International Journal of Principles and Applications of Information Science and Technology
December 2007, Vol.1, No.1

Evolutionary Software Test Data Generation Using Sequence Comparison

H. Turgut Uyar, A. Şima Uyar, and Emre Harmancı

Istanbul Technical University, Dept. of Computer Engineering, Istanbul, Turkey
{uyar,etaner,harmanci}@itu.edu.tr

Abstract. Evolutionary algorithms have been applied to the dynamic structural test data generation problem. The fitness evaluation methods proposed so far either suffer from poor guidance characteristics or can not be applied uniformly. We propose a new method based on the pairwise sequence comparison techniques defined in bioinformatics. Interpreting the desired and generated paths as sequences, aligning and comparing these paths gives an intuitive measure about their similarity. We also define a sequence distance which measures the amount of change needed to convert the generated path into the desired path. The similarity and distance scores can then be used for evaluating candidate solutions in a search-based test data generation problem. Our preliminary tests show that this method produces promising results.

1 Introduction

Testing is a critical and costly activity in software development and techniques for automating this process have been explored extensively [5]. Metaheuristic search techniques have been successfully applied to this problem (see [9] for a detailed survey), especially in the field of structural test data generation, where the problem is generating input values for program components in order to cover specific paths or statements.

Given a code which takes some input parameters, if the aim is to find a set of input values that will follow a desired path through this code, search methods need to be able to evaluate input value sets to determine which of these are better than others. The basic approach for evaluating input value sets in dynamic structural test data generation methods can be summarized as follows:

1. represent a set of input values as a candidate solution
2. apply these input values to the code and observe the generated path
3. compare the generated path with the desired path and assign a fitness value to the input set based on this comparison

Various evaluation methods have been proposed for the test data generation problem, mostly based on distances to convert incorrect branching decisions into correct ones and on correctly covered program constructs. Many of these methods provide little guidance for the search to cover hard-to-reach portions of

the code due to their unfavorable characteristics such as creating large plateaus in the search space. Besides, many methods suffer from implementation difficulties such as the need to partition the problem into subproblems for handling different parts of the code or the need for a structural analysis of the code.

We present a new fitness evaluation method based on the pairwise sequence comparison techniques in bioinformatics. A short summary of this study has been published in [13]. If we consider the desired path and the path of executed statements as sequences, finding the similarity and/or distance between these sequences corresponds to the pairwise sequence comparison problem in bioinformatics. To compare two sequences, first they are aligned and then similarity and distance measures are calculated based on the aligned sequences [10]. We aim to show that this method can provide good guidance characteristics while at the same time it is easy to implement.

For discussing the issues in fitness evaluation, we will consider two commonly used benchmark problems:

Triangle classification ([2], [9], [12]) The function takes three input parameters which represent the sides of a triangle and returns the type of the triangle (one of "invalid", "scalene", "isosceles", and "equilateral"). An example implementation for the problem in the Python programming language is given in Figure 1. This problem is suitable for demonstrating basic issues.

```
 1:    def triangle(a, b, c):
 2:        if a > b:
 3:            a, b = b, a
 4:        if a > c:
 5:            a, c = c, a
 6:        if b > c:
 7:            b, c = c, b
 8:        if a + b <= c:
 9:            result = INVALID
10:        else:
11:            result = SCALENE
12:            if a == b and b == c:
13:                result EQUILATERAL
14:            elif a == b or b == c:
15:                result ISOSCELES
16:        return result
```

Fig. 1. Triangle classification.

Finding the minimum and maximum elements in a list ([8]): The function takes a list of numbers and three integers (the indices of the first and last elements and a step size) which select a sublist. It returns the minimum and maximum of

the elements in the sublist. An example implementation for the problem in the Python programming language is given in Figure 2. This problem is suitable for demonstrating issues concerning loop structures.

```
 1:    def minmax(low, high, step, A):
 2:        min = max = A[low]
 3:        i = low + step
 4:        while i < high:
 5:            if max < A[i]:
 6:                max = A[i]
 7:            if min > A[i]:
 8:                min = A[i]
 9:            i = i + step
10:        return min, max
```

Fig. 2. Finding the minimum and maximum elements.

