GENERATION OF TESTS AGAINST A GREEDY ALGORITHM FOR THE KNAPSACK PROBLEM USING AN EVOLUTIONARY ALGORITHM

Vladimir Mironovich and Maxim Buzdalov

ITMO University
Computer Technologies Department
49 Kronverkskiy prosp., Saint-Petersburg, Russia
mironovich.vladimir@gmail.com, mbuzdalov@gmail.com

Abstract: *Generation of tests for programming challenge tasks can be difficult when it is needed to cover wrong solutions (i.e. greedy algorithms) that use certain tricks (i.e. random shuffling of input data) to decrease the number of tests with wrong answers. In this paper, generation of tests for the knapsack problem is considered. Several tests that make a certain class of incorrect solutions fail with high probability are generated using an evolutionary algorithm.*

Keywords: *knapsack problem, greedy algorithm, randomized algorithm, algorithm verification, test generation, evolutionary algorithms*

1 Introduction

Programming challenges [1, 2] are competitions where participants solve various tasks by writing computer programs. The programs are judged by running on several tests and checking the answers. Each program must adhere certain time and memory limits. If a program gives correct answers for all these tests while not exceeding time and memory limits, it is considered to be correct.

While preparing a programming challenge task, jury members should construct a test suite that filters out as many incorrect (i.e. giving wrong answers on some tests) or inefficient (i.e. exceeding time and/or memory limits on some tests) solutions. This can be difficult in the case of incorrect solutions that are specially crafted to fail at as few test cases as possible. This paper is dedicated to one combination of a problem and a solution: the knapsack problem [16] and a greedy algorithm, which cannot solve the knapsack problem, but after many runs with shuffled input data it is able to produce a correct answer with high probability. A problem of good test generation, i.e. generation of a test, for which the greedy algorithm with shuffling is rarely able to produce a correct answer, is approached with an evolutionary algorithm.

The rest of the paper is structured as follows. Section 2 discusses the related work, mainly from the area of software engineering. Section 3 describes a programming challenge task which is essentially a knapsack problem with certain limits, and a correct solution to it, as intended by the jury. In Section 4.2, an incorrect solution to that problem, which is based on greedy algorithm with input data shuffling, is described. Section 5 is dedicated to initial exploration of the search space by random test generation. Section 6 describes the evolutionary algorithm and the experiment results. Section 7 concludes.

2 Related work

This section covers the related work in the field of test generation, as well as the work related to the knapsack problem.

2.1 High-coverage test generation

Development of software heavily relies on test suites that cover as many lines of code, instructions or execution paths as possible. Constructing such test suites is traditionally done by coders or by testers, which is difficult and expensive. Tools like Pex [18], CREST [7], CUTE [17], PathCrawler [22] construct test suites that achieve high coverage of lines, instructions or executable paths of the tested code. However, implementations of the incorrect algorithms of knapsack problems can sometimes be completely covered by a single randomly-generated test. In other words, high line coverage does not guarantee that the inputs are found where the answers differ from the correct ones.

2.2 Worst-case execution time test generation

Another area of automated construction of tests is generation of tests that show the worst-case execution time of the tested program.

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).

Search-based software testing contributed a number of papers to this area. Most of the works on evolutionary worst-case execution time test generation consider the case of real-time software testing [4, 5, 10–14, 19–21]. Another application where worst-case execution time matters is test generation for programming challenge tasks [8]. There are some papers on related topics of testing algorithms of discrete mathematics [6] and SAT solvers [15].

3 The "Bibliophile" programming challenge task

This paper is dedicated to test generation for a certain instance of the knapsack problem. This instance was proposed by one of the authors of this paper, as a jury member, at Petrozavodsk training camp in winter of 2009 as a part of a training challenge.

Out of 43 total solutions, seven were accepted, 13 received the Wrong Answer outcome, 16 got Time Limit Exceeded and five got Runtime Error. However, none of accepted solutions were actually correct – for each of them, a test can be constructed which made it output a wrong answer or fail with a runtime error. This can be explained by the poor quality of the test set. The aim of the current research is to fix this situation and to prevent such situations in the future.

