Chapter V

A Combinational Approach to Fitness Computation for Program Test Data Generation using Genetic Algorithm

5.1. Introduction

In program-based testing, flag variables, enumeration variables, unstructured control flow and state behaviour make search difficult. For example, in the program below,

1. if ( d== 0 )
2. flag = 0;
3. if (flag)
4. result = 0;
5. else
6. result = n / d;

Statement 4 can be executed only when the value of \( d \) is zero. An objective function based on the branch condition in line 3 gives no guidance for search and in such cases, metaheuristic search performs no better than random search (McMinn and Holcombe, 2006). Program transformations, substitution by predicates used in assigning values and application of an extended chaining approach have been explored (Baresel and Sthamer, 2003; McMinn and Holcombe, 2006; Wappler et al., 2007). Taking an alternate approach, (Baresel, et al., 2002) has explored the design of fitness functions with the object of guiding search. In this chapter we describe a combinational fitness computation approach that combines branch and execution path details. The computation is based on the dynamic identification of a target path based on the program elements that must be covered, e.g., statement 2 and branch (3,4) in the program fragment above, and the actual execution path. Initial experiments indicate that this approach may result in better performance in terms of number of generations required and the coverage achieved as compared to pure path based and branch based approaches.
5.2. Test data generation using GA

Before one can use a metaheuristic technique, such as genetic algorithm, to generate test data for a program \( P \) under test, it must go through a setup process. The steps in this process are:

1. Choose an appropriate test adequacy criterion.
2. Select a representation for test data to be input to program \( P \).
3. Define a fitness function.
4. Instrument the program \( P \) to create program \( P_t \). The instrumented program \( P_t \) is used directly for test data generation.
5. Select suitable genetic algorithm operators and parameters.

Test data is generated to meet the requirements of a particular test data adequacy criterion. In our case the criterion is the branch coverage criterion. The next step is selecting a suitable representation for test data. Each test data, i.e., the inputs \( x_1, ..., x_n \), for one execution of \( P \), is represented as a concatenated sequence of fixed lengths binary representations of \( x_1, ..., x_n \). The length of the binary substring representing \( x_i \) is chosen to represent the largest legal value that can be input to \( P \). The fitness of each individual is computed by first determining a target node sequence and then computing a fitness function. This is described in detail in Section 5.2.1. The choice of suitable GA operators and parameters is important as it can significantly affect GA performance. This is usually determined through experimentation.

Once the setup is complete, the GA can be executed for test data generation. It is possible that infeasibility may prevent complete coverage from being achieved. This must be identified and eliminated. One simple scheme for dealing with infeasibility is 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. How this may be done is not discussed here. If the branch is found to be infeasible then it is marked as traversed and the search is rerun. After a traversal of all the branches has been attempted, the test data generation process outputs the test set together with the percentage of test branches that are satisfied by the test set.

5.2.1. Fitness Computation

In this section we describe the fitness computation procedure. Fitness computation is a two-step process. In the first step a target node sequence is computed dynamically and in the
second step the actual execution path is compared with the target node sequence to compute fitness.

5.2.1.1. Preliminaries

A control flow graph (CFG) of a program $P$ is a directed graph $G = (N, E, s, e)$ where $N$ is a set of nodes, $E$ is a set of edges, and $s$ and $e$ are unique entry and exit nodes to the graph. Each node $n \in N$ corresponds to a statement in the program, with each edge $e = (n_i, n_j) \in E$ representing a transfer of control from node $n_i$ to $n_j$. Nodes corresponding to decision statements are referred to as branching nodes. Outgoing edges from these nodes are referred to as branches. The predicate determining whether a branch is taken is referred to as a branch predicate. A path through a CFG is a sequence $(n_1, n_2, \ldots, n_m)$ of nodes such that for $i$, $1 \leq i \leq m$, $(n_i, n_{i+1}) \in E$. A program path is a path that begins at the entry node $s$ and terminates at the exit node $e$. A node sequence through a CFG is a sequence $(n_1, n_2, \ldots, n_k)$ such that for $i$, $1 \leq i \leq k$, there is a path in the CFG with $n_i$ as the start node and $n_{i+1}$ as the end node of the path, i.e., $n_{i+1}$ is reachable from $n_i$ in the CFG. An $I$-program path is the program path traversed on executing $P$ on the encoded inputs in an individual $I$, from the current population, in a GA run.

