2011 10th International Conference on Machine Learning and Applications

Choosing Best Fitness Function with Reinforcement Learning

Arina Afanasyeva
National Research University
of Information Technologies, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
Email: afanasyevarina@gmail.com

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

***Abstract*—This paper describes an optimization problem with one target function to be optimized and several supporting functions that can be used to speed up the optimization process. A method based on reinforcement learning is proposed for choosing a good supporting function during optimization using genetic algorithm. Results of applying this method to a model problem are shown.**

I. Introduction

There exist several kinds of optimization problems, such as the scalar optimization problem and the multicriteria optimization problem [1]. In this research, a different optimization problem is considered. The problem is to maximize a certain, *target* function. For each calculation of the target function, several *supporting* functions are calculated as well. Some of the supporting functions may correlate with the target one, and, in fact, it may be more efficient to optimize some of the supporting functions in certain phases of optimization, even if during this process the target function sometimes decreases. Such problems often arise in automated performance test generation [2].

Genetic algorithms (GA) are often used for solving difficult optimization problems [3]. One of the drawbacks of using GAs is that the process of optimization may take a long time, so, if an optimization problem with supporting functions is considered, testing each of the supporting functions is very inefficient. This paper proposes the method which allows choosing the most efficient fitness function automatically during the run of the genetic algorithm. Moreover, when different functions are efficient at different stages of optimization process, the method chooses the currently optimal function dynamically.

Let us introduce some concepts. Consider a set of fitness functions (FFs). The goal of the genetic algorithm is to breed an individual with the best value of one of these functions, which we call the *target FF*. The rest of the functions will be called *supporting* FFs. The FF that is used in the genetic algorithm at the moment will be called the *current FF*.

A model problem, where use of different supporting functions at different stages of optimization is more efficient, is used to demonstrate that reinforcement learning [4] allows to choose the optimal current FF during the run of the GA.

II. Model problem

Let an individual be a bit string of a fixed length n with x bits set to 1. The target fitness function is

$$g(x) = \lfloor \frac{x}{k} \rfloor. \quad (1)$$

Supporting fitness functions are $h_1(x) = \min(x, p)$ and $h_2(x) = \max(x, p)$, where p is a positive integer. We will call p the *switch point*.

It is most efficient to use the function h_1 as the current fitness of the individuals with the number of bits less than p. For the other individuals, the function h_2 should be used.

The task for the proposed method is to dynamically switch GA to the most appropriate current FF basing on the kind of the individuals in the current generation. In other words, the method should set the current FF to h_1 first and switch it to h_2 when the individuals in the current generation reach the switch point.

III. Basics of reinforcement learning

Many of the reinforcement learning (RL) algorithms do not require preparation of a training set [5]. Such algorithms are convenient to use with GA, as there is no need to make several GA runs in order to collect some training information. Furthermore, the incremental nature of some reinforcement learning algorithms seems to be promising for dynamic switch of the current fitness function during the GA run time. So we chose reinforcement learning to optimize GA.

Recall some basic concepts of RL. The *agent* applies some *actions* to the *environment*. After each action the agent receives some representation of the environment's *state* and some numeric *reward*. The agent should maximize the total amount of reward it receives.

It is essential for the algorithms we used that the RL task can be represented as a *Markov decision process (MDP)*. The MDP consists of

- set of states S;
- set of actions A;
- reward function $R : S \times A \to \mathbb{R}$;
- transition function $T : S \times A \times S \to \mathbb{R}$, such as the probability of transition from the s state to the s' caused by taking the action a is $T(s, a, s')$.

It is shown in the next section how the model problem was represented in terms of MDP.

IV. Reinforcement learning task

Consider the representation of the model problem as a reinforcement learning task. Set of actions of the agent is represented by fitness functions: $A = g, h_1, h_2$. Applying some action means choosing the corresponding fitness function as the current one. It is followed by creation of a new generation of individuals in the GA.