3.1 The knapsack problem

The *0-1 knapsack problem* (will be referred to as simply *knapsack problem* in the rest of the paper) is a well-known combinatorial optimization problem. Given N items, each having the weight w_i and the cost c_i, and a knapsack with the weight capacity of W. One needs to find a subset of items of the maximum total cost that will fit into the knapsack, i.e. to determine $x_i \in \{0, 1\}$ such that $\sum_{i=1}^{N} w_i x_i \leq W$ and $\sum_{i=1}^{N} c_i x_i$ is maximum possible.

The knapsack problem is NP-hard [9], however, it is often referred to as "the easiest NP-problem", because there exist algorithms that solve a large fraction of big instances in polynomial (even linear) time [16].

3.2 The "Bibliophile" task

The "Bibliophile" task can be reduced to the knapsack problem with the following limits:

- all item weights are integers in the range $[1; W_{max}]$ where $W_{max} = 2009$;
- all item costs are equal to their respective weights (the subset sum problem);
- there are N items, $1 \leq N \leq 5000$;
- the capacity of the knapsack W does not exceed 10^9;
- the time limit is two seconds;
- the memory limit is 256 megabytes.

One is asked to output not only the maximum cost of the subset of items that can be fit to the knapsack, but the numbers of items themselves.

The time limit of two seconds roughly corresponds to 300–500 millions of simple operations with integers in a single thread on most personal computers and laptops. The exact limit on the number of operations depends on the clock speed of the CPU and different levels of random-access memory, as well as on the access pattern to the memory.

4 Solutions

This section describes one of the correct solutions to the "Bibliophile" challenge task, as well as the incorrect solution which is studied in this paper.

4.1 Correct solution

The standard pseudo-polynomial time algorithm runs using $O(N \cdot W)$ time and $O(N \cdot W)$ space (this is needed to restore the actual list of items). In this problem, W is bounded by 10^9, but when W is not less than the sum of weights of items, the answer is trivial. So we can think that $W \leq W_{max} \cdot N$. This transforms the complexity of the discussed algorithm into $O(N^2 \cdot W_{max})$. The maximum value of $N^2 \cdot W_{max}$ is $5000^2 \cdot 2009 = 5.0225 \cdot 10^{10}$, which is significantly greater than the allowed number of operations. This renders the algorithm unusable to solve this problem.

However, in Pisinger's PhD thesis [16] a similar but more efficient algorithm is described. It is shown that if one constructs a greedy solution (in the case of the subset sum problem, this is an arbitrary set of items that can be fit into the knapsack and is maximal by inclusion), then the optimal solution can always be achieved by the following steps:

- if the total weight of the current item set is less than W, add one item that is not in the set;
- if the total weight of the current item set is greater than W, remove one item from the set.

Note that in any state of the algorithm the total weight of the item set will never be outside the range $[W - W_{max} + 1; W + W_{max} - 1]$. This makes it possible to implement a pseudo-polynomial time algorithm with time and memory complexity of $O(N \cdot W_{max})$. As the maximum value of $N \cdot W_{max}$ is $5000 \cdot 2009 = 1.0045 \cdot 10^7$ and the implementation constant is not too large for both time and memory, such an algorithm will solve all possible instances of the problem under time and memory limits.

4.2 Solution 7511

This section describes a solution which was submitted during the Petrozavodsk training camp under the ID of 7511. It implements an algorithm which constructs the correct answer with a certain small probability. Such algorithms can produce correct answers with a guarantee only if they are allowed to work for infinite time. Practical implementations of such algorithms run them for a certain period of time and give out the best found answer.

The idea of this algorithm is to perform a number of random shuffles of items and run a greedy algorithm on the items. The algorithm tracks the best answer of all greedy answers. When the execution time of the algorithm exceeds 1.5 seconds, it terminates and prints the best found answer. The pseudocode of the algorithm is given in Algorithm 1.

