RESEARCH-ARTICLE

Operator Selection using Improved Dynamic Multi-Armed Bandit

JANY BELLUZ, National Higher School of Computer Science and Applied Mathematics, Saint Martin d'Heres, Auvergne-Rhone-Alpes, France

MARCO GAUDESI, Polytechnic of Turin, Turin, TO, Italy

GIOVANNI SQUILLERO, Polytechnic of Turin, Turin, TO, Italy

ALBERTO PAOLO TONDA, Paris-Saclay Food and Bioproduct Engineering, Massy, Ile-de-France, France

Published: 11 July 2015

GECCO '15: Genetic and Evolutionary Computation Conference
July 11 - 15, 2015
Madrid, Spain

Conference Sponsors:
SIGEVO

https://doi.org/10.1145/2739480.2754712
ISBN: 9781450334723

Operator Selection using Improved Dynamic Multi-Armed Bandit

Jany Belluz
Grenoble INP - Ensimag
681, rue de la passerelle -
Domaine universitaire - BP 72
Saint Martin d'Hères, France
jany.belluz@ensimag.fr

Marco Gaudesi,
Giovanni Squillero
DAUIN, Politecnico di Torino
Corso Duca degli Abruzzi, 129
Torino, Italy
{marco.gaudesi,
giovanni.squillero}@polito.it

Alberto Tonda
UMR 782 GMPA, INRA
1 av. Lucien Brétignères
78850, Thiverval-Grignon,
France
alberto.tonda@grignon.inra.fr

ABSTRACT

Evolutionary algorithms greatly benefit from an optimal application of the different genetic operators during the optimization process: thus, it is not surprising that several research lines in literature deal with the self-adapting of activation probabilities for operators. The current state of the art revolves around the use of the Multi-Armed Bandit (MAB) and Dynamic Multi-Armed bandit (D-MAB) paradigms, that modify the selection mechanism based on the rewards of the different operators. Such methodologies, however, update the probabilities after each operator's application, creating possible issues with positive feedbacks and impairing parallel evaluations, one of the strongest advantages of evolutionary computation in an industrial perspective. Moreover, D-MAB techniques often rely upon measurements of population diversity, that might not be applicable to all real-world scenarios. In this paper, we propose a generalization of the D-MAB approach, paired with a simple mechanism for operator management, that aims at removing several limitations of other D-MAB strategies, allowing for parallel evaluations and self-adaptive parameter tuning. Experimental results show that the approach is particularly effective with frameworks containing many different operators, even when some of them are ill-suited for the problem at hand, or are sporadically failing, as it commonly happens in the real world.

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods, and Search*

Keywords

Adaptivity; Operator selection; Multi-Armed Bandit; Self-adapting

GECCO '15 July 11–15, 2015, Madrid, Spain

ISBN 978-1-4503-3472-3/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739480.2754712

1. INTRODUCTION

With each passing year, evolutionary algorithms (EAs) are more and more adopted in industries as optimization techniques, thanks to their ability of efficiently navigating the search space of NP-hard problems, thus delivering acceptable solutions in a reasonable amount of time [17]. Nevertheless, their wide-spread adoption is slowed down by the expert knowledge needed to fine-tune parameters such as population size, type of individual selection, and – more importantly – activation probabilities for the genetic operators. It is no coincidence if the EAs most popular outside the evolutionary community require little to no parameter tuning at all [19, 9, 6].

Several research lines focus on self-adapting EAs' parameters during the optimization, in order to free the users from the burden of finding the optimal settings for their specific application. A considerable number of articles in literature deal with the self-adaptation of the activation probabilities of the genetic operators. Regardless the strategy, that may range from encoding probabilities directly into the genome, to inserting heuristic triggers in the algorithm, an optimal choice of the best operators can significantly enhance the optimization process. However, properly tweaking the probabilities of application for a certain operator over the different phases of the process is quite challenging, especially when an evolutionary framework includes several specialized mutations, and different kinds of crossovers.

In order to deal with such complexity, the scientific community traditionally resorted to the Multi-Armed Bandit (MAB) paradigm: initially developed by game theorists [11, 1], this framework tries to balance the exploration for better operators with the exploitation of the best ones. Since the optimal operator to apply during the evolution is likely to change during the evolutionary process, the problem has been re-stated to be solved as a Dynamic Multi-Armed Bandit (D-MAB) approach.

