Generation of Tests for Programming Challenge Tasks Using Helper-Objectives

Arina Buzdalova, Maxim Buzdalov, and Vladimir Parfenov

St. Petersburg National Research University
of Information Technologies, Mechanics and Optics,
49 Kronverkskiy prosp., Saint-Petersburg, Russia, 197101
{abuzdalova,mbuzdalov}@gmail.com,
parfenov@mail.ifmo.ru

Abstract. Generation of performance tests for programming challenge tasks is considered. A number of evolutionary approaches are compared on two different solutions of an example problem. It is shown that using helper-objectives enhances evolutionary algorithms in the considered case. The general approach involves automated selection of such objectives.

Keywords: test generation, programming challenges, multi-objective evolutionary algorithms, multi-objectivization, helper-objectives.

1 Introduction

Programming challenge tasks are given at programming contests [1, 2]. Generally, a task consists of a problem formulation, input data format and output data format. Solutions are checked using pre-written tests. A test represents input data. In order to pass the test, a solution should provide correct output data within certain time and memory limits.

The goal of our research is to automatically generate performance tests. To clarify what exactly a performance test in this paper is, we say that the aim of performance test generation is to create such a test that running time of the tested solution on this tests exceeds the time limit. In order to generate tests, evolutionary algorithms [3] are used, as proposed in our previous work [4].

Although the running time is the objective to optimize, using it as a fitness function is not efficient. We propose using some helper-objectives [5] instead of or along with the running time objective. Such approach is inspired by multi-objectivization and helper-objective optimization techniques [5–7].

The exclusive part of this paper is consideration of helper-objectives for two different solutions and comparative analysis of 10 algorithms which were used to generate tests against these solutions.

2 Research Goal

The goal of the research is to explore different evolutionary approaches for performance test generation. The approaches to be explored are listed in Section 3. We generate performance tests against two different solutions of an example programming challenge task. The task, its solutions and corresponding helper-objectives are described below.

2.1 Programming Challenge Task

As in [4], we consider a programming challenge task “Ships. Version 2”. This task is located at the Timus Online Judge [2] under the number 1394.

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 $N \leq 99$, $2 \leq M \leq 9$, $1 \leq s_i \leq 100$, $\sum s_i = \sum h_j$. The time limit is one second, and the memory limit is 64 megabytes.

Due to the fact that this problem is NP-hard [8] and the 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 which makes them exceed the time limit.

2.2 Helper-Objectives

The target objective to be maximized is running time of a programming challenge task solution. However, it is inefficient to use running time as a fitness function, since it is platform-dependent, quantified and noisy [4]. In order to solve this issue, we suggest including counters in the solution code. The counters can be used as helper-objectives [5]. The pseudocodes of the *Solution-1* and *Solution-2* with the included helper-objectives are shown below.

Solution-1 with included helper-objectives: I, P, R

```
Read the input data
I := 0, P := 0, R := 0
while(solution not found)
    Randomly shuffle ships
    Call the recursive dynamic programming based ship arranging procedure
        For each call to this procedure, R := R + 1
        In each innermost loop, P := P + 1
    if (solution is found)
        Write the answer
    else
        I := I + 1
    end if
end while
```

Solution-2 with included helper-objectives: I, L, Q

```
Read the input data
I := 0, L := 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
        I := I + 1
        L := L + last
        last := 0
    end if
end while
Q := 1000000000 * I + last
```

The main difference between the solutions is the implementation of the ship arranging procedure. Unfortunately, we are not able to put more detailed code in this article, because of the programming challenge rules that prevent publication of solutions. However, it will be obvious from the experiment results that the solutions have different performance.

3 Approach

In this section different evolutionary approaches of test generation are described. Evolutionary algorithms can be used to optimize either a single objective or several ones. The objective can stay the same during the evolutionary algorithm run (a *fixed objective*), or we can select the objective to be optimized at each stage of the optimization process (a *dynamic objective*). The evolutionary algorithms, as well as helper-objective selection strategies are described below.

3.1 Evolutionary Algorithms

Single-Objective Genetic Algorithm (GA). The single-objective evolutionary algorithm is a genetic algorithm (GA) with the population size of 200. To create a new population 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 similar to [4] 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. If for 1000 generations the best fitness value does not change, then the current population is cleared and initialized with newly created individuals.

Multi-Objective Evolutionary Algorithm (NSGA-II). For optimization of more than one objective, a fast variant of the NSGA-II algorithm [9] proposed in [10] 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.

3.2 Helper-Objective Selection

Selection by M. T. Jensen. We consider two selection methods. The first one was proposed by M. T. Jensen [5]. According to this method, a helper-objective is chosen randomly from the set of helper-objectives and is being optimized for a fixed number of populations. Then the next helper-objective is chosen, and so on. This method implies using two-objective evolutionary algorithm, where the first objective is the running time and the second one is a helper-objective.

