WORST-CASE EXECUTION TIME TEST GENERATION USING GENETIC ALGORITHMS WITH AUTOMATED CONSTRUCTION AND ONLINE SELECTION OF OBJECTIVES

Nikita Kravtsov, Maxim Buzdalov, Arina Buzdalova and Anatoly Shalyto

ITMO University
Computer Technologies Department
49 Kronverkskiy prosp., Saint-Petersburg, Russia
{linn1024,mbuzdalov,abuzdalova}@gmail.com, shalyto@mail.ifmo.ru

Abstract: *Worst-case execution time test generation can be difficult if tested programs use complex heuristics. Such programs may fail only on very small subsets of possible input data. Previous works show that evolutionary optimization (in particular, genetic algorithms) is a suitable tool for test generation under such conditions. We present an approach of automated integration of counters in the source code. There are two types of counters: one for counting the number of procedure calls, and another one for counting the number of loop executions. The values of these counters at the end of the program execution, as well as the execution time, serve as optimization objectives. We also propose two new methods for online selection of objectives. Together with the counter integration approach, they augment the already existing test generation method and increase its degree of automation. The experimental results for three example programs and for several objective selection algorithms are presented.*

Keywords: *helper-objectives, auxiliary objectives, worst-case execution time, performance testing*

1 Introduction

What is the input data which makes your program work too long? What is the maximum delay between an input event and an output action? These questions are related to the problem of determining the worst-case execution time of a program or a procedure.

These questions are undecidable, as almost all questions about programs, but one may hope to produce approximate answers in reasonable time. So it is reasonable to apply search-based optimization techniques, such as genetic algorithms, to the problem of worst-case execution time test generation [4].

Tools like Pex [19], CREST [6], CUTE [16] construct tests that achieve high coverage of lines or instructions of the tested code. However, they are not designed to construct tests with high execution time.

Theoretically, the test with the worst-case execution time can be found in tests that cover all feasible execution paths (like the one created by the PathCrawler tool [22]). Such paths can be ordered to omit some of them and to speed up the search [23]. However, it is difficult or impossible to apply the latter approach to programs containing loops or recursion (actually, the ideas from [23] were tested only for programs without loops).

Most of the works on evolutionary worst-case execution time test generation consider the case of real-time software testing [4,9,20,21]. Another application where worst-case execution time matters is generation of tests for programming challenge tasks [8]. There are some papers on related topics of testing algorithms of discrete mathematics [5] and SAT solvers [12].

In most of the papers on worst-case execution time test generation, the execution time serves as a fitness function. However, measurements of the execution time contain noise introduced by the operating system scheduling and the measured time is often proportional to a certain time quant. To overcome these difficulties, the paper [8] advise to include some counters into the source code of the tested program. Each of these counters is associated with a loop or a procedure and incremented each time the loop is entered or the procedure is called. The values of these counters, as well as the execution time, serve as fitness functions.

The approach based on optimizing the values of counters focuses the search to find tests which cause long time execution not due to, for example, cache misses, but due to weaknesses in the algorithm itself. Such property can make this approach promising for genetic software improvement (the GISMOE approach [10]).

There are two problems associated with this approach: the first problem is integration of the counters into the source code, and the second problem is the selection of the “correct” counter to optimize. The first of these problems was not addressed properly in the previous papers, as the counters were integrated by hand [8]. The second problem was solved using the EA+RL approach [7], which is based on reinforcement learning [18].

In this paper we extend the approach proposed in the paper [8]. The first difference is that the counters are integrated automatically into the source code by the ANTLR tool [13]. Due to this change, the number of counters, which was three or four previously [8], increases and becomes about 30 now. The second difference is that we use for objective selection not only reinforcement learning algorithms, but also two new algorithms.

We present the experimental results of the proposed approach on three solutions to the programming challenge task described in [8]. The results demonstrate that the presented algorithms for automated objective selection demonstrate an advantage over the reinforcement learning algorithms and over random fitness function selection.