The most successful MAB-based operator selection frameworks share a common requirement: new individuals should be evaluated between each operator application [4, 12, 7]. However, industrial use cases for EA optimization usually feature computationally intensive evaluation functions, that may run complex simulations [3] or even test a candidate solution on a physical device [8]. Such problems make it mandatory to parallelize the evaluation process, which is clearly incompatible with the current limitations of MAB-based operator selection schemes. Finally, in use cases where

the structure of a solution is extremely complex and several operators are available, even an operator that produces good solutions can fail sporadically, and such a possibility should be acknowledged and correctly handled as part of the operator selection strategy, whereas failing operators are often completely removed from previous D-MAB approaches.

In this paper, we propose a generalization of the D-MAB approach, aimed at removing the requirements for sequential evaluations. The proposed approach makes more sensible choices during the period in which we must select λ operators in a row without receiving any new information. Indeed, our strategy improves over naïvely selecting λ times the same operator by maintaining a part of exploration during the same period. The new mechanism is tuned to be fully exploitable in μGP, an industrial-grade generic evolutionary toolkit [16].

The rest of the paper is organized as follows. Section 2 briefly presents the necessary concepts to introduce the scope of this work. Section 3 illustrates the proposed approach, while Section 4 shows the experimental results on a set of test cases. Finally, Section 5 summarizes the conclusions drawn from the experience.

2. BACKGROUND

This section briefly introduces the necessary notions related to the scope of this work. First, the D-MAB framework is described; then, an overview of previous operator-selection framework is presented.

2.1 The Multi-Armed Bandit Framework

The "Exploration vs. Exploitation" dilemma has been intensively studied in game theory [2], especially in the context of the MAB framework [11, 1]. Let's consider an hypothetical slot machine with N arms (the bandit); at time t_k, the i-th arm, when selected, gets a reward 1 with probability p_i, and 0 otherwise. A solution to the MAB problem is a decision making algorithm that selects an arm at every time step, with the goal of maximizing the cumulative reward gathered during the process.

Generalizing the widely-studied Upper Confidence Bound (UCB) algorithm [1], that shows how to maximize the cumulative reward with optimal convergence rate on simple 0-1 arms, the player should select at each time step t the arm i that maximizes the following quantity:

$$\hat{p}_{i,t} + C \cdot \sqrt{\frac{\log \sum_k n_{k,t}}{n_{i,t}}} \tag{1}$$

where $n_{i,t}$ is the number of times the i-th arm has been activated from the beginning of the process to time t; while $\hat{p}_{i,t}$ denotes the average empirical reward received from arm i. C is a (critical) scaling factor not present in the original UCB definition, where normalization was not necessary, and it controls the trade-off between exploration, favored by the right term of the equation; and exploitation, favored by the left part of the equation, that pushes for the option with the best average empirical reward.

2.2 DMAB and Operators Selection in EA

The MAB problem can be intuitively applied to operator selection in EAs: every arm of the bandit can be mapped to one operator. Using the UCB metric, the algorithm keeps exploring all arms, while favoring good operators. However, contrary to the theoretical bandit, an evolutionary run is a dynamic environment, in which the standard MAB algorithm would require a considerable amount of time to detect that the best operator has changed. To solve this issue, [4] proposed to use a statistical change detection test, creating the Dynamic MAB (D-MAB) algorithm. Specifically, the Page-Hinkley test [10] is used to detect whether the empirical rewards collected for the best current operator undergo an abrupt change. In the D-MAB, if the PH test is triggered, suggesting that the current best operator is no longer the best one, the MAB algorithm is restarted.

An overview of the most successful D-MAB based mechanisms can be found in [7]. Further works build on the D-MAB algorithm most notably by comparing various credit assignment mechanisms and measuring how well they complement the D-MAB selection scheme. In [12], for example, the authors propose to combine the D-MAB with a credit assignment scheme called *Compass*, that evaluates the performance of operators by considering not only the fitness improvements from parent to offspring, but also the way they modify the diversity of the population, and their execution time.

