Generation of Tests for Programming Challenge Tasks Using Multi-Objective Optimization

Maxim Buzdalov
University ITMO
49 Kronverkskiy prosp.
Saint-Petersburg, Russia
mbuzdalov@gmail.com

Arina Buzdalova
University ITMO
49 Kronverkskiy prosp.
Saint-Petersburg, Russia
abuzdalova@gmail.com

Irina Petrova
University ITMO
49 Kronverkskiy prosp.
Saint-Petersburg, Russia
petrova@rain.ifmo.ru

ABSTRACT

In this paper, an evolutionary approach to generation of test cases for programming challenge tasks is investigated. Multi-objective and single-objective evolutionary algorithms, as well as helper-objective selection strategies, are compared. Particularly, a previously proposed method of choosing between helper-objectives with reinforcement learning is considered. This method is applied to the multi-objective evolutionary algorithm for the first time. Results of the experiment show that the most reasonable approach for the considered problem is using multi-objective evolutionary algorithm with automated helper-objective selection.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Data generators*; I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Experimentation, Performance

Keywords

Programming challenges, genetic algorithms, multi-objective, helper-objectives, reinforcement learning, testing

1. INTRODUCTION

In this paper, a method to generate test cases for programming challenge tasks using multi-objective evolutionary algorithms is described. A programming challenge [1, 2, 4, 23] is a competition where participants compete in writing computer programs which solve certain problems. A programming challenge task includes the formulation of the problem, requirements for the input and output data, time and memory limits.

In most types of programming challenges the correctness of solutions is checked by running them on a number of prewritten 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.

Test cases for programming challenge tasks are typically generated either by hand 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 depends very much on the human factor.

One of the ways to make the situation better is to automate the process of test case creation. In this work, test case generation is performed using single-objective evolutionary algorithms (EAs), as well as multi-objective ones (MOEAs) [10, 11]. The latter is exclusive for this paper. The previous works on this topic show that the single-objective evolutionary approach is suitable for generation of tricky test cases impossible for a human to come up with [6, 7].

We generate test cases for testing solution performance time. Performance time is a hard objective to be optimized, since it is noisy and takes only several different values, at least in the case of the considered problem. So we use the multi-objectivization approach [18, 21] and the helper-objectives approach [16, 19]. Both approaches are based on using some additional objectives to be optimized. Incorporating such objectives allows to use MOEAs and compare them with single-objective EAs.

2. PROBLEM DESCRIPTION

In this section a programming challenge task and its sample solution are considered. The aim of the research, which is to efficiently generate test cases for the considered solution, is also discussed in detail.

2.1 Task Statement

As in [6], we consider the programming challenge task "Ships. Version 2". This task is located at the Timus Online Judge under the number 1394 [3].

The task formulation is as follows. There are N ships, each of length s_i, and M havens, each of length h_j. Ships need to be allocated 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:

- $N \le 99$, $2 \le M \le 9$, $1 \le s_i \le 100$;
- $\sum s_i = \sum h_j$;
- the time limit is 1 second;

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

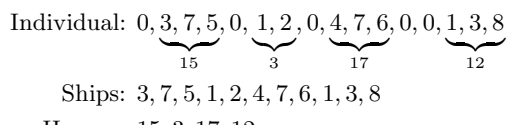

Figure 1: Individual: ships and havens

- the memory limit is 64 megabytes.

This problem is a special case of the multiple knapsack problem, which is known to be NP-hard in the strong sense [22] (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.

2.2 Sample Solution

For this paper, we chose a solution to the challenge task described above which needed the smallest number of generations of the genetic algorithm [6] to defeat. The solution uses a recursive greedy algorithm with elements of dynamic programming, which is applied to random permutations of the input data until it finds an answer. Following the approach in [6], we introduce three counters: `P`, which is incremented on each data permutation, `R`, which is incremented on every call of the recursive procedure, and `I`, which is incremented in the inner loops residing in the recursive procedure. Another counter, `T`, which is the target objective and equals the running time of the solution on the test case, in milliseconds, is added by the testing framework. The described counters are used as objectives in evolutionary algorithms considered below.

