2012 11th International Conference on Machine Learning and Applications

Generation of Tests for Programming Challenge Tasks on Graph Theory using Evolution Strategy

Maxim Buzdalov
St. Petersburg National Research University
of Information Technologies, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
Email: mbuzdalov@gmail.com

***Abstract*—In this paper, an automated method for generation of tests against inefficient solutions for programming challenge tasks on graph theory is proposed. The method is based on the use of $(1 + 1)$ evolution strategy and is able to defeat several kinds of inefficient solutions. The proposed method was applied to a task from the Internet problem archive, the Timus Online Judge.**

I. Introduction

In most of the popular types of programming challenges [1]–[3] the correctness of solutions for the programming tasks is checked by running them on a number of pre-written tests under time, memory and other limits, and then checking the answer it gives. When successfully compiled and running on a test, a solution may end up with one of the following outcomes [4], [5]:

- 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 output file contains an incorrect answer;
- Accepted (AC) — the output file contains the correct answer.

A correct solution is a solution which gets Accepted outcome on every possible tests which fits the constraints given in the problem statement. The aim of the test set is to defeat solutions which are wrong or inefficient, that is, an ideal test set must contain at least one test for every imaginable wrong or inefficient solution.

The quality of test set for 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.

Tests for programming challenge tasks, except for the most trivial cases, are generated either by hand, for small-sized tests, or by programs written by jury members that create tests according to some predetermined patterns or at random. Thus, generation of such tests requires deep knowledge of the programming task and its possible solutions, and the quality of the tests depends very much on the human factor.

One of the ways to make the situation better is to automate the process of test creation as deep as possible. In this work, evolution strategy is used to generate tests that challenge the inefficient solutions. The use of evolution algorithm, such as evolution strategy, is ideologically inspired by a number of works on unit test generation [6], [7]. To the best knowledge of the author, there are no works, except for his own [8], that address automation of tests generation for programming challenge tasks.

The rest of the paper is structured as follows. In Section II, the method of test generation is described briefly. Section III describes a statement of the problem for which tests were generated. In Section V, examples of inefficient solutions and corresponding fitness functions are shown. Section VI presents the results of the experiment, and Section VII concludes.

II. The Method

The main idea of the method of test generation using evolutionary algorithms, such as evolution strategy, is to represent searching of a suitable test as a search problem.

A test should be represented as an individual of an evolutionary algorithm, and evolutionary operators, such as mutation operator, should be defined on this representation.

A fitness function should be defined to represent the desired quality of the test quantitatively. As this paper concentrates on test generation against inefficient solutions, the fitness function should generally be proportional to the amount of the resource which the inefficient solution uses too much: time, memory or something similar. However, using straightforward measures, such as passed time or consumed memory, as fitness function directly introduces problems with precision of the measurements and random noise [8].

To overcome this difficulty, a fitness function should be designed individually for each solution or for each class of the solutions. Usually this involves modifying the source code

978-0-7695-4913-2/12 $26.00 © 2012 IEEE
DOI 10.1109/ICMLA.2012.194

of the solution in order to calculate some value which will be the fitness value. Concrete examples of fitness functions for several types of solutions are introduced later in Section V.

As the test representation, evolutionary operators on this representation and fitness function are all defined, an evolutionary algorithm is run in order to optimize the fitness function until the termination criterion is met. It is mostly enough to stop the algorithm when, on the best tests evolved, the solution under test starts consuming more resources than allowed and, consequently, starts receiving negative outcomes.

III. Task Statement

The proposed method was tested on a programming task named "Work for Robots", which is located at Timus Online Judge [9] under the number of 1695 [10].

The formalized statement looks as follows: given an undirected graph, count the cliques in this graph. A clique is a full subgraph of the given graph (including an empty subgraph). The constraints follow:

- the number of vertices N does not exceed 50;
- the graph does not have loops or multiple edges between the same pair of vertices;
- the time limit is 2 seconds;
- the memory limit is 64 megabytes.

This problem is an NP-hard problem. However, there exists a solution which runs in time and space of $O(2^{N/2})$. The solution uses the "meet-in-the-middle" approach, which, briefly speaking, partitiones the graph into two parts, solves some subproblems for these parts and then merges the results together.

