# Solving CSPs using self-adaptive constraint weights: how to prevent EAs from cheating

**A.E. Eiben**
Free University Amsterdam
and Leiden University

**B. Jansen**
Leiden University

**Z. Michalewicz**
UNC-Charlotte, USA, and
Polish Academy of Sciences

**Ben Paechter**
Napier University

## Abstract

This paper examines evolutionary algorithms (EAs) extended by various penalty-based approaches to solve constraint satisfaction problems (CSPs). In some approaches, the penalties are set in advance and they do not change during a run. In other approaches, dynamic or adaptive penalties that change during a run according to some mechanism (a heuristic rule or a feedback), are used. In this work we experimented with self-adaptive approach, where the penalties change during the execution of the algorithm, however, no feedback mechanism is used. The penalties are incorporated in the individuals and evolve together with the solutions.

## 1 Introduction

A **constraint satisfaction problem** (CSP) is a pair $\langle S, \phi \rangle$, where $S$ is a Cartesian product of sets $S = D_1 \times \ldots \times D_n$ (called the free search space), and $\phi$ is a formula (Boolean function on $S$). A **solution of a constraint satisfaction problem** is an $\bar{s} \in S$ with $\phi(\bar{s}) = true$. Usually a CSP is stated as a problem of finding an instantiation of variables $v_1, \ldots, v_n$ within the finite domains $D_1, \ldots, D_n$ such that constraints (relations) $c_1, \ldots, c_m$ prescribed for (some of) the variables hold. The feasibility condition (the formula $\phi$) is then given by the conjunction of the given constraints.

Evolutionary algorithms are usually considered to be ill-suited for solving constraint satisfaction problems. One of the reasons for this is that the traditional search operators are "blind" to the constraints, so that parents satisfying a certain constraint could produce offspring which violate it. Furthermore, while EAs have a "basic instinct" to optimise, a CSP has no objective function — just a set of constraints to be satisfied.

There are several approaches that have been devised to attempt to address this problem [12, 13]. These include repair algorithms, decoders, specialized (constraint-preserving) operators and heuristic operators. These methods are inter-related and some approaches use a combination of them.

Repair algorithms work on the principle that a while a child may have some unsatisfied constraints, a heuristic method can be used to attempt to alter the chromosome directly so that a larger number of constraints are satisfied.

Decoders make use of an indirect representation and a "growth" engine which converts the genotype into a phenotype in a way that attempts to maximise the number of satisfied constraints. For example, the "growth" engine might be a greedy algorithm, and the indirect representation might parameterise that algorithm by defining the order in which the variables should be considered.

The use of specialized operators involves defining mutation and recombination operators so that they are either certain or likely to preserve the satisfied constraints of the parents. Often this method is combined with a seeding operation which uses a heuristic to ensure that at least some members of the initial population have a larger number of constraints satisfied than would randomly created chromosomes.

Heuristics can help to solve CSPs by adding some "intelligence" into the operators. The operators can be designed so that they tend to increase the number of constraints satisfied, by use of some knowledge about the problem. This can be combined with "targeted mutation" — where mutation is targeted towards those variables which are causing constraints to be broken [7, 16, 15, 14].

Whichever of the above methods is used, some objective function is required for the EA to operate. This is normally constructed by having a penalty scheme, where the breaking of constraints adds to the penalty. Usually weights are given to the individual constraints so that:

$$f(s) = \sum_{i=1}^{m} w_i \times \chi(s, c_i), \tag{1}$$

where $s$ is a candidate solution, $c_i$ ($i \in \{1, \ldots, m\}$) are the given constraints to be satisfied, and

$$\chi(s, c_i) = \begin{cases} 1 & \text{if } s \text{ violates } c_i \\ 0 & \text{otherwise.} \end{cases}$$

Changes in the weights will cause the EA to put more or less effort into the satisfaction of any particular constraint. More importantly, weight changes alter the shape and characteristics of the search space. In order to solve the problem most effectively, we need to have weights that transform the search space into one that the EA finds easy to navigate through. A heuristic might tell us that the constraints which are hardest to satisfy should be given the highest weights. But this has two problems. Firstly, we are left with the problem of determining which constraints are hardest to solve, and secondly, the heuristic may not be correct!