2 Integration of Counters

The tested programs may come in two different ways: as separate executables or as procedures that can be called directly to compute all necessary fitness measures. However, the time needed to execute a separate program may be quite long on some platforms (e.g. Windows) compared to the execution time of the useful part of the tested program. Instead, we decided to run both genetic algorithm and tested programs in a single environment. As we implemented our genetic algorithm on the Java platform, we rewrite the tested programs in Java, so an invocation of the tested program corresponds to an invocation of a certain method of a Java class.

We implemented automated integration of counters into the source code written in Java programming language, however, the applicability of the method is not limited to Java only. There are two types of code locations where an increment of a counter should be inserted:

- beginnings of the methods (except for the `main` method);
- beginnings of the loops.

We created a small tool [2] based on ANTLR [13] to properly recognize locations where an increment should be done and to insert the instructions for incrementing the counters, as well as the declarations of the counters and the static methods to access and clear their values.

As the tested programs may execute for large amounts of time, it is needed to set a certain time limit and to interrupt the program when the time limit for a single run is exceeded. Java platform does not offer any means to stop a thread under a certain time limit. To solve the problem, the time consumed by the current thread since the invocation starts is checked in the same code fragments where the counters are incremented. When the time limit is exceeded, an exception is thrown, which is caught in the fitness evaluation procedure and processed accordingly. To reduce the number of system calls, these checks are performed every 2^{18} counter increments.

This simplified way of inserting counters has a problem in the multithreaded setting, when several threads may alter the values of counters concurrently. This, however, can be solved by loading as many copies of the tested program as needed using separate instances of Java classloader.

3 Objective Selection Algorithms

The previous section was dedicated to automated integration of counters to the source code. These counters, along with the execution time, serve as optimization objectives. However, we don't know which objectives are really suitable for optimization, so we need to select them during optimization. In this section, we propose two new objective selection algorithms and describe the already proposed ones.

3.1 Algorithm Alg-1

The first algorithm, which we call ALG-1 in this paper, works as follows. It consists of *meta-iterations*, the i-th meta-iteration (where i starts from zero) has the size of 2^i iterations. In each meta-iteration of size x, the following is performed:

- for each $t \in [1; K]$ where K is the number of objectives:
  1. The t-th objective is selected.
  2. The genetic algorithm is reinitialized (all individuals are replaced with randomly generated ones).
  3. x iterations of the genetic algorithm are performed.
  4. If an optimal individual is found, the algorithm stops.

The idea behind this algorithm is as follows: if optimization by the t-th objective yields the result for T iterations in average, the algorithm will eventually be optimizing the t-th objective for at least T iterations, and the number of iterations before this event does not exceed $2K \cdot T$, where K is the number of objectives.

Table 1: Tested solutions. LOC = Lines of Code.

Solution	Original language	Original LOC	Java LOC	Loop counters	Method counters
2208365	Pascal	104	118	14	2
3589819	C++	168	186	18	11
3600147	C++	221	246	20	12

3.2 Algorithm Alg-2

The second algorithm, which we call Alg-2, differs from Alg-1 only in that it does not restart the evolutionary algorithm after switching objectives. This has the following consequences:

- In the case when optimization by the t-th objective causes the generation of the genetic algorithm to degenerate (e.g. all individuals become equal), optimization by the $(t+1)$-th and subsequent objectives may yield worse results.
- In the case when objectives have similar landscapes, the algorithm can yield better results by not restarting the genetic algorithm, because each objective will be optimized from a "good" basis of the previous objective.

This means that Alg-2 may be worse or better than Alg-1 under different conditions. The subsequent experimental research will show that this is indeed true.