2 Related Work on Evolutionary Software Testing

Choosing an appropriate fitness evaluation function is a critical step when applying an evolutionary algorithm to any problem because good fitness functions will guide the search towards better solutions. In the next sections, we will discuss the main methods that have been proposed. A recent survey and comparison of fitness functions for path testing can be found in [14].

2.1 Branch Distances

A commonly used evaluation method is computing *branch distances*. If the control flow continues with the unwanted branch at a condition statement, the branch distance at that statement is defined as the difference of the actual and necessary values of the variables in order to let the flow take the intended branch. Since solutions with smaller branch distances are preferred, the problem becomes one of minimizing the branch distances and the fitness value 0 indicates an optimum solution. For the triangle classification example, if the desired path is $< 2, 4, 5, 6, 8, 11, 12, 14, 15, 16 >$ and the input values are $a = 23$, $b = 66$, $c = 69$, the $a == b$ *or* $b == c$ predicate on line 14 will be false and flow will continue at line 16, whereas the predicate should be true and control should have been transferred to line 15. The distance for the $a == b$ predicate is $|23-66| = 43$ and the distance for the $b == c$ predicate is $|66 - 69| = 3$. If any of these predicates were true, the compound predicate would have been true, therefore the branch distance is the minimum of the two distances, 3.

If there are multiple branching statements along the path, one way of calculating the total branch distance of the generated path is to sum up all branch

distances. In the example given above, for the same desired path, the input set $a = 23$, $b = 66$, $c = 69$ will fail on the predicates on lines 4 and 14 with the distances 47 and 3, respectively, making the total distance 50.

The major problem with branch distances is that they can not be applied to all predicate types. They can easily be computed when the predicate is a comparison between two numbers but it is hard to define a distance for the case when the predicate is simply a boolean flag.

Another problem is that inputs which cause big portions of the code getting skipped with small branch distances get better fitness values than inputs which generate paths that follow the desired path more closely but fail with larger distances. For example, the input set $a = 1$, $b = 10$, $c = 15$ will get the branch distance 20 (15 for line 4 and 5 for line 8). Considering only branch distances makes this second input set seem to be a better solution than the first set, which did not fail at line 8.

Taking the first branch distance as the total distance is another method, but it only helps in some cases. In the example, this method also does not solve the explained problems with the given input sets.

2.2 Control Structures

Another evaluation method is to take into account the number of control structures in the desired path which are also covered by the generated path. An *approximation level* can be defined to measure the number of correct branches taken to reach a desired program construct [15].

In the triangle classification example, all paths will reach line 8. For a path to reach line 15, the predicates on lines 8, 12, and 14 must be false, false and true, respectively. The fitness of a path can simply be evaluated as the number of correct branches taken. In this case, for the example, the first input set is assigned the fitness value 2, whereas the second set is assigned the value 0.

The major problem with approximation levels is that fitness values are chosen from a very small set and solutions that fail on the same predicate can not be compared. In the example, there are 4 distinct fitness values (0 to 3). This makes the search space consist of several plateaus and the search can not be guided towards better solutions within these plateaus.

Another problem is that evaluation makes sense only for one program construct such as a selection. If there are multiple such constructs, approximation levels for each construct will be different. For example, the above discussion for the triangle classification problem is relevant only for lines 8-15 and does not cover the first three if-constructs on lines 2-7. To handle this situation, *partial aims* are defined where each program construct is evaluated separately [15] which makes the problem more complicated.

2.3 Combined Approaches

Fitness evaluation methods which combine branch distance and approximation level methods have also been proposed. For example, primarily using approxi-

mation levels and preferring the solution with the smaller branch distance when the approximation levels are equal provides better guidance for the search. This method would also solve the problem with the total branch distance definition because taking the distance of the first incorrect branch would be suitable to compare two equal paths. Multiobjective evaluation functions have also been proposed where the fitness value is a function of the normalized approximation level and branch distance scores [9]. However, the code under test will still need to be partitioned to overcome the difficulties with evaluating multiple control structures, especially in the presence of loops.

We should point out that the *chaining approach* [4] is a major technique used in the field that solves some of the mentioned issues and introduces some other difficulties. Since we are proposing a new fitness evaluation function, the chaining approach is out of the scope of this paper.

