Generation of Tests for Programming Challenge Tasks Using Evolution Algorithms

Maxim Buzdalov
Saint-Petersburg State University of IT, Mechanics and Optics
197101, 49 Kronverkskiy prosp.
Saint-Petersburg, Russia
mbuzdalov@gmail.com

ABSTRACT

In this paper, an automated method for generation of tests in order to detect inefficient (slow) solutions for programming challenge tasks is proposed. The method is based on genetic algorithms.

The proposed method was applied to a task from the Internet problem archive — the Timus Online Judge. For this problem, none of the existed solutions passed the generated set of tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Data generators*

General Terms

Algorithms, Experimentation, Performance

Keywords

Programming challenges, genetic algorithms, testing

1. INTRODUCTION

Testing of software is a difficult problem. In a classic book [12], it is stated that testing takes 50% of time and more than 50% of cost spent on development of a piece of software. The following properties are shared by most of enterprise software from the point of view of testing:

- in majority of cases, the logic of the program is tested;
- suboptimal answers are often acceptable;
- the program to be tested either exists or is to be written by the same developer who writes tests [8].

In most of the popular types of programming challenges [1, 2, 5], the correctness of solutions for the programming tasks is checked by running them on a number of pre-written tests (input files) under time, memory and other limits, and then checking the answer (the output file). When successfully compiled and running on a test, a solution may end up with one of the following outcomes [3, 14]:

- Time Limit Exceeded (TL) — the solution exceeded the time limit set for the problem;
- Memory Limit Exceeded (ML) — the solution exceeded the memory limit set for the problem;
- Runtime Error (RE) — the solution terminated unexpectedly, most probably because of some runtime errors (division by zero, array index out of bounds) or uncaught exceptions;
- Presentation Error (PE) — the output file does not match the required format;
- Wrong Answer (WA) — the answer is incorrect;
- Accepted (AC) — the answer is correct.

Creation of tests for programming tasks differs from the enterprise software testing in a number of points. First, the logic of a solution is typically much simpler, so almost every test covers all the paths. Second, partially correct solutions, as well as solutions giving suboptimal answers are inacceptable or are ranked lower than the fully correct ones. Third, the programs to be tested do not exist yet (except for ones written by jury members), as tests are typically made prior to the competition.

The quality of the test sets for the programming tasks mostly determines the quality of the challenge itself. In other words, weak tests allow a number of incorrect solutions to pass the tests, so skilled participants who are trained to invent correct solutions are given less chance to win than unskilled ones who write incorrect solutions in the hope it will be accepted.

One of the ways to make the situation better is to make the process of test creation more automated. In this work, genetic algorithms are used to generate tests that challenge the inefficient solutions. The use of evolution algorithms is inspired by works on unit test generation [7, 15].

2. THE APPROACH

The approach being described is designed for generation of tests against inefficient solutions for those kinds of programming tasks where the usual ways of test generation produce weak tests (i.e., tests that are likely to be passed by some wrong or inefficient solutions). Such tasks often correspond

GECCO'11, July 12–16, 2011, Dublin, Ireland.

to the computer science problems that are known to have a number of heuristics that do not improve the worst-case performance but improve the average-case performance, thus making it very hard to find the worst-case tests.

One may take the knapsack problem as an example. This problem is known to be NP-complete. However, there are many algorithms that, on random data, perform as fast as $O(N)$ where N is the number of items [13]. The problem of finding hard test data is examined in [9] but the approach described there fails when the task is to find a hard test with certain small limits on weights and capacities of items.

The approach consists of the following steps:

- choosing a representation of a test as an individual of the genetic algorithm;
- designing genetic operators for the representation of a test;
- implementing a solution to generate tests against;
- designing a fitness function for a test, based on the chosen solution;
- evolving a number of tests using the genetic algorithm, the test representation and the solution-based fitness function.

2.1 Test Representation

The choice of a test representation is determined by the data included in the test, and the constraints on the test data, described in the task statement. The representation should ensure that, for a randomly-generated representation, the rate of correct tests (i.e., tests that meet the constraints) is reasonably high. In the example of a task below the concrete example of the representations is given.

