Policy Learning in Resource-Constrained Optimization

Richard Allmendinger
University of Manchester
School of Computer Science
Manchester, UK
allmendr@cs.man.ac.uk

Joshua Knowles
University of Manchester
School of Computer Science
Manchester, UK
j.knowles@manchester.ac.uk

ABSTRACT

We consider an optimization scenario in which resources are required in the evaluation process of candidate solutions. The challenge we are focussing on is that certain resources have to be committed to for some period of time whenever they are used by an optimizer. This has the effect that certain solutions may be temporarily non-evaluable during the optimization. Previous analysis revealed that evolutionary algorithms (EAs) can be effective against this resourcing issue when augmented with static strategies for dealing with non-evaluable solutions, such as repairing, waiting, or penalty methods. Moreover, it is possible to select a suitable strategy for resource-constrained problems offline if the resourcing issue is known in advance. In this paper we demonstrate that an EA that uses a reinforcement learning (RL) agent, here Sarsa(λ), to learn offline when to switch between static strategies, can be more effective than any of the static strategies themselves. We also show that learning the same task as the RL agent but online using an adaptive strategy selection method, here D-MAB, is not as effective; nevertheless, online learning is an alternative to static strategies.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods, control theory

General Terms

Algorithms

Keywords

Evolutionary computation, reinforcement learning, bandit algorithms, closed-loop optimization, dynamic optimization

1. INTRODUCTION

We are currently interested in applying evolutionary algorithms (EAs) to optimization problems in which candidate solutions are evaluated by conducting experiments, e.g. physical or biochemical experiments (similar to [14]), including: combinatory drug discovery [6], intrument configuration optimization [12], and quantum control [17]. The particular challenge we are focussing on in this process is that resources required to conduct certain experiments may not be available at each time step during the optimization. This may lead to situations where solutions $\vec{x}$ that are *feasible* candidate solutions to the problem may nevertheless be temporarily *non-evaluable*. Alternatively, we can think of the problems of interest as ones that feature a static fitness function f and feasible search region X but dynamic constraints that restrict the set of solutions evaluable at each time step during the optimization. We call these dynamic constraints *ephemeral resource constraints* (ERCs), and the set of evaluable solutions (or *evaluable search region*) defined by the ERCs at time step t we denote by E_t. Figure 1 illustrates a typical situation in ERC optimization problems (ERCOPs) with respect to the distribution of solutions across E_t and X. The particular ERC type considered here is what we call a *commitment relaxation ERC*. In instrument configuration optimization, for example, this ERC may simulate scenarios where certain instrument settings, once selected, cannot be changed during a working day but only the next day after relaxation of the instrument has taken place at night.

In recent papers [1, 2], we introduced different types of ERCs, studied their effect on evolutionary search, and proposed various (static) constraint-handling strategies — such as repairing, waiting, and penalizing strategies — for use within an EA for dealing with ERCs. An important observation arising from that was that patterns of performance impact seen on the same ERC type, and independent of the fitness landscape, are quite similar; whereas, between the different ERCs they are much more different. This observation is good news as it means that one can perform an offline analysis of ERCOPs, given the common case that the ERCs are known in advance. For all material relating ERCOPs to some other problem types [5, 4, 10], which we cannot cover here, the reader is referred to [1].

Inspired by the above mentioned observation, this paper investigates whether an EA that learns offline when to switch between static strategies during the optimization process is more efficient than the static strategies themselves. Offline learning is performed by a reinforcement learning (RL) agent, and this agent aims at optimizing the average fitness of the final population (also known as the *reward*) by selecting the most suitable static strategy (which serve as the *actions*) in a given state during the optimization.

In the context of evolutionary search, RL has been used

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'11, July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...$10.00.

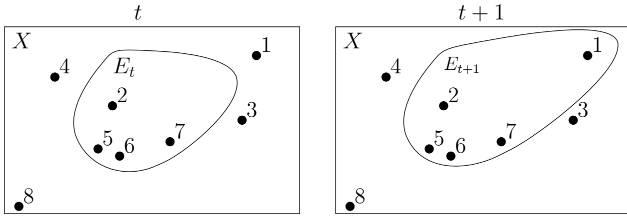

Figure 1: An illustration of how a population consisting of the solutions 1,2,3,4,5,6,7 and 8 might be distributed across the feasible search space X and the evaluable search space E_t. At time step t, only solutions 2,5,6 and 7 can be evaluated while solutions 1,3,4 and 8 must be repaired to be evaluable. Each evaluation might cause a change in E_t.

for both online [8] and offline learning [13]. However, while previous work was mainly concerned with tuning and controlling of EA parameter values and operators, we use RL to control the selection of constraint-handling strategies. Although this is somewhat similar to Zhang and Dietterich's work [20], which studies how to incrementally repair a single infeasible schedule, our work is different mainly in that we are not dealing with a scheduling problem. The fact that we use RL in combination with a population-based optimizer rather than a repair-based scheduler is another difference.

For different *NK* landscapes, we compare the performance of the RL-based EA against each of the static constraint-handling strategies and an EA that learns the same task as the RL agent but online using an adaptive strategy selection method based on the UCB algorithm [3, 7]. The next section describes the commitment relaxation ERC in more detail. The different constraint-handling strategies including the learning-based strategies we consider are then given in Section 3. An experimental study described in Section 4 gives the results of the strategies and also a case study that deals with the issue of selecting a suitable strategy for an ERCOP with partially unknown search space properties. Section 5 concludes the paper.