3.3 Other Algorithms

We compare the proposed algorithms with the ones used in [8]:

- Random objective selection. Each objective, including the execution time and the counter values, is selected at random for each iteration of the genetic algorithm. In the discussion of results, we will call it simply "Random".
- The Delayed Q-learning algorithm [17]. This is a reinforcement learning algorithm, which is used together with the EA+RL method as in [8]. A single reinforcement learning state is used. The reward is calculated as a difference between the best execution times in consecutive generations. This algorithm will be denoted as "Delayed".

4 Tested Programs

In this section, the programming challenge task "Ships. Version 2" [8], is described. We also present three solutions to this task, which are the programs we generate tests for.

4.1 Ships. Version 2

Programming challenges [1] are competitions for individuals or teams which require participants to solve problems by writing computer programs. Each problem statement specifies the problem to be solved, the format of input and output data, the limits for the input data, as well as maximum allowed amounts of time and memory (a time limit and a memory limit) that a solution may use. The solution is tested by running it on one or more predefined tests. If it gives correct answers on all the tests while not exceeding the time and memory limits, it is considered to be correct.

"Ships. Version 2" is a programming challenge task available at the Timus Online Judge [3]. The statement of this problem is as follows. Given N $(2 \le N \le 99)$ ships and M $(2 \le M \le 9)$ havens. Each ship has a length S_i $(1 \le S_i \le 100)$, and each haven has a positive length H_i. The sum of lengths of ships is equal to the sum of lengths of havens. One needs to write a program that assigns each ship to a haven, such as that for each haven the sum of lengths of ships assigned to it does not exceed the length of the haven. It is guaranteed that the solution always exists. The time limit is one second. The memory limit is 64 megabytes.

If bounds on N and M are removed, the problem "Ships. Version 2" can be shown to be NP-hard in the strong sense [14] by reduction to the Exact Cover by 3-Sets problem in the same way as this is proven for the multiple knapsack problem [11]. This means that no solution is known that solves all instances of the problem in time which is bounded by a polynomial of the input size or of the numbers in the input. This, in turn, means that it is very unlikely to solve all instances of this problem under one second on a single core of a typical (for the year 2014) desktop, server, or laptop computer.

4.2 Solutions

We analyse three different solutions to this problem with identifiers 2208365, 3589819, and 3600147. The first solution was originally implemented in Pascal, and two other solutions were implemented in C++. To be able to run them from Java code without creating a new process for each test, which is expensive, we translated them to Java.

For each of these solutions at least one test is known which makes the solution work for more than five seconds on a single core. Let us call such tests **good** tests. Note that the time limit of five seconds is used when test generation is performed. This is a measure to ensure that implementations of these solutions in different programming languages using different compilers will not run faster than one second. Also note that we don't check the memory limit in the current research.

Table 1 presents, for each of the solutions, its original language, numbers of lines of code in the original language and in Java port, and the number of loop and method counters integrated as described in Section 2.

5 Genetic Algorithm

Apart from the fitness function being optimized, the same genetic algorithm as in [8] is used in this paper. We outline it for the sake of consistency.

5.1 Individual Encoding

An individual is a list of integer numbers from 0 to 100. Each positive integer in this list produces a ship, and each interval of consecutive positive integers produces a haven.

Let $S_1, \ldots, S_N$ be a sequence of ships generated from an individual, and $H_1, \ldots, H_K$ be a sequence of havens. A test case which is generated from these sequences has the first and the last ships swapped, e.g. $S_N, S_2, \ldots, S_{N-1}, S_1$, so that solutions can not solve the problem too easily by assigning ships to havens greedily.

This kind of test case encoding satisfies two most difficult conditions from Section 4.1: first, the sum of lengths of ships is equal to the sum of lengths of havens, and second, the solution always exists. Note that, for a test case generated at random rather than using the described encoding, checking the latter condition actually implies solving the original problem, but in even harder conditions.

5.2 Evolutionary Operators

The same evolutionary operators are used as in [8]. A new individual is generated by putting $L = 50$ randomly generated integers to a list. The integers are generated as follows: zero is selected with the probability of 1/5, otherwise a positive value is selected equiprobably from a range of $[1; 100]$. The size of list was experimentally chosen to be equal to 50.