The strategy described in this section can be used for def-use coverage, path coverage and branch coverage with dependencies taken into account. Here we describe the strategy in the context of branch coverage with dependencies. A branch is said to be dependent on a node or a node sequence if the execution of corresponding statement or statement sequence is necessary for the branch to be traversed, i.e., the corresponding branch predicate takes the necessary truth value for the branch to be traversed only if the statement or statement sequence is executed. The following code fragment taken from (McMinn and Holcombe, 2006) illustrates this situation. In the code fragment, the true branch (6,7) is traversed if node (3) is.

```c
void flag_example(int a, int b)
{
    int flag = 0;
    if (a == 0)
        flag = 1;
    if (b != 0)
        flag = 0;
    if (flag)
```
5.2.1.2. Computation of Target Node Sequence

When a genetic algorithm is used to generate test data, in each generation, distinct I-program paths may be traversed by the individuals in the population. The fitness of an individual must be computed on the basis of how close is the I-program path to covering the target branch and the nodes or the node sequence on which it is dependent. Since there is a node-branch sequence that must be covered, this first requires the computation of a target node sequence, or possibly a target path, that includes the target branch, dependent nodes and as many nodes as possible in the I-program path under consideration. The I-program path is then compared with the computed target node sequence to determine fitness. It may be noted that the target node sequence may be different for each individual in the population since nodes from the individual’s I-program path need to be considered. This makes the target node sequences dynamic and avoids the use of fixed program paths to compute fitness.

We now describe the computation of the target node sequence. The algorithm begins with the node sequence \( A = (s, \text{node sequence on which the branch to be covered depends}, \text{end nodes of the branch to be covered}, e) \) and then fills in as much of the I-program path under consideration so as to obtain a valid node sequence that begins with \( s \) and ends with \( e \). Let \( G \) be the CFG of the program under test. Let the target branch be \((b_1, b_2)\) and the node sequence on which this branch depends be \( (d_1, \ldots, d_k), \) \( k \geq 1, \) and \( d_i \) are all distinct for \( i = 1, \ldots, k. \) Let the I-program path under consideration be \( I = (n_1 = s, n_2, \ldots, n_p = e) \) and let \( T \) be the target node sequence under construction. The algorithm for constructing the target node sequence is given in Figure 5.1. This is followed by a detailed example. The algorithm uses node sequence \( A = (a_i, a_2, \ldots, a_q) \) which is initialized as \( (s, d_1, \ldots, d_k, b_1, b_2, e) \) in Step 1 to construct the target node sequence. Thus \( A \) includes all the nodes that must be present, in the given sequence, in the target node sequence. In Step 2, the \( T \) is initialized to the empty sequence. In Step 6, in iteration \( i, \) we select a maximal sub-path \( J = (n_6, \ldots, n_g) \) in \( I, \) to fill in between \( a_i \) and \( a_{i+1}. \) If the I-program path \( I \) includes either or both \( a_i \) and \( a_{i+1}, \) then accordingly we will have \( a_i = n_f \) or \( a_{i+1} = n_g. \) On the other hand if both \( a_i \) and \( a_{i+1} \) are not covered by \( I \) then \( (a_i, n_f, \ldots, n_g, a_{i+1}) \) is a valid subsequence in \( G. \) In Step 7 head \( (a_i, n_f, \ldots, n_g, a_{i+1}) \) is appended to \( T, \) where head \( (n_1, \ldots, n_k) \) gives \( (n_1, \ldots, n_{k-1}). \) Thus minimally, \( a_i \) is appended to \( T \) and the last element in
the list, which is not \( a_i \), is left out. Further, by appending \( e \) to \( T \) in step 9, it is ensured that \( T \) is minimally \( A \).

1. \( A = (s, d_1, \ldots, d_k, b_1, b_2, e); \quad // A=(a_1,a_2,\ldots,a_q) \)
2. \( T = ( ); \quad // \text{initialize to empty} \)
3. for \( i = 1 \) to \( q-1 \)
4. \{ \)
5. \( J = ( ); \)
6. Find maximal sub-path \( J=(n_{i_1}, \ldots, n_{i_g}) \) in \( I \), not already included in \( T \) such that \((a_i \neq n_{i_1}, \ldots, n_{i_g} \neq a_{i+1})\) is a valid subsequence in \( G \). Here ‘?’ is interpreted as ‘=’ if \( a_j, j=i,i+1 \), is in \( I \) or ‘,’if \( a_j \) is not included in \( I \); Let \( K = (a_i \neq n_{i_1}, \ldots, n_{i_g} \neq a_{i+1}) \).
7. Append subsequence head\((K)\) to \( T \);
8. } \)
9. Append \( e \) to \( T \);

Figure 5.1 Algorithm for constructing the target node sequence.