2. COMMITMENT RELAXATION ERCS

Generally speaking, a commitment relaxation ERC forces or commits an optimizer to a specific variable value combination for some (variable) period of time whenever it uses this particular combination. Consider the motivating example of optimizing the configuration of an instrument, such as a mass spectrometer. Here, a commitment relaxation ERC can simulate the scenario where a particular instrument property, such as the laser frequency of a mass spectrometer, once set to a particular setting H, cannot be changed during the rest of the 'working day' but only the next day after the instrument 'relaxes' during the night. We refer to a 'working day' as an *epoch*, where V denotes its duration, and the activation period $k(j), 0 \leq k(j) \leq V$, to be the duration of the period of time we have to commit to a particular setting H during the jth epoch. Note, the length of the activation period may change with each new epoch depending on when the particular setting H is selected by the optimizer. To describe the setting H we can conveniently use the notion of schemata. For example, assuming a binary representation of solutions of string length $l = 5$, we would

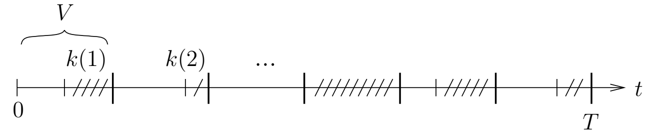

Figure 2: An illustration of how a commitment relaxation ERC may partition the optimization time into epochs of length V, and how it may be potentially activated. The activation period $k(j)$ during the jth epoch is represented by the dashed part.

Algorithm 1 Implementation of a comm. relax. ERC

```
1: commitmentRelaxationERC(x⃗, t){
2: if t − last_activation ≥ k then
3:     if x⃗ ∈ H then
4:         last_activation = t; k = V − t mod V
5:     return true      // x⃗ is evaluable
6: else if x⃗ ∉ H then
7:     return false;    // x⃗ is not evaluable
8: else
9:     return true;}     // x⃗ is evaluable
```

use $H = (*1**0)$ to state that a commitment is associated with the instrument setting for which the value of bit position 2 and 5 is set to 1 and 0, respectively; the $*$ is a wildcard symbol which means that a bit position can have any value. We refer to H as the *constraint schema*.

Figure 2 illustrates the partition of the optimization time into epochs, and a possible distribution of activation periods. From this figure it is apparent that the total number of constraint activations during the optimization can vary between $0 \leq j \leq \lceil T/V \rceil$ (T is the total optimization time). That is, we might be lucky and the ERC may be never activated, e.g. if solutions belonging to H do not lie on an optimizer's search path, but already one activation may introduce enough solutions from H into the population such that future activations might be more likely.

The corresponding implementation of a commitment relaxation ERC is defined by Algorithm 1. The method *commitmentRelaxationERC*$(\vec{x}, t)$ is defined by the parameters V and H, and it takes as input a candidate solution $\vec{x}$ that is to be checked for evaluability and the current (global) time step t. The output is a boolean value indicating whether $\vec{x}$ is evaluable or not. The method maintains two auxiliary variables, *last_activation* and k, required to update the internal state of the constraint: Line 2 to 4 are responsible for the activation of the ERC and the setting of the activation period, while Line 6 ensures that solutions have to be in H; initially we set *last_activation* $= k = 0$.

In future, we will denote a commitment relaxation ERC of this form by *commRelaxERC*(V, H). In the experimental study we consider the case where several commitment relaxation ERCs with different non-overlapping constraint schemata $H_i, i = 1, ..., r$, coexist. In this case, we need to consider three aspects: (i) a solution is non-evaluable if it violates at least one ERC, (ii) a repaired solution has to satisfy all activated ERCs and not only the ones that were violated, and (iii) it needs to be checked whether a repaired solution activates an ERC that was not activated before. A *repaired solution* is one that has undergone repairing in the sense that its genotype has been modified to make it evaluable.

3. CONSTRAINT-HANDLING STRATEGIES FOR ERCOPS

This section introduces five static constraint-handling strategies and two learning-based strategies, which perform learning over the static strategies. The strategies are applicable not only to commitment relaxation ERCs but (in similar form) also to other ERC types. Three of the static strategies (forcing, regenerating, and the subpopulation strategy) apply repairing and two (waiting and penalizing) avoid it in order to prevent drift-like effects in the search direction.

3.1 Static constraint-handling strategies

1. Forcing: This strategy forces a non-evaluable solution $\vec{x}$ into the constraint schemata H_i of all activated ERCs. In other words, all bits that do not match the order-defining bit values of the schemata H_i of all activated ERCs are flipped, and the solution so obtained is returned for evaluation. Repairing strategies of this kind have been used previously, e.g. to solve combinatorial optimization problems (see e.g. [11])

2. Regenerating: Upon encountering a non-evaluable solution, similar to the death penalty approach [10], this strategy iteratively generates new solutions from the empirical distribution of the current offspring population (i.e. it generates new offspring from the current parent set) until it generates one that is evaluable, i.e. falls into the schemata H_i of all activated ERCs, or until L trials have passed without success. In the latter case, we select the solution, generated within the L trials, that has the smallest sum of Hamming distances to the order-defining bits of the schemata H_i of all activated ERCs and apply forcing to it; ties between several equally-closest solutions are broken randomly. Thus the method always returns an evaluable solution.