The mutation operator replaces every integer in the individual with a probability of $1/L$ with an integer generated randomly as above.

The following variation of two-point crossover operator is used. Assume that the elements of the individual are indexed from 1 to L. First, an exchange length X is selected randomly from a range of $[1; L]$. Second, an offset F_1 in the first individual is selected randomly from the range of $[1; L - X + 1]$. Third, an offset F_2 in the second individual is selected randomly from the same range independently of F_1. Last, the list subranges $[F_1, F_1 + X - 1]$ and $[F_2, F_2 + X - 1]$ from the first and the second individuals, respectively, are exchanged.

6 Experiments

In this section, the experiments are presented for the solutions listed in Section 4.2 using the objective selection algorithms from Section 3. For each solution and each objective selection algorithm, 100 runs were conducted. Each run is terminated if either a good test is generated or the iteration limit of 50000 iterations is reached. The hardware used in the experiment contains four AMD Opteron® 6234 processors at 2.4 GHz, with six cores each, which constitutes 24 cores in total.

The results of the experiments are presented in Table 2. For each solution, the p-values for all pairs of objective selection algorithms, that were computed by the Wilcoxon rank sum test implemented in R [15], are presented in Table 3. The null hypothesis used in the test is that the true location shift is equal to 0. It can be seen that:

- good tests (i.e. the tests for which the time limit of five seconds is exceeded) cannot be constructed for the solution 2208365 using the automatically constructed objectives and random selection algorithm or the one based on the Delayed Q-learning algorithm. The difference between our results and the results from [8] can be explained by the fact that in our work 16 objectives were used instead of three.

Table 2: Experimental results

Selection algorithm	Successful runs	Minimum of successful	Maximum of successful	Average of successful	Deviation of successful
			2208365		
Random	0/100	—	—	—	—
Delayed	0/100	—	—	—	—
ALG-1	100/100	3526	34123	10820.96	5846.99
ALG-2	95/100	545	46353	13590.56	9199.17
			3589819		
Random	0/100	—	—	—	—
Delayed	0/100	—	—	—	—
ALG-1	0/100	—	—	—	—
ALG-2	54/100	9461	48034	30837.78	12525.17
			3600147		
Random	100/100	66	47815	7928.87	7963.17
Delayed	100/100	100	36230	11204.14	9324.75
ALG-1	100/100	1073	21208	5054.70	3727.80
ALG-2	100/100	55	49933	7731.13	9616.52

Table 3: p-values for all pairs of objective selection algorithms.

	Random	Delayed	ALG-1	ALG-2
		2208365		
Random	—	1.0	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$
Delayed	1.0	—	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$
ALG-1	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$	—	0.023
ALG-2	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$	0.023	—
		3589819		
Random	—	1.0	1.0	$< 2.2 \cdot 10^{-16}$
Delayed	1.0	—	1.0	$< 2.2 \cdot 10^{-16}$
ALG-1	1.0	1.0	—	$< 2.2 \cdot 10^{-16}$
ALG-2	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$	—
		3600147		
Random	—	0.016	0.017	0.19
Delayed	0.016	—	$6.0 \cdot 10^{-6}$	0.00065
ALG-1	0.017	$6.0 \cdot 10^{-6}$	—	0.82
ALG-2	0.19	0.00065	0.82	—

- ALG-1 seems to perform slightly better than ALG-2 for 2208365, as the number of successful runs is greater and the mean for successful runs is less for ALG-1, although this difference is not very big from statistical point of view (p-value is 0.023).
- no algorithm except for ALG-2 generated good tests for the solution 3589819. The latter did it in 54 out of 100 runs.
- for the solution 3600147, all algorithms showed rather good performance (all runs finished under 50 000 iterations). Although, as for 2208365, ALG-1 seems to be better than ALG-2 in mean number of iterations, statistically this hypothesis is not supported (p-value is 0.82). All other pairs, except for Delayed vs. ALG-1, do not differ very much statistically, but to a smaller degree.