Algorithm 1 The pseudocode of solution 7511

```
x ← item weights (equal to costs)
N = |x|
W ← knapsack capacity
A ← {} – the best found answer

while execution time is less than 1.5 seconds do
    a ← greedy solution for x
    if a is better than A then
        A ← a
    end if
    shuffle x randomly
end while
```

Let the probability of finding the correct answer on a certain test T be $p(T)$. Then the probability of not finding the correct answer in k iterations is $(1 - p(T))^k$. The aim of test generation is to minimize the value of $p(T)$ such that the expected running time until the answer is found is maximized. In fact, test generation may stop when the expected running time, or its estimation, exceeds the time limit for several times, because the probability of finding the correct answer empirically becomes small in this case.

5 Search space exploration

All experiments in this paper were conducted on a computer with an Intel® Core™ i7-3520M CPU clocked at 3.4 GHz in the hyperthreading mode and 8 gigabytes of RAM.

We tried to do random test generation to determine the areas of search spaces to focus on. Different values of N, the number of items, are tried. We determine the knapsack capacity as the total weight of items multiplied

Table 1: Difficulty of random tests for large ranges of N and C. Gray cells mean that a test was generated, for which the solution gives a wrong answer.

C\N	10	15	20	25	30	50	100	300	1000	5000
0.05	$1.0 \cdot 10^1$	$6.7 \cdot 10^1$	$3.1 \cdot 10^2$	$1.2 \cdot 10^3$	$3.1 \cdot 10^3$	$2.6 \cdot 10^3$	$2.3 \cdot 10^3$	$2.5 \cdot 10^3$	$2.1 \cdot 10^3$	$5.9 \cdot 10^2$
0.10	$3.3 \cdot 10^1$	$3.1 \cdot 10^2$	$2.1 \cdot 10^3$	$4.2 \cdot 10^3$	$2.6 \cdot 10^3$	$2.5 \cdot 10^3$	$2.4 \cdot 10^3$	$2.3 \cdot 10^3$	$2.2 \cdot 10^3$	$6.3 \cdot 10^2$
0.15	$7.6 \cdot 10^1$	$9.3 \cdot 10^2$	$3.9 \cdot 10^3$	$2.5 \cdot 10^3$	$2.6 \cdot 10^3$	$2.7 \cdot 10^3$	$2.6 \cdot 10^3$	$2.7 \cdot 10^3$	$2.2 \cdot 10^3$	$8.2 \cdot 10^2$
0.20	$1.6 \cdot 10^2$	$1.9 \cdot 10^3$	$3.2 \cdot 10^3$	$2.7 \cdot 10^3$	$2.8 \cdot 10^3$	$2.8 \cdot 10^3$	$3.1 \cdot 10^3$	$2.7 \cdot 10^3$	$2.4 \cdot 10^3$	$1.0 \cdot 10^3$
0.25	$2.4 \cdot 10^2$	$2.9 \cdot 10^3$	$2.9 \cdot 10^3$	$2.8 \cdot 10^3$	$2.8 \cdot 10^3$	$2.9 \cdot 10^3$	$3.0 \cdot 10^3$	$3.0 \cdot 10^3$	$2.5 \cdot 10^3$	$1.0 \cdot 10^3$