3. Subpopulation strategy: Let us assume there is only one ERC, i.e. $r = 1$. In this case, alongside the actual population, this strategy maintains also a *subpopulation* SP of maximum size J that contains the fittest solutions from H_1 evaluated so far. A non-evaluable solution is then dealt with by generating a new solution based on this subpopulation. If the maximum population size of SP, J, is not reached, then a new solution from H_1 is generated at random, otherwise we apply one selection and variation step using the same algorithm as the one we augment the constraint-handling strategies on; if the new solution is non-evaluable, which may happen due to mutation, we apply forcing to it. If $r > 1$, then the number of subpopulations we need is upper-bounded by 2^r, the power set of the total number of ERCs. A solution is then generated using the subpopulation that is defined by the (set of) schemata H_i of all activated ERCs.

4. Waiting: This strategy does not repair but it *waits* with the evaluation of a non-evaluable solution and the generation of new solutions until the activation periods of all ERCs that are *violated* by the solution have passed; i.e. the optimization freezes. The freezing period is bridged by submitting as many what we are calling *null solutions* as required until the solution becomes evaluable; null solutions have the effect that the optimizer can 'wait' for a time step without evaluating a solution. Note, in the implementation of this strategy we use here we realize this by setting the (global) time counter directly to the end of the longest activation period of all violated ERCs (see Algorithm 2, Line 19).

5. Penalizing: Like waiting, this strategy does not repair. However, instead of freezing the optimization, a non-evaluable solution is *penalized* by assigning a poor objective value c to it (which is similar to a static penalty function method [10]), and the time counter is incremented as if it had been evaluated. The effects are that evaluable solutions may be accidentally generated during an activation period, and that evaluated solutions coexist with non-evaluated ones in the same population. However, due to selection pressure in parental and environmental selection, non-evaluated solutions are likely to be discarded as time goes by. As we will use the strategy within an elitist EA, and because we use a c that is the minimal fitness in the search space, a non-evaluated solution will never be inserted into a population that is filled with evaluated solutions in the first place.

Algorithm 2 EA with static constraint-handling strategies

Require: ERC_1,...,ERC_r (set of ERCs), f (objective function), T (time limit), μ (parent population size), #Strategy (number of selected static constraint-handling strategy)
1: $t = 0$ (global time counter), $Pop = \emptyset$ (current population)
2: **while** $|Pop| < \mu \wedge t < T$ **do**
3: generate solution $\vec{x}$ at random
4: $\vec{x} = \text{functionWrapper}(\vec{x})$
5: $Pop = Pop \cup \{\vec{x}\}, t{+}{+};$
6: **while** $t < T$ **do**
7: generate one offspring $\vec{x}$ by selecting two parents from Pop, and then recombining and mutating them
8: $\vec{x} = \text{functionWrapper}(\vec{x})$
9: form new Pop by selecting the best μ solutions from the union population $Pop \cup \{\vec{x}\}, t{+}{+};$
10:
11: functionWrapper($\vec{x}, t$){
12: $y_t = \text{null}$
13: **if** $\vec{x}$ satisfies the ERCs ERC_1,...,ERC_r **then**
14: $\vec{x}_t = \vec{x};\ y_t = f(\vec{x}_t)$
15: **else**
16: **if** #Strategy = 1 ∨ #Strategy = 2 ∨ #Strategy = 3 **then**
17: $\vec{x}_t = \text{repair}(\vec{x}, t);\ y_t = f(\vec{x}_t)$
18: **if** #Strategy = 4 **then**
19: $t = t + \tau;\ \vec{x}_t = \vec{x};\ y_t = f(\vec{x}_t)$ // τ is the number of time steps we have to wait until $\vec{x}$ is evaluable
20: **if** #Strategy = 5 **then**
21: $\vec{x}_t = \vec{x};\ y_t = c$ // c is a constant, representing poor fitness
22: return $\vec{x}_t$}

The pseudocode in Algorithm 2 shows how we manage commitment relaxation ERCs and how the different static strategies are integrated into an EA with a $(\mu + 1)$-ES reproduction scheme; this is the same EA as we will use later in the experimental study. Time steps refer here to function evaluations of a single solution, but, alternatively, they may refer, for example, to real time units (e.g. seconds), a calender period (e.g. Tuesday 2-4pm), or something else. Note, the method of Algorithm 1 is called at Line 13 of Algorithm 2.

3.2 Offline learning: RL-based strategy

RL is a computational approach to learning from interaction of an agent with an environment whereby the aim of an RL agent is to learn some optimal policy π, a mapping from an environmental state $s_t \in S$ to an action $a_t \in A(s_t)$, so as to maximize some *discounted return*:

$$R_t = \sum_{j=0}^{\infty} \gamma^j r_{t+j+1}, \tag{1}$$

Table 1: EA parameter settings

Parameter	*Setting*
Parent population size μ	50
Per-bit mutation probability	$1/l$
Crossover probability	0.7

where $r_{t+1} \in \Re$ is a numerical reward received after taking action a_t in state s_t, and γ ($0 \leq \gamma \leq 1$) is the *discount rate*. In learning problems where the agent-environment interaction can be divided into sub-sequences (known as *episodes*), such as single runs of an EA, an agent aims at maximizing the return at the end of each episode. For a comprehensive introduction to the field of RL please refer to [18].