3. APPROACH

In order to generate test cases for the described task, we propose a number of evolutionary methods. We compare single-objective optimization and multi-objective one. Due to performance reasons and suggestions proposed in [16], only two-objective MOEAs are considered. The first objective is the target one and the second one is a helper-objective. The details described below are common for both EAs and MOEAs.

3.1 Individual Encoding

We use a special test encoding scheme similar to one proposed in [6]. 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 Figure 1).

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 2.1: 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 is at least as hard as solving the problem.

3.2 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.

3.3 Automated Selection of Helper-Objectives

The helper-objective to be optimized can be either chosen manually and stay the same during the run (*fixed* helper-objective), or automatically selected by some heuristics at different stages of the run (*dynamic* helper-objective). The strategies used to select a dynamic helper-objective are considered below.

It is proposed to randomly choose the helper-objective in [16] and optimize it for some fixed period, then choose another one and continue. Each helper should be tried at least once. The disadvantage of this approach is that the problem specific is not taken into account. It is mentioned in [16] that some adaptive selection method is needed.

We propose to use another selection strategy that is based on reinforcement learning (RL) [25, 17, 15]. This approach allows to learn and use some features of the particular problem. It is based on the EA + RL method, that we previously proposed in [9, 8, 5]. To our knowledge, it is the only method where reinforcement learning is used to choose objectives, or fitness functions [14, 20].

The EA + RL method, or MOEA + RL in case of multi-objective optimization, is described below after [9]. In order to set the reinforcement learning task [25], we should define the set of actions A, a definition of the environment states $s \in S$ and the reward function $R : S \times A \to \mathbb{R}$. Let x be an individual evolved by the evolutionary algorithm. Denote the i-th generation 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 generation G_i.

Let us define the reward function $R : S \times A \to \mathbb{R}$, which is calculated after choosing the criterion f_i in the state s_{i-1} and generating G_i. It depends on the difference between

fitness of individuals at sequential generations and is the highest when fitness increases:

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

where k is a discount parameter. Notice that the higher is the reward, the bigger is the increase of the target fitness.

In the single-objective EA only one objective is optimized at once. Reinforcement learning is used to choose and dynamically change this objective. Since reinforcement learning maximizes the total reward, which depends on the target fitness, EA is guided to optimize the target objective. In MOEA, reinforcement learning is used to choose the helper-objective, which is optimized together with the target one.

4. EXPERIMENT

During the experiment, test cases are generated using both EA and MOEA. Every objective is optimized as a fixed one in EA and MOEA. Selection strategies for choosing a dynamic helper-objective are used in both EA and MOEA. The settings of the EA, the MOEA and the reinforcement learning algorithm are described below.

All variations of the single-objective EA are run for 100 times. Most of the MOEA variations are run for 20 times, since it takes a very long time to perform the runs. The exception is the MOEA with reinforcement strategy, which is run for 100 times. All the results are averaged.

4.1 Settings

In both EA and MOEA the size of the generation is 200. An algorithm is terminated either when an individual, for which the running time of the tested solution exceeds five seconds, is evolved, or 10000 generations are processed.

In the single-objective EA, which is a genetic algorithm, to create a new generation a tournament selection with tournament size of 2 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 generation, the elitist strategy is used with the elite size of five individuals. If for 1000 generations the best fitness value does not change, then the current generation is cleared and initialized with newly created individuals.

For optimization of more than one objective, a fast variant of the NSGA-II algorithm [12] proposed in [13] is used. In the random selection strategy, a helper-objective is chosen randomly and being optimized during 50 generations. When each objective is tried, the process repeats.

Delayed Q-learning is chosen to be the reinforcement learning algorithm [24]. It is restarted every 50 generations, which prevents stagnation. The update period is $m = 0.5$, the bonus reward $\varepsilon = 0.001$ and discount factor $\gamma = 0.1$. The discount parameter used to calculate the reward is set to $k = 0.5$. All the parameter values are set on the basis of preliminary experiment results.

