Evolutionary Intelligence manuscript No.
(will be inserted by the editor)

Exploratory Analysis of an On-line Evolutionary Algorithm in Simulated Robots

Evert Haasdijk · S.K. Smit · A.E. Eiben

Received: 16-02-2012 / Accepted: 19-09-2012

Abstract In traditional evolutionary robotics, robot controllers are evolved in a separate design phase preceding actual deployment; we call this off-line evolution. Alternatively, robot controllers can evolve while the robots perform their proper tasks, during the actual operational phase; we call this on-line evolution. In this paper we describe three principal categories of on-line evolution for developing robot controllers (encapsulated, distributed, and hybrid), present an evolutionary algorithm belonging to the first category (the $(\mu + 1)$ ON-LINE algorithm), and perform an extensive study of its behaviour. In particular, we use the Bonesa parameter tuning method to explore its parameter space. This delivers near-optimal settings for our algorithm in a number of tasks and, even more importantly, it offers profound insights into the impact of our algorithm's parameters and features. Our experimental analysis of $(\mu + 1)$ ON-LINE shows that it seems preferable to try many alternative solutions and spend little effort on refining possibly faulty assessments; that there is no single combination of parameters that performs well on all problem instances and that the most influential parameter of this algorithm – and therefore the prime candidate for a control scheme – is the evaluation length τ.

Keywords evolutionary robotics · on-line evolution · scientific testing · parameter tuning · Bonesa

Parts of this work were made possible by the European Union FET Proactive Initiative: Pervasive Adaptation funding the SYMBRION project under grant agreement 216342

Evert Haasdijk[1] · S.K. Smit[1,2] · A.E. Eiben[1]
[1] Vrije Universiteit Amsterdam
[2] TNO, The Hague
E-mail: e.haasdijk@vu.nl, selmar.smit@tno.nl, a.e.eiben@vu.nl

1 Introduction

Imagine a collective of autonomous robots that find themselves in a dynamic environment that they (or rather, their designers) didn't know to expect. Obviously, they cannot rely on pre-defined behaviour determined by fixed controllers to cope robustly with the unknown challenges in such a scenario. Rather, the robots have to learn to cope with their environment and any changes in it, just as they have to learn to react to changes in their own bodies, for instance as a result of hardware failure or reconfiguration. An essential capability, therefore, of such robots is the ability to adapt their controllers – to learn – in the face of challenges they encounter in a hands-free manner, without supervision – human or otherwise.

Our research is inspired by the vision of robots one day being able to adapt like this as so eloquently articulated by Nelson, Barlow, and Doitsidis [29]:

> "Advanced autonomous robots may someday be required to negotiate environments and situations that their designers had not anticipated. The future designers of these robots may not have adequate expertise to provide appropriate control algorithms in the case that an unforeseen situation is encountered in a remote environment in which a robot cannot be accessed. It is not always practical or even possible to define every aspect of an autonomous robot's environment, or to give a tractable dynamical systems-level description of the task the robot is to perform. The robot must have the ability to learn control without human supervision."

To provide such levels of autonomous adaptivity through evolution, an evolutionary algorithm must run on-board, without any external master computer to execute evolutionary operators or evaluations. Crucially, the robots' con-

trollers change on the fly, as they go about their proper tasks: adaptation occurs *on-line*, during the operational period of the robots.

The overwhelming majority of evolutionary robotics research to date, however, involves *off-line* adaptation during the development stage prior to the operational period. Additionally, the evolutionary algorithm that develops the robot controllers actually doesn't run on the robots themselves at all, but it runs on a separate computer that centrally maintains the population of robot controllers (or rather, their genetic encodings) and performs selection and genetic operators. The robots themselves – be it real or simulated – only come into play to evaluate candidate controllers. Once the controllers are deployed on the actual robots genuinely to perform their tasks, the evolutionary process is terminated and the controllers are modified no more – at least not through evolution. Nolfi and Floreano [30] provide ample illustration of this approach to evolutionary robotics. Without detracting from the impressive work with this approach to controller evolution, it cannot provide the on-line adaptive capabilities that we envisage.

Essentially, the off-line approach to evolutionary robotics is no different from other fields of evolutionary computation as we have known them since the 1960s — Evolution Strategies, Genetic Algorithms, Genetic Programming, Differential Evolution, to name but a few. Traditional evolutionary robotics and mainstream evolutionary algorithms share the centralised management of evolutionary operators, where would-be parents do not select mates and produce offspring in autonomously, but are being selected by an 'oracle' (the outer loop of the main evolutionary algorithm) and undergo variation operators passively.

For the on-line adaptive capabilities without supervision as we envisage, this is not an appropriate scheme. For that, the adaptation mechanism – the evolutionary algorithm, in our case – must operate in a decentralised, autonomous fashion. We distinguish three ways in which the evolutionary algorithm can be implemented on the robots themselves without recourse to some external overseer that orchestrates evolution or assesses fitness:

Encapsulated In this case, the complete evolutionary algorithm runs within the robot itself. In effect, the robot orchestrates its own evolution without referral to other robots or external assessments of its performance;

Distributed In systems with multiple robots, each robots contains only a single individual. Robots select mates from their neighbours