3 Pairwise Sequence Comparison as a Fitness Measure

Pairwise sequence alignment can be defined as the process of finding matching patterns that occur in the same order in two sequences. In biology, sequence alignment is used to discover functional, structural or evolutionary similarities between sequences such as DNA or protein sequences.

There are two types of sequence alignment approaches: global alignment techniques are used for aligning entire sequences, whereas local alignment techniques are used for aligning highly matching subsequences. Global alignment is more appropriate if the sequences are quite similar, and they have similar lengths. Local alignment is more appropriate if the sequences are highly similar in some subsequences but dissimilar in the remaining parts, or they are very different in length, or they share a region with specific functionality.

Two metrics are used for scoring the alignment of two sequences: one which defines the similarity between two sequences based on the number of matching items in the aligned sequences, and the other which defines the distance between two sequences based on the number of mismatched items. In an optimal alignment, mismatching items are arranged and gaps are inserted to bring as many matching items in alignment as possible to obtain the highest possible score, while at the same time keeping the aligned sequences as short as possible.

After the alignment, for each location on the aligned sequences, one of three situations will occur: a *match* if the elements are identical, a *mismatch* if the elements are different, and a *gap* if one of the elements is a gap. To determine issues such as whether gaps should be preferred to mismatches or how important it is to have matches, scores are assigned to each of the three situations given above. These scores are parameters of the algorithm and different scores result in different alignments. Dynamic programming approaches which guarantee the optimal alignment between two sequences have been proposed for both global and local sequence alignment [11]. Detailed information on pairwise sequence alignment and scoring can be found in [7] and [10].

For the triangle classification example, the inputs $a = 23$, $b = 66$, $c = 69$ generate the path $< 2, 4, 6, 8, 11, 12, 14, 16 >$, and the inputs $a = 1$, $b = 10$, $c = 15$ generate the path $< 2, 4, 6, 8, 9, 16 >$. Possible alignments of these sequences with the desired path $< 2, 4, 5, 6, 8, 11, 12, 14, 15, 16 >$ are given in Figure 3.

```
desired:    2  4  5  6  8 11 12 14 15 16
generated:  2  4  -  6  8 11 12 14  - 16

             (a) a=23 b=66 c=69

desired:    2  4  5  6  8 11 12 14 15 16
generated:  2  4  -  6  8  9  -  -  - 16

             (b) a=1 b=10 c=15
```

Fig. 3. Aligned paths for the triangle classification example.

3.1 The Proposed Fitness Evaluation Method

We are proposing a new fitness evaluation method based on the pairwise similarity of the desired path and the path generated by the input set. After aligning these paths, the fitness will be defined as the ratio of the number of matches to the length of the aligned sequences. If the two sequences are identical, all elements in the aligned sequences will match and this will produce a fitness of 1. Therefore, a fitness value of 1 will indicate an optimal solution. As seen in Figure 3, this evaluation method assigns the fitness values $8/10 = 0.8$ and $5/10 = 0.5$ to the input sets in the triangle classification example.

This method has the advantage that it can be applied uniformly to all program constructs, including loops. For the minimum and maximum element finding example, if the path $< 2, 3, 4, 5, 7, 9, 4, 5, 7, 8, 9, 4, 10 >$ is desired and the path $< 2, 3, 4, 5, 7, 8, 9, 4, 5, 7, 8, 9, 4, 5, 7, 8, 9, 4, 10 >$ is generated, a possible alignment is given in Figure 4, for which the fitness would be calculated as $13/19 = 0.68$.

```
desired:    2 3 4 5 7 - 9 4 5 7 8 9 4 - - - - - 10
generated:  2 3 4 5 7 8 9 4 5 7 8 9 4 5 7 8 9 4 10
               |- pass1 -|- pass2 -|- pass3 -|
```

Fig. 4. Aligned paths for the minimum-maximum finding example.

3.2 Sequence Distance

Pairwise sequence comparison suffers from a similar problem as approximation level based fitness evaluation in that two different input data sets that generate the same path can not be compared. To overcome this problem, we will define a *sequence distance* as a measure of the amount of change needed to convert the generated path to the desired path, thus providing guidance information to the search algorithm.