3. PROPOSED APPROACH

We propose a D-MAB selection strategy that not only allows operators to sporadically fail without being completely removed from the process, but is also able to consecutively apply several operators without needing a performance feedback after each application. For our approach, we consider the following EA structure:

```
operators ← {available operators and their MAB state};
policy.init(operators);
parents ← {some random individuals};
until reached a stop condition do
    offspring ← [];
    applications ← [];
    policy.before_selections(operators);
    until λ successful operator applications do
        op ← policy.select(operators);
        children ← op.apply(parents);
        if children = ∅ then
            policy.failure(op);
        else
            policy.success(op);
            applications.append((op, children));
            offspring.append(children);
    evaluate(offspring);
    policy.reward(parents, offspring, applications);
    parents ← selection(parents, offspring);
```

Algorithm 1: Outline of our target EA

This general structure is shared by different EAs, such as the μGP toolkit [16], that will be our case study. In this type of architecture the evaluation phase can be easily parallelized, and operators can occasionally fail without being removed from the selection process. We adopt the general term "operators" to denote both mutation and crossover operators, as one operator in μGP is not bound to operate on a specific number of parents nor to produce a given offspring size.

During the current generation, the only information that the policy can gather is whether the selected operator actually produced children, through the functions `policy.success()` and `policy.failure()`. After the evaluation phase, the policy can access more information: the fitness of the newly produced offspring makes tournaments possible.

3.1 Notations

Each operator is considered as an arm of a MAB and is associated to several counters. First, *op.pending* keeps track of the number of successful applications awaiting a reward in each generation. Then, the *op.enabled* and *op.tokens* counters account for this operator's applicability. An operator is said to be *enabled* when it can produce new offspring and has already proved so at least once. Furthermore, an enabled operator can be selected only if it has a positive number of *tokens*. We use these tokens to limit the computation time spent in unadapted operators. Finally, we maintain a short history of the last obtained rewards in *op.window* and the classical D-MAB statistics, in $op.n$, $op.\hat{p}$, $op.m$ and $op.n$ [12]. Algorithm 2 covers the initialization of these variables. The initial number of tokens assigned to operators is set to a small number, such as $\kappa = 3$.

```
function policy.init(operators) is
    foreach op in operators do
        // Intra generational call count
        op.pending ← 0;
        // Failure handling statistics
        op.enabled ← False;
        op.tokens ← κ;
        // Compass-like window
        op.window ← queue of size τ;
        // D-MAB statistics
        op.n ← 0;
        op.p̂ ← 0;
        op.m ← 0;
        op.M ← 0;
```

Algorithm 2: Initialization of the operator statistics

3.2 Operator failures

We define a "failure" as the application of an operator that does not result in any new usable solution. This can happen for three reasons: the operator always produces an invalid individual (e.g., it removes or adds an element when the problem solution is a fixed-length list); the operator is never applicable (e.g., it performs a Gaussian mutation, but the genome is a bit string); its execution can sporadically fail based on the structure of the selected parents or other details (e.g., it concatenates two genomes and creates an individual too long to be valid). Such a notion, adopted in μGP and not quite common in classic combinatorial optimization, is essential to alleviate the burden of selecting the perfect set of operators [13].

In μGP, λ genetic operators are used in each generation. There is no distinctions between mutations and recombinations. As *failures* are not included in the count, more than λ operators could be selected and activated in order to reach the desired number. It is necessary to distinguish operators that always fail because inapplicable to the current problem, from operators that do not provide good solutions in the specific phase of the optimization process, or sporadically fail.

We adaptively handle inapplicable operators by considering all operators *disabled* until they prove their usefulness building one valid solution. All operators are called periodically to check whether some emergent characteristic of the population enables them to work in a later stage of evolution. As a consequence, when an unforeseen edge case is encountered during its execution, an operator can just fail without any penalty: this also makes it easier to add new operators to a framework, as foreseeing their effect on all possible genomes is not necessary.