3.2.1 Algorithm

We use Sarsa(λ) [15] as the RL agent or algorithm to learn when to switch between static constraint-handling strategies during an EA run. An episode is a single EA run, and the static strategies described above form the set of actions $A(s)$. In Sarsa(λ) the idea is to learn an *action-value function* $Q(s,a) = E_\pi\{R_t | s_t = s, a_t = a\}$, a prediction of the return obtainable after taking action a when in state s and then following some policy π. This prediction is updated by

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))e_t(s,a), \quad (2)$$

where α is the learning rate. The eligibility trace e is a kind of memory that keeps track of how eligible a previously visited state-action pair is for being updated based on currently obtained rewards. There are various ways eligibility traces can be updated. We use replacing traces, which are updated as follows:

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t, \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise,} \end{cases} \quad (3)$$

where λ is the *decay factor* of the trace. The trace e is set to 0 at the beginning of each episode. The policy according to which we select an action in a state is determined by the ϵ-greedy method. This method takes a random action a with probability ϵ, and with probability $1 - \epsilon$ it takes the greedy action a^* for which $Q(s_t, a^*) = \max_a Q(s_t, a)$. A more detailed description of Sarsa(λ) can be found in [18].

To characterize a state s we use two variables: the current population average fitness and the current time step. The fitness values are normalized so that they lie in the range $[0; 1]$, while the optimization time is limited by T. Each of the two variables is binned into 5 equally-sized intervals, resulting in the intervals $\left[\frac{j}{5}; \frac{j+1}{5}\right]$ and $\left[\frac{T\cdot j}{5}; \frac{T\cdot(j+1)}{5}\right], j = 0, 1, 2, 3, 4$, respectively. The only reward we provide is the average fitness of the population at the end of an episode.

There are some further aspects related to the learning problem and algorithm we want to mention. First, note that a non-evaluable solution may be encountered at different stages of the optimization process or not at all in a single EA run. In other words, the number of states visited in an episode, and the number of non-evaluable solutions encountered in a state, may vary between episodes and states, respectively. In the case where a non-evaluable solution is encountered, the first action selected in a particular state is applied to all non-evaluable solutions encountered in this state. This selection approach tends to perform better than allowing an optimizer to reselect actions when in one and the same state, because it is more direct in terms of credit assignment. Finally, to encourage exploration we use the idea of optimistic initial values [16]. We initialize all entries of $Q(s,a)$ with value 1.0.

3.3 Online learning: UCB-based strategy

To learn online when to switch between different static constraint-handling strategies, we use the dynamic multi-armed bandit (D-MAB) algorithm [7]. The idea of this adaptive strategy selection method is to consider the learning problem as a multi-armed bandit problem with the static strategies serving as independent arms. D-MAB extends the *upper confidence bound 1* (UCB1) algorithm [3] with the statistical Page-Hinkley test, which has the purpose to detect changes in the sequence of rewards obtained, and then to restart the multi-armed bandit. The reader is referred to the paper (ibid.) for a more detailed description of D-MAB.

D-MAB requires that the play of an arm is followed by a subsequent reward. We provide a reward immediately after the play of an arm, and it is the (normalized) raw fitness of the resulting solution. Note that if the arm associated with the strategy, *waiting*, is played, then the reward may be available only a few time steps later. Also, in the case of *penalizing*, the reward will always be some poor fitness value c. However, this does not mean that penalizing is never selected because (i) UCB1 maintains always a certain degree of exploration, and (ii) a restart of the multi-armed bandit triggered by the Page-Hinkley test puts all arms in the same initial position. Alternative credit assignment schemes and UCB algorithms were tested but the combination used in this paper tends to perform, on average, best and most robustly on the test problems considered.

4. EXPERIMENTAL ANALYSIS

4.1 Experimental setup

We augment the constraint-handling strategies on a genetic algorithm with a $(\mu + 1)$ES reproduction scheme (see Algorithm 2 of Section 3); this choice is guided by simplicity but accounts for our beliefs that elitism is generally useful in this domain. The algorithm also uses binary tournament selection (with replacement) for parental selection, uniform crossover [19], and bit flip mutation. The parameter settings of the EA and the constraint-handling strategies are given in Table 1 and 2, respectively. The parameters $\lambda_{\rm PH}$ and δ involved in D-MAB are related to the Page-Hinkley test, while the scaling factor C controls the balance between the exploration of currently poor (or unused) constraint-handling strategies and the exploitation of the currently best strategies; we tested different settings for these parameters but the values given in Table 2 tend to perform, on average, best and most robustly on the test problems considered. For the RL-based strategy, denoted in Table 2 by RL-EA, we use a training and testing scheme (similar to [13]). In the training phase, the RL agent estimates the function $Q(s,a)$, while, in the testing phase, the Q-function is frozen and the greedy actions a^* are always selected. As specified in Table 2, the testing phase involves 100 episodes or EA runs, and this is also the number of runs for which we run the other constraint-handling strategies; i.e. any results shown are average results across 100 EA runs. To allow for a fair

Table 2: Parameter settings of constraint-handling strategies

Strategy	*Parameter*	*Setting*
Regenerating	Number of regeneration trials L	100000
Penalizing	Fitness c assigned to non-evaluable solutions	0
Subpopulation strategy	Maximal size of all subpopulations SP_h, J_h	25
RL-EA	Decay factor λ	1.0
	Discount rate γ	1.0
	Learning rate α	0.1
	Probability ϵ of selecting a random action	0.1
	#Training episodes	5000
	#Testing episodes	100
D-MAB	Threshold parameter λ_{PH}	0.1
	Tolerance parameter δ	0.01
	Scaling factor C	1

comparison of the strategies, we use a different seed for the random number generator for each EA run but the same seeds for all strategies. We also use a different seed for each episode of the training phase of RL-EA.