7 Conclusion

We presented an approach to worst-case execution time test generation. The first stage is to augment the source code of the tested program with counters which store the number of invocations of each method and each loop. The second stage is to run a single-objective genetic algorithm, where the fitness function is determined for each iteration by an objective selection algorithm.

The approach was evaluated on three solutions to the programming challenge task “Ships. Version 2”. For all the solutions, the test that makes the solution work for more than five seconds was evolved if the objective selection algorithm Alg-2 was used.

The presented approach is different from [8] in two aspects. First, it is more automated because it does not require a human to insert the counters into the source code. Second, two new objective selection algorithms, which do not need parameter tuning, are proposed and have shown better results, so they can be recommended as a default option for an implementation of the presented approach in a software tool for test generation.

The source code for the experiments, including the tested solutions, is available at GitHub [2].

References

[1] ACM International Collegiate Programming Contest. URL: http://en.wikipedia.org/wiki/ACM_ICPC

[2] Source code for experiments (a part of this paper). URL: https://github.com/mbuzdalov/papers/tree/master/2014-mendel-tests-counters

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

[4] Alander, J.T., Mantere, T., Turunen, P.: Genetic Algorithm based Software Testing. In: Proceedings of the 3rd International Conference on Artificial Neural Networks and Genetic Algorithms, pp. 325–328. Norwich, UK (1997)

[5] Arkhipov, V., Buzdalov, M., Shalyto, A.: Worst-Case Execution Time Test Generation for Augmenting Path Maximum Flow Algorithms using Genetic Algorithms. In: Proceedings of the International Conference on Machine Learning and Applications, vol. 2, pp. 108–111. IEEE Computer Society (2013)

[6] Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 443–446 (2008)

[7] Buzdalova, A., Buzdalov, M.: Increasing Efficiency of Evolutionary Algorithms by Choosing between Auxiliary Fitness Functions with Reinforcement Learning. In: Proceedings of the International Conference on Machine Learning and Applications, vol. 1, pp. 150–155 (2012)

[8] Buzdalova, A., Buzdalov, M., Parfenov, V.: Generation of Tests for Programming Challenge Tasks Using Helper-Objectives. In: 5th International Symposium on Search-Based Software Engineering, *Lecture Notes in Computer Science*, vol. 8084, pp. 300–305. Springer (2013)

[9] Gross, H.G., Mayer, N.: Search-based Execution-Time Verification in Object-Oriented and Component-Based Real-Time System Development. In: Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 113–120 (2003)

[10] Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation **XX**, 1–18 (2014)

[11] Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc., New York, NY, USA (1990)

[12] Moreno-Scott, J.H., Ortíz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Challenging Heuristics: Evolving Binary Constraint Satisfaction Problems. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 409–416. ACM (2012)

[13] Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007)

[14] Pisinger, D.: Algorithms for Knapsack Problems. Ph.D. thesis, University of Copenhagen (1995)

[15] R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2013). URL `http://www.R-project.org/`

[16] Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005)

[17] Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L.: PAC Model-free Reinforcement Learning. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 881–888 (2006)

[18] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, USA (1998)

[19] Tillmann, N., de Halleux, J.: Pex — White Box Test Generation for .NET. In: Tests And Proofs. Second International Conference, pp. 134–153 (2008)

[20] Tlili, M., Wappler, S., Sthamer, H.: Improving Evolutionary Real-Time Testing. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1917–1924. ACM (2006)

[21] Wegener, J., Mueller, F.: A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. Real-Time Systems **21**(3), 241–268 (2001)

[22] Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In: Lecture Notes in Computer Science Volume, vol. 3463, pp. 281–292 (2005)

[23] Williams, N., Roger, M.: Test Generation Strategies to Measure Worst-Case Execution Time. In: ICSE Workshop on Automation of Software Testing, pp. 88–96 (2009)