Molecular biology defines three mutation operations to convert a source sequence into a destination sequence [10]. If the source and destination sequences correspond to the generated and desired paths, then these operations describe how to convert the generated path into the desired path:

1. *insertion*: A subsequence in the destination sequence is missing from the source sequence, i.e. it has to be inserted into the source sequence. In our problem, this case typically occurs when a selection or repetition construct is not entered when it should. This case can be identified as a gap in the aligned source sequence.
2. *deletion*: The source sequence contains a subsequence not found in the destination sequence, i.e. it has to be deleted from the source sequence. In our problem, this case typically occurs when a selection or repetition construct is entered when it should not. This case can be identified as a gap in the aligned destination sequence.
3. *substitution*: The item in the source sequence is different from the item in the destination sequence. In our problem, this case typically occurs when the wrong branch is taken at an if-else construct. This case can be identified as a mismatch in the aligned sequences. In our problem we will consider this case as a combination of deletion and insertion mutations.

For example, in Figure 3a, to convert the generated path into the desired path, two insertion mutations are needed to add the lines 5 and 15 that correspond to the gaps in the generated path. In Figure 3b, instead of interpreting the change from line 9 to line 11 as a substitution mutation, we will apply a deletion mutation for line 9 and an insertion mutation for lines 11-12-14-15.

In molecular biology, each gap is treated as a single mutation with the penalty value based on the length of the gap. The gap opening penalty d is the branch distance of an incorrect branch. It also makes sense that incorrect branches which cause longer portions of the code being skipped or inserted should get higher penalties. But instead of a linear penalty, we propose a logarithmic penalty as in Equation 1, where the total sequence distance is the sum of all penalties.

$$pen(d, g) = d(1 + ln(g)) \tag{1}$$

For the example in Figure 4, assuming that the branch distance on line 7 (pass 1) is 10 and on line 4 (pass 3) is 20, the total sequence distance is:

$$10(1 + ln(1)) + 20(1 + ln(5)) = 62.19$$

Interpreting substitution mutations as deletion and insertion mutations of length 1 results in the penalty calculation $d(1 + ln(1)) = d$. This scheme makes it easier to implement than handling substitution mutations separately. Though we have used the same penalty method for all of insertion, deletion and substitution mutations, it is also possible to use different calculations for each type of mutation.

4 Empirical Analysis

The outlines and relevant parameters of the evolutionary algorithm and the sequence alignment algorithm are explained below along with the test setup and a discussion of the results.

4.1 The Evolutionary Algorithm

Evolutionary Algorithms (EAs) [3], an umbrella term covering several slightly differing techniques, are population based optimization approaches inspired from classical Mendelian genetics and Darwin's evolutionary theory. Genetic Algorithms (GAs), Genetic Programming (GP) and Evolutionary Strategies (ES) are the earlier representatives of EAs, developed approximately around the same timeframe independently by different groups. GAs and ES are originally for search and optimization tasks while GP is more for machine learning tasks and automatic program generation. In the original proposals, they mainly differed in the representation of solution candidates and the operators they used, e.g. GAs used binary coded strings, ES used real-coded vectors whereas GP used trees. However, over time, due to the requirements of the various applications that emerged, the distinction between these approaches became less clear and various representations and operators were used with each. Therefore it is appropriate to refer to all such implementations as EAs in general.

We used a standard steady-state EA with duplicate elimination. Through duplicate elimination, all individuals in the population are genotypically different. This ensures a good level of diversity in the population. In a steady-state EA, at each iteration of the algorithm, a new individual is created from two parents through selection, crossover and mutation and is inserted into the population through a replacement scheme. The algorithmic flow of such a steady-state algorithm is given in Algorithm 1. Further details about steady-state EAs can be found in [3].

Representation Each individual in an EA, i.e. a solution candidate, consists of a chromosome and a fitness value. For our problem, a chromosome consists of genes each of which corresponds to an input parameter for the code being tested. In traditional GAs, genes are binary valued. Therefore in earlier studies, parameter values were converted into binary representations. For example, if an integer parameter is defined to take on values in the interval $[0, 7]$, it was represented with a string of 3 bits. However, this is inefficient, especially in

Algorithm 1 The steady-state evolutionary algorithm.

```
randomly initialize population
while max no of fitness evaluations not reached do
  select parents
  create child through cross-over
  mutate child
  if child is not duplicate then
    if fitness of child better than fitness of worst then
      child replaces worst in population
    end if
  else
    discard child
  end if
end while
```

cases where there is redundancy. It is more natural to represent each parameter as it is, that is, as an integer if it takes on integer values, as a floating point number if it takes on real values or binary values if it is a decision variable or a boolean variable. For both of our sample test problems, the parameters take on integer values within defined intervals. Therefore, the genes in our examples are represented as integers. For example, in Section 3 for the triangle classification example, it was shown that the inputs $a = 23$, $b = 66$, $c = 69$ generate the path $< 2, 4, 6, 8, 11, 12, 14, 16 >$. In the EA, the chromosome for this case is $23 - 66 - 69$.