2.2 Model Solutions

For the approach to work, there must be some solutions written with the purpose of generating tests against. This is not a new idea, as, in the "traditional" workflow of preparing of problems, some inefficient solutions are written anyway to demonstrate that the test set filters all of them. However, in this type of workflow the solutions which are considered to be inefficient by the described approach, often are mistakenly counted as "correct" ones. If it is possible to add tests after the challenge starts, then the accepted solutions may be used for generation of tests as well.

2.3 Fitness Functions

There is no universal fitness function which is applicable to every instance of test generation problem. Consider the following fitness function: the number of instructions executed while the solution is running. Using it, the genetic algorithm may fail to find a good test if the size of such test is relatively small, and running time of the solution on random test is determined by the size of a test. It is the case of the knapsack problem [13, 9].

The way adopted in this paper is to use a fitness function specific to the problem and the particular solution under test. It is possible to develop more fine-grained fitness functions, which allow to breed better tests, or same quality tests in less time.

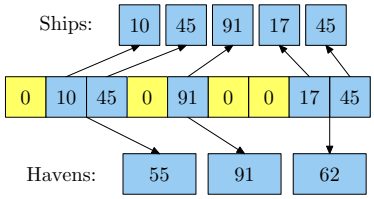

Figure 1: A test encoded by a sequence of integers

3. EXAMPLE: "SHIPS. VERSION 2"

In this section, we consider a programming task named "Ships. Version 2", which is located at Timus Online Judge [6] under the number 1394 [4].

3.1 Task Statement

Given N ships, each having a length of s_i, and M havens, each having a length of h_j, one needs to assign the ships to the havens, such that the total length of all ships assigned to the j-th haven does not exceed h_j. The constraints follow:

- $N \le 99$, $2 \le M \le 9$, $1 \le s_i \le 100$;
- $\sum s_i = \sum h_j$;
- the correct assignment always exists;
- time limit: 1 second;
- memory limit: 64 megabytes.

The task is clearly seen to have relationship with the multiknapsack problem [13]. This problem is known to be NP-hard *in the strong sense*: there are no known solutions that have running time bounded by a polynomial of the constraints. One may expect that the programming task presented above should be unsolvable. Nevertheless, there were 260 accepted solutions at the time of June, 15, 2009, which means that the test set was quite weak.

3.2 Test Encoding

To overcome the difficulties imposed by both test constraints and the genetic algorithms, a special encoding algorithm was designed. First, a test is encoded by a sequence of integers, which helps to fit the majority of constraints, including the hardest one, by design. Second, the sequence is encoded by a tree-shaped generator, which helps larger blocks of sequence elements to survive.

3.2.1 Test from a Sequence

The biggest problem with the test encoding is that there is a strong constraint — the correct assignment of ships to havens should always exist. Checking this constraint is equivalent to solving the whole problem. If using the direct encoding, the ratio of correct tests among all possible tests will be close to zero.

To overcome this, a special encoding was developed. The test is encoded by a sequence of integer numbers in the range from 0 to 100, where positive numbers denote the ships, and the contiguous intervals of positive numbers denote the havens (see Fig. 1). The sequence may have an arbitrary length. Sequences are not generated directly, but are produced from the presentation described below.

One can see that, if the order of items is not taken into account, every possible test can be encoded as a sequence. The

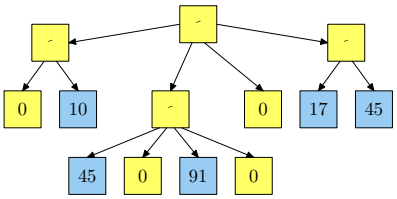

Figure 2: A tree-shaped generator

rate of incorrect tests generated from a random sequence is relatively small. The integer sequence is used as an intermediate representation between a test and an individual.