The obvious answer to this problem is to let the EA evolve its own weights. The so-called SAW-ing mechanism [8, 9, 10] achieves this in an adaptive[1] fashion: the run is periodically stopped and weights of unsatisfied constraints are raised. While this mechanism has been successful on many problems it still has a heuristic component, the feedback mechanism, and two new parameters: the time elapsed between two weight updates and the weight increment. A straightforward way to get around these drawbacks is a self-adaptive approach with the weights given to each constraint included in the chromosome. But this option can bring its own problems, since the EA might improve the value of the objective function by evolving the weights rather than the variables. In other words a chromosome might "cheat" by saying "OK, so I don't satisfy that constraint, but I don't think that constraint is very important", and decreasing the corresponding weight. In particular, if the weights are zeros, the penalty is zero as well!

This observation has led to the belief that the self-adaptation of constraint weights for CSPs (and for optimization problems as well) is not possible. However, this is based on the assumption that an EA will proceed by assigning a fitness to each chromosome, and then using some selection or replacement strategy that is based on that fitness. In fact this need not be the case, and the technique introduced here neatly solves the problem. If we use tournament selection, then a universal fitness value is not required — just some way of comparing chromosomes. This allows us to delay deciding on the weights to use until the tournament competitors are known. As this point we can use the maximum of each of the weights, across all competitors, and so eliminate cheating.

[1] For a thorough treatment of the notions adaptive, self-adaptive etc. parameter control in EAs, see [6].

In this paper we report on an experimental investigation of this self-adaptive method for setting constraint weights on CSP's. It is self-adaptive because the influence of the user on the weights is completely eliminated. There is not even a weak impact as in the heuristic feedback mechanism in an adaptive scheme; we leave the weight calibration entirely on the EA itself. The underlying motivation is formed by the belief that evolution is powerful enough to calibrate itself. In the area of evolution strategies self-adaptivity is a standard feature with many experimental and theoretical support for its usefulness [1, 17]. However, there are two crucial differences between those findings in ES and our investigation here. First, the known results in ES concern continuous parameter optimization problems, while we investigate discrete constraint satisfaction problems. Second, in ES it is the mutation stepsize — and sometimes the direction — that is self-adapted. We introduce and study self-adaptation of the fitness function itself.

The paper is organized as follows. In section 2 we discuss the self-adaptive algorithm used in all experiments. Section 3 presents the test problems, while section 4 shows the experimental setups (algorithms and performance measures). Section 5 discusses the experimental results and section 6 concludes the paper.

## 2 The Self-Adaptive Algorithm

The technique described in this paper is self-adaptive in the sense that certain parameters, namely the constraint weights that define the evaluation function, are included in the chromosomes. Thus they are subject to evolutionary process, and they undergo recombination, mutation, and selection, just as the problem variables in the chromosomes.

### 2.1 Representation, evaluation, and selection

We represent a candidate solution of a given CSP by an integer vector $\vec{v}$, where $v_i$ stands for the $i$-th variable; its values are taken from the domain $D_i$. An individual $\vec{x}$ consists of two parts, $\vec{x} = \langle \vec{v}, \vec{w} \rangle$, where $\vec{v}$ (of length $n$) holds the instantiations of the problem variables and $\vec{w}$ (of length $m$) contains the constraint weights (positive integers).

Rather than a fitness value to be maximized, we use the total penalty (to be minimized) as an evaluation function. For a given individual $\langle \vec{v}, \vec{w} \rangle$ it is defined as follows:

$$g(\vec{v}, \vec{w}) = \sum_{i=1}^{m} w_i * \chi(\vec{v}, c_i),$$

where

$$\chi(\vec{v}, c_i) = \begin{cases} 1 & \text{if } \vec{v} \text{ violates } c_i \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, an individual $\langle \vec{v}, \vec{w} \rangle$ is a solution for the given CSP if and only if $g(\vec{v}, \vec{w}) = 0$ and $w_i > 0$ for all $i$.