The target \( T \) so constructed is a node sequence and may not be a path. This thus avoids comparing the \( I \)-program path with a path that may not be feasible. Furthermore, the algorithm takes into account \( I \)-program paths that may include loop traversals. Consider the five cases shown in Figure 5.2. The portion of the program flow graph covered by an \( I \)-program path is shown for each case in the figure. In cases 1, 2 and 3, \( d_i \) and \( d_{i+1} \) are not included in the \( I \)-program path. In all the cases, a maximal node sequence from \( d_i \) to \( d_{i+1} \) will include all the loop iterations in the \( I \)-program path. For instance, in case 1, if the \( I \)-program path is \((s, u, u, \ldots, v, w, e)\) then the maximal node sequence included from \( d_i \) to \( d_{i+1} \) will be \((u, u, \ldots, v)\). In case 2 if the \( I \)-program path is \((s, u, z, u, z, u, v, w, e)\) then the maximal node sequence included from \( d_i \) to \( d_{i+1} \) will be \((u, z, u, z, u, z)\). In case 3 if the \( I \)-program path is \((s, u, z, u, z, u, v, w, e)\) then the maximal node sequence included from \( d_i \) to \( d_{i+1} \) will be \((z, u, z, u, z, u, v)\). In case 4, if the \( I \)-program path is \((s, u, u, \ldots, u, v, w, e)\) with \( u \) occurring \( n \) times and \( d_i \) and \( d_{i+1} \) occurring \( n-1 \) times then the maximal node sequence included from \((\text{first})\) \( d_i \) to \((\text{last})\) \( d_{i+1} \) will be \((d_{i+1}, u, d_i, u, d_{i+1}, \ldots, u, d_i)\) with \( u \) occurring \( n-2 \) times and \( d_i \) and \( d_{i+1} \) occurring \( n-2 \) times each. In case 5, a maximal path from \( d_i \) to \( d_2 \) will cover all iterations up to the last. The sub-path of \( I \) traversed in the last iteration will then be used to fill in the node sequence between \( d_2 \) through \( d_k \).
An Example

The construction of target node sequence can be explained by an example. Consider the program and its CFG in Figure 5.3. Let the target branch be (16, 17). The dependent nodes for this branch are 3 and 9. Let the I-program path under consideration be $I = (s, 0, 2, 4, 5, 7, 8, 10, 11, 12, 13, e)$. To compute the target node sequence, we first consider the node sequence $(v, 3, 9, 16, 17, e)$. Subsequent steps are as follows:

- Consider pair $(s, 3)$. A valid subsequence from $I$ that is selected is $(s, 0, 2)$. This gives the sequence $(s, 0, 2, 3)$ from which head$(s, 0, 2, 3)$ gives $(s, 0, 2)$ as the target node sequence.

- We now consider $(3, 9)$. Considering nodes in $I$, the target node sequence is augmented to $(s, 0, 2, 3, 5, 7, 8)$. 

- Now considering the pair $(9, 16)$, we get $(s, 0, 2, 3, 5, 7, 8, 9, 11)$ as the new target node sequence.

- Now consider $(16, 17)$. Since I deviates at 11 taking a critical branch, the target node sequence is now $(s, 0, 2, 3, 5, 7, 8, 9, 11, 16)$

- From $(17, 25)$ we get $(s, 0, 2, 3, 5, 7, 8, 9, 11, 16, 17)$ as the new target node sequence.
• From (25,e) we get \(s,0,2,3,5,7,8,9,11,16,17,25\).

• Finally appending \(e\), we have \(s,0,2,3,5,7,8,9,11,16,17,25, e\) as the target node sequence for \(I\).

5.2.2. The Fitness Function

Fitness computation is based on a combination of two criteria

1. Branch distance and approach level. This is based on the proposal of Pachauri and Gursaran (Pachauri and Gursaran, 2012a)

2. Path similarity. This is based on the description by Lin and Yeh (Lin and Yeh, 2001). Also described in Section 2.3.1.

The fitness of an individual \(x\), using criteria (1) above only, is computed as in (McMinn, 2004)

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

A critical branch (Korel, 1990a) is a branch that leads the execution away from the target in a path through the program. The approach level is a count of the number of branching nodes in the shortest path from the node \(s\), in the flow graph to the branching node with the critical branch. 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

\[
\text{Normalized branch distance} = \frac{1}{(1.001^{\text{distance}})}
\]

Table 5.1 Branch distance computation.

<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></td>
</tr>
</tbody>
</table>

where, distance, or branch distance is computed at the first node, from node \(s\), with the critical branch. Table 5.1 shows the computation of branch distance for different conditions.

