Reinforcement Learning for Online Control of Evolutionary Algorithms

A.E. Eiben, M. Horvath, W. Kowalczyk, and M.C. Schut

Department of Computer Science, Vrije Universiteit Amsterdam
{gusz, mhorvath, wojtek, schut}@cs.vu.nl

Abstract. The research reported in this paper is concerned with assessing the usefulness of reinforcment learning (RL) for on-line calibration of parameters in evolutionary algorithms (EA). We are running an RL procedure and the EA simultaneously and the RL is changing the EA parameters on-the-fly. We evaluate this approach experimentally on a range of fitness landscapes with varying degrees of ruggedness. The results show that EA calibrated by the RL-based approach outperforms a benchmark EA.

1 Introduction

During the history of evolutionary computing (EC), the automation of finding good parameter values for EAs have often been considered, but never really achieved. Related approaches include meta-GAs [1, 6, 15], using statistical methods [5], "parameter sweeps" [11], or most recently, estimation of relevance of parameters and values [10]. To our knowledge there is only one study on using reinforcement learning (RL) to calibrate EAs, namely the mutation step size [9]. In this paper we aim at regulating "all" parameters. To position our work we briefly reiterate the classification scheme of parameter calibration approaches in EC after [2, 4].

The most conventional approach is *parameter tuning*, where much experimental work is devoted to finding good values for the parameters *before* the "real" runs and then running the algorithm using these values, which remain fixed during the run. This approach is widely practicised, but it suffers from two very important deficiencies. First, the parameter-performance landscape of any given EA on any given problem instance is highly non-linear with complex interactions among the dimensions (parameters). Therefore, finding high altitude points, i.e., well performing combinations of parameters, is hard. Systematic, exhaustive search is infeasible and there are no proven optimization algorithms for such problems. Second, things are even more complex, because the parameter-performance landscape is not static. It changes over time, since the best value of a parameter depends on the given stage of the search process. In other words, finding (near-)optimal parameter settings is a dynamic optimisation problem. This implies that the practice of using constant parameters that do not change during a run is inevitably suboptimal.

Such considerations have directed the attention to mechanisms that would modify the parameter values of an EA on-the-fly. Efforts in this direction are mainly driven by two purposes: the promise of a parameter-free EA and performance improvement.

The related methods – commonly captured by the umbrella term *parameter control* can further be divided into one of the following three categories [2, 4]. *Deterministic parameter control* takes place when the value of a strategy parameter is altered by some deterministic rule modifying the strategy parameter in a fixed, predetermined (i.e., user-specified) way without using any feedback from the search. Usually, a time-dependent schedule is used. *Adaptive parameter control* works by some form of feedback from the search that serves as input to a heuristic mechanism used to determine the change to the strategy parameter. In the case of *self-adaptive parameter control* the parameters are encoded into the chromosomes and undergo variation with the rest of the chromosome. The better values of these encoded parameters lead to better individuals, which in turn are more likely to survive and produce offspring and hence propagate these better parameter values. In the next section we use this taxonomy/terminology to specify the problem(s) to be solved by the RL-based approach.

2 Problem definition

We consider an evolutionary algorithm to be a mechanism capable of optimising a collection of individuals, i.e., a way to self-organise some collective of entities. Engineering such an algorithm (specifically: determining the correct/best parameter values) may imply two different approaches: one either *designs* it such that the parameters are (somehow) determined beforehand (like in [10]), or one includes a component that *controls* the values of the parameters during deployment. This paper considers such a control component.

Thus, we assume some problem to be solved by an EA. As presented in [10], we can distinguish 3 layers in using an EA:

- **Application layer**: The problem(s) to solve.
- **Algorithm layer**: The EA with its parameters operating on objects from the application layer (candidate solutions of the problem to solve).
- **Control layer**: A method operating on objects from the algorithm layer (parameters of the EA to calibrate).

The problem itself is irrelevant here, the only important aspect is that we have individuals (candidate solutions) and some fitness (utility) function for these individuals derived from the problem definition. Without significant loss of generality we can assume that the individuals are bitstrings and the EA we have in mind is a genetic algorithm (GA). For GAs the parameter calibration problem in general means finding values for variation operators (crossover and mutation), selection operators (parent selection and survivor selection), and population size. In the present investigation we consider four parameters: crossover rate p_c, mutation rate p_m, tournament size k[1], and population size N. This gives us a paramater quadruple $\langle N, k, p_m, p_c \rangle$ to be regulated. Other components and parameters are the same as for the simple GA that we use as benchmark, cf. Section 4. The rationale behind applying RL for parameter calibration is that we add

[1] Because the population size can vary we use *tournament proportion or tournament rate* (related to the whole population), rather than tournament size.

an RL component to ("above") the GA and use it to specify values for $\langle N, k, p_m, p_c \rangle$ to the underlying GA. Monitoring the behavior of the GA with the given parameters enables the RL component to calculate new, hopefully better, values – a loop that can be iterated several times during a GA run. Within this context, the usefulness of the RL approach will be assessed by comparing the performance of the benchmark GA with a GA regulated by RL.