Performance problems can arise if an operator is computationally intensive, builds very good solutions, but fails most of the time. Such an operator might get called repeatedly and use a considerable amount of CPU time for the production of few viable children. To avoid this situation, we use the idea of *failure tokens*, that is, a maximum number of failed calls allowed per generation. In this case, we set that a disabled operator will be tested again once every 10 generations. Once enabled, an operator will have a budget of λ authorized failures per generation, as shown by the first function of Algorihtm 3. The rest of our failure handling mechanism is implemented in the second function as a "filter" before the real selection strategy. Finally, the last two functions account for the consumption of tokens and the enabling of operators.

3.3 Credit assignment

A well-known approach to aggregate the different goals of exploration and exploitation is the Compass method [14, 12]. Compass associates the application of an operator with the variation of two characteristics of the population on which it operates: ΔD (mean diversity) and ΔQ (mean fitness). Its execution time T is also stored. These three values, averaged over a window of the last τ applications of the operator, are used to compute a reward. The meta-parameter Θ defines a compromise between the two first criteria ΔD and ΔQ: according to the authors, it affects the *Exploration vs. Exploitation* (EvE) orientation of the algorithm, and this compromise value is divided by the execution time of the operator to produce its reward.

This technique cannot be translated directly into all EAs, and, specifically, it requires several adjustment to be applied in μGP. Despite the several characteristics that needed to be left out, its underlying ideas proved to be extremely useful. First of all, μGP does not use the Θ angle: the tool already features a self-adapted σ parameter that controls how much the offspring diverges from the parent in mutation operators, and effectively regulates the amount of exploration [16].

As for diversity preservation and promotion, μGP provides different mechanisms such as fitness sharing, fitness scaling, delta entropy, and fitness holes, which encapsulate the *diversity vs. quality* problem into the comparison of individuals, either during selection of parents (fitness hole) or selection of survivors (scaled fitness) [15, 5, 16].

Moreover, even the ΔQ criteria is thus not available in μGP: the toolkit does not make any assumption about the fitness function. As in many industrial problems, candidate solutions can only be ranked, but their relative goodness cannot be evaluated on any absolute scale.

We therefore replace the $(\Delta Q, \Delta D, \Theta)$ triple with a single measure: we organize a soccer-style tournament between

```
function policy.before_selections(operators) is
    foreach op ∈ operators do
        op.pending ← 0;
        if op.enabled then
            op.tokens ← λ;
        else
            if current generation mod 10 = 0 then
                op.tokens ← 1;

function policy.select(operators) is
    foreach op in operators where ¬ op.enabled do
        if op.tokens > 0 then
            return op;
    enabled ← {op ∈ operators | op.enabled};
    if enabled = ∅ then
        redistribute κ tokens to all operators;
        return any operator;
    selectable ← {op ∈ enabled | op.tokens > 0};
    if selectable = ∅ then
        redistribute λ tokens to all enabled operators;
        selectable ← enabled;
    if ∃o ∈ selectable | o.n = 0 then
        return argmin_{o∈selectable} o.n + o.pending;
    return policy.real_select(selectable);

function policy.failure(operator) is
    operator.tokens ← operator.tokens − 1;

function policy.success(operator) is
    operator.pending ← operator.pending + 1;
    if ¬operator.enabled then
        operator.enabled ← True;
        operator.tokens ← λ;
```

Algorithm 3: Execute at least once all operators before using DMAB and limit failure rate

all the parents and all the freshly evaluated offspring, and reward the offspring proportionally to their final rank. This procedure provides a comparison of the new offspring with respect to their parents.

We choose not to consider the execution time T: μGP has been designed to target industrial-scale problems where the evaluation cost is usually predominant over the time spent in the evolutionary loop [18]. Moreover, the architecture has been designed to impose the lowest possible coupling between the evaluator and the evolutionary algorithm, and may not be meaningful or correlated with the generating operator.

One feature of interest remains from Compass: the time window of the last τ values of $(\Delta Q, \Delta D, \Theta)$. Our selection scheme will indeed use a window of past fitness improvements to distribute rewards. However, the original version of Compass[14] and the more recent work to pair it with D-MAB [12] differ: while the former computes a mean, the latter argues that using extreme values (i.e. the max) yields better results, borrowing the idea from Whitacre et al. [21]. We choose to compromise between the two options by computing a weighted sum of the values, assigning exponentially decreasing weights ($w = \vartheta^i = \frac{1}{2^i}$) to the sorted values. Whenever required, the compromise could be further tuned adjusting the discount $\vartheta \in [0, 1]$: a value of $\vartheta = 0$ gives the maximum, a value of $\vartheta = 1$ the mean.