Reinforcement Learning Selection (RL). The other selection method is EA + RL method [7]. The fitness function is chosen with reinforcement learning from a set that includes the target objective and the helper ones. The choice is influenced by a reward that depends on the target objective (running time) growth. So the target objective is already taken into account and a single-objective evolutionary algorithm can be used.

In this work, delayed Q-learning algorithm [11] is used. It is restarted every 50 generations, which aims at preventing stagnation. The update period is $m = 5$, the bonus reward is $\varepsilon = 0.001$ and discount factor is $\gamma = 0.1$. 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.

4 Experiment

Tests for each considered solution were generated using all the considered algorithms with each compatible objective. Each algorithm was run for 100 times, then the results were averaged. The termination condition was either evolving of a test that made the solution to exceed the time limit (a *successful run*), or reaching the population number limit, which was 10000 populations.

The results for the Solution-1 are shown in the Table 1. T denotes the running time of the solution, σ is the diversity of the population number in a run. *Populations* refer to the mean number of populations needed to exceed the time limit, the smaller it is the more efficient the corresponding algorithm is. Note that using running time as a fitness function is inefficient, as was expected.

In the fixed objective case, multi-objective optimization significantly outperforms single-objective one, no matter what helper-objective is used. In the dynamic objective case, multi-objective optimization is also good enough. Although in this example NSGA-II with a fixed objective outperforms all the other approaches, using dynamic objective can be more preferable in general, as shown below.

Table 1. Results of test generation for the Solution-1

Algorithm	Fitness functions	Successful runs, %	Populations	
			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
NSGA-II + RL	all	99	895	1215
NSGA-II + Jensen	all	100	882	786

The results for the Solution-2 are shown in the Table 2. In the fixed objective case, only the objective `Q` is efficient. Although this objective provides the best performance, we usually do not know this in advance and should perform runs with each helper-objective.

At the same time, all the dynamic objective methods are efficient. In dynamic objective approach one run is enough, the most efficient objective is chosen automatically. So the dynamic helper objective approach is both general and efficient one.

Table 2. Results of test generation for the Solution-2

Algorithm	Fitness functions	Successful runs, %	Generations	
			Mean	σ
Fixed objective				
GA	`Q`	95	3815	3466
GA	`I`	54	12669	12873
GA	`L`	51	13755	14082
GA	`T`	0	–	–
NSGA-II	`T, Q`	95	2217	3136
NSGA-II	`T, I`	45	15861	16723
NSGA-II	`T, L`	20	41330	44768
Dynamic objective				
GA + RL	all	80	5817	6160
NSGA-II + RL	all	72	6679	7764
NSGA-II+Jensen	all	75	6103	7076

5 Conclusion

A number of approaches for generation of performance tests against programming challenge solutions were compared. It was shown that using helper-objectives significantly improves the optimization process. We suggest using multi-objective evolutionary algorithms with dynamic helper-objectives, which is a general and efficient method. Further work involves formalization of a class of problems that can be efficiently solved using the proposed approach. Another future goal is implementation of automated insertion of helper-objectives in the solution code, since currently such insertion is made manually.

References

1. ACM International Collegiate Programming Contest, `http://cm.baylor.edu/welcome.icpc`
2. Timus Online Judge. The Problem Archive with Online Judge System, `http://acm.timus.ru`
3. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer (2003)
4. Buzdalov, M.: Generation of Tests for Programming Challenge Tasks Using Evolution Algorithms. In: GECCO Conference Companion on Genetic and Evolutionary Computation, pp. 763–766. ACM, New York (2011)
5. Jensen, M.T.: Helper-Objectives: Using Multi-Objective Evolutionary Algorithms for Single-Objective Optimisation. Journal of Mathematical Modelling and Algorithms 3(4), 323–347 (2004)
6. Knowles, J.D., Watson, R.A., Corne, D.W.: Reducing Local Optima in Single-Objective Problems by Multi-objectivization. In: Zitzler, E., Deb, K., Thiele, L., Coello Coello, C.A., Corne, D.W. (eds.) EMO 2001. LNCS, vol. 1993, pp. 269–283. Springer, Heidelberg (2001)
7. Buzdalova, A., Buzdalov, M.: Increasing Efficiency of Evolutionary Algorithms by Choosing between Auxiliary Fitness Functions with Reinforcement Learning. In: 11th International Conference on Machine Learning and Applications, pp. 150–155. IEEE (2012)
8. Pisinger, D.: Algorithms for Knapsack Problems. PhD Thesis, University of Copenhagen (1995)
9. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. Transactions on Evolutionary Computation 6(2), 182–197 (2002)
10. D'Souza, Rio G. L., Chandra Sekaran, K., Kandasamy, A.: Improved NSGA-II Based on a Novel Ranking Scheme. Computing Research Repository. ID: abs/1002.4005 (2010)
11. Strehl, A.L., Li, L., Wiewora, E., Langford, J., Littman, M.L.: PAC model-free reinforcement learning. In: 23rd International Conference on Machine Learning, pp. 881–888 (2006)