Lin and Yeh (Lin and Yeh, 2001) have extended the Hamming distance function to measure the distance between two given paths and then to compute the similarity measure that determines the fitness of the currently executed path with respect to the target path. Given a target path and the (current) actual path traversed, the similarity between them is calculated
from \( n \)-order sets of ordered and cascaded branches for each of the paths. For each order, the
distance between the set for each path is given by the symmetric difference between them.
This distance is normalized and the similarity between these \( n \)-order sets is found by
subtracting the value 1 from this normalized distance. The total similarity, denoted
SIMILARITY, between two paths is the sum of the similarities of all \( n \)-order sets, each one
associated with a weighting factor which is usually found by experience. The exact
computation of SIMILARITY is elaborated in Section 2.3.1.

In order to bring the traversed I-program path closer to the target node sequence, the fitness
of an individual is computed using path similarity, normalized branch distance and approach
level. Similarity considers the closeness if the whole I-program path with the target node
sequence and the latter two are used to correct deviations from the target sequence. Thus the
fitness of an individual \( x \) is computed as:

\[
\text{Fitness}(x) = \left( \text{normalized branch distance} + \text{normalized SIMILARITY} \right)/2 + \text{approach level}
\]

5.3. Experiments

Experiments were carried out to determine the performance improvement with the proposed
fitness computation scheme. The program used for experiments is given in Figure 5.3. Dependency
relationships in the program are shown in Table 5.2.

As can be seen, these are not trivial and are in fact
difficult to cover with metaheuristic techniques which
do not account for dependencies explicitly. Further,
the program takes real inputs with which it becomes
even more difficult to satisfy, for instance, equality conditions.

For branch coverage as the selected test data adequacy criterion, the program in Figure 5.3
was used to compare the following:

1. (Case I) The path testing approach of Lin and Yeh (Lin and Yeh, 2001), referred to as
   PTA hereafter. In the approach, a path is selected as a target and test cases are evolved
   that will lead the program execution to achieve the target path. A fitness function
   named ‘similarity’, as described above, is used to determine which test cases should
   survive if the final test case has not been found. The proposed approach does not
suggest a particular strategy for selecting a target path, but it is assumed that the path is feasible. For the benchmark program, the following paths were selected for coverage

\[ a. \quad 0,2,4,5,7,8,10,11,12,13,25 \]
\[ b. \quad 0,2,4,5,7,8,10,11,12,14,25 \]
\[ c. \quad 0,1,25 \]
\[ d. \quad 0,2,3,5,7,8,10,11,15,16,18,19,25 \]
\[ e. \quad 0,2,4,5,6,8,10,11,15,16,18,20,21,25 \]
\[ f. \quad 0,2,4,5,7,8,9,11,15,16,18,20,22,23,25 \]
\[ g. \quad 0,2,3,5,6,8,9,11,15,16,17,25 \]
\[ h. \quad 0,2,3,5,7,8,10,11,15,16,18,20,22,24,25 \]

The selected paths together constitute a minimal cover of all the branches in the program and are also basis paths (Pressman, 2010).

2. (Case II) The usual search based test data generation approach for branch coverage, referred to as NBD hereafter. This approach has been followed by a number of authors (Baresel et al., 2002; McMinn and Holcombe, 2006). Each branch is considered in turn, in some sequence, and becomes the new test goal for which search is performed. The fitness of each individual \( x \) in the population is computed as (Pachauri and Gursaran, 2012a)

\[
\text{fitness} (x) = \text{approach level} + \text{normalized branch distance}
\]

where \( \text{approach level} \) is a count of the number of predicate nodes in the shortest path from the first predicate node, from the start node, to the predicate node with the critical branch. The \( \text{normalized branch distance} \) is calculated according to the formula

\[
\text{Normalized Branch Distance} = \left( 1 / (1.001^{\text{distance}}) \right)
\]

In order to improve performance, branches were ordered for consideration for coverage using the path prefix strategy and the genetic algorithm used elitism and memory (Pachauri and Gursaran, 2011).

3. Case (III) The dynamic approach proposed in this chapter, referred to as DA hereafter, together with NBD. For branches that are dependent on other nodes being executed, the strategy outlined in this chapter is used. For all the other branches, scheme (2) was
used. Table 5.2 shows the identified dependencies. We do not discuss how these dependencies may be computed in this chapter.

Parameter and operator settings for the genetic algorithm are given in Table 5.3. For each of PTA, NBD and DA, and population size, hundred experiments (called runs hereafter) were carried out and the following statistics were collected as in the case of pilot experiments:

- Mean number of generations. The maximum number of generations for the experiments was taken to be $10^7$. Accordingly, the termination criterion for each experiment is either full branch coverage or $10^7$ generations whichever occurs earlier. It is possible that full branch coverage is not achieved even after $10^7$ generations. The mean number of generations thus does not tell us whether full branch coverage is achieved, hence the second statistic.