The state of the environment depends on the state of the GA. It is a vector of the fitness functions ordered by the value of $(f(x_c) - f(x_p))/f(x_c)$, where f is a fitness function to be measured, x_p is the number of bits set to one in the best individual from the previous generation, and x_c is the number of bits set to one in the best individual from the current generation.

The value of the reward function depends on changes of the fitness of the best individual, which appear after the creation of the next generation caused by the action of the agent. Before giving the formula for the reward function, we will define an auxiliary function D_f as the following:

$$D_f(x_1, x_2) = \begin{cases} 0 & \text{if } f(x_2) - f(x_1) < 0 \\ 0.5 & \text{if } f(x_2) - f(x_1) = 0 \\ 1 & \text{if } f(x_2) - f(x_1) > 0 \end{cases}$$

The reward function is

$$R(s, a) = D_g(x_s, x_{s'}) + c(D_{h_1}(x_s, x_{s'}) + D_{h_2}(x_s, x_{s'})), \quad (2)$$

where $c \in [0, 1]$ is a real-valued parameter that allows to variate the supporting contribution of fitness functions to the reward, s, s' are the previous and the new state respectively, a is the action that caused transition from s to s', x_s and $x_{s'}$ are the number of bits set to one in the best individuals that correspond to the states s and s' respectively.

V. Method description

We used ε-greedy Q-learning [5] and Delayed Q-learning [6] algorithms. They can be classified as incremental model-free reinforcement learning algorithms [7]. Model-free algorithms were chosen because they have relatively low space and computational complexities. Such characteristics allow not to slow down the GA which solves the model problem.

An incremental algorithm that controls the GA was implemented. Each step of this algorithm consists of the learning step and the GA step. At the learning step, the agent chooses the current fitness function. At the GA step, this function is used to create the next generation of the individuals.

VI. Experiment

The model problem was solved using GA that was controlled by ε-greedy Q-learning and Delayed Q-learning. A normal GA without any learning was considered as well. The following subsections describe this experiment and its results.

A. Description of the experiment

During the experiment all the implemented algorithms were run with different combinations of the model problem parameters values and the learning algorithm parameters values.

The following parameters of the model problem were used:

- l — the length of an individual;
- p — the switch point;
- k — the divisor in the target function g (see 1).

The parameters of learning are different for different algorithms. The algorithm based on Delayed Q-learning had the following parameters [6]:

- m — the update period of Q-value estimates;
- ϵ — the bonus reward;
- γ — the discount factor.

The following parameters of the ε-greedy Q-learning algorithm were adjusted:

- ε — the exploration probability;
- α — the learning speed;
- γ — the discount factor.

The preliminary experiment showed that the parameter c from the reward function definition (see 2) should be of approximately 0.5. It allowed to give preference to the target function and to take into account the values of the supporting functions at the same time. Values of supporting functions change more frequently that allows to gather more experience and increases learning performance.

During the main experiment more than thousand combinations of parameter values were processed. Each configuration was run 50 times in order to average the results. We tried to search for the best parameters with gradient descent, but it turned to be inefficient due to stochastic nature of genetic algorithms. The parameters considered below were manually chosen from the set of calculated configurations.

The crossover probability in all algorithms was equal to 0.7. Two mutation probabilities were used: 0.01 (the *high* one) and 0.003 (the *low* one). This lead to different kinds of the environment. Use of the high mutation probability resulted in slowing down the GA. More specifically, the GA evolved individuals with new fitnesses relatively rare, which resulted in the reward function value changing too slow. It is hard to gather experience using such GA, but Delayed Q-learning showed good results. Greedy Q-learning turned to be more applicable for use with low mutation probability. These facts are illustrated in the next subsections using the problem parameters $l = 400, p = 266, k = 10$.

B. Performance results for high mutation probability

Consider the results for the big mutation probability 0.01. The plots of the GA runs controlled by Delayed Q-learning and normal uncontrolled GA runs can be seen on Fig. 1. The horizontal axis corresponds to the number of a generation. The vertical axis reflects average values of the target FF of the best individual from the corresponding generation.