For the reasons explained in section 1 above, we use tournament selection. Given a tournament of 2 individuals, $\langle \vec{v}_1, \vec{w}_1 \rangle$ and $\langle \vec{v}_2, \vec{w}_2 \rangle$, we define $\vec{w}_{max_i} = \max(w_{1_i}, w_{2_i})$ for all $1 \leq i \leq m$ and compare $g(\vec{v}_1, \vec{w}_{max})$ with $g(\vec{v}_2, \vec{w}_{max})$. The winner of the tournament is the individual with the lowest $g(\cdot, \vec{w}_{max})$ value. This mechanism can be straightforwardly generalized to $k$-tournament.

### 2.2 Other components and parameters

In our experiments we used 1-point crossover (applied with probability $p_c = 1.0$), a mutation operator randomly changing a value with $p_m = 0.1$, and a population size of 10 in a steady state fashion[2]. In each cycle two parents are chosen. These parents create two offspring (crossover plus mutation). The new generation of 10 is selected from the 12 individuals by a given replacement mechanism (we experimented with a few options here; see section 4). The maximum number of fitness evaluations is 100,000.

## 3 Test cases

We ran experiments on randomly generated binary CSPs with $n = 15$ variables and a uniform domain size $|D_i| = 15$ $(1 \leq i \leq 15)$. The values of the weights (integers) ranged between 1 and 50. The problem instances were generated by the `randomCsp` program written by J.I. van Hemert[3], based on earlier work of Gerry Dozier [3, 4]. This program can generate random CSPs, for any given combination of constraint density $d$ (the ratio of constraints w.r.t. all possible constraints) and constraint tightness $t$ (the ratio of allowed vs. not allowed value combinations). This allows a systematic testing and an assessment of an algorithm's niche, i.e., the identification of the type of problems for which that algorithm performs well. It is known that for some combinations of $d$ and $t$ the problem instances are easy, they are solvable in a short time. For other values the generated instances are unsolvable and the two regions are separated by the so-called mushy region (phase transition). Instances here have typically only one solution and they are hard to solve [2].

We tested our algorithms on 25 different combinations of constraint tightness and density. For each $d \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ and $t \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ we generated 10 different instances. This amounted to 250 problem instances in total.

## 4 Experimental setups

All experiments were performed by two categories of evolutionary algorithms: (1) steady-state and (2) $(\mu, \lambda)$ EA. We discuss these in turn.

### 4.1 Steady-state EA

We implemented four algorithm variants, differing in the selection mechanisms. For both the parent selection and the replacement mechanism we tested uniform random choice and 4-tournament selection. Furthermore, we experimented with an intensified 10-tournament scheme in the replacement mechanism. This led to the following four variants:

- Variant A(r 4):
  - random parent selection (2 parents will be uniform randomly selected)
  - 4-tournament replacement (select 4 chromosomes randomly and delete the worst one, where "worst" is defined by the above function $g$ and $\vec{w}_{max}$).
- Variant B(4 r):
  - 4-tournament parent selection (choose 4 chromosomes randomly and the best one, based on $g$ and $\vec{w}_{max}$, will be used to create children through crossover and mutation)

[2]Dozier's microgenetic algorithm as well as our own earlier research support small populations for CSPs.

[3]See www.wi.leidenuniv.nl/~ jvhemert

- random replacement.

- Variant C(4 4):
  - 4-tournament parent selection
  - 4-tournament replacement
- Variant D(4 10):
  - 4-tournament parent selection
  - 10-tournament replacement (very close to worst-fitness replacement)

With each algorithm variant we executed 10 independent runs on each problem instance. The results shown in table 1 are thus based on the 100 runs for each combination of $(d, t)$. The **Success Rate** (SR) values give the percentage of instances where a solution has been found. The **Average Number of Evaluations to Solution** (AES) is the number of fitness evaluations, i.e. the number of newly generated candidate solutions, in *successful* runs.