Fitness Evaluation The proposed method which is explained in Section 3.1 is used to evaluate the fitness of individuals. When two individuals are compared, the one with the better sequence similarity score is preferred. In case the two individuals have identical similarity scores, the one with the smaller sequence distance is preferred. If the similarity score of an individual is 1, the individual is an optimal solution.

Initialization The first population of the EA is initialized randomly within the allowed intervals for each parameter. No individuals are allowed to be duplicates.

Selection Two individuals are selected as parents from the population to undergo crossover to produce an offspring. Binary tournament selection [3] is used for the selection step. In binary tournament selection, for each parent, two individuals are selected randomly from the whole population and the one with the higher fitness becomes the parent.

Crossover One offspring individual is generated from two parents through crossover. For the crossover step, *uniform crossover* technique is chosen in this

study. Uniform crossover ensures a good mixing of parameters from both parents. In uniform crossover, the offspring receives each gene from either of its parents with equal probability. For example for parents $P_1 = 13 - 24 - 56$ and $P_2 = 36 - 12 - 25$ a possible offspring may be $13 - 12 - 25$ where the first gene comes from P_1 and the second and third genes come from P_2.

Mutation Mutation is the step where new gene values are introduced into the population. Mutation at each gene occurs with a predefined mutation probability p_m which is usually very small and is commonly taken to be $p_m = 1/ChromosomeLength$. Typically mutation is based on a neighborhood relationship defined between the points in the search space. In a binary representation, mutation is implemented through randomly changing the value of the gene from 0 to 1 or vice versa. The corresponding mutation approach for integer representations is called *random resetting* where the value of the gene is randomly changed to another value within the allowed interval for that parameter. However, this mutation approach frequently disrupts the search process since it does not use any neighborhood information. Another mutation mechanism used with integer representations is called *creep mutation*. Creep mutation preserves the neighborhood information for integers since the integer value of the gene is either incremented or decremented by a fixed step size. We used creep mutation with a step size of 1 in this study.

Replacement In each iteration of a steady-state EA, a new individual is created and is inserted into the population according to a replacement strategy. In the most commonly used replacement scheme [3] for steady-state EAs, the new individual replaces the worst individual in the population if it is better. Because of duplicate elimination, if the newly created individual is a duplicate of any existing individual, it is discarded. Otherwise, it is evaluated and its fitness is compared to the fitness of the currently worst individual in the population. If the child has better fitness, it replaces the worst individual in the population.

Termination The program is run until an optimal solution, i.e. an individual with a fitness value of 1, is found or until a predefined number of fitness evaluations have been performed.

4.2 Sequence Alignment

A time-efficient method for sequence alignment, *FastLSA*, is described in [1]. However, since each fitness evaluation requires that the code under test will be executed and therefore takes a long time, the time performance of the sequence alignment algorithm plays a secondary role. For easier implementation, we used a simple linear sequence alignment method based on Hirshberg's algorithm [6] which provides the same solution quality.

The algorithm first constructs a dynamic programming matrix by comparing elements in both paths and using the match, mismatch and gap scores. These

are selected as 2, -1, and -1, respectively. These are commonly used values in sequence alignment applications and we have observed that they produce suitable alignments for the test data generation problem.

The matrix is populated according to the algorithm given in Algorithm 2. The nodes on the desired path constitute the columns and the nodes on the generated path constitute the rows. A row and a column are inserted into the table to represent the gap element. This insertion can be done at the beginning or at the end; element calculation will start at the intersection of the inserted row and column (which will have the value 0) and proceed toward the other corner. In our study, we inserted the gap row and column at the beginning and therefore proceeded from the upper left corner toward the lower right corner.