The graph which constitutes the test is represented as an adjacency matrix. The $(1 + 1)$ evolution strategy, which uses the test representation and mutation operators described below, is used as an optimization algorithm.

IV. Test Representation and Mutation Operations

The graph is represented by an adjacency matrix — a square matrix of size of $N \times N$, where N is the number of vertices of the graph. The elements of the matrix are boolean values.

The analysis of possible solutions shows that N should be set to 50 for all the solutions.

Two mutation operators are used. The first operator works as follows:

- the number of cells to be mutated K is selected from one of these values: 10, 100, 1000;
- a randomly selected cell (X, Y) and its counterpart (Y, X) are flipped. This operation is done K times.

The second operator flips a cell at once. The order of cells to flip is determined at random before the start of the optimization.

These two operators are performed in turn. It can be said that the first operator performs an explorative type of search, while the second one works exploitatively.

V. Fitness Functions

In this section, some approaches to design of the fitness function is described.

A. Backtracking

The purely backtracking solutions are the easiest to challenge. For solutions of this type, the fitness is set to the number of calls of the main (recursive) function. An example of the recursive function, together with the fitness function evaluation, is given below.

```
int fitness = 0;
long go(long mask) {
    if (mask == 0) {
        return 1;
    }
    if ((mask & (mask - 1)) == 0) {
        return 2;
    }
    //Increment the fitness
    ++fitness;
    int lastBit = numberOfTrailingZeros(mask);
    long rec1 = mask ^ (1L << lastBit);
    long rec2 = rec1 & edgesOf[lastBit];
    return go(rec1) + go(rec2);
}
```

The key fact which makes the solutions of this type slow is that they often call the function for the same argument many times, while the function, except for the fitness increase, has no side effects.

B. Backtracking with open hashing

One of the ways to speed up the backtracking solution is to remember the calculated key-value pairs in a hash map. To decrease the constant implementation factor, the open hashing is used. The example of a solution that uses the open hashing is given below (most of code borrowed from the previous example is omitted).

```
int hashSize = 1000003;
long[] key = new long[hashSize];
long[] val = new long[hashSize];
int indexFor(long k) {
    int i = (int) (k % hashSize);
    while (key[i] != 0 && key[i] != k) {
        i = (i + 1) % hashSize;
    }
    return i;
}
long go(long mask) {
    /* ... */
    int k = indexFor(mask + 1);
    if (key[k] != 0) return val[k];
    key[k] = mask + 1;
    /* ... */
    return val[k] = result;
}
```

A typical solution of this kind works fast enough, and the only way to make it fail is to overfull its hash table. The corresponding fitness function is computed the following way after the program itself has done with calculations:

```
int fitness = 0;
for (int i = 0; i < hashSize; ++i) {
    if (key[i] != 0) {
        ++fitness;
    }
}
```

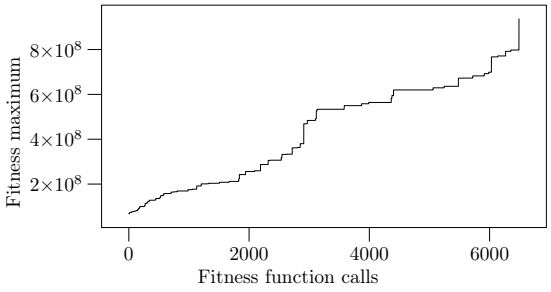

Fig. 1. Example of a FF plot for backtracking solutions

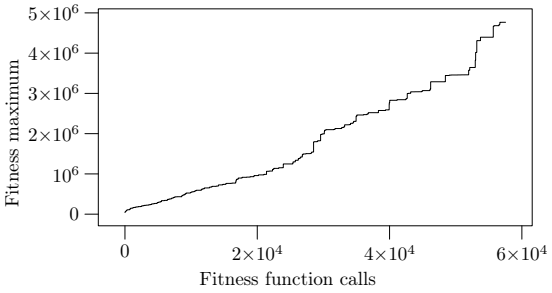

Fig. 2. Example of a FF plot for hashing solutions

C. Excessive memory allocation

Another way to stop calculating the same values many times, at least some of them, is using a cache for a part of the arguments.