### 4.2 $(\mu, \lambda)$ EA

In the evolution strategies community there is much experience with self-adaptation. Although those experiences concern different problems (continuous parameter optimization vs. discrete constraint satisfaction) and self-adaptation of a different algorithm parameter (mutation step size vs. evaluation function), it is natural to ask whether conclusions drawn there would hold in our problem context too. In particular, it is generally believed in ES that successful self-adaptation has two preconditions:

1. a surplus of offspring, typically 7 times as much as the size of the parent population;
2. "forgetfulness", i.e. a $(\mu, \lambda)$ strategy, discarding the parents immediately and only considering the offspring for inclusion in the new generation.

From this perspective our algorithms are ill-suited to perform self-adaptation, since in the terms of $\mu$ and $\lambda$ our steady-state mechanism amounts to a $(10 + 2)$ strategy. This motivated a second series of experiments where the algorithm setup adheres to the recommendations from ES. To this end we redesigned the population model and implemented a $(\mu, \lambda)$ EA with the same parameters as before, $\mu$ being 10 and varying $\lambda = 26, 50, 70$. Recall, that parent selection is always uniform random in ES. As for the survivor selection (replacement strategy), we tested these algorithms with both 4-tournament and 10-tournament selection. That is, the 10 members of the new generation are chosen from the offspring by independently executing 10 tournaments with size 4, respectively 10. This gave us 6 new algorithm variants to test (3 values of $\lambda$, 2 tournament sizes). The results, based again on 10 independent runs on each problem instance, are presented in table 2 and table 3, for 4-tournament and 10-tournament, respectively.

## 5 Evaluation of Results

The results on the steady-state EA variants indicate the soundness of the basic ideas behind this research, self-adaptation of penalties in combination with tournament selection does work. There are differences between the algorithm variants. Apparently applying the selective pressure in the parent selection step (and using uniform selection in the replacement strategy) is inferior to the other setups. The best option seems to be the C(4 4) variant, (not too high) selective pressure for parent selection *and* survivor selection.

For the $(\mu, \lambda)$ style EAs it holds that, independently from the applied selective pressure (i.e., the tournament size), 26 offspring are not enough. The best setup seems to be a medium offspring size and a strong selective pressure, the (10,50) EA with 10-tournament.

It is very interesting to look at the outcomes from the perspective of self-adaptation itself. That is, to see what setup allows the best self-adaptation. The best algorithm in the (10+2) scheme is C(4 4), while the best ($\mu$,$\lambda$) algorithm is (10,50). Comparing their performance we see an advantage of the (10+2) method. This is in contrast with the general recommendations in evolution strategies, where it is suggested that a comma-strategy with many offspring is necessary to have self-adaptation work (and thus to have the best algorithm performance). Our results do not support those recommendations. It seems that they must be restricted to continuous parameter optimization and the self-adaptation of the mutation parameters (stepsize and rotation angles). Our results with discrete constraint satisfaction problems and self-adaptation of the evaluation function point into another direction. Although the experimental support for general recommenations is not sufficient at this moment, our results indicate a challenging research subject.

## 6 Conclusions

Our results show that the initial intuition of the possibly "cheating" EA (minimizing weights, instead of solving constraints) is not correct. The easy instances (upper left corner in the tables) are always solved, and the EA can maintain a non-zero success rate even in the mushy region, where the phase transition takes place [2].

Further research is being performed along different lines. First we are comparing the self-adaptive approach (as introduced here) to the adaptive approach applied in the the SAW-ing EA for constraint satisfaction [10]. An additional idea is combine both mechanisms and to use an evaluation function based on a double sum: one being self-adaptive, one being "SAW"-ed.

As for studying the phenomenon of self-adaptation, we plan to test $(10 + 50)$ strategies versus $(10, 50)$ strategies. We need to perform more experiments to analyse why the new mechnism works and why the ES-based conjectures on the advantages of $(\mu, \lambda)$ EAs with many offspring are invalid in our problem context. To this end, the exact effects of the self adaptive mechanism on the constraint weights and thus on the fitness landscape need to be studied.

## References


[1] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[2] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th IJCAI-91*, volume 1, pages 331–337, Morgan Kaufmann, 1991. Morgan Kaufmann.

[3] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In IEEE [11], pages 306–311.