To this end, we investigate RL that can perform on-the-fly adjustment of parameter values. This has the same functionality as self-adaptation, but the mechanics are different, i.e., not by co-evolving parameters on the chromosomes with the solutions. Here, RL enables the system to learn from the actual run and to calibrate the running EA on-the-fly by using the learned information in the same run.

The research questions implied by this problem description can now be summarized as follows.

1. Is the performance of the RL-enhanced GA better than that of the benchmark GA?
2. How big is the learning overhead implied by using RL?

As for related work, we want to mention that including a control component for engineering self organising applications is not new - the field of autonomic computing recognises the usefulness of reinforcement learning for control tasks [12]. Exemplar applications are autonomous cell phone channel allocation, network packet routing [12], and autonomic network repair [8]. As usual in reinforcement learning problems, these applications typically boil down to finding some optimal *control policy* that best maps actions to system states. For example, in the autonomic network repair application, a policy needs to be found that optimally decides on carrying out costly test and repair actions in order to let the network function properly. The aim of our work is slightly different than finding such a control policy: we assume some problem on the application level that needs to be solved by an EA on the algorithm layer. As explained before, we consider the self organisation to take place on the algorithm level rather than the application level (as is the case for autonomic computing applications).

3 Reinforcement Learning

Our objective is to optimize the performance of an EA-process by dynamically adjusting the control parameters as mentioned above with help of reinforcement learning. The EA-process is split into a sequence of *episodes* and after each episode an adjustment of control parameters takes place. The state of the EA-process (measured at the end of every episode) is represented by a vector of numbers that reflect the main properties of the current population: mean fitness, standard deviation of fitness, etc. In a given state an action is taken: new control parameters are found and applied to EA to generate a new episode. The quality of the chosen action, the *reward*, is measured by a function that reflects the progress of the EA-process between the two episodes. Clearly, our main objective is to apply reinforcement learning to learn the function that maps states into actions in such a way that the overall (discounted) reward is maximized. In this paper we decided to represent states and actions by vectors of parameters that are listed in Table 1. The reward function could be chosen in several ways. For example, one could

consider improvement of the best (or mean) fitness value, or the success rate of the breeding process. In [9] four different rewarding schemes were investigated and following their findings we decided to define reward as the improvement of the best fitness value.

Index	State Parameter	Type	Range
s_1	Best fitness	$\mathbb{R}$	0-1
s_2	Mean fitness	$\mathbb{R}$	0-1
s_3	Standard deviation of the fitness	$\mathbb{R}$	0-1
s_4	Breeding success number	$\mathbb{N}$	0-control window
s_5	Average distance from the best	$\mathbb{R}$	0-100
s_6	Number of evaluations	$\mathbb{N}$	0-99999
s_7	Fitness growth	$\mathbb{R}$	0-1
$s_8 - s_{11}$	Previous action vector		
Index	Control Parameter	Type	Range
c_1	Population size	$\mathbb{N}$	3-1000
c_2	Tournament proportion	$\mathbb{R}$	0-1
c_3	Mutation probability	$\mathbb{R}$	0-0.06
c_4	Crossover probability	$\mathbb{R}$	0-1

Table 1. Components of State and Action vectors

3.1 The Learning Algorithm

Our learning algorithm is based on a combination of two classical algorithms used in RL: the Q-learning and the SARSA algorithm, both belonging to the broader family of Temporal Difference (TD) learning algorithms, see [14] and [7]. The algorithms maintain a table of state-action pairs together with their estimated discounted rewards, denoted by $Q(s, a)$. The estimates are systematically updated with help of the so-called *temporal difference*:

$$r_{t+1} + \gamma Q(s_{t+1}, a^*_{t+1}) - Q(s_t, a_t)$$

where r, s, a denote reward, state and action, indexed by time, and γ is the reward discount factor. The action a^*_{t+1} can be either the best action in the state s_{t+1} (according to the current estimates of Q) or an action (not necessarily optimal) which is actually executed (in the exploration mode of the learning algorithm). When the best action is chosen we talk about *on-policy TD control* (SARSA learning), otherwise we talk about *off-policy TD control* (Q-learning), [14].