The learning parameters here are $m = 100, \epsilon_1 = 0.2, \gamma = 0.01$. This set of values is one of the sets that provided high

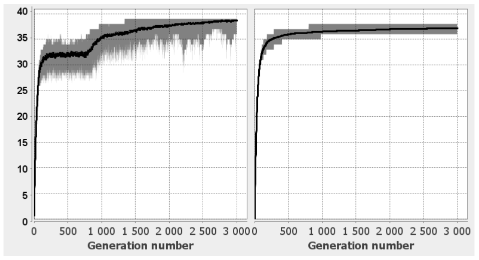

Fig. 1. GA run with Delayed Q-learning (left) and no learning (right)

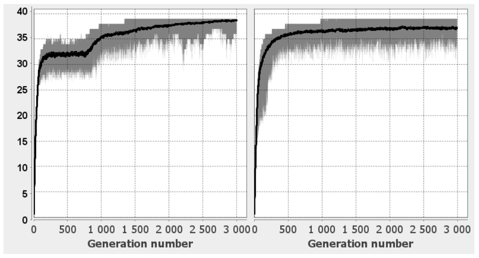

Fig. 2. GA run with Delayed (left) and ε-greedy Q-learning (right)

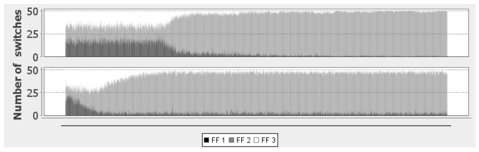

Fig. 3. Number of current FF choices made by Delayed (top) and ε-greedy Q-learning (bottom)

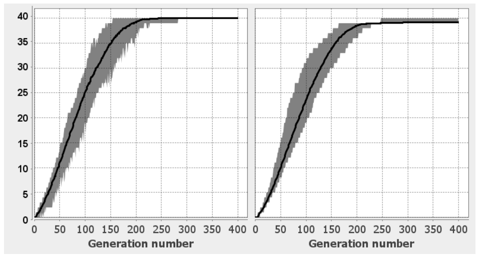

Fig. 4. GA run with ε-greedy Q-learning (left) and no learning (right)

performance during the experiment. The GA with learning gets better individuals faster, as the learning algorithm chooses the proper current FF while the normal GA uses the target FF as the current one. The horizontal part of the Delayed Q-learning plot corresponds to the period of gathering experience. At this period each fitness function is chosen with equal probability.

Fig. 2 compares GA runs controlled by Delayed Q-learning and ε-greedy Q-learning. The parameters of the Delayed Q-learning algorithm are the same as on the Fig. 1. The parameters of the ε-greedy Q-learning algorithm are $\varepsilon = 0.1, \alpha = 0.1, \gamma = 0.01$. These values provided the best performance of the ε-greedy Q-learning based algorithm. The exploration strategy used in this algorithm allows to reflect changes of the state faster. The ε-greedy Q-learning algorithm does not need to gather experience for some fixed period of time unlike the Delayed Q-learning algorithm does. At the same time, the average fitness provided by the ε-greedy controlled GA is less than the average fitness achieved using Delayed Q-learning. In other words, the Delayed Q-learning based algorithm shows better results during the long run.

C. Choice of FF for high mutation probability

Fig. 3 shows number of different current fitness functions chosen by the learning algorithms. Let us call the *first interval* the numerical interval that ends with the switch point. The numerical interval that starts with the switch point is the *second interval*, respectively. FF1 denotes to the supporting function h_1 that is efficient on the first interval, FF2 denotes to h_2 that is efficient on the second interval, FF3 corresponds to the target function g. The ε-greedy algorithm is more flexible on the first interval. It manages to choose h_1 as the current FF. Delayed Q-learning algorithm needs time to gather experience, so it has no time to choose the most advantageous current FF on the first interval. Although it switches to h_2 at the second interval rather quickly. So both Delayed and ε-greedy Q-learning based algorithms eventually choose h_2 at the second interval.