0.30	$3.5 \cdot 10^2$	$4.0 \cdot 10^3$	$2.9 \cdot 10^3$	$2.9 \cdot 10^3$	$2.9 \cdot 10^3$	$3.1 \cdot 10^3$	$3.2 \cdot 10^3$	$3.1 \cdot 10^3$	$2.6 \cdot 10^3$	$1.3 \cdot 10^3$
0.35	$4.9 \cdot 10^2$	$4.4 \cdot 10^3$	$3.0 \cdot 10^3$	$3.1 \cdot 10^3$	$3.0 \cdot 10^3$	$3.1 \cdot 10^3$	$3.2 \cdot 10^3$	$3.1 \cdot 10^3$	$2.8 \cdot 10^3$	$1.2 \cdot 10^3$
0.40	$6.1 \cdot 10^2$	$4.6 \cdot 10^3$	$3.2 \cdot 10^3$	$3.3 \cdot 10^3$	$3.3 \cdot 10^3$	$3.7 \cdot 10^3$	$3.7 \cdot 10^3$	$4.1 \cdot 10^3$	$3.3 \cdot 10^3$	$1.6 \cdot 10^3$
0.45	$7.0 \cdot 10^2$	$4.8 \cdot 10^3$	$3.5 \cdot 10^3$	$3.4 \cdot 10^3$	$3.8 \cdot 10^3$	$3.7 \cdot 10^3$	$3.8 \cdot 10^3$	$3.9 \cdot 10^3$	$3.4 \cdot 10^3$	$1.9 \cdot 10^3$
0.50	$8.2 \cdot 10^2$	$4.9 \cdot 10^3$	$3.7 \cdot 10^3$	$3.8 \cdot 10^3$	$3.8 \cdot 10^3$	$4.3 \cdot 10^3$	$4.4 \cdot 10^3$	$4.4 \cdot 10^3$	$4.0 \cdot 10^3$	$2.2 \cdot 10^3$
0.55	$9.7 \cdot 10^2$	$6.2 \cdot 10^3$	$4.1 \cdot 10^3$	$4.3 \cdot 10^3$	$4.0 \cdot 10^3$	$4.2 \cdot 10^3$	$4.5 \cdot 10^3$	$4.4 \cdot 10^3$	$4.4 \cdot 10^3$	$2.7 \cdot 10^3$
0.60	$9.6 \cdot 10^2$	$7.7 \cdot 10^3$	$4.3 \cdot 10^3$	$4.4 \cdot 10^3$	$4.6 \cdot 10^3$	$5.0 \cdot 10^3$	$4.9 \cdot 10^3$	$5.2 \cdot 10^3$	$4.8 \cdot 10^3$	$3.2 \cdot 10^3$
0.65	$9.2 \cdot 10^2$	$9.6 \cdot 10^3$	$5.1 \cdot 10^3$	$5.0 \cdot 10^3$	$5.2 \cdot 10^3$	$5.5 \cdot 10^3$	$5.4 \cdot 10^3$	$6.0 \cdot 10^3$	$5.4 \cdot 10^3$	$3.8 \cdot 10^3$
0.70	$8.8 \cdot 10^2$	$1.1 \cdot 10^4$	$5.7 \cdot 10^3$	$5.5 \cdot 10^3$	$5.9 \cdot 10^3$	$5.8 \cdot 10^3$	$6.8 \cdot 10^3$	$6.9 \cdot 10^3$	$6.9 \cdot 10^3$	$5.0 \cdot 10^3$
0.75	$8.2 \cdot 10^2$	$1.4 \cdot 10^4$	$7.5 \cdot 10^3$	$6.4 \cdot 10^3$	$6.2 \cdot 10^3$	$7.3 \cdot 10^3$	$7.1 \cdot 10^3$	$8.3 \cdot 10^3$	$8.1 \cdot 10^3$	$5.9 \cdot 10^3$
0.80	$7.0 \cdot 10^2$	$1.4 \cdot 10^4$	$1.4 \cdot 10^4$	$7.9 \cdot 10^3$	$7.6 \cdot 10^3$	$8.0 \cdot 10^3$	$8.5 \cdot 10^3$	$1.0 \cdot 10^4$	$1.0 \cdot 10^4$	$8.6 \cdot 10^3$
0.85	$4.9 \cdot 10^2$	$1.0 \cdot 10^4$	$5.1 \cdot 10^4$	$1.4 \cdot 10^4$	$1.0 \cdot 10^4$	$1.1 \cdot 10^4$	$1.2 \cdot 10^4$	$1.3 \cdot 10^4$	$1.3 \cdot 10^4$	$1.1 \cdot 10^4$
0.90	$4.1 \cdot 10^2$	$6.4 \cdot 10^3$	$6.6 \cdot 10^4$	$8.8 \cdot 10^4$	$2.1 \cdot 10^4$	$1.3 \cdot 10^4$	$1.5 \cdot 10^4$	$1.7 \cdot 10^4$	$1.9 \cdot 10^4$	$1.8 \cdot 10^4$
0.95	$2.1 \cdot 10^2$	$2.6 \cdot 10^3$	$2.4 \cdot 10^4$	$1.6 \cdot 10^5$	$3.4 \cdot 10^5$	$3.5 \cdot 10^4$	$2.9 \cdot 10^4$	$3.5 \cdot 10^4$	$3.7 \cdot 10^4$	$3.8 \cdot 10^4$