As noticed in [14], both learning strategies have different characteristics concerning convergence speed and ability of finding optima. Therefore, our version of reinforcement learning will be switching between on- and off-policy control at random, with a pre-specified frequency δ.

The approach outlined above works with discrete tables of state-action pairs. In our case, however, both states and actions are continuous. Therefore, during the learning process we will maintain a table of observed states, taken actions and obtained rewards and use this table as a trainig set for modeling the function Q with help of some regression model: a neural network, weighted nearest-neighbour algorithm, regression tree, etc. This, in turn, leads to a yet another problem: given an implicit representation of Q and a current state s, how can we find an optimal action a^* that maximizes $Q(s, a)$? For the purpose of this paper we used a genetic algorithm to solve this sub-problem. However, one could think about using other (perhaps more efficient) optimization methods.

There are two more details that we have implemented in our RL-algorithm: periodical retraining of the Q-function and a restricted size of the training set. Retraining the regression model of Q is an expensive process, therefore it is performed only when a substantial number of new training cases are generated; we will call this number a *batch size*. Using all training cases that were generated during the learning process might be inefficient. For example, "old" cases are usually of low quality and they may negatively influence the learning process. Moreover, a big training set slows down the training process. Therefore we decided to introduce an upper limit on the number of cases that are used in retraining, *memory limit*, and to remove the oldest cases when necessary. The pseudo-code of our training algorithm is presented below:

1 Initialize Q abitrarily
2 Initialize ε
3 **Repeat** (for each episode)
4 Ask the controlled system for initial state s
5 Choose an action a' according to the optimization over the function $Q(s, a')$
6 $a =$ randomize a' with ε probability.
7 **Repeat** (for each step of the episode)
8 Do action a, and observe r, s'
9 Choose an action a' that oprimizes the function $Q(s', a')$
10 $a'' =$ randomize a' with ε probability.
11 Add new training instance to Q: $\langle s, a, r + \gamma(\delta Q(s', a') + (1 - \delta)Q(s', a''))\rangle$
12 Re-train Q if the number of new cases reached the *batch size*
13 $s = s'$
14 $a = a''$
15 (until s is not a terminal state)
16 Decrease ε

The randomization process that is mentioned in lines 6 and 10 uses several parameters. Reinforcement learning has to spend some effort on exploring the unknown regions of the policy space by switching, from time to time, to the *exploration mode*. The probability of entering this mode is determined by value of the parameter ε. During the learning process this value is decreasing exponentially fast, until a lower bound is reached. We will refer to the initial value of ε, the discount factor and the lower bound as ε-initial value, ε-discount factor and ε-minimal, respectively.

In exploration mode an action is usually selected at random using a uniform probability distribution over the space of possible actions. However, this common strategy could be very harmful for the performance of the EA. For instance, by decreasing the

population size to 1 the control algorithm could practically kill the EA-process. To prevent such situations we introduced a new mechanism for exploration that explores areas that are close to the optimal action. As the optimal action is found with help of a separate optimization process, we control our exploration strategy with a parameter that measures the *optimization effort*. Clearly, the smaller the effort, the more randomness in the exploration process. As mentioned earlier, in this research we used a separate genetic algorithm to find optimal actions. Therefore, we can express the optimization effort in terms of the rate of decrease of the number of evaluations in the underlying genetic process.

3.2 System Architecture

The overall architecture of our learning system is shown in Figure 3.2. It consists of three components: General Manager, State-Action Evaluator and Action Optimizer.

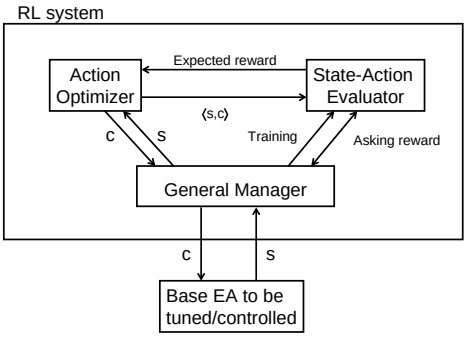

Fig. 1. The architecture of a RL-controller for EA.

General Manager is responsible for managing the whole process of RL. It maintains a training set of state vectors, together with taken actions and rewards, activates the training procedure for modeling the Q function and calls Action Optimizer to chose an action in a given state.

