Pairwise Sequence Comparison for Fitness Evaluation in Evolutionary Structural Software Testing

H. Turgut Uyar
Istanbul Technical University
Dept. of Computer Eng.
Istanbul, Turkey
uyar@ce.itu.edu.tr

A. Şima Uyar
Istanbul Technical University
Dept. of Computer Eng.
Istanbul, Turkey
etaner@ce.itu.edu.tr

Emre Harmancı
Istanbul Technical University
Dept. of Computer Eng.
Istanbul, Turkey
harmanci@ce.itu.edu.tr

ABSTRACT

Evolutionary algorithms are among the metaheuristic search methods that have been applied to the structural test data generation problem. Fitness evaluation methods play an important role in the performance of evolutionary algorithms and various methods have been devised for this problem. In this paper, we propose a new fitness evaluation method based on pairwise sequence comparison also used in bioinformatics. Our preliminary study shows that this method is easy to implement and produces promising results.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms

Keywords

search-based software engineering, structural software testing, automated software test data generation, evolutionary algorithms, pairwise sequence alignment

1. INTRODUCTION

Testing is a critical and costly activity in software development and techniques for automating this process have been explored extensively [2], especially in the field of structural test data generation, where the aim is to generate input values for program components in order to cover specific paths or statements [4].

A basic approach for using search methods to solve this problem can be summarized as follows:

- represent a set of input values as a candidate solution
- apply these input values to the code under test and observe the generated path
- compare the generated path with the desired path and assign a fitness value to the input set

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
ACM 1-59593-186-4/06/0007.

In this paper, we present a new fitness evaluation method based on the pairwise sequence comparison technique also used in bioinformatics [5]. We tested the proposed method using common benchmark problems and our preliminary results show that the guidance characteristics of this method are similar to existing methods but it is easier to implement and is uniformly applicable to a wide range of problem instances.

2. RELATED WORK

Some of the proposed fitness evaluation methods provide little guidance in the search for covering hard-to-reach portions of the code, and some methods suffer from implementation difficulties. The basic approaches are [4]:

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. Solutions with smaller branch distances get better fitness values.

A major problem with this method 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. Loops pose another problem with this scheme in that it becomes harder to determine whether a predicate has failed or not for a branch inside a loop.

Control Structures. An *approximation level* is defined to measure the number of correct branches taken to reach a desired program construct. Solutions with higher approximation levels get better fitness values.

A major problem with this method is that solutions which fail on the same predicate can not be compared. This makes the search space consist of plateaus and the search can not be guided towards better solutions within plateaus. Another important problem is that if there are multiple selection or repetition constructs, approximation levels for each construct will be different and each construct will have to be evaluated separately, using *partial aims* [6].

Combined Approaches. Branch distance and approximation level methods can be combined, for example by using a multiobjective function where the fitness value is a function of the normalized approximation level and branch distance scores.

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

Figure 1: Path alignment example.

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 [5]. For each location on the aligned sequences, one of three situations will occur:

- a *match*: the elements are identical
- a *mismatch*: the elements are different
- a *gap*: one of the elements is a gap

Assuming that the desired path of line numbers through a code is $< 2, 4, 5, 6, 8, 11, 12, 14, 15, 16 >$ and the generated path is $< 2, 4, 6, 8, 9, 16 >$, a sample alignment is given in Figure 1.

Two metrics are used for scoring the alignment: one which defines the similarity based on the number of matches, and the other which defines the distance based on the number of mismatches and gaps.

3.1 Sequence Similarity

We are proposing a new fitness evaluation method based on the pairwise similarity of the desired and generated paths. After aligning these paths, the fitness is defined as the ratio of the number of matches to the length of the aligned sequences. If the two paths are identical, all elements will match and this will produce a fitness of 1.

3.2 Sequence Distance

The sequence similarity method by itself suffers from the same problem as the approximation level method because two input sets that generate the same path can not be compared. To overcome this, we define a *sequence distance* to measure the amount of change needed to convert the generated path to the desired path using the following operations adapted from bioinformatics [5]:

1. *insertion*: A subsequence in the desired path is missing from the generated path. This case occurs when a branch is not entered when it should and can be identified as a gap in the aligned generated path.
2. *deletion*: The generated path contains a subsequence not found in the desired path. This case occurs when a branch is entered when it should not and can be identified as a gap in the aligned desired path.
3. *substitution*: The item in the generated path is different from the item in the desired path. This case typically occurs when the wrong branch is taken at a predicate and can be identified as a mismatch.

We compute a penalty for each gap as:

$$pen(d, g) = d + (g - 1)e$$

where g is the length of the gap, d is the gap opening penalty (in our case, the branch distance) and e is the gap-extension penalty. The sum of individual gap penalties is the sequence distance.

4. EMPIRICAL ANALYSIS

We tested the proposed method with a few common benchmark functions such as triangle classification [4] using a standard steady-state evolutionary algorithm with elitism and duplicate elimination [1]. Each gene represents an input parameter for the code being tested, where gene values are integers in the range [1,1000]. For sequence alignment, we used a linear sequence alignment algorithm based on Hirshberg's approach [3].

For each test, we measured the success rate, the number of fitness evaluations and the first hitting times. Our results can be summarized as follows:

- Using only sequence similarity resulted in very poor success rates, even in moderately complex cases.
- Using only sequence distances performed slightly better, still with unacceptable success rates in most cases.
- Using sequence similarity and preferring the solution with smaller sequence distance when similarities are equal resulted in very high success rates (100% for nearly all cases) and an acceptable performance.

5. CONCLUSIONS AND FUTURE WORK

The proposed sequence similarity and sequence distance approaches roughly have the same characteristics as approximation level and branch distance methods. Their advantage is that they are easy to implement and they can be uniformly applied to all parts of the code. Besides, since they evaluate the paths in their entirety instead of processing them partially, they have the potential of better exploiting the desirable characteristics of an input set.

Our preliminary 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 effect of parameter settings such as the match, mismatch and gap penalties for the sequence alignment can also be explored. The major drawback of the current implementation is that the tester specifies the desired path. Work on eliminating this requirement is also needed.

6. REFERENCES

[1] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 1st edition, Nov. 2003.

[2] Mark Fewster and Dorothy Graham. *Software Test Automation*. ACM Press, 1999.

[3] D. S. Hirshberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[4] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, June 2004.

[5] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edition, June 2004.

[6] 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, 2002.