process of artificial intelligence [4]. The other implementation of clonal selection algorithm is in the field of function optimization, which had gained a tremendous attention of the researchers. A detailed survey of clonal selection algorithm, its application and various modified version of clonal selection algorithm is given by [5], but none had worked for automated software test data generation in the literature.

During the past few years Software test data generation technique is modified into search based software test data generation technique, [6, 7, 8, 9], which converts the software test data generation technique into a search problem, which in turn, changes into function minimization problem. We too had used the clonal selection algorithm for the test data generation technique for branch coverage with some modification according to the requirement, as originally proposed by castro and zuben [4].

III. TEST DATA GENERATION FOR BRANCH COVERAGE USING METAHEURISTIC TECHNIQUE

In this section we first describe in a general way the steps for test data generation with search based technique or metaheuristic technique.

Let \( P \) be the program under test, then a general sequence of steps for test data generation using a metaheuristic technique is described in Figure 1.

1. Choose an appropriate test adequacy criterion. This in our case is the branch coverage criterion.
2. Setup the metaheuristic technique.
   a. Select a representation for test data to be input to program \( P \).
   b. Define a fitness function.
c. Instrument the program \( P \) to create program \( P_t \). The instrumented program \( P_t \) is used directly for test data generation.

d. Select suitable metaheuristic technique parameters.

3. Generate test data.
   a. Run the metaheuristic technique for test data generation using \( P_t \) for fitness computation.
   b. Identify and eliminate infeasibility.
   c. Regenerate test data if necessary.

**Figure 1 Steps for Test Data Generation using a Metaheuristic Technique**

Test data is generated to meet the requirements of a particular test data adequacy criterion. The criterion in our case is the branch coverage criterion. The setup phase begins with the choice of a suitable representation for test data and the identification of a suitable fitness function. The inputs for one execution of \( P \), i.e., a single test data, are represented in a binary string also called a *binary individual*. For instance, if the input to \( P \) is a pair of integers \( x = (I_1, I_2) \), then this pair is represented as shown in Figure 2. The length of the substrings representing \( I_1 \) and \( I_2 \) are chosen to represent the largest legal value that can be input to \( P \). The length of the complete string is the sum of the lengths of the two substrings.

**Figure 2 Binary string representations.**

The fitness of a binary individual is computed as

\[
\text{Fitness}(x) = \text{Approximation Level} + \text{Normalized Branch Distance}
\]

As opposed to the usual practice of formulating the generation problem as a fitness minimization problem, in this work it is taken to be a maximization problem. The definition of approximation level and normalized branch distance is also different from [6] although the basic idea is similar. This was done with the view to examine new fitness functions. A *critical branch* [3], as defined earlier, is a branch that leads the execution away from the target branch in a path through the program.

The *approximation Level* is a count of the number of predicate nodes in the shortest path from the first predicate node, from the start node, in the flow graph to the predicate node with the critical branch. See Figure 3 for an example. The shortest path is chosen to avoid loops and take care of multiple paths that may be followed to reach the critical branch. The *Normalized Branch Distance* is calculated according to the formula

\[
\text{Normalized Branch Distance} = \frac{1}{1.001 \times \text{distance}}
\]

where, *distance*, or *branch distance*, as defined in [10, 11], is computed at the node with the critical branch using the values of the variables and constants involved in the predicates used in the conditions of the branching statement. However, the definition of normalized branch distance is different from the definition of [6] as the problem is a maximization problem. Table 1 shows the computation of branch distance for different conditions. Entries one through five are the same as in [3].

Table 1 also describes the computation of branch distance in the presence of logical operators **AND** (&&) and **OR** (||). In both these cases, the definition takes into account the fact that branch distance is to be minimized whereas the fitness is to be maximized. Figure 3 illustrates branch distance computation for different cases.