Action Optimizer contains an optimisation procedure (in our case: a genetic algorithm referred to as AO-EA) which is responsible for seeking an optimal action (a vector of control parameters). In other words, for a given state s the module seeks an optimum of the function $Q(s, \_)$ that is maintained by the State-Action Evaluator module.

State-Action Evaluator maintains a function that estimates the expected discounted reward values for arbitrary state-action pairs. The function is implemented as a regression model (a neural network, weighted nearest-neigbour, regression tree, etc.) and can be retrained with help of a suitable learning alrgoritm and a training set that is maintained by the General Manager Module.

Parameter	Value
Reward discount factor (γ)	0.849643
Rate of on- or off-policy learning (δ)	0.414492
Memory limit	8778
Exploration probability (ε)	0.275283
ε-discount factor	0.85155
ε-minimal	0.956004
Probablility of uniform random exploration	0.384026
Optimization effort	0.353446

Table 2. Parameter settings of the RL system

4 Experiments

The test suite[2] for testing GAs is obtained through the Multimodal Problem Generator of Spears [13]. We generate 10 landscapes of 1, 2, 5, 10, 25, 50, 100, 250, 500 and 1000 binary peaks whose heights are linearly distributed and where the lowest peak is 0.5. The length L of these bit strings is 100. The fitness of an individual is measured by the Hamming distance between the individual and the nearest peak, scaled by the height of that peak.

We define an adaptive GA (AGA) with on-the-fly control by RL. The AGA works with control heuristics generated by RL on the fly. RL is thus used here at runtime to generate control heuristics for the GA.

The setup of the SGA is as follows (based on [3]). The model we use is a steady-state GA. Every individual is a 100-bitstring. The recombination operator is 2-point crossover; the recombination probability is 0.9. The mutation operator is bit-flip; the mutation probability is 0.01. The parent selection is 2-tournament and survival selection is delete-worst-two. The population size is 100. Initialisation is random. The termination criterion is $f(x) = 1$ or 10,000 evaluations.

The parameters of the RL system have to be tuned, which has been done through extensive tuning and testing resulting in the parameter settings shown in Table 2. We used the REPTree algorithm [16] as the regression model for the State-Action Evaluator.

As mentioned in the introduction, the Success Rate (SR), the Average number of Evaluations to a Solution (AES) and its standard deviation (SDAES), and the Mean Best Fitness (MBF) and its standard deviation (SDMBF) are calculated after 100 runs of each GA.

The results of the experiments are summarised in Figures 2, 3 and 4. The experiments 1-10 on the x-axis correspond the different landscapes with 1, 2, 5, 10, 25, 50, 100, 250, 500 and 1000 binary peaks, respectively.

The results shown in Figures 2, 3 and 4 contain sufficient data to answer our research questions from Section 2 – at least for the test suite used in this investigation. The first research question concerns the performance of the benchmark SGA vs. the RL-enhanced variant. Considering the MBF measure it holds that the AGA consistently outperforms the SGA. More precisely, on the easy problems SGA is equally good, but

[2] The test suite can be obtained from the webpage of the authors of this paper.

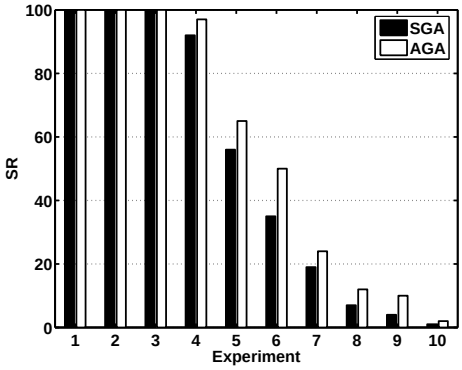

Fig. 2. SR results for SGA and AGA.

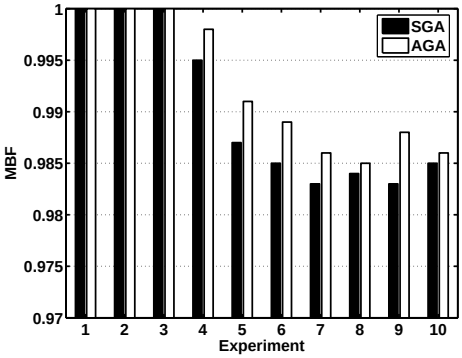

Fig. 3. MBF results for SGA and AGA.

