2013 IEEE Congress on Evolutionary Computation
June 20-23, Cancún, México

Adaptive Selection of Helper-Objectives for Test Case Generation

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

Arina Buzdalova
St. Petersburg National Research University
of Information Technologies, Mechanics and Optics
49 Kronverkskiy prosp.
St. Petersburg, Russia, 197101
Email: abuzdalova@gmail.com

***Abstract*—In this paper a method of adaptive selection of helper-objectives in evolutionary algorithms, which was previously applied to model problems only, is applied to generation of test cases for programming challenge tasks. The method is based on reinforcement learning. Experiments show that the proposed method performs equally well compared to the best helper-objectives selected by hand.**

I. Introduction

Single-objective optimization can be enhanced by adding helper-objectives, or helpers [1], but how should we choose the most efficient ones, and when should we use the particular helper? A method designed to solve these issues was proposed in the previous works [2]–[5]. The method is called EA + RL. It turned to be successful in solving some model problems. In the current work it is applied to a practically important problem of software testing, namely, generation of test cases for programming challenge tasks.

The paper is structured as follows. In Section I-A the details of the helper-objective approach are given. In Section I-B there is some basic information on programming challenge tasks. Section II describes a problem of test case generation, and Section III gives an overview on the method of its solving. In Section IV the experiment results are presented. Section V concludes.

A. Helper-Objective Approach

There are several techniques that involve some additional objectives in order to enhance performance of evolutionary algorithms (EAs). In multiobjectivization technique [6] all the objectives are optimized simultaneously by some multi-objective algorithms (MOEAs) [7], [8]. In this technique the objectives should be specially developed in order to increase the optimization performance. It was shown that adding an inefficient objective leads MOEAs to fail on the considered model problems [4].

Helper-objective approach also involves using MOEAs, but it requires a strategy of choosing the helper to be optimized at the current population [1]. The strategy can be either random, or ad-hoc [9]. The random one is general, but it does not take advantage of problem characteristics. At the same time, ad-hoc strategies can be efficient, but they lack generality.

Previously proposed EA+RL method incorporates helper-objectives into single-objective EA. It requires less computational effort than MOEA-based methods, which makes it more applicable to such resource-consuming problems as test case generation.

EA + RL provides an adaptive strategy of helper selection based on reinforcement learning [10]–[12]. Reinforcement learning is used to select the most efficient helper to be optimized in the current population of the evolutionary algorithm. It was shown that reinforcement learning algorithms manage to choose efficient objectives and to ignore the inefficent ones [4]. The selection strategy used in EA + RL is problem independent and it allows to learn some features of the problem as well, thus the method seems to increase both efficiency and generality of the helper-objective approach.

There are several works that investigate using reinforcement learning for adjustment of EAs. In some of them tuning of numerical parameters such as mutation probability and population size is considered [13], [14], in other papers evolutionary operators selection [15], [16] is investigated. Using reinforcement learning as a strategy of choosing helper-objectives in EAs was proposed in EA + RL method for the first time.

B. Programming Challenge Tasks

A programming challenge [17]–[20] is a competition where participants compete in writing computer programs which solve certain problems. A programming challenge task includes the formulation of the problem, the format of the input and output data, the constraints on the input data, the output data correctness criteria and the time and memory limits, which the solutions should admit to.

In most types of programming challenges the correctness of solutions is checked by running them on a number of pre-written test cases and then checking the answer they give. If a solution produces a correct answer for each test case while not exceeding time and memory limits, it is considered to be correct.