Each element in the matrix is calculated using the values of its neighbors calculated so far. As can be seen in the algorithm, when an element is calculated, at most three values based on its already computed neighbors (horizontal, vertical, and diagonal) will be considered and their maximum will be selected.

Algorithm 2 Computing the dynamic programming matrix.

```
rows ← length of desired path + 1
cols ← length of generated path + 1
for all i in rows do
  for all j in cols do
    horizontal ← horizontal neighbor + gap
    vertical ← vertical neighbor + gap_score
    if row node = column node then
      diagonal ← diagonal neighbor + match_score
    else
      diagonal ← diagonal neighbor + mismatch_score
    end if
    matrix[i][j] ← max(horizontal, vertical, diagonal)
  end for
end for
```

After the matrix is populated, it is traversed to align the sequences according to the algorithm given in Algorithm 3. Traversal begins either at the upper left or lower right corner and proceeds toward the other corner. At each iteration, the neighbor with the maximum value is selected as the next element. If the move from the current element to the next element is diagonal, corresponding nodes in both the desired and generated paths are selected. Horizontal or vertical moves correspond to inserting gaps to either of the paths.

For example, if the desired path is $< 2, 4, 5, 6, 8, 11, 12, 14, 15, 16 >$ and the generated path is $< 2, 4, 6, 8, 9, 16 >$, the resulting matrix is given in Figure 5 and a the outcome of the alignment algorithm as marked by the arrows is given in Figure 3b. Note that the alignment starts at cell $(2, 2)$ and at each transition the element before the transition is taken from the corresponding row or column. For example, on the transition from $(9, 11)$ to $(16, 12)$, the node 9 will be added to

Algorithm 3 Aligning the sequences.

```
current ← corner element
while current not reached the other corner do
  h ← horizontal neighbor of current
  v ← vertical neighbor of current
  d ← diagonal neighbor of current
  next ← max(h, v, d)
  if next = diagonal then
    insert column node into aligned desired path
    insert row node into aligned generated path
  else if next = horizontal then
    insert column node into aligned desired path
    insert gap into aligned generated path
  else
    insert gap into aligned desired path
    insert row node into aligned generated path
  end if
  current ← next
end while
```

the aligned generated path and the node 11 will be added to the aligned desired path. It is also worth pointing out that the algorithm can produce different outcomes due to selection among neighboring options. Our algorithm favors the diagonal neighbor in such a case.

	G	2	4	5	6	8	11	12	14	15	16
G	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
2	-1	2	1	0	-1	-2	-3	-4	-5	-6	-7
4	-2	1	4	3	2	1	0	-1	-2	-3	-4
6	-3	0	3	3	5	4	3	2	1	0	-1
8	-4	-1	2	2	4	7	6	5	4	3	2
9	-5	-2	1	1	3	6	6	5	4	3	2
16	-6	-3	0	0	2	5	5	5	4	3	5

Fig. 5. Sequence alignment example.

4.3 Test Problems

The proposed method was tested with the triangle classification and minimum-maximum element finding problems discussed throughout this paper. The mutation and crossover probabilities are chosen as recommended in literature. The settings of the population size and the maximum number of fitness evaluations used as a termination condition are determined experimentally. The input pa-

rameter ranges are selected such that they generate a moderately sized search space.

For the triangle classification problem, the code given in Figure 1 was used and following desired paths were tested:

- $< 2, 4, 5, 6, 8, 11, 12, 14, 15, 16 >$: This is a non-trivial path which is generated when $a = b > c$. This path will be referred to as $T1$ in the test results.
- $< 2, 4, 6, 8, 11, 12, 13, 16 >$: This is the hardest path which is generated when $a = b = c$. This path will be referred to as $T2$ in the test results.

All three input parameters are represented as integers in the range [1,1000]. The population size for the steady-state algorithm is 100. The probability for the uniform crossover operator is 1 and the probability for the creep mutation operator is $1/l$, where l is the length of the chromosome, in this case 3. A maximum of 10000 solutions are evaluated and search stops if an optimum solution is found.

For the minimum-maximum element finding problem, the code given in Figure 2 was used and following desired paths were tested:

- $< 2, 3, 4, 5, 7, 9, 4, 5, 7, 8, 9, 4, 10 >$: This path is taken from [8] and will be referred to as $M1$ in the test results.
- $< 2, 3, 4, 5, 7, 9, 4, 5, 7, 8, 9, 4, 5, 6, 7, 9, 4, 5, 6, 7, 9, 4, 10 >$: This is a longer and more complicated path than the above and will be referred to as $M2$ in the test results.