This leads us to Algorithm 4 for reward distribution.

```
function policy.reward(parents, offspring,
applications) is
    tournament ← parents ∪ offspring;
    sort tournament by increasing fitness;
    foreach (op, children) in applications do
        improvement ← max_{children} tournament.rank(child) / tournament.size();
        op.window.append(improvement);
        W ← sorted op.window in decreasing order;
        r ← (Σ_{i=0}^{W.size()} W_i 2^{-i}) / (Σ_{i=0}^{W.size()} 2^{-i});
        // D-MAB algorithm from DaCosta
        op.p̂ ← 1/(op.n+1) (op.n op.p̂ + r);
        op.n ← op.n + 1;
        op.m ← op.m + (op.p̂ − r + δ);
        op.M ← max(op.M, op.m);
        if op.M − op.m > γ then
            reset all MAB statistics of all operators;
```

Algorithm 4: Credit assignment

3.4 Operator selection

The limitation we found to the D-MAB selection scheme lies in the exclusive dependence of an operator's selection on its obtained rewards: in our case it means that, during one generation, D-MAB will select the same operator λ times. Experimental results will show that this strategy is suboptimal with higher values of λ, because when the D-MAB takes the decision to exploit an operator it does so λ times in a row without any exploration, and conversely, when exploration is needed for an operator, it is called λ times even if it's the worst available.

We propose to mitigate this "all or nothing" intragenerational effect in the following way: we consider that operator applications during the generation receive immediately a fake reward equal to their current estimated reward. I.e., given an operator *op*, we simply increment the number of successful executions *op.n* for each successful application while maintaining the three other MAB statistics (*op.r̂*, *op.m*, *op.M*) to their original values.

This makes the DMAB scores vary enough during the generation to allow exploration to happen. After the evaluation of all candidates, the other D-MAB statistics are updated as usual using the actual rewards. We call this strategy PDMAB (Parallelized D-MAB).

```
function policy.real_select(operators) is
    total_n ← Σ_{o∈operators} o.n + o.pending;
    return argmax_{o∈operators} o.p̂ + C √(log total_n / (o.n + o.pending))
```

Algorithm 5: PDMAB strategy

4. EXPERIMENTAL RESULTS

4.1 Reference strategies

We compare our custom strategy with both the original DMAB and a completely random selection, used as a baseline soundness check. We use the following values for the meta-parameters: $\gamma = 1$, $C = \sqrt{2}$, $\delta = 0.15$. Since all the considered strategies are DMAB-based and share the same parameters, the comparison requires no off-line parameter tuning.

The simulations and actual tests have been carried with various values of $\lambda = 1, 20, 50, 100$. The steady case ($\lambda = 1$) is not interesting in our comparison, but we ran it to make sure that our PDMAB strategy is actually equivalent to standard DMAB. The other extreme, $\lambda = 100$, will not be discussed either because we found that intermediary values of λ already illustrate our point well enough.

4.2 Simulated behavior

We begin by implementing the strategies in a reduced environment that simulates operator application and individual evaluation by directly drawing a reward from selected statistical distributions. This allows us to compute the cumulative regret of the different strategies and make empiric predictions about their asymptotic behavior, with the strong hypothesis of a "perfect" credit assignment strategy, which would provide an exact representation of the quality of operators.

We use several "operator reward landscapes": a simple case with one constantly good operator and ten bad ones, a dynamic version of this test where the role of the good operator changes smoothly several times during the evolution, and finally an environment with several mediocre operators and an occasionally very good one. All these situations are tested with and without an added white noise.

These simulations show that the accumulated regret of the pure D-MAB strategy augments as λ grows, as shown by figure 1. On the first graph, which represents the static situation, we see that the DMAB accumulates more regret than PDMAB during the learning phase, at the beginning. Indeed, when the DMAB selects a bad operator for exploration, it is applied 50 times in a row, while the PDMAB distributes the exploration over time more smoothly. Even if on this static case the difference between the two approaches is progressively reduced as PDMAB catches up on exploration, in a dynamic setting the first part is repeated after each reset, which means the DMAB accumulates a lot of regret after each change in the environment. This is shown by the second and third graphs.