D. The case of low mutation probability

Use of the low mutation probability 0.003 leads to different results. As it was mentioned, ε-greedy Q-learning showed better results here. What is more, a universal combination of its parameters was found. It is $\varepsilon = 0.3, \alpha = 0.6, \gamma = 0.01$. This combination leads to good performance with most of the considered combinations of the model problem parameters.

In the Fig. 4 the best instance of the ε-greedy Q-learning based algorithm is compared with the genetic algorithm that uses g (see 1) as fitness function. The Fig. 4 shows that it performs better than the GA used in the model problem.

Best instances of Delayed Q-learning and ε-greedy Q-learning based algorithms are compared in the Fig. 5. The Delayed Q-learning based algorithm is still better than the normal GA, but its performance becomes worse than the performance of the second algorithm. Notice that, in the case of low mutation probability, there is no constant part in the Delayed algorithm plot. It is because GA manages to evolve better individuals even with an inefficient current function chosen.

Low mutation probability results in speeding up the GA. So results provided by the target FF during some fixed period of time slightly differ from the results provided by the proper supporting FF. Consequently, it is hard for the Delayed algorithm to choose between the supporting function and the target one. At the same time greedy algorithm chooses the proper supporting functions at the both intervals without delay. The Fig. 6 reflects the choices performed by the two considered algorithms.

VII. Conclusion

An approach that allows choosing the current fitness function dynamically in a genetic algorithm in order to speed up

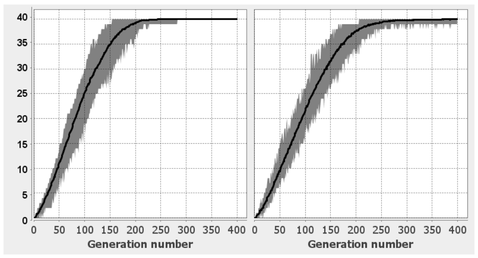

Fig. 5. GA run with ε-greedy (left) and Delayed Q-learning (right)

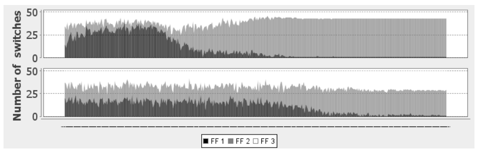

Fig. 6. Number of current FF choices made by ε-greedy (top) and Delayed Q-learning (bottom)

the search of the best individual is described. This approach is based on the reinforcement learning. It showed good results in solving the model problem. Two different Q-learning algorithms were used. The ε-greedy Q-learning provided switching to the proper fitness function on the spot. Delayed Q-learning provided higher average fitness during the long run in the case of high mutation probability. In the case of low mutation probability, ε-greedy Q-learning turned to be more efficient. The proposed method can be extended in order to be used with other optimization algorithms, not only genetic ones. Application of reinforcement learning seems to be promising in the area of choosing supporting optimization criteria.

The possible directions of future work are the following:

- Investigation of the model-based reinforcement learning applicability. Model-based learning can allow taking into account features of genetic algorithms, but it is also likely to require more resources.
- Implementation of the special algorithms for non-stationary environment [8]. It is promising since a genetic algorithm is likely to be better described as a non-stationary environment.
- Application of the proposed approach to the solution of some practical problems, particularly, to the generation of tests described in [2].
- Application of the method to other optimization algorithms.

REFERENCES

[1] M. Ehrgott, *Multicriteria optimization*. Springer, 2005.

[2] 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.

[3] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1996.

[4] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

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

[6] A. L. Strehl, L. Li, E. Wiewiora, J. Langford, and M. L. Littman, “Pac model-free reinforcement learning,” in *ICML-06: Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 881–888.

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, vol. 4, p. 237285, 1996.

[8] B. C. D. Silva, E. W. Basso, A. L. C. Bazzan, and P. M. Engel, “Dealing with non-stationary environments using context detection,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML 2006)*. ACM Press, 2006, pp. 217–224.