It is assumed that if for a certain task there exists an algorithm or its implementation which may produce an incorrect answer (e.g. greedy algorithm, or bugs in implementation, or incomplete case analysis) or may exceed the time or memory

978-1-4799-0454-9/13/$31.00 ©2013 IEEE

TABLE I. PROCESS TIME MEASUREMENT: LINUX, CONFIG_HZ = 100

Program No	Execution time, ms					
1	0	10	10	10	10	10
	10	10	10	10	10	10
2	60	50	60	60	60	60
	60	60	60	50	60	60
3	200	200	200	200	200	200
	200	200	200	200	210	200
4	470	480	470	470	470	470
	470	470	470	470	480	470
5	930	920	930	920	930	920
	920	930	920	920	920	920
6	1630	1620	1640	1620	1610	1640
	1600	1600	1600	1610	1620	1630

TABLE II. PROCESS TIME MEASUREMENT: WINDOWS VISTA SP1

Program No	Execution time, ms					
1	15	0	0	15	0	0
	15	0	0	15	15	0
2	31	46	31	46	31	31
	46	31	46	31	46	46
3	140	140	140	171	156	171
	156	156	125	125	171	140
4	296	359	375	421	406	312
	265	312	296	265	281	375
5	656	656	609	453	625	640
	640	500	593	593	578	656
6	1125	968	1078	1093	1109	1125
	1140	1140	1031	1125	1156	1093

limit on some test cases, then at least one such test case exists in the testset of the task.

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

One of the ways to deal with these issues is to automate the process of test case creation as deep as possible. In this work, test case generation is performed using evolution algorithms. The use of evolution algorithms is ideologically inspired by a number of works on unit test generation [21], [22]. The previous works on this topic show that the evolutionary approach is suitable for generation of tricky test cases that are unlikely for a human to come up with [23], [24].

II. PROBLEM DESCRIPTION

In this paper generation of test cases for solutions of a particular programming challenge task is described. In particular, generation of test cases against inefficient solutions is addressed. In [23] it is shown that the running time of a program is often a bad objective to optimize for two reasons. First, it is noisy because of operation system scheduling algorithms and hardware events. Second, the measured time intervals are quantized.

In tables I and II the process time measurements for 12 runs of six different test programs are provided. Two different computers were used, one with Linux and another one with Windows. It can be seen that time intervals on Linux with CONFIG_HZ kernel option set to 100 are multiples of 10 milliseconds, while on Windows Vista the quant size is between 15 and 16 milliseconds. Both of these values constitute a significant part of a typical time limit of two seconds used in programming challenges.

From the above it can be concluded that the running time of a program is a noisy function with relatively small number of discrete values, which makes it hard to optimize.

An approach to address this problem is proposed in [23] and later extended in [24]. The idea of this approach is to integrate one or more counters to the source code of the program. When a solution finished working with a test, the values of these counters can be used as fitness functions.

Given a certain solution to a programming challenge task and the counters integrated into its source code, the question of what counters should be optimized to generate test cases more efficiently is open. In [23], [24] this question was solved by trial and error. The goal of this research is to automate the selection of the most efficient fitness functions using reinforcement learning.

III. METHOD DESCRIPTION

EA + RL method is based on guiding an evolutionary algorithm with reinforcement learning. According to [3], [4] the principal scheme of the method is as follows. The reinforcement learning *agent* interacts with the *environment* associated with the evolutionary algorithm. It chooses the fitness function to be used in the next population. The fitness function is selected from a set of all given objectives, which includes the helper objectives and the objective to be optimized, or the *target* objective, as well. Then the agent gets a *reward* based on the difference of the target fitness in two sequential populations as well as a *state* mapped from the newly evolved population and the process repeats. The agent uses a reinforcement learning algorithm that maximizes the total reward. The higher is the reward, the bigger is the increase of the target fitness, so good choices of the agent lead to the better performance of the optimization algorithm.

Let us describe the interaction between reinforcement learning agent and the EA more formally. The basis of this formalization was originally proposed in [3] and developed in [4].

In order to set the reinforcement learning task [10], we should define the set of actions A, a definition of the environment states $z \in Z$ and the reward function $R : Z \times A \to \mathbb{R}$. Let x be an individual evolved by the evolutionary algorithm. Denote the i-th population by G_i. The set of actions A corresponds to the set of all objectives, consisting of t — the target objective and the elements of H — the set of helper-objectives: $A = H \cup \{t\}$. Taking an action means choosing some objective $f_i \in A$ as the fitness function that is used in the population G_i.