<table>
<thead>
<tr>
<th>Decision Type</th>
<th>Branch Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 a &lt; b</td>
<td>a – b</td>
</tr>
<tr>
<td>2 a &lt;= b</td>
<td>a – b</td>
</tr>
<tr>
<td>3 a &gt; b</td>
<td>b – a</td>
</tr>
<tr>
<td>4 a &gt;= b</td>
<td>b – a</td>
</tr>
<tr>
<td>5 a == b</td>
<td>Abs(a – b)</td>
</tr>
<tr>
<td>6 a != b</td>
<td>Abs(a – b)</td>
</tr>
<tr>
<td>7 a &amp;&amp; b</td>
<td>a + b</td>
</tr>
<tr>
<td>8 a</td>
<td>b</td>
</tr>
</tbody>
</table>

In general, in order to generate test data to satisfy the branch coverage criterion using a clonal selection algorithm, the sequence in which the branches will be selected for coverage must be defined. A chosen branch may become difficult to cover if the corresponding branch predicate is not reached by any of the test data or individuals in the current population. One of the proposals made by Pachauri and Gursaran [12] for sequencing is the *path prefix strategy*. We adopt this strategy for the experiments described in this paper. Further, each time a branch is traversed for the first time, it may be necessary to store the test data that traverse the branch and inject these into the population when the sibling branch is selected for traversal. This is referred to as *memory* and is used in this paper. In order to ensure that individuals reaching the sibling branch of the target are not destroyed by the clonal
selection algorithm operators, elitism is adopted. Up to 10% of fit individuals, with a minimum of one individual, are carried forward to the next generation. Furthermore, it is also possible to initialize the population each time a new branch is selected for coverage or leave it uninitialized. In the experiments described in this paper, the population is not initialized.

Infeasibility may prevent test data from being generated to satisfy C. It may be dealt with as follows. If the search is attempting to traverse a particular branch, but is unable to do so over a sufficiently large, predetermined, number of iterations, then the search run is aborted and the branch is manually examined for infeasibility. If the branch is found to be infeasible then it is marked as traversed and the search is rerun.

In Figure 1, Step 2d involves the selection of appropriate parameter values for the chosen metaheuristic technique and step 3a involves an application of the metaheuristic technique. Implementation of Step 3a is described in next Section

IV. CLONAL SELECTION ALGORITHM FOR TEST DATA GENERATION

The modification in clonal selection algorithm that suits the process of test data generation is described in figure 4. Several adjustments are made in clonal selection algorithm explained in the caspto and zuben [4]. The algorithm suggested by them was primarily for function optimization process, so few changes are made for using clonal selection algorithm for test data generation techniques.

1. Start with Initial random population.
2. While (all branches are not covered)
   
   3. Evaluate the affinity of each an antibody
   4. Select a branch as target branch to be covered using Path-prefix Strategy.
   5. While (target branch not traversed)
      
      6. Generate two clones of each individual.
      7. Hypermutate each clone.
      8. Reselect individuals from the pool of parents and clones.
      9. Replace individuals with poor affinity.
   } end while
   } end while

Figure 4 clonal selection algorithm for test data generation

The algorithm starts with an initial random population. The initial population is evaluated for fitness of each individuals (from here an antibody will be used in place of individuals and affinity is used with fitness value). After the fitness evaluation step, target branch is selected using the path-prefix strategy explained by prather and myers in [13]. On selecting the target branch, the target branch along with the pool of antibodies is given to clonal selection algorithm to find an antibody covering the desired target branch. The steps of clonal selection for the test data generation technique starts with generating the clones, in this case two clones of each an antibody are generated. In the next step the clones are hypermutated. Hypermutation of clones is explained later. After hypermutation of clones, the antibodies equals to the population size of algorithm are selected from the pool of parents and the hypermutated clones with the binary tournament selection method. Then 10% of antibodies, with lower affinity, are replaced by newly created random antibodies. The loop continues, until an antibody traversing the selected target branch is not created.

The process of hypermutation differentiate it from other evolutionary search algorithm especially from the genetic algorithm, as it does not include crossover operator, it has mutation operator only, and that too implemented in a different manner than genetic algorithm. The process of hypermutation can be in many ways. Every clone is mutated either with the fixed number of bits to be mutated or with the fixed probability of mutation. In the first method the number of bits to be mutated are fixed, with the minimum number of one bit, to some percentage of length of chromosome but in the second method all depend on the probability of mutation, which is kept low, ranges from 0.01 to 0.04 only.

V. EXPERIMENTAL SETUP AND RESULTS

Few experiments were conducted to tune the parameters for the steps of hypermutation; reselection and replacement (steps 7, 8, 9 of figure 4) of antibodies with poor affinity, but much of the experimentation were done for the hypermutation step. Experiments were conducted on the benchmark program (Sthamer Triangle classifier problem) from [14, 15]. This program classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral, right angle triangle or scalene. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It’s CFG has 29 Nodes with 13 predicate nodes. The maximum nesting level is 12. It has equality conditions with AND operator and complex relational operators.