4.2 Experimental study

We treat this experimental study as a case study to demonstrate both the performance of the different constraint-handling strategies and one way in which a suitable strategy for an ERCOP may be selected in a real-world application. The case study is based on a simplified version of a real-world optimization problem, involving the configuration of an instrument (similar to [12]). Variables represent detector voltage, temperature ramp, laser, and other settings of a mass spectrometer. The fitness of a configuration is determined by running a sample through it, and measuring properties of the instrument's output signal. ERCs arise in this problem because certain instrument setting combinations once selected for an experiment (an evaluation) can only be changed the next day, after relaxation has taken place at night.

A configuration is represented by a binary string of length $l = 30$. Available are $T = 2000$ time steps for the optimization, and the ERCs, which are known *a priori*, are two commitment relaxation ERCs: $commRelaxERC(V = 20, H = (10101***...))$ and $commRelaxERC(V = 20, H = (*...**101))$. Little is known of the fitness landscape before optimization begins. However, as in [12], it would be expected that there is some degree of epistasis in the problem. We would not know whether the two schemata represent good or poor instrument configurations.

As algorithm designers, we are now faced with the challenge to select an optimization algorithm or constraint-handling strategy for the above described ERCOP. The common approach is to first design appropriate problem functions that simulate the problem at hand, and then to test several algorithms on these functions and use the best one for the real-world problem.

In terms of test functions f, we know from previous analysis that the type of resourcing issue is largely responsible for the effect on the performance, with similar effects observable for different fitness landscapes. Nevertheless, selecting or generating a fitness landscape that mimics the problem at hand more accurately increases the confidence in the selected constraint-handling strategy. In this case study, we know about the fitness landscape that there is a certain degree of epistasis and that multiple optima might exist. Hence, we should be considering test functions that feature different degrees of epistasis. Suitable test functions are, for example, NK landscapes [9], which involve two tunable parameters: the total number of bits N, and the number of bits that interact epistatically at each of the N loci, K. The parameter K allows us to control conveniently the degree of epistasis, whereby larger values of K represent more epistasis. We consider four different settings for K, namely, $K = 1, 2, 3,$ and 4, while N is fixed by the instrument setup to $N = 30$; these K values cover reasonable degrees of epistasis.

Figure 3 shows the population average fitness obtained with the different constraint-handling strategies on the four NK landscapes as a function of the time counter (we do not show the standard error as it was negligible); the average fitness of the best solution in a population is in alignment with the population average fitness. From the plots we can see that the ERCs affect the performance of an EA. Also, the plots confirm what we mentioned before that similar patterns are obtained for all the test functions. In fact, with respect to the static strategies, we observe a trend that the subpopulation strategy performs best in the initial stages of the optimization, forcing and penalizing in the middle part of the optimization, and waiting in the final stages of the optimization. Also, the time step at which waiting outperforms penalizing shifts further to the right as the degree of epistasis increases.

For RL-EA, the results in Figure 3 were obtained using a training phase involving 5000 different NK landscapes with $N = 30$ and $K = 2$. After that training phase, the same frozen Q-function was used in the testing phase of all four NK landscape types. Consequently, this allows us to assess the robustness of the learnt control policy as training and testing are done in environments with different properties (at least for the cases $K \neq 2$). From the figure we can see that RL-EA is the best performing strategy on all four NK landscape types at $T = 2000$. For a low level of epistasis, RL-EA is even able to match almost the performance of an EA optimizing in an ERC-free environment. We observe also that the performance advantage over waiting, the second best strategy, tends to increase with the degree of epistasis; when comparing the two strategies against each other using more than 100 algorithmic runs, then, according to the Kruskal-Wallis test (significance level of 5%), RL-EA is also significantly better than waiting for $K = 3$ and $K = 4$. This aspect indicates the robustness of RL-EA. On the other hand, although D-MAB is not able to perform as well as RL-EA, it is able to match the performance of waiting for $N = 30, K = 4$. Another benefit of offline learning is that if the optimization time would be shorter, say around $T = 500$, then RL-EA would learn a different control policy, while D-MAB does not. That is, RL-EA adapts to the real-world problem at hand; we have confirmed this experimentally (results not shown).

Figure 4 shows the greedy action a^* in each state s learnt by the RL agent. From the plot it is apparent that the agent learnt to use mainly waiting at the beginning of the optimization process, penalizing in the middle part of the optimization, and, depending on the population average fitness, either forcing, waiting, or the subpopulation strategy,