Let us define the reward function $R : Z \times A \to \mathbb{R}$, which is calculated after choosing the criterion f_i in the state z_{i-1} and generating G_i. It depends on the difference between fitness of individuals at sequential populations and is the highest when

fitness increases:

$$R(z_{i-1}, f_i) = \frac{\sum_{x \in G_i} t(x) - \sum_{x \in G_{i-1}} t(x)}{\sum_{x \in G_i} t(x)} + \\ + k \sum_{f \in H} \frac{\sum_{x \in G_i} f(x) - \sum_{x \in G_{i-1}} f(x)}{\sum_{x \in G_i} f(x)},$$

where k is a discount parameter.

This reward takes into account all individuals in a population. In previous works the reward depended on the target fitness of the best individual only [3], [4]. It was revealed from preliminary experiments that the new reward definition is more efficient, at least for test case generation. The preliminary experiments also showed that using a single state z is efficient.

Despite new reward and state definitions, the principle scheme of the EA + RL method did not change since [3]. The pseudocode of the method is cited in Listing 1.

Listing 1 The EA + RL method.

```
Initialize the RL agent
Set the number of the current population: i ← 0
Generate the initial population G_0
while (EA termination condition is not reached) do
    Evaluate the state z_i and pass it to the RL agent
    Get the FF for the next population f_{i+1} from the RL agent
    Evolve the next population G_{i+1}
    Calculate the reward: r ← R(z_i, f_{i+1}) and pass it to the RL agent
    Increase the number of evolved populations: i ← i + 1
end while
```

IV. Experiment

During the experiment, test cases for a sample solution were generated using a single-objective evolutionary algorithm, EA + RL and a MOEA. 100 runs of each algorithm were performed. The aim of the experiment was to confirm that EA + RL performs well enough to replace manual choosing between helpers and to compare it with the MOEA-based method proposed in [1]. The detailed description of the experiment and analysis of the results are presented in the following subsections.

A. Task Statement

As in [23], we consider a programming challenge task "Ships. Version 2". This task is located at the Timus Online Judge [25] under the number 1394 [26].

The task formulation is as follows. There are N ships, each of length s_i, and M havens, each of length h_j. It is needed to allocate ships to the havens, such that the total length of all ships assigned to the j-th haven does not exceed h_j. It is guaranteed that the correct assignment always exists. The constraints are as follows:

- $N \leq 99$, $2 \leq M \leq 9$, $1 \leq s_i \leq 100$;
- $\sum s_i = \sum h_j$;
- time limit is 1 second;
- memory limit is 64 megabytes.

This problem is a special case of the multiple knapsack problem, which is known to be NP-hard in strong sense [27] (i.e. no solutions are known which have a running time polynomial of any numbers in the input). Due to this fact and high limits on the input data, it is very unlikely that every possible problem instance can be solved under the specified time and memory limits. However, for the most sophisticated solutions it is very difficult to construct a test case which makes them exceed the time limit.

B. Sample Solution

For this problem all known solutions implement branch-and-bound algorithms with different initial approximations and various heuristics. We chose one of these solutions which is generally quite fast, but its running time increases drastically as the complexity of the test case grows. This behaviour allows to perform many experiments in a short amount of time.

The structure of the solution is given in Listing 2. According to the approach from [23], we introduced three counters: "iterations", "length", and "tuple". The initialization algorithms for these counters are included in the above mentioned listing.

Listing 2 Scheme of a sample solution

```
Read the input data
iterations ← 0, length ← 0, last ← 0
while (solution not found) do
    Randomly shuffle ships and havens
    last ← 0
    Call the recursive ship arranging procedure
        For each call to this procedure, last ← last + 1
    if (solution is found) then
        Write the answer
    else
        iterations ← iterations + 1
        length ← length + last
        last ← 0
    end if
end while
tuple ← 10^9 · iterations + last
```

Another counter, "time", which equals the running time in milliseconds of the solution on the test case is added by the testing framework.