3.2.2 *Sequence from a Tree-Shaped Generator*

To allow for grouping of consecutive sequence elements, tree-shaped generators are introduced. A tree-shaped generator is a rooted tree. Each leaf of the tree contains a single sequence element. Each internal node produces a sequence that is a concatenation of sequences produced by its children. The sequence generated by the generator as a whole is produced by its root (see Fig. 2).

The depth of trees is unlimited. A generator is created using a stack of tree nodes with the following algorithm. In each of K iterations, one of two steps is performed:

- With the probability of 3/4, a leaf containing a randomly generated integer is pushed onto the stack. The integer is set to 0 with the probability of 1/7, and the positive values are chosen equiprobably.
- With the probability of 1/4, a random number of nodes is popped from the stack, then a new node which has these nodes as children is created and pushed onto the stack.

With $K = 108$, the average depth of randomly created generators is approximately equal to 12. K is selected in such a way that the tests created from generators are nearly maximal in size. It is shown below that the optimal size for a particular solution is effectively determined by the genetic algorithm.

The idea of tree-shaped generators is largely inspired by the genetic programming [10]. The tree-shaped generator is used as an individual for the genetic algorithm.

3.2.3 *Crossover and Mutation Operators*

The genetic operators on tree-shaped generators resemble the corresponding operators on parse trees. The crossover operator exchanges randomly selected subtrees of the parent trees. The mutation operator replaces a randomly selected subtree with a randomly generated tree of the same height.

3.3 Fitness Functions

As explained in Section 2.3, one needs to design a fitness function, considering the solution under the testing. For some typical solutions, the choice of the fitness function is explained below.

- The solution consists of a recursive function, which calls itself with various parameters. For those solutions, the fitness is the number of calls of such a function. If there are more than one such function, some or all of them are counted.
- The solution permutes the input data in random ways and tries to assign ships to havens in some greedy or dynamic programming based way. If the arrangement fails, one more permutation is tested, and so on. In this case, the fitness is the number of permutations performed until the answer is found.
- Some of the solutions consist of several parts, which are given some time to execute. If some part finds the answer, the execution is terminated, otherwise, the next part tries to find the answer. For such solutions, the fitness is a vector of fitness values for each of the part. These vectors are comparable lexicographically. To cover all the variations of such solutions, the time intervals for each part are set to the time limit, so, in the case of successful test generation, no variation can pass the test.
- For solutions that do not fall into the categories above, a special approach, depending on the kind of solution, is developed. Once this is done, an algorithm which tries to apply the fitness function to an arbitraty solution can be added to the framework for future use.
- If the solution gives wrong answer for the given test, the fitness value is set to the best possible value. This is because the main aim of test generation is to find a test that makes the solution fail. Using the ideas described in this paper, one can get the wrong answer verdict either by accident or by using a special fitness function. However, this verdict shows that the solution is wrong, so the aim is achieved and the test generation can be stopped.

For the solutions that use a random number generator, the random seed was fixed, and the time limit was increased. Tests generated this way were able to successfully fight the solutions of that type with non-fixed random seeds with a probability very close to 1.0.

3.4 Evolution Algorithm

In this research, little attention was paid for the optimality of the genetic algorithm scheme. The scheme described below was designed by several trial-and-error attempts. More work on this topic is left for the future.

In the scheme used in the research, the number of individuals in the generation S is selected between 200 and 500. To create next generation, the following steps are made.

- The number of new individuals generated is set to S.
- For every pair of new individuals to be generated, two parents are selected independently using a tournament selection. To perform the selection, 8 individuals are chosen independently at random. In each tournament, the probability for the fittest individual to win is 0.9.
- The best S individuals of parents, children and $S/4$ random individuals proceed to the next generation.

The presented algorithm contains a strong elitist selection, so it is prone to stagnation. When the stagnation occurs, that is, in 50 generations there is no fitness increase, the algorithm enters a new stage with all-random generation. The best individual from the killed generation is retained in a special individual bank for the future. When the next stage

of the algorithm reaches the level of that individual in fitness, the saved individual is injected in the new generation. The recombination of old and new optimums sometimes result in fitness higher in several orders.