For each population size, hundred experiments (or runs) were carried out and the following statistics were collected.

- Mean number of generations. It may be noted that the termination criterion for each experiment is either full branch coverage or 107 generations whichever occurs earlier. The number of generations to termination over hundred experiments is used to compute the mean. The mean does not tell us if all the branches were covered.
- Mean percentage coverage achieved.
Table 2 Experimental settings

<table>
<thead>
<tr>
<th>Parameter/ Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>6, 10, 16, 20, 26, ..., 110.</td>
</tr>
<tr>
<td>Mutation Probability</td>
<td>Varies on the basis of affinity</td>
</tr>
<tr>
<td>Number of point mutation</td>
<td>Varies on the basis of affinity</td>
</tr>
<tr>
<td>Selection Method</td>
<td>Binary tournament</td>
</tr>
<tr>
<td>Fitness Function</td>
<td>AL&amp;NBD, approximation level with normalized branch distance</td>
</tr>
<tr>
<td>Population Initialization</td>
<td>Initialize once at the beginning of the run</td>
</tr>
<tr>
<td>Population Replacement</td>
<td>Replacement of 10% antibodies with poor affinity</td>
</tr>
<tr>
<td>Maximum Number of Generations</td>
<td>$10^7$</td>
</tr>
</tbody>
</table>

(a) The first pilot experiment generates the initial random population and the steps are same as explained in figure 4 except the step 7. Here one bit of each clone is mutated irrespective of its affinity. So that each clone is different from the parent but this strategy doesn’t work as can be seen form the graphs of figure 5 and 6.

(b) For the second experiment, clones are hypermutated on the basis of mutation probability ($p_m$) based on their affinity, i.e. $p_m$ is set to be 0.01 for top 20% of antibodies, 0.02 for the next 20%, 0.03 for the next 30% and 0.04% for all the rest antibodies. Figure 5 and 6.

(c) For the third experiment, clones are hypermutated by numerous point mutations, which are based on their affinity. Number of point mutation is set to be 3% of length of chromosomes for top 20% of antibodies, 4% for the next 20%, 5% for the next 30% and 7% for all the rest antibodies. Figure 5 and 6.

Figure 5 shows the mean number of generations for (a), (b) and (c) mentioned above and Figure 6 shows the mean percentage coverage for different population sizes and all the three strategies. As can be seen form the figure 5. Strategy explained above does not give expected results neither in terms of generations consumed nor with the mean percentage coverage, but figure shows slight improvement from (a) to (b) and marginal improvement from (b) to (c).

However the thread of (b) in figure 6 shows sufficient improvement in mean percentage coverage. This give rise to the chances for some more modification either with some type genetic material exchange mechanism or some type of local search to be implemented to achieve full branch coverage and with reasonable lower number of generations consumed.

VI. CONCLUSION AND FUTURE WORK

Although Clonal selection algorithm has the capability of optimizing multimodal functions and maintaining local solutions and furthermore Clonal selection algorithm can exploits and explore the solution space in parallel and in effective manner, but it lacks with poor convergence rate and difficulties in achieving high quality solutions in less time bound. Such problem can be sorted out on embedding some sort of neighborhood search along within. Inspite the clonal selection algorithm is used as automated software test data generation technique, but it does not give good results either in form of mean number of generations or in terms of mean percentage coverage. These results can further be improved on embedding some type of local search along with hypermutation. The effect of local search might improve the results with respect to mean number of generations consumed and mean percentage coverage.

REFERENCES


(a<=0 || b<=0 || c<=0)
Target Missed
Approximation Level 0
Branch Distance = Min ((a-0), (b-0), (c-0))
(a>10000 || b>10000 || c>10000)
Target Missed
Approximation Level 1
Branch Distance = Min ([[10000-a], (10000-b), (10000-c)])
(c==a+b)
Target Missed
Approximation Level 2
Branch Distance = Abs(c-(a+b))
(a=b & & b<=c || c<=2)
Target Missed
Approximation Level 3
Branch Distance = Min ((a-b) + (b-c), (c-2))
(b!=a+c)
Target Missed
Approximation Level 4
Branch Distance = -Abs(b - (a+c))

Figure 3 Approximation Level and Branch Distance