C. Genetic Algorithms

In this section, individual encoding, evolutionary operators and genetic algorithms used in the paper are described.

Individual encoding. To reduce the search space by satisfying a number of constraints imposed on a valid test in Section IV-A, we use a special test encoding scheme similar to one proposed in [23]. The 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 (see Fig. 1).

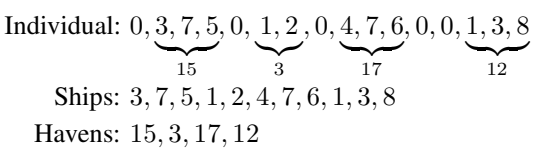

Fig. 1. Ships and havens generated from an individual

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 the 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 IV-A: 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 is at least as hard as solving the problem. Some other conditions may be violated (e.g. the number of havens may be out of bounds), but the probability of this event is relatively small, and if such a test case is ever generated, all fitness values are set to zero without running a solution on this test case.

Evolutionary operators. 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 50 is chosen experimentally, and, despite the fact not every test case can be produced, the results are good nevertheless.

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 second individuals respectively are exchanged.

Single-objective algorithm. A single-objective genetic algorithm is used for optimization of each single objective and for working with reinforcement learning. The size of the population is 200. To create a new population, a tournament selection with tournament size of two and the probability of selecting a better individual of 0.9 is used. After that, the crossover and mutation operators are applied with the probability of 1.0. To form a new population, the elitist strategy is used with the elite size of five individuals. The genetic algorithm is terminated either when an individual, for which the running time of the tested solution exceeds five seconds, is evolved, or 10000 populations are processed.

For optimization of single objectives, an additional heuristic is applied. If for 1000 populations the best fitness value does not change, then the current population is cleared and initialized with newly created individuals.

Multi-objective algorithm. The proposed method is compared with the MOEA-based method proposed in [1]. For optimization of more than one objective, a fast variant of the NSGA-II algorithm [28] proposed in [29] is used. The population size of 200 is used. Except for the version of tournament selection and nondominated sorting based selection strategy, which is traditionally used in NGSA-II algorithms, the evolutionary operation pipeline is the same as in the single-objective case. The termination criterion is also the same as in the previous section. The objectives being optimized are the target objective and a helper one, thus the optimization is two-objective. Such algorithm is claimed to be the best in average [1].

D. Reinforcement Settings

Two different reinforcement learning algorithms were implemented: Q-learning [10] and Delayed Q-learning [30]. In Q-learning algorithm the ε-greedy strategy with $\varepsilon = 0.3$ was used. The learning speed and the discount factor were $\alpha = 0.4$ and $\gamma = 0.001$ respectively.

Delayed Q-learning was restarted every 50 populations to prevent stagnation. The update period was $m = 5$, the bonus reward $\varepsilon = 0.001$ and discount factor $\gamma = 0.1$.

The discount parameter used to calculate the reward was set to $k = 0.5$. All the parameter values were set on the basis of preliminary experiment results.

E. Mean and Diversity Recalculation

As described in Section IV-C, for performance reasons, the number of populations in the experiment is limited to 10000. This means that for some runs the goal of the optimization (evolving a good test case) may not be reached, and it is impossible to calculate the average of the number of populations to finish the optimization.

However, it is possible to estimate this value, if we assume that the algorithm is restarted when the number of populations reaches 10000 and the goal of optimization is not reached. Let E_S be the average of the number of populations for successful runs[1], R be the ratio of successful runs, G be the maximum number of populations until restart. Then the expectation of the number of populations until success E can be estimated by the equation:

$$E = E_S \cdot R + (G + E) \cdot (1 - R),$$

which, after solving this equation, transforms to

$$E = E_S + \frac{1 - R}{R} G. \tag{1}$$

To find a similar formula for the standard deviation, we need first to compute the expectation for the squared number

[1]A run is considered to be successful if the goal of optimization is reached: a test case is generated such that the solution executed on this test case exceeds the time limit.