[4] G. Dozier, J. Bowen, and A. Homaifar. Solving constraint satisfaction problems using hybrid evolutionary search. *IEEE Transactions on Evolutionary Computation*, 2(1):23–33, 1998.

[5] A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors. *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, Berlin, 1998. Springer.

[6] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[7] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In IEEE [11], pages 542–547.

[8] A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press, 1997.

[9] A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.

[10] A.E. Eiben, J.I. van Hemert, E. Marchiori, and A.G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In Eiben et al. [5], pages 196–205.

[11] *Proceedings of the 1st IEEE Conference on Evolutionary Computation*. IEEE Press, 1994.

[12] Z. Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. In J.R. McDonnell, R.G. Reynolds, and D.B. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*, pages 135–155. MIT Press, 1995.

[13] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[14] B. Paechter, R.C. Rankin, A. Cumming, and T.C. Fogarty. Timetabling the classes of an entire university with an evolutionary algorithm. In Eiben et al. [5].

[15] P. Ross, D. Corne, and H. Fang. Improving evolutionary timetabling with delata evaluation and directed mutation. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[16] P. Ross and E. Hart. An adaptive mutation scheme for a penalty based graph-colouring GA. In Eiben et al. [5], pages 795–802.

[17] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

| density | alg | tightness 0.1 | | 0.3 | | 0.5 | | 0.7 | | 0.9 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SR | AES | SR | AES | SR | AES | SR | AES | SR | AES |
| 0.1 | A(r 4) | 1.00 | 1 | 1.00 | 19 | 1.00 | 64 | 1.00 | 129 | 1.00 | 224 |
| | B(4 r) | 1.00 | 1 | 1.00 | 14 | 1.00 | 64 | 1.00 | 158 | 1.00 | 293 |
| | C(4 4) | 1.00 | 1 | 1.00 | 14 | 1.00 | 40 | 1.00 | 82 | 1.00 | 129 |
| | D(4 10) | 1.00 | 1 | 1.00 | 12 | 1.00 | 38 | 1.00 | 81 | 1.00 | 127 |
| 0.3 | A(r 4) | 1.00 | 26 | 1.00 | 298 | 1.00 | 2223 | 1.00 | 8088 | 0.49 | 35472 |
| | B(4 r) | 1.00 | 19 | 1.00 | 529 | 1.00 | 25387 | 0.01 | 59476 | 0.00 | - |
| | C(4 4) | 1.00 | 17 | 1.00 | 174 | 1.00 | 906 | 1.00 | 5940 | 0.43 | 37464 |
| | D(4 10) | 1.00 | 19 | 1.00 | 177 | 1.00 | 1063 | 1.00 | 4909 | 0.42 | 33130 |
| 0.5 | A(r 4) | 1.00 | 121 | 1.00 | 3272 | 0.14 | 39121 | 0.00 | - | - | - |
| | B(4 r) | 1.00 | 118 | 0.61 | 41872 | 0.00 | - | 0.00 | - | 0.00 | - |
| | C(4 4) | 1.00 | 80 | 1.00 | 2445 | 0.27 | 42044 | 0.00 | - | 0.00 | - |
| | D(4 10) | 1.00 | 71 | 1.00 | 3006 | 0.17 | 46861 | 0.00 | - | 0.00 | - |
| 0.7 | A(r 4) | 1.00 | 532 | 0.08 | 36456 | 0.00 | - | 0.00 | - | 0.00 | - |
| | B(4 r) | 1.00 | 806 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | C(4 4) | 1.00 | 407 | 0.07 | 61677 | 0.00 | - | 0.00 | - | 0.00 | - |
| | D(4 10) | 1.00 | 463 | 0.07 | 19623 | 0.00 | - | 0.00 | - | 0.00 | - |
| 0.9 | A(r 4) | 0.54 | 14326 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | B(4 r) | 0.49 | 32640 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00- | - |
| | C(4 4) | 0.55 | 10867 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | D(4 10) | 0.51 | 13531 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |

Table 1: Success rates and the corresponding AES values for the steady-state style self-adaptive EAs

| density | alg | tightness 0.1 | | 0.3 | | 0.5 | | 0.7 | | 0.9 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SR | AES | SR | AES | SR | AES | SR | AES | SR | AES |
| 0.1 | (10, 26) | 1.00 | 1 | 1.00 | 38 | 1.00 | 120 | 1.00 | 243 | 1.00 | 403 |
| | (10, 50) | 1.00 | 1 | 1.00 | 57 | 1.00 | 138 | 1.00 | 274 | 1.00 | 398 |
| | (10, 70) | 1.00 | 1 | 1.00 | 69 | 1.00 | 170 | 1.00 | 286 | 1.00 | 453 |
| 0.3 | (10, 26) | 1.00 | 42 | 1.00 | 592 | 1.00 | 10467 | 0.10 | 61211 | 0.00 | - |
| | (10, 50) | 1.00 | 67 | 1.00 | 502 | 1.00 | 1928 | 1.00 | 10021 | 0.49 | 41950 |
| | (10, 70) | 1.00 | 85 | 1.00 | 578 | 1.00 | 1863 | 1.00 | 10013 | 0.45 | 35949 |
| 0.5 | (10, 26) | 1.00 | 199 | 1.00 | 23397 | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 50) | 1.00 | 235 | 1.00 | 4919 | 0.24 | 48430 | 0.00 | - | 0.00 | - |
| | (10, 70) | 1.00 | 265 | 1.00 | 5764 | 0.15 | 48968 | 0.00 | - | 0.00 | - |
| 0.7 | (10, 26) | 1.00 | 889 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 50) | 1.00 | 791 | 0.08 | 40870 | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 70) | 1.00 | 1060 | 0.04 | 26297 | 0.00 | - | 0.00 | - | 0.00 | - |
| 0.9 | (10, 26) | 0.61 | 23176 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 50) | 0.43 | 19080 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 70) | 0.46 | 17188 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |

Table 2: Success rates and AES values for the $(\mu, \lambda)$ self-adaptive EAs with 4-tournament selction

| density | alg | tightness | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0.1 | | 0.3 | | 0.5 | | 0.7 | | 0.9 | |
| | | SR | AES | SR | AES | SR | AES | SR | AES | SR | AES |
| 0.1 | (10, 26) | 1.00 | 1 | 1.00 | 38 | 1.00 | 116 | 1.00 | 215 | 1.00 | 441 |
| | (10, 50) | 1.00 | 1 | 1.00 | 53 | 1.00 | 137 | 1.00 | 256 | 1.00 | 374 |
| | (10, 70) | 1.00 | 1 | 1.00 | 70 | 1.00 | 173 | 1.00 | 287 | 1.00 | 403 |
| 0.3 | (10, 26) | 1.00 | 37 | 1.00 | 512 | 1.00 | 7589 | 0.13 | 32238 | 0.00 | - |
| | (10, 50) | 1.00 | 60 | 1.00 | 467 | 1.00 | 1718 | 1.00 | 10117 | 0.56 | 45350 |
| | (10, 70) | 1.00 | 81 | 1.00 | 505 | 1.00 | 1483 | 1.00 | 6551 | 0.29 | 48050 |
| 0.5 | (10, 26) | 1.00 | 178 | 1.00 | 22083 | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 50) | 1.00 | 210 | 1.00 | 4937 | 0.28 | 43913 | 0.00 | - | 0.00 | - |
| | (10, 70) | 1.00 | 256 | 1.00 | 4000 | 0.09 | 37170 | 0.00 | - | 0.00 | - |
| 0.7 | (10, 26) | 1.00 | 856 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 50) | 1.00 | 908 | 0.04 | 65283 | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 70) | 1.00 | 664 | 0.07 | 38329 | 0.00 | - | 0.00 | - | 0.00 | - |
| 0.9 | (10, 26) | 0.61 | 23991 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 50) | 0.43 | 12801 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |
| | (10, 70) | 0.43 | 21236 | 0.00 | - | 0.00 | - | 0.00 | - | 0.00 | - |

Table 3: Success rates and AES values for the $(\mu, \lambda)$ self-adaptive EAs with 10-tournament selction