The list parameter consists of 100 integers in the range [1,1000]. The other three parameters are integers in the range [1,100]. As with the previous test setup, the population size is 100, the crossover probability is 1 and the mutation probability is $1/l$, this time l being 103. A maximum of 50000 solutions are evaluated and search stops if an optimum solution is found.

To observe the effects of similarity and distance scores, three evaluation methods have been tested for each problem:

- sequence similarity only (optimum fitness is 1)
- sequence distance only (optimum fitness is 0)
- sequence similarity and sequence distance: if the similarity scores are equal, the solution with the smaller distance score is preferred

4.4 Test Results

For each of the three evaluation methods, each path is tested for 50 runs and the following quantities are calculated:

- success rate (SR)
- in successful runs: average number of fitness evaluations (EAv) and the standard deviation (ESd)

– in unsuccessful runs: average number of first hitting times (FAv) and the standard deviation (FSd), where the first hitting time is defined as the number of fitness evaluations it has taken to find the overall best fitness

Using only sequence similarity evaluation resulted in very poor success rates. In fact, other than the relatively simple path $T1$ in the triangle classification problem, it failed to find a solution in any test. In the $T1$ test, the success rate was 50%, with an average of 447.88 fitness evaluations and a standard deviation of 409.44. An average first hitting time of 34.76 with a standard deviation of 65.82 indicates that a suboptimum is found fairly early and searching further did not improve the performance. From these results, we can conclude that similarity-only evaluation is not suitable even in moderately complex problems.

Using only sequence distances performed slightly better. As can be seen in Table 1, it had a 100% success rate in the $T1$ test. It also occasionally did succeed in the $T2$ test but a 28% success rate is still not acceptable. Using sequence similarity together with sequence distance resulted in much better performance, as can be seen in Table 2. The tests for $T1$, $T2$ and $M1$ were successful all the time, whereas the test for $M2$ failed only once.

Table 1. Distance only.

P	SR	EAv	ESd	FAv	FSd
T1	100%	523.66	213.47	-	-
T2	28%	1266.21	598.78	567.36	257.03

Table 2. Similarity and distance.

P	SR	EAv	ESd	FAv
T1	100%	609.96	299.26	-
T2	100%	2284.38	1024.79	-
M1	100%	1980.86	1190.46	-
M2	98%	6449.71	6019.87	49807

5 Conclusions and Future Work

The proposed sequence similarity and sequence distance methods roughly have the same guidance characteristics as approximation level and branch distance methods, respectively. The advantage of the proposed methods is that they can be applied to all parts of the code under test in the same way. Therefore, the proposed fitness evaluation method is intuitive and easy to implement. Besides, since they evaluate the paths in their entirety instead of processing them partially, they have the potential of better exploiting the desirable characteristics the input sets might have. Our current results are promising but more research is needed to better analyze the performance on various problems and to identify possible pitfalls in the algorithm. The major drawback of the current implementation is that the tester specifies the desired path. Work on eliminating this requirement is currently being done.

References

1. K. Charter, J. Schaeffer, and D. Szafron. Sequence alignment using fastlsa. In *International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, pages 239–245, 2000.

2. J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, Oct. 1999.
3. A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 1st edition, Nov. 2003.
4. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
5. M. Fewster and D. Graham. *Software Test Automation*. ACM Press, 1999.
6. D. S. Hirshberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
7. N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press, Aug. 2004.
8. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, Aug. 1990.
9. P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, June 2004.
10. D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edition, June 2004.
11. S. B. Needleman and C. D. Wunsch. A general method applicable to search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.
12. R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.
13. H. T. Uyar, A. S. Uyar, and A. E. Harmanci. Pairwise sequence comparison for fitness evaluation in evolutionary structural software testing. In *GECCO '06: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1959–1960. ACM Press, 2006.
14. A. Watkins and E. M. Hufnagel. Evolutionary test data generation: A comparison of fitness functions. *Software-Practice and Experience*, 36:95–116, 2005.
15. J. Wegener, K. Buhr, and H. Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1233–1240, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.