TABLE III. EXPERIMENT RESULTS

Algorithm	Fitness functions	Successful runs, %	Average populations number	σ
GA + Delayed Q-learning	all	90	4411	4352
GA	tuple	89	4497	4716
GA + Q-learning	all	85	4580	5186
NSGA-II + Random	all	82	4820	5598
GA + Random	all	76	6074	6788
GA	iterations	74	8182	7459
GA	length	51	15030	13969
GA	time	1	990319	994987

of populations to success Q. Let Q_S be this value for the succeeded runs under G populations, then:

$$Q = Q_S \cdot R + (Q + 2GE + G^2) \cdot (1 - R),$$

and, after solving the equation:

$$Q = Q_S + \frac{1-R}{R}(G^2 + 2GE). \quad (2)$$

Let the standard deviation of the number of populations for the successful runs be D_S and the standard deviation of the number of populations until success be D. By their definitions, $D_S^2 = Q_S - E_S^2$ and $D^2 = Q - E^2$. In latter, substitute Q from (2):

$$D^2 = Q_S + \frac{1-R}{R}(G^2 + 2GE) - E^2.$$

Replacing Q_S by $D_S^2 + E_S^2$ and extracting the square root, we get the final formula for D:

$$D = \sqrt{E_S^2 - E^2 + D_S^2 + \frac{1-R}{R}(G^2 + 2GE)}. \quad (3)$$

F. Results

The results of the runs are presented in Table III. The algorithms are sorted by the increase of the mean number of populations needed to evolve a "good" test case. The mean and the diversity were calculated using the formulae 1 and 3 respectively. A test case is considered to be"good" if it makes the solution exceed the time limit. Successful runs are the runs in which a good test case was evolved.

GA corresponds to the single-objective genetic algorithm. GA + Delayed Q-learning and GA + Q-learning are implementations of the EA + RL method using the corresponding reinforcement learning algorithms. NSGA-II + Random corresponds to the algorithm from [1]. In GA + Random fitness function is randomly chosen from all the objectives at the each population of the single-objective genetic algorithm.

We can see that, among the fixed objectives, the "tuple" function is the best one, followed by "iterations", then "length", then "time". As it was mentioned in Section II, the running time is indeed a bad objective to optimize. Results for all these functions are clearly distinguishable, and p-value for all of them, which was calculated using the ANOVA test [31], is far less than 10^{-3}.

The top four results produced by the delayed Q-learning, the "tuple" function, the ε-greedy Q-learning and the NSGA-II algorithm form a statistically indistinguishable group with their total p-value of 0.945. The results of the GA + Random seem to be more or less different from the top group: the pairwise p-values between the GA + Random and the members of the top group are 0.04, 0.058, 0.082, and 0.156, respectively. In general, it can be said that using reinforcement learning is in average more efficient than choosing random objectives.

V. CONCLUSION

A previously proposed method of adaptive helper selection was applied to a practically significant problem of generation test cases for programming challenge tasks. The features of this object domain were described in detail. Particularly, it was shown that the running time fitness function is inefficient and should be replaced by some helpers. Mean and diversity recalculation was proposed. It may be useful in experiments when an optimal solution could not be found steadily by an evolutionary algorithm.

It was shown that the proposed reinforcement learning based method is efficient enough to replace manual selection of helper-objectives. The method was also compared with the NSGA-II-based selection strategy from [1]. The proposed method is at least as efficient as the NSGA-II-based one. The statistically significant difference can possibly be obtained with more algorithm runs.

REFERENCES

[1] M. T. Jensen, "Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation," *Journal of Mathematical Modelling and Algorithms*, vol. 3, no. 4, pp. 323–347, 2004.

[2] A. Afanasyeva and M. Buzdalov, "Choosing best fitness function with reinforcement learning," in *Proceedings of the Tenth International Conference on Machine Learning and Applications, ICMLA 2011*, vol. 2. Honolulu, HI, USA: IEEE Computer Society, 2011, pp. 354–357.