The second graph of figure 1 showcases the situation that is most representative of a real evolutionary run: at each stage of the evolution one operator is more than the others (left). While the D-MAB is the best strategy for a steady-state algorithm (equivalent to PDMAB for $\lambda = 1$, not shown), it starts accumulating more regret than PDMAB for $\lambda = 20$ (middle) and becomes clearly suboptimal for $\lambda \geq 50$ (right). On the third graph, we explain the advantage of PDMAB by the fact that its continuous exploration allows it to reset faster than the classic DMAB when the intermittent operator changes its state.

4.3 Full-stack benchmark

We ran two full tests, using μGP as the evolutionary engine. First, a specially crafted variation on OneMax is used to validate our simulated results on a simple and well known problem. Then, the newer Lamps benchmark demonstrates the applicability of the proposed strategies to Group Evolution [20].

Our version of One Max consists in maximizing the number of ones in an initially null string of 10,000 bits, using two groups of mutation operators: *static* and *dynamic*. The chances of creating an individual fitter than the parent is constant in the former group, while it depends on the current phase of the optimization process for the latter group. In more detail, static operators are: *RandomFlip* (randomly change the value of a bit), *Set*, and *Reset* (assign a given value to the bit). Dynamic operators are: *LocalMajoritySet*, consider a bit's immediate neighbors in the string and, if both are of the same value, set it to the same value; *GlobalMajoritySet*, considers the whole string and assign the bit the most common value. In an analogous way, we define the *Local-* and *GlobalMinorityReset* operators. Obviously with such definitions, when the individual fitness reaches the middle value of 5,000, the performance of the global operators will be brutally inverted, while the quality of the local version will slowly change during the whole evolution.

In order to stress the failure handling mechanism, all these operators have been defined very lazily: for example, if the "set one bit" operator randomly selects a 1, it will fail instead of trying again. In addition to that, we threw in all the available μGP operators, however irrelevant they might be. Most of them will fail constantly and stay disabled, the others will at most reset the whole bit string and be scorned by the D-MAB.

The experimental results show that the OneMax benchmark perfectly reproduces the simulated behavior, as shown by figure 2. The dominance of PDMAB over DMAB is statistically significant with a p-value of 0 using the Kolmogorov-Smirnov two-sample test, for both value of λ.

The results on Lamps with $\lambda = 20, 50$ are not statistically significant, with p-values of 0.13 and 0.5. We therefore conclude that our approach is at least as good as the standard DMAB on group evolution problems, and we propose to improve these results in a later work through an in-depth study of credit assignment and operator selection in the context of group evolution.

5. CONCLUSIONS

In this paper, we presented an extension to the Dynamic Multi-Armed Bandit approach for operator selection for evolutionary algorithms. Our approach brings the performance benefits of the DMAB strategy to industrial-grade generational algorithms such as μGP, with the added benefit of simplifying its usage, removing the need to select operators beforehand, and relaxing the constraints on operator definitions, as operators can fail sporadically.

Our simulation and experimental results on two benchmarks demonstrate that the extension indeed yields in all setups a performance at least equal to the DMAB approach for specific problems and clearly better in other cases.

Further works will focus on the validation of this selection scheme on more complex problems, and will conduct a more precise study of the proposed credit assignment.