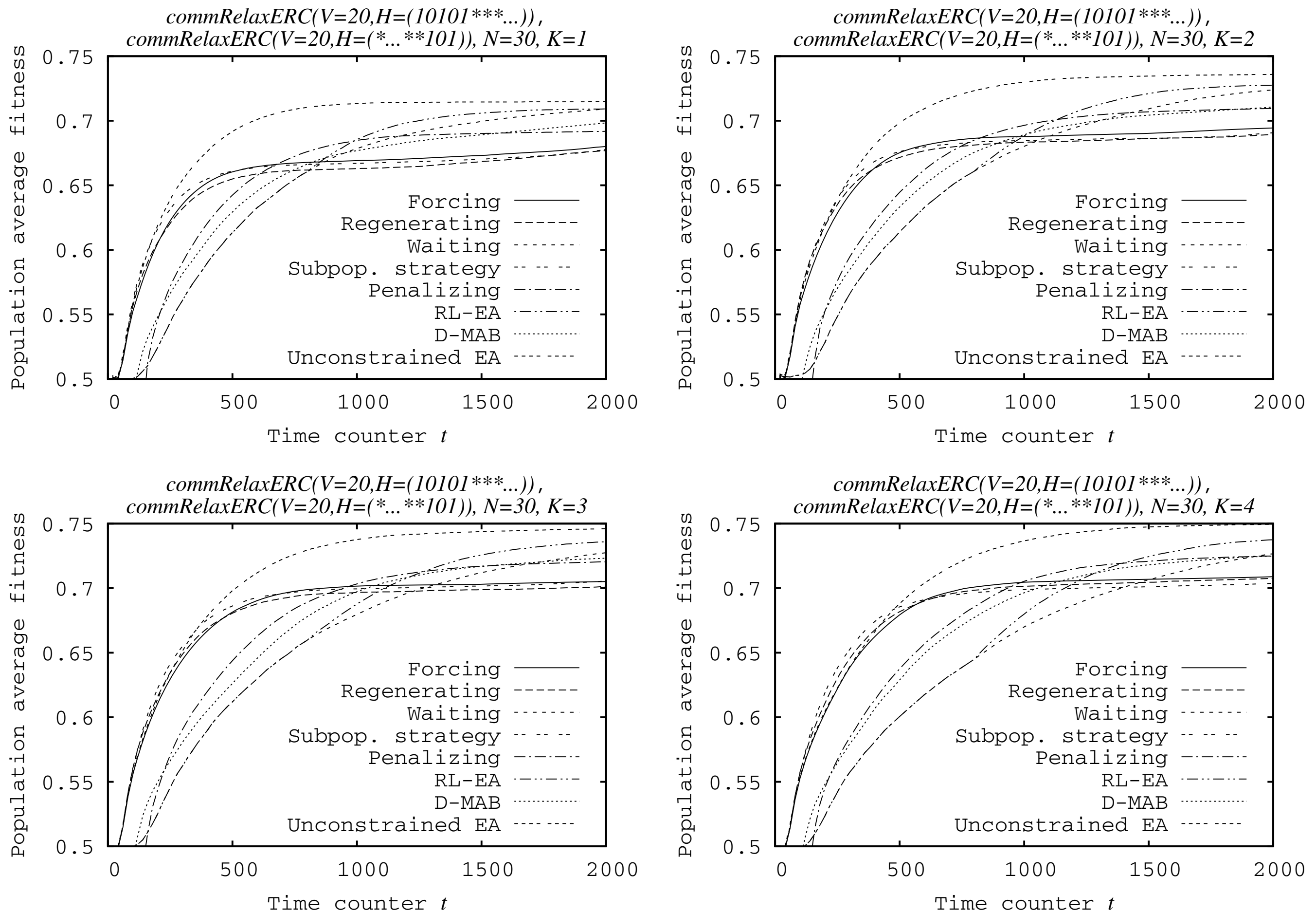

Figure 3: Plots showing the population average fitness obtained by the different constraint-handling strategies on NK landscapes with $N = 30$ and $K = 1$ (top left), $K = 2$ (top right), $K = 3$ (bottom left), and $K = 4$ (bottom right) as a function of the time counter t; results are averaged over 100 independent runs using a different randomly generated NK problem instance for each run. All instances were subject to the commitment relaxation ERCs $commRelaxERC(20, H = (10101 * * * ...))$ and $commRelaxERC(20, H = (*... * *101))$. The results of 'Unconstrained EA' were obtained by running the EA on the same problem instances but without the ERCs.

in the final part of the optimization. From Figure 3 one may conclude that a policy that uses a repairing strategy (forcing, regenerating, or subpopulation strategy) at the beginning of the optimization, and waiting or penalizing towards the end, should also perform well. The agent did not learn this policy because the repairing strategies may lead quickly to a homogeneous population containing many solutions that fall into both or either of the constraint schemata. If the schemata are poor, then one should escape from this population state but this is difficult as diversity needs to be again introduced into the population and this takes too long using waiting or penalizing.

Figure 5 illustrates what strategies are used on average by D-MAB during different periods of the optimization process of $N = 30, K = 4$. From the plot we can see that penalizing is selected least often, which is due to the low fixed reward associated when playing the associated arm. A trend is obvious that waiting is used less often than the repairing strategies at the end of the optimization process, which is in alignment with the policy learnt by the RL agent. The reason that the performance of D-MAB is nevertheless poorer than the one of RL-EA is that the repairing strategies are used too often throughout the optimization process but in particular at the beginning of the optimization. As mentioned before, this may result in a homogeneous population state from which it is difficult to escape; the fact that the total number of plays increases towards the end of the optimization confirms that D-MAB actually ends up in this poor population state. On NK instances with less epistasis, the Page-Hinkley test is triggered less often, and waiting is used almost as often as the repairing strategies throughout the optimization.