3.5 Experiment and Results

At the time of the experiment, there were 260 accepted solutions. The process of test generation proceeded as follows, while there were accepted solutions on the test server:

- Among accepted solutions, the slowest one is chosen.
- One to three tests are generated against this solution. Generation of each test takes from 1 to 24 hours.
- These tests are added to the server, and all accepted solutions are rejudged.

In total, 25 solutions were used to generate tests. From the new tests, the "best" 11 tests were selected (a test T is considered to be "best", if no other test beats a superset of solutions beaten by T). These tests were added to the testing server under numbers from 48 to 58.

The majority of tests have the similar sizes: the number of ships is from 30 to 35, and the number of havens is 9. One may notice that the number of ships is significantly smaller than the limit. For solutions that are structurally different from the majority, the tests also differ. For example, one of the tests against some "special" solution has 84 ships and 6 havens.

In most of the trees representing the best tests, there are several similar subtrees, which have the size of order of 10. These subtrees are either equal or differ in small number of leaves. When comparing the best individuals from the adjacent generations, it can be seen that these subtrees are copied by the crossover operating on similar individuals. This effect is achieved using the tree representation at no cost. To get a similar effect using string-based individuals, one needs to use a two-point crossover, but in this case, there will be no grouping of consecutive sequence elements.

At the precise moment, in a year and a half after the generated tests were uploaded to the server, there are only nine accepted solutions, all of them submitted after the time of the test upload. Compared to 260 previously accepted ones, it is a very small number, which demonstrates the high quality of generated tests.

4. CONCLUSIONS

The approach of test generation for programming challenge tasks using genetic algorithms was introduced. It is designed to generate tests against inefficient solutions. The approach was demonstrated on the example of a programming task from the Timus Online Judge. The experiment showed the efficiency of the presented approach: the newly generated tests defeated all the solutions accepted by the time of experiment.

5. FUTURE WORK

5.1 Tuning the Search Algorithm

The scheme of the genetic algorithm presented in the paper is definitely not an optimal one. The possible ways of improve the situation include:

- altering the selection schemes;
- tuning the parameters of the algorithm itself and of the genetic operators;
- automated selection of the most efficient genetic operator or even the individual representation.

Genetic algorithms are not the only evolutionary algorithms that work in this area. For simpler individuals, evolution strategies are expected to perform equally well or even better [11].

5.2 Other Classes of Incorrect Solutions

The fitness functions described in the paper are designed to find test for solutions inefficient in terms of execution time. However, the similar approach could be used to defeat the solutions that use too much memory, even if they fit the time limit.

To generate tests against the solutions which are able to give wrong answers, another approach, similar to the one in [7, 15], should be developed. However, due to specifics of the programming tasks, the standard code coverage metrics, such as instruction coverage or branch coverage, may show poor results (i.e., the code is covered completely, but not all the possible cases are tested). New or more advanced coverage metrics should be invented for this to work.

6. REFERENCES

[1] ACM International Collegiate Programming Contest. http://cm.baylor.edu/welcome.icpc.

[2] International Olympiad in Informatics. http://www.ioinformatics.org.

[3] NEERC Contest Rules. http://neerc.ifmo.ru/information/contest-rules.html.

[4] Problem "Ships. Version 2". http://acm.timus.ru/problem.aspx?num=1394.

[5] Programming Contests at TopCoder. http://www.topcoder.com/tc.

[6] Timus Online Judge. The Problem Archive with Online Judge System. http://acm.timus.ru.

[7] J. T. Alander, T. Mantere, and P. Turunen. Genetic Algorithm Based Software Testing. In *Artificial Neural Nets and Genetic Algorithms*, pages 325–328, Wien, Austria, 1998. Springer-Verlag.

[8] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[9] V. Chvatal. Hard Knapsack Problems. *Operations Research*, 28(6):1402–1411, 1980.

[10] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[12] G. J. Myers. *The Art of Software Testing, Second Edition*. John Wiley & Sons, Inc., 2004.

[13] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, February 1995.

[14] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*. Springer Verlag, New York, 2003.

[15] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, 2004.