The example solution below caches the arguments if they have only $K = 24$ lowest bits.

```
//The memory is allocated statically
int cache[1 << 24];
long long go(long long mask) {
    /* ... */
    if (mask < cache.length &&
            cache[mask]) {
        return cache[mask] - 1;
    }
    /* ... */
    if (mask < cache.length) {
        cache[mask] = result + 1;
    }
    return result;
}
```

If the parameter K is lower, then the solution times out. But if $K \geq 24$, then the cache array reaches or exceeds 64 megabytes in size. One may expect that this solution gets "Memory Limit Exceeded" outcome. It is not true generally, since the memory for the cache array is allocated by pages of, typically, 4 kilobytes by the operating system. The pages that were not used are not counted in total memory consumption.

To make a solution of this kind fail, one needs to make it "touch" all the pages it allocates. So, the fitness function is the number of memory pages used.

VI. The Results of the Experiment

At the moment of start of the experiment, there were 86 accepted solutions. Of these, ten solutions were selected to generate tests against them. One of them was later detected to be correct, and the remaining nine were defeated by nine newly generated tests. Examples of fitness function plots for some of the solutions belonging to three different types described above are shown on Fig. 1–3. For Fig. 1, the fitness function was equal to the number of calls to the recursive function equivalent to the function described in Section V-A. For Fig. 2, the fitness function was equal to the size of the hash table. For Fig. 3, the fitness function was equal to the number of used memory pages.

After adding these tests to the server (they received the numbers from 43 to 52 without 47), 45 previously accepted solutions failed to pass the new tests. In Table I there is a short summary on the number of solutions defeated by each of the generated tests with the indication of the outcome (WA, TL and ML, the meaning of the outcomes were explained in Section I).

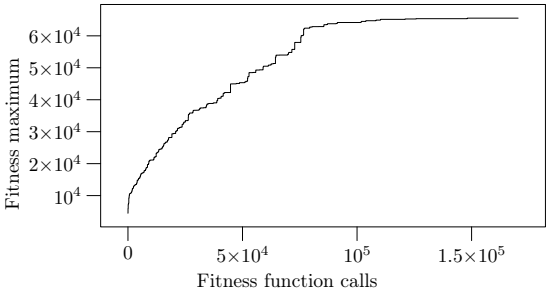

Fig. 3. Example of a FF plot for solutions with excessive memory allocation

TABLE I
Efficiency of newly generated tests

No. of test	Number of solutions not passing the test			
	All	WA	TL	ML
43	1	0	1	0
44	2	0	2	0
45	2	0	2	0
46	1	0	1	0
48	9	0	8	1
49	13	0	13	0
50	12	0	12	0
51	2	2	0	0
52	3	0	0	3
Total	45	2	39	4

VII. Conclusion

The method of test generation for programming challenge tasks on graph theory against inefficient solutions was described. This method is based on the use of $(1 + 1)$ evolution strategy.

The method was demonstrated on the example of the problem "Work for Robots", a programming task from the Timus Online Judge. The experiments showed the efficiency of the presented approach, as after adding newly generated tests more than a half of previously accepted solutions were rejected.

REFERENCES

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

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

[3] Programming Contests at TopCoder. [Online]. Available: http://www.topcoder.com/tc

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

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

[6] J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," in *Artificial Neural Nets and Genetic Algorithms*. Wien, Austria: Springer-Verlag, 1998, pp. 325–328.

[7] P. Tonella, "Evolutionary testing of classes," in *ISSTA*, 2004, pp. 119–128.

[8] M. Buzdalov, "Generation of tests for programming challenge tasks using evolution algorithms," in *Proceedings of the 2011 GECCO Conference Companion on Genetic and Evolutionary Computation*, New York, US, ACM, 2011, pp. 763–766.

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

[10] Problem "Work for Robots". [Online]. Available: http://acm.timus.ru/problem.aspx?num=1695