- Mean percentage coverage achieved.

Additionally ANOVA was carried out using SYSTAT 9.0 to determine if there is a significant difference in mean number of generations.

<table>
<thead>
<tr>
<th>Parameter/ Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Population Size</td>
<td>6, 10, 16, 20, 26, …, 110.</td>
</tr>
<tr>
<td>2 Crossover type</td>
<td>Two point crossover</td>
</tr>
<tr>
<td>3 Crossover Probability</td>
<td>1.0</td>
</tr>
<tr>
<td>4 Mutation Probability</td>
<td>0.01</td>
</tr>
<tr>
<td>5 Selection Method</td>
<td>Binary tournament</td>
</tr>
<tr>
<td>8 Population Initialization</td>
<td>Initialize once at the beginning of the GA run</td>
</tr>
<tr>
<td>9 Population Replacement Strategy</td>
<td>Elitism with upto 10% carry forward</td>
</tr>
<tr>
<td>10 Maximum Number of Generations</td>
<td>$10^7$</td>
</tr>
<tr>
<td>11 Memory</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**5.4. Results**

<table>
<thead>
<tr>
<th>Population Size</th>
<th>p value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.0</td>
</tr>
<tr>
<td>10</td>
<td>0.0</td>
</tr>
<tr>
<td>16</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Table 5.4 shows that the differences mean number of generations is significant over all populations for the chosen alpha (α) value of 0.05. In Figure 5.4 and Figure 5.5, it can be seen that the least value, for all populations is achieved with Case III which implements the methodology described in this chapter. Full branch coverage (Figure 5.5) is achieved for all population sizes in all runs with Case III which is not the case with Cases I and II. On analysis it was found that branch (16, 17), which is dependent on nodes 3, 6 and 9 was not easily covered by Case I and II. Branch (16, 17) requires condition (tri>3), which can only be true when all the sides of the triangle are equal, i.e. the triangle is an equilateral triangle, and this condition is difficult to achieve because no guidance is available for search.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.0</td>
</tr>
<tr>
<td>26</td>
<td>0.0</td>
</tr>
<tr>
<td>30</td>
<td>0.0</td>
</tr>
<tr>
<td>36</td>
<td>0.0</td>
</tr>
<tr>
<td>40</td>
<td>0.0</td>
</tr>
<tr>
<td>46</td>
<td>0.0</td>
</tr>
<tr>
<td>50</td>
<td>0.0</td>
</tr>
<tr>
<td>56</td>
<td>0.0</td>
</tr>
<tr>
<td>60</td>
<td>0.0</td>
</tr>
<tr>
<td>66</td>
<td>0.0</td>
</tr>
<tr>
<td>70</td>
<td>0.0</td>
</tr>
<tr>
<td>76</td>
<td>0.0</td>
</tr>
<tr>
<td>80</td>
<td>0.0</td>
</tr>
<tr>
<td>86</td>
<td>0.0</td>
</tr>
<tr>
<td>90</td>
<td>0.001</td>
</tr>
<tr>
<td>96</td>
<td>0.0</td>
</tr>
<tr>
<td>100</td>
<td>0.003</td>
</tr>
<tr>
<td>106</td>
<td>0.001</td>
</tr>
<tr>
<td>110</td>
<td>0.0</td>
</tr>
</tbody>
</table>
int triangle(double i, double j, double k)
{
    int tri = 0;
0. if ((i<=0) || (j<=0) || (k<=0))
1.     return 4;
2. else-if (i==j)
3.     tri += 1;
else
4.     // do nothing
5.     if (i==k)
6.     tri += 2;
else
7.     // do nothing
8.     if (j==k)
9.     tri += 3;
else
10.    // do nothing
11.    if (tri==0)
12.        {
13.            if ((i+j<=k) || (j+k<=i) || (i+k<=j))
14.                tri = 4;
else
15.                tri = 1;
16.                return tri;
17.        }
18.    else
19.        {
20.            if (tri>3)
21.                tri=3;
else-if ((tri==1) && (i+j>k))
22.                tri = 2;
else-if ((tri==2) && (i+k>j))
23.                tri = 2;
else-if ((tri==3) && (j+k>i))
24.                tri = 2;
else
25.                tri = 4;
26.        }
27.    return tri;
}
Figure 5.3 Source Code and CFG of the Benchmark Program
Figure 5.4 Mean Number of Generations

Figure 5.5 Mean Percentage Coverage