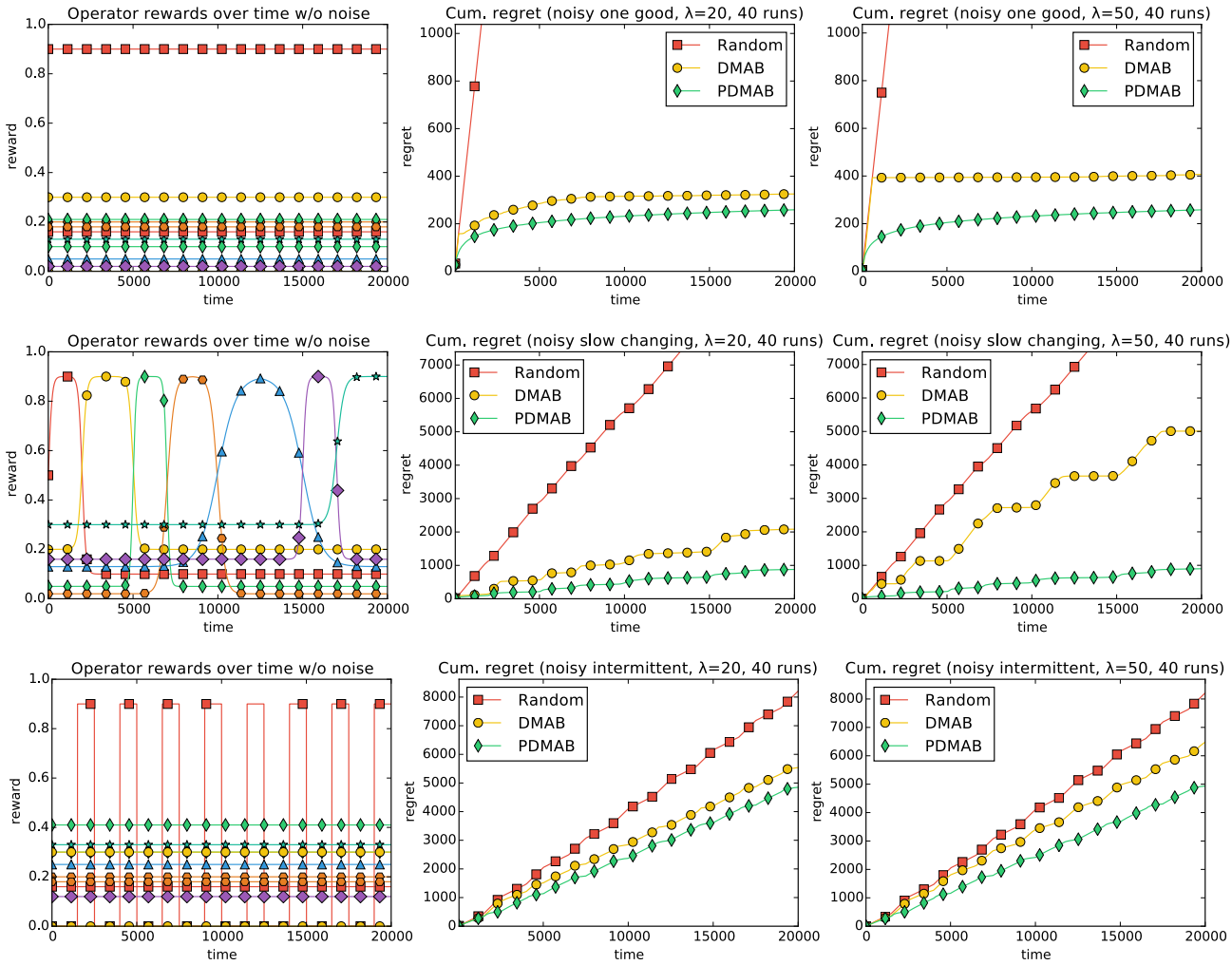

Figure 1: Simulation results for the three strategies (baseline Random, reference DMAB and modified PDMAB), over the three operator setups, for $\lambda = 20, 50$. Lower regret is better.

6. REFERENCES

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[2] N. Cesa-Bianchi and G. Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006.

[3] F. Corno, E. Sánchez, and G. Squillero. Evolving assembly programs: how games help microprocessor validation. *Evolutionary Computation, IEEE Transactions on*, 9(6):695–706, 2005.

[4] L. DaCosta, A. Fialho, M. Schoenauer, and M. Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 913–920. ACM, 2008.

[5] K. Deb and D. E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann Publishers Inc., 1989.

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

[7] Á. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1-2):25–64, 2010.

[8] S. Gandini, W. Ruzzarin, E. Sanchez, G. Squillero, and A. Tonda. A framework for automated detection of power-related software errors in industrial verification processes. *Journal of Electronic Testing*, 26(6):689–697, 2010.

[9] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[10] D. V. Hinkley. Inference about the change-point from cumulative sum tests. *Biometrika*, 58(3):509–523, 1971.