Overall, the strong performance of the RL-EA is encouraging, but we want to mention that in order to achieve that performance, some tuning of the agent may be required. This is due to two aspects. First, the stochastic nature of an EA in the sense that: (i) taking the same action in a particular state but in different episodes may cause an EA to end up in different future states, and (ii) visiting the same state-action pairs in different episode may result in different rewards obtained at the end of an episode. Second, since a different problem instance is used in each training episode,

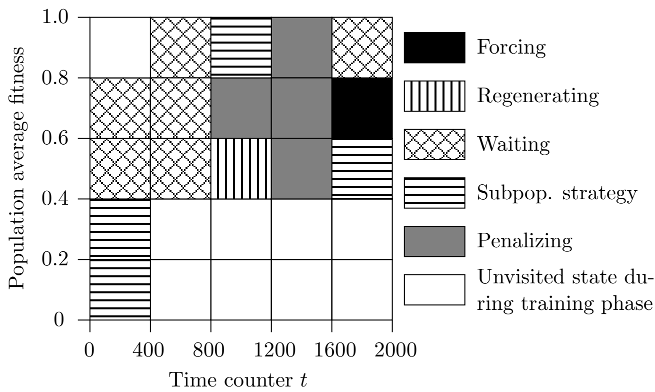

Figure 4: A plot showing the greedy actions a^* learnt by the RL agent for each state s. Training was done on NK landscapes with $N = 30$ and $K = 2$.

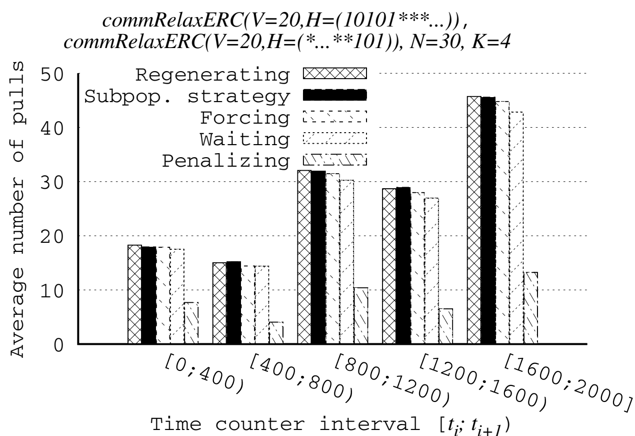

Figure 5: A plot showing the average number of times each strategy is played by D-MAB at different periods of the optimization process.

the constraint schemata of the two ERCs may represent both good and poor instrument configurations. Hence, as the quality of a schema has an impact on what strategy is most appropriate, the current optimal control policy learnt by the RL agent may change constantly during training. The approach taken here to deal with these two issues was to train the agent for different numbers of training episodes using also different settings of parameters involved in the agent algorithm. The parameter setting combination that performed, on average, best and most robustly on the training or validation problems is used in this paper. Figure 6 illustrates how the population average fitness may be affected by the number of training episodes, and different settings of the decay factor λ and the discount rate γ.

Overall, based on the results shown, we would select RL-EA for the real-world instrument optimization problem as it performs best after 2000 time steps. Let us assume that the real-world problem is a NK landscape instance with $N = 30$ and $K = 3$. Hence, to verify our selection, we perform one run with all strategies on a single newly generated NK landscape instance with $N = 30$ and $K = 3$; the results are shown in Figure 7, and note that RL-EA uses the control policy of Figure 4. The plot confirms the findings made

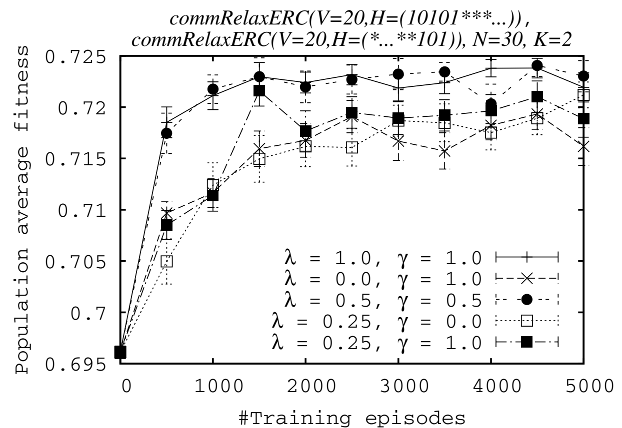

Figure 6: A plot showing the population average fitness obtained by RL-EA for different values of the decay factor λ and the discount rate γ as a function of the number of training episodes. The population average fitness values themselves represent average values across 30 learning trials; each trial used different randomly generated problem instances and random number generator seeds for training and testing, but the same instances and seeds for any combination of λ and γ values. Training and testing was done on NK landscapes with $N = 30$ and $K = 2$

on the test functions. In particular, RL-EA, D-MAB, and waiting, perform best at the end of the run while the static repairing strategies perform best at the beginning of the optimization. Clearly, the result on a single instance might still be different from the result we obtained from averaging over many instances, even though the same type of test function is used for both experiments. Nevertheless, this case study demonstrates how one can approach and solve an ERCOP beginning with the definition of the ERCs, modelling the simulated environment, selection of appropriate test functions, and finally tuning and comparing different optimizers and selecting the most suitable one to be used in real world.

5. CONCLUSION

In this paper we have considered an optimization scenario in which resources are required in the evaluation process of candidate solutions. The particular challenge we focussed on was that certain resources had to be committed to or used for some period of time whenever they have been used by the optimizer; we referred to this type of resourcing issue as a commitment relaxation ERC. Several strategies to deal with this ERC have been proposed including static strategies, such as repairing, waiting, and penalizing strategies, and two learning-based strategies. The learning-based strategies aim at learning when to switch between the static strategies during the optimization process. However, while one strategy learns this task offline using a reinforcement learning (RL) agent, here Sarsa(λ), the other strategy performs online learning using the UCB algorithm extended with the statistical Page-Hinkley test to detect changes in the sequence of rewards obtained. Offline learning is possible in this optimization scenario because the performance of an EA depends largely on the type of ERC, with similar effects observable for different fitness landscapes.