by a constant C and rounded down to the nearest integer, so we also tried different values of C. As a measure of "difficulty" of a test, we consider the number of iterations of the algorithm until the correct answer is found multiplied by N. Table 1 presents the results averaged over 1000 runs for several values of $N \in [10; 5000]$ and $C \in [0.05; 0.95]$.

As one can see from Table 1, a configuration with $N = 30$ and $C = 0.95$ produced better tests in average. For this configuration, one test was generated that made the solution give a wrong answer. However, running a solution for 1000 times on that test revealed that the wrong answer is given only in 504 out of 1000 runs. This result needs to be improved.

In the next section, we attempt to generate better tests in that region using an evolution algorithm.

6 Evolutionary Algorithm

To generate tests, we use a $(1 + 1)$ evolutionary algorithm.

For a run of the evolutionary algorithm, we fix the number of items N and a knapsack capacity ratio C. The individual is a list of N item weights. The knapsack capacity is determined as a maximum of the smallest weight and the sum of item weights multiplied by C rounded down to the nearest integer. The initialization is done by generating each item weight uniformly at random from the range $[1; 2009]$. The mutation operator changes each item weight w with a probability of $2/N$ to $\min(2009, \max(1, w + random(-Q, Q)))$, where $Q = 12$.

The fitness function is the number of iterations of the solution until a correct answer is found. We start the solution using a fixed random seed to ensure that the fitness value for a certain test is the same for different invocations. The evolution terminates when the time needed to find a correct answer exceeds 10 seconds. We motivate this enlarged time limit (10 seconds vs 1.5 seconds) by the hope that a test that makes a solution with a fixed random seed run for too long will also make a solution with an arbitrary random seed run for a reasonably long period of time. The algorithm is restarted from scratch when there is no fitness increase for 1000 iterations.

6.1 Experiment Results

The evolutionary algorithm is run 10 times for $N = 30$ and $C = 0.95$. In each of the runs, the optimization goal was reached – a test was generated, for which the solution with the fixed random seed could not find the correct answer in 10 seconds. The tests are shown in Table 2 along with the number of fitness evaluations until the test is found, the number of evolutionary algorithm restarts and the number of times a solution with an arbitrary random seed fails to find the correct answer in 1.5 seconds.

Table 2: Results of evolutionary algorithm

	N	W	Items	Evaluations	Restarts	Fails
1	30	27997	131 1911 2004 856 308 76 376 208 302 208 1837 1455 1300 926 1220 815 836 151 1056 90 696 631 1830 1661 811 1450 1812 1470 1629 1415	178173	127	813/1000
2	30	27516	57 1576 1616 1303 854 1299 593 1527 1560 1880 1023 1869 1573 1851 179 643 221 1180 1966 698 1475 230 44 593 449 517 239 97 44 1809	86098	58	788/1000
3	30	30775	1681 678 8 316 643 987 214 1756 574 1608 1242 49 166 1503 2002 1474 1953 1322 1814 1592 1561 1256 1743 67 1465 1672 7 226 1836 980	67857	51	734/1000
4	30	27666	1581 1084 1631 1582 30 1757 827 926 958 175 67 1566 1123 411 54 1326 1934 830 201 553 384 744 1208 202 742 1168 725 1838 1628 1868	43309	34	552/1000
5	30	27997	131 1911 2004 856 308 76 376 208 302 208 1837 1455 1300 926 1220 815 836 151 1056 90 696 631 1830 1661 811 1450 1812 1470 1629 1415	80671	50	798/1000
6	30	27618	1165 252 1960 100 1460 1511 455 711 79 1173 1291 349 21 96 1934 532 1653 1980 1131 1210 340 698 1681 873 617 217 1599 858 1831 1295	33187	24	797/1000
7	30	27464	608 67 1 1871 1906 365 1747 42 510 1560 29 988 657 290 1526 1313 1664 507 1003 1691 1678 1303 1022 1031 1124 588 999 411 555 1854	166855	120	521/1000
8	30	27997	131 1911 2004 856 308 76 376 208 302 208 1837 1455 1300 926 1220 815 836 151 1056 90 696 631 1830 1661 811 1450 1812 1470 1629 1415	99434	61	811/1000
9	30	24285	730 1321 1835 1 1005 687 725 316 1914 1037 579 439 726 1137 1165 1734 1896 1 1 1785 701 216 970 226 79 561 602 988 1183 1004	101993	72	810/1000
10	30	28234	1243 1834 330 926 1498 1858 1514 631 1818 1107 502 324 623 1047 1061 1891 945 1608 251 1397 23 1386 62 1683 167 986 1089 184 23 1710	18974	14	531/1000