[11] T. L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

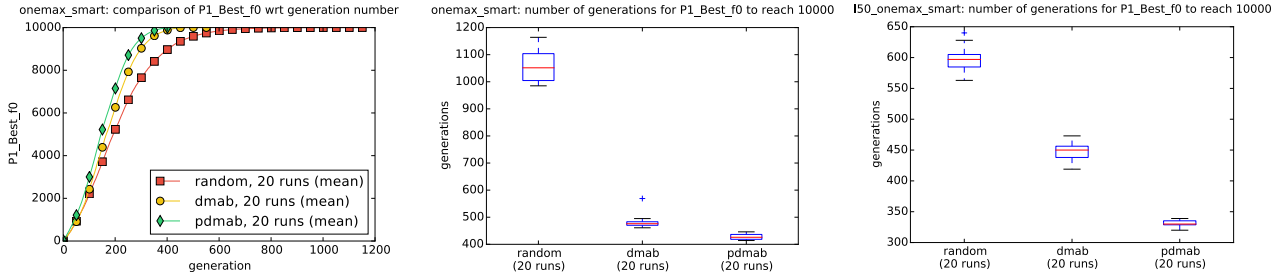

Figure 2: Experimental results for OneMax: fitness over time for $\lambda = 20$, number of generations before convergence for $\lambda = 20, 50$. Fewer generations is better.

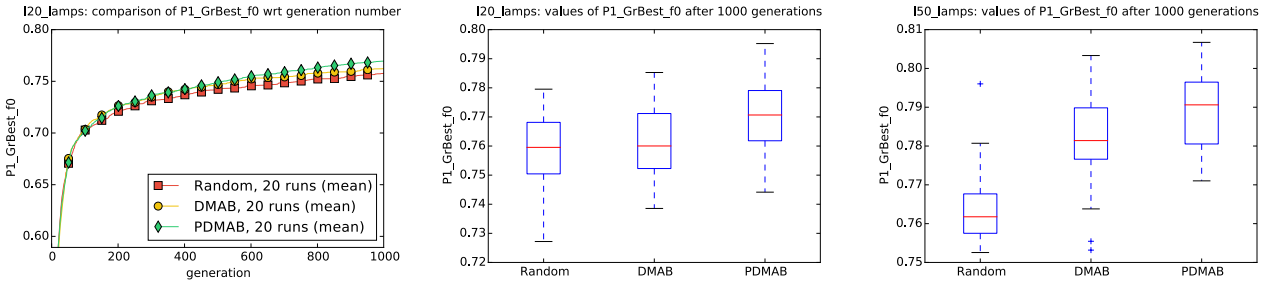

Figure 3: Experimental results for Lamps: fitness over time for $\lambda = 20$, maximum group fitness after 1000 generations for $\lambda = 20, 50$. Higher fitness is better.

[12] J. Maturana, Á. Fialho, F. Saubion, M. Schoenauer, and M. Sebag. Extreme compass and dynamic multi-armed bandits for adaptive operator selection. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 365–372. IEEE, 2009.

[13] J. Maturana, F. Lardeux, and F. Saubion. Autonomous operator management for evolutionary algorithms. *Journal of Heuristics*, 16(6):881–909, 2010.

[14] J. Maturana and F. Saubion. A compass to guide genetic algorithms. In *Parallel Problem Solving from Nature–PPSN X*, pages 256–265. Springer, 2008.

[15] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming*, pages 204–217. Springer, 2003.

[16] E. Sanchez, M. Schillaci, and G. Squillero. *Evolutionary Optimization: the μGP toolkit*. Springer, 2011.

[17] E. Sanchez, G. Squillero, and A. Tonda. *Industrial applications of evolutionary algorithms*, volume 34. Springer, 2012.

[18] E. Sanchez, G. Squillero, and A. Tonda. *Industrial applications of evolutionary algorithms*, volume 34. Springer, 2012.

[19] M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

[20] A. Tonda, E. Lutton, and G. Squillero. Lamps: A test problem for cooperative coevolution. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2011)*, pages 101–120. Springer, 2011.

[21] J. M. Whitacre, T. Q. Pham, and R. A. Sarker. Use of statistical outlier detection method in adaptive evolutionary algorithms. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1345–1352. ACM, 2006.