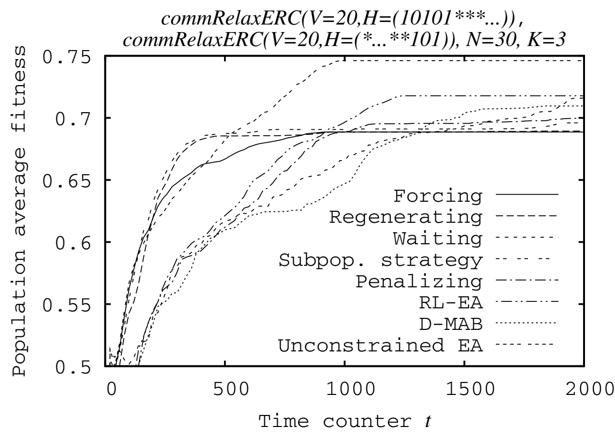

Figure 7: A plot showing the population average fitness obtained in a single run on a randomly generated *NK* landscape with $N = 30$ and $K = 3$ as a function of the time counter t; the instance was subject to the two commitment relaxation ERCs $commRelaxERC(20, H = (10101 *** ...))$ and $commRelaxERC(20, H = (*... **101))$.

We analyzed the performance of the constraint-handling strategies and demonstrated how one may approach and solve a new ERC optimization problem (ERCOP) in the common case where knowledge of the fitness landscape is poor. The analysis concluded that ERCs affect the performance of an EA but an RL-based strategy is able to get close to the performance of an EA optimizing in an ERC-free environment, particularly on fitness landscapes with little epistasis. The online learning algorithm did not perform as well as the RL-based algorithm, mainly because it does not look ahead in the optimization process and so may end up quickly in a poor population state from which it is difficult to escape; this is a general issue of online learning within an ERCOP scenario and it is yet unclear how to avoid it. Static repairing strategies are well suited if little optimization time is available, while a static waiting or penalizing strategy is more suited for longer optimization times.

Our study has of course been very limited, and there remains much else to learn about the effects of ERCs and how to handle them. Our current research is looking at the design and tuning of RL agents for dynamic optimization problems that also account for the stochasticity present in evolutionary search. Analyzing constraint-handling and learning strategies on different and perhaps more realistic fitness landscapes than *NK* landscapes is another avenue we are pursuing.

6. REFERENCES

[1] R. Allmendinger and J. Knowles. Ephemeral resource constraints in optimization and their effects on evolutionary search. Technical Report MLO-20042010, University of Manchester, 2010.

[2] R. Allmendinger and J. Knowles. On-line purchasing strategies for an evolutionary algorithm performing resource-constrained optimization. In *Proceedings of PPSN*, pages 161–170, 2010.

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

[4] P. A. N. Bosman and H. L. Poutré. Learning and anticipation in online dynamic optimization with evolutionary algorithms: the stochastic case. In *Proceedings of GECCO*, pages 1165–1172, 2007.

[5] J. Branke. *Evolutionary Optimization in Dynamic Environments*. Kluwer Academic Publishers, 2001.

[6] D. Calzolari, S. Bruschi, L. Coquin, J. Schofield, J. D. Feala, J. C. Reed, A. D. McCulloch, and G. Paternostro. Search algorithms as a framework for the optimization of drug combinations. *PLoS Computational Biology*, 4(12):e1000249, 2008.

[7] L. Da Costa, A. Fialho, M. Schoenauer, and M. Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of GECCO*, pages 913–920, 2008.

[8] A. E. Eiben, M. Horvath, W. Kowalczyk, and M. Schut. Reinforcement learning for online control of evolutionary algorithms. In *Engineering Self-Organising Systems*, volume 4335 of *Lecture Notes in Computer Science*, pages 151–160, 2007.

[9] S. Kauffman. Adaptation on rugged fitness landscapes. In *Lecture Notes in the Sciences of Complexity*, pages 527–618, 1989.

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

[11] R. Nakano. Conventional genetic algorithm for job shop problems. In *Proceedings of ICGA*, pages 474–479, 1991.

[12] S. O'Hagan, W. B. Dunn, M. Brown, J. Knowles, and D. B. Kell. Closed-loop, multiobjective optimization of analytical instrumentation: gas chromatography / time-of-flight mass spectrometry of the metabolomes of human serum and of yeast fermentations. *Analytical Chemistry*, 77(1):290–303, 2005.

[13] J. E. Pettinger and R. M. Everson. Controlling genetic algorithms with reinforcement learning. Technical report, University of Exeter, 2003.

[14] I. Rechenberg. Case studies in evolutionary experimentation and computation. *Computer Methods in Applied Mechanics and Engineering*, 2-4(186):125–140, 2000.

[15] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[16] J. Schmidhuber. Curious model-building control systems. In *Proceedings of IJCNN*, pages 1458–1463, 1991.

[17] O. M. Shir, M. Emmerich, T. Bäck, and M. J. J. Vrakking. The application of evolutionary multi-criteria optimization to dynamic molecular alignment. In *Proceedings of CEC*, pages 4108–4115, 2007.

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

[19] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of ICGA*, pages 2–9, 1989.

[20] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of IJCAI*, pages 1114–1120, 1995.