[3] A. Afanasyeva and M. Buzdalov, "Optimization with auxiliary criteria using evolutionary algorithms and reinforcement learning," in *Proceedings of 18th International Conference on Soft Computing MENDEL 2012*, Brno, Czech Republic, 2012, pp. 58–63.

[4] A. Buzdalova and M. Buzdalov, "Increasing efficiency of evolutionary algorithms by choosing between auxiliary fitness functions with reinforcement learning," in *Proceedings of the 11th International Conference on Machine Learning and Applications, ICMLA 2012*, vol. 1, 2012, pp. 150–155.

[5] A. Buzdalova and M. Buzdalov, "Adaptive selection of helper-objectives with reinforcement learning," in *Proceedings of the 11th International Conference on Machine Learning and Applications, ICMLA 2012*, vol. 2, 2012, pp. 66–67.

[6] J. D. Knowles, R. A. Watson, and D. Corne, "Reducing local optima in single-objective problems by multi-objectivization," in *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, ser. EMO '01. London, UK: Springer-Verlag, 2001, pp. 269–283.

[7] K. Deb, *Multi-objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[8] C. Coello, G. Lamont, and D. V. Veldhuisen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, ser. Genetic and Evolutionary Computation Series. Springer, 2007.

[9] D. F. Lochtefeld and F. W. Ciarallo, "Helper-objective optimization strategies for the job-shop scheduling problem," *Applied Soft Computing*, vol. 11, no. 6, pp. 4161 – 4174, 2011.

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[11] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[12] A. Gosavi, "Reinforcement learning: A tutorial survey and recent advances," *INFORMS Journal on Computing*, vol. 21, no. 2, pp. 178–192, 2009.

[13] S. Müller, N. N. Schraudolph, and P. D. Koumoutsakos, "Step size adaptation in evolution strategies using reinforcement learning," in *Proceedings of the Congress on Evolutionary Computation*. IEEE, 2002, pp. 151–156.

[14] A. E. Eiben, M. Horvath, W. Kowalczyk, and M. C. Schut, "Reinforcement learning for online control of evolutionary algorithms," in *Proceedings of the 4th international conference on Engineering self-organising systems ESOA'06*. Springer-Verlag, Berlin, Heidelberg, 2006, pp. 151–160.

[15] J. E. Pettinger and R. M. Everson, "Controlling genetic algorithms with reinforcement learning," in *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco: Morgan Kaufmann Publishers Inc., 2002, p. 692.

[16] Y. Sakurai, K. Takada, T. Kawabe, and S. Tsuruta, "A method to control parameters of evolutionary algorithms by using reinforcement learning," in *Proceedings of the 2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems*, ser. SITIS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 74–79.

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

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

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

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

[21] 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.

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

[23] 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.

[24] M. Buzdalov, "Generation of tests for programming challenge tasks on graph theory using evolution strategy," in *Proceedings of the 11th International Conference on Machine Learning and Applications, ICMLA 2012*, vol. 2, 2012, pp. 62–65.

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

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

[27] D. Pisinger, "Algorithms for Knapsack Problems," Ph.D. dissertation, University of Copenhagen, February 1995.

[28] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-II," *Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[29] R. G. L. D'Souza, K. C. Sekaran, and A. Kandasamy, "Improved NSGA-II based on a novel ranking scheme," *Computing Research Repository*, vol. abs/1002.4005, 2010.

[30] A. L. Strehl, L. Li, E. Wiewiora, J. Langford, and M. L. Littman, "PAC model-free reinforcement learning," in *Proceedings of the 23rd International Conference on Machine Learning (ICML 2006)*, 2006, pp. 881–888.

[31] D. S. Soper. Analysis of variance (ANOVA) calculator - one-way ANOVA from summary data (online software). [Online]. Available: http://www.danielsoper.com/statcalc