Table 1: Test generation results

Algorithm	FFs	OK, %	Generations	
			Mean	σ
Fixed objective				
GA	`I`	99	2999	1986
GA	`R`	93	3153	3742
GA	`P`	54	12621	12770
GA	`T`	0	–	–
NSGA-II	`T`, `I`	100	203	119
NSGA-II	`T`, `R`	100	440	381
NSGA-II	`T`, `P`	100	448	360
Dynamic objective				
GA + RL	all	65	9636	9538
GA + Random	all	57	13602	7929
NSGA-II + RL	all	99	895	1215
NSGA-II + Random	all	100	882	786

4.2 Mean and Diversity Recalculation

For performance reasons, the number of generations 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 generations until finish.

However, it is possible to estimate this value, if we assume that the algorithm is restarted when the number of generations reaches 10000 and the goal of optimization is not reached. Let E_S be the average of the number of generations for successful runs, R be the ratio of successful runs, G be the maximum number of generations until restart, D_S be the standard deviation of the number of generations for successful runs. For the sake of conciseness, we provide the formulae for the estimated total expectation E and deviation D without proof:

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

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

4.3 Results

Results of the experiment are presented in Table 1. "Random" stands for the random strategy used to select a dynamic helper, which was described in Section 3.3. "RL" stands for the reinforcement-based strategy.

One can conclude that, for this problem, multi-objective optimization performs better than the single-objective one. Nevertheless, it is worth noting that in the single-objective case reinforcement helper selection strategy outperforms the random one.

Which kind of helpers is more efficient – a fixed or a dynamic one? It can be seen that two-objective MOEA with a fixed objective yields the best performance. On the other hand, in the general case, using such approach means performing a number of runs for trying all the objectives, since we have no prior knowledge about the efficiency of different objectives.

Using dynamic helper approach means that one run is enough. In average, MOEA with dynamic helper performs worse than the best fixed-objective algorithm, but noticeably better than the original EA that optimizes performance time of the tested solution only. So we suggest using either

NSGA-II + Random or NSGA-II + RL for test case generating, since results obtained using random and reinforcement helper selection strategies are similar in the MOEA case.

5. CONCLUSION

Generation of tests for a programming challenge task was performed using different kinds of evolutionary algorithms. A number of helper-objectives were implemented. Multi-objective evolutionary algorithms turned to be more efficient for this problem than single-objective ones. Using a multi-objective evolutionary algorithm along with dynamic helper-objective is both a general and efficient approach.

A reinforcement helper-objective selection was considered. It outperformed the conventional random selection in the single-objective case, but was similar in the multi-objective case. Future work involves defining a reasonable state and reward definitions in order to increase the efficiency of reinforcement learning. It will be essential to show that with these definitions helper-objective selection problem can be modeled as some kind of Markov decision process.

6. REFERENCES

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

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

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

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

[5] 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*, volume 2, pages 354–357, Honolulu, HI, USA, 2011. IEEE Computer Society.

[6] 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*, pages 763–766, New York, US, ACM, 2011.

[7] 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*, volume 2, pages 62–65, 2012.

[8] 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*, volume 2, pages 66–67, 2012.

[9] 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*, volume 1, pages 150–155, 2012.

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

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

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

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

[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*, pages 151–160. Springer-Verlag, Berlin, Heidelberg, 2006.

[15] A. Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.

[16] M. T. Jensen. Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation. *Journal of Mathematical Modelling and Algorithms*, 3(4):323–347, 2004.

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

[18] 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*, EMO '01, pages 269–283, London, UK, 2001. Springer-Verlag.

[19] D. F. Lochtefeld and F. W. Ciarallo. Helper-objective optimization strategies for the job-shop scheduling problem. *Applied Soft Computing*, 11(6):4161 – 4174, 2011.

[20] 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*, pages 151–156. IEEE, 2002.

[21] F. Neumann and I. Wegener. Can single-objective optimization profit from multiobjective optimization? In J. Knowles, D. Corne, K. Deb, and D. R. Chair, editors, *Multiobjective Problem Solving from Nature*, Natural Computing Series, pages 115–130. Springer Berlin Heidelberg, 2008.

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

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

[24] 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)*, pages 881–888, 2006.

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