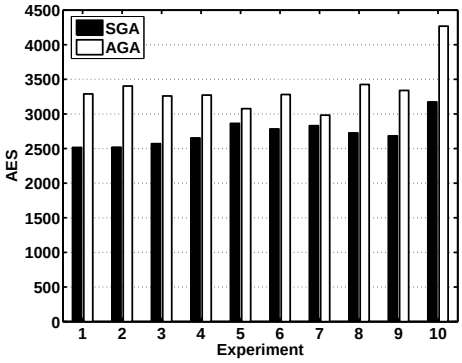

Fig. 4. AES results for SGA and AGA.

as the number of peaks (problem hardness) is growing, the adaptive GA becomes better. The success rate results are in-line with this picture: the more peaks the greater the advantage of the adaptive GA. Considering the third performance measure, speed defined by AES, we obtain another ranking. The SGA is faster than the AGA. This is not surprising, because of the RL learning overhead.

We are also interested in the overhead caused by reinforcement learning. From the systems perspective this is measurable by the lenghts of the GA runs. The AES results indicate the price of using RL in the on-line mode: approximately 20-30% increase of effort.[3] From the users perspective there is an overhead as well. The RL extension needs to be implemented (one-time investment) and the RL system needs to be calibrated. This latter one can take substantial time and/or innovativeness. For the present study we used a semi-automated approach through a meta-RL to optimize the parameters of our RL controlling the GA. We omit the details here, simply remarking that the RL parameter settings shown in Table 2 have been obtained by this approach.

5 Conclusions and Further Research

This paper described a study into the usefulness of reinforcement learning for online control of evolutionary algorithms. The study shows: firstly, concerning fitness and succes rate, the RL-enhanced GA outperforms the benchmark GA; concerning speed (number of evaluations), the RL-enhanced GA is outperformed by the benchmark GA. Secondly, also for the overhead of RL the user needs to tune the RL parameters causing overhead.

For future work, we consider a number of options. Firstly, our results indicate that on-the-fly control can be effective in design problems (given time interval, in search of optimal solution). To find best solutions to a problem, we hypothesize it is better to concentrate on solving the problem rather than finding the optimal control of the problem. This hypothesis requires further research. Secondly, the RL systems may be given more degrees of freedom: choice of probability of applying different operators, type of selection mechanism, include special operators to jump out of local optima. Finally, whereas RL in the presented work controls global parts of the EA, we consider the inclusion of local decisions like selection of individuals or choosing the right operator for each individual.

References

1. J. Clune, S. Goings, B. Punch, and E. Goodman. Investigations in meta-gas: panaceas or pipe dreams? In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 235–241, New York, NY, USA, 2005. ACM Press.
2. A. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[3] We assume that fitness evaluations constitute the huge majority of computational efforts running a GA.

3. A. Eiben, E. Marchiori, and V. Valko. Evolutionary algorithms with on-the-fly population size adjustment. In X. et al, editor, *Parallel Problem Solving from Nature, PPSN VIII*, volume 3242 of *LNCS*, pages 41–50. Springer, 2004.
4. A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.
5. O. Francois and C. Lavergne. Design of evolutionary algorithms – a statistical perspective. *IEEE Trans. on Evolutionary Computation*, 5(2):129–148, 2001.
6. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern.*, 16(1):122–128, 1986.
7. L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
8. M. Littman, N. Ravi, E. Fenson, and R. Howard. Reinforcement learning for autonomic network repair. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, pages 284–285. IEEE Computer Society, 2004.
9. S. D. Mueller, N. N. Schraudolph, and P. D. Koumoutsakos. Step size adaptation in evolution strategies using reinforcement learning. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 151–156. IEEE Press, 2002.
10. V. Nannen and A. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In *Proceedings of IJCAI'07, the 2007 International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 2007. to appear.
11. M. E. Samples, J. M. Daida, M. Byom, and M. Pizzimenti. Parameter sweeps for exploring GP parameters. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1791–1792, Washington DC, USA, 25-29 June 2005. ACM Press.
12. B. D. Smart. Reinforcement learning: A user's guide. Tutorial at International Conference on Autonomic Computing (ICAC 2005), 2005.
13. W. Spears. *Evolutionary Algorithms: the role of mutation and recombination*. Springer, Berlin, Heidelberg, New York, 2000.
14. R. S. Sutton and A. G. Barto. *Reinforcement Learning: an Introduction*. MIT Press, 1998.
15. G. Wang, E. D. Goodman, and W. F. Punch. Toward the optimization of a class of black box optimization algorithms. In *Proc. of the Ninth IEEE Int. Conf. on Tools with Artificial Intelligence*, pages 348–356, New York, 1997. IEEE Press.
16. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2 edition, 2005.