One can note that in 7 out of 10 runs, the number of times the solution failed to find an answer is significantly higher than in the found test generated at random.

7 Conclusion

We presented an approach for test generation to the knapsack problem. The generated tests make a greedy solution to the knapsack problem, which uses random shuffling of input data, return wrong answers with a high probability. The preliminary experiments with random test generation showed that good tests have moderately small number of items and knapsack capacities that differ only slightly from the total sum of item weights. The resulting tests were generated using an evolutionary algorithm.

The source code for the experiments is available at GitHub [3] for reproduction.

Acknowledgement: This work was financially supported by the Government of Russian Federation, Grant 074-U01.

References

[1] ACM International Collegiate Programming Contest. URL: http://en.wikipedia.org/wiki/ACM_ICPC
[2] International Olympiad in Informatics. URL: http://www.ioinformatics.org

[3] Source code for experiments (a part of this paper).
URL: https://github.com/mbuzdalov/papers/tree/master/2014-mendel-knapsacks
[4] Alander, J.T., Mantere, T., Moghadampour, G.: Testing Software Response Times Using a Genetic Algorithm. In: Proceedings of the 3rd Nordic Workshop on Genetic Algorithms and their Applications, pp. 293–298 (1997)
[5] Alander, J.T., Mantere, T., Moghadampour, G., Matila, J.: Searching Protection Relay Response Time Extremes using Genetic Algorithm – Software Quality by Optimization. Electric Power Systems Research **46**(3), 229–233 (1998)
[6] 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)
[7] 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)
[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] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
[10] Gross, H.G.: Measuring Evolutionary Testability of Real-Time Software. Ph.D. thesis, University of Glamorgan (2000)
[11] Gross, H.G.: A Prediction System for Evolutionary Testability Applied to Dynamic Execution Time Analysis. Information and Software Technology **43**(14), 855–862 (2001)
[12] Gross, H.G., Jones, B.F., Eyres, D.E.: Structural Performance Measure of Evolutionary Testing Applied to Worst-Case Timing of Real-Time Systems. IEEE Proceedings - Software **147**(2), 25–30 (2000)
[13] Gross, H.G., Mayer, N.: Evolutionary Testing in Component-based Real-Time System Construction. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 207–214 (2002)
[14] 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)
[15] 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)
[16] Pisinger, D.: Algorithms for Knapsack Problems. Ph.D. thesis, University of Copenhagen (1995)
[17] Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005)
[18] Tillmann, N., de Halleux, J.: Pex — White Box Test Generation for .NET. In: Tests And Proofs. Second International Conference, pp. 134–153 (2008)
[19] Tlili, M., Wappler, S., Sthamer, H.: Improving Evolutionary Real-Time Testing. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1917–1924. ACM (2006)
[20] 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)
[21] Wegener, J., Sthamer, H., Jones, B.F., Eyres, D.E.: Testing Real-Time Systems using Genetic Algorithms. Software Quality Journal **6**(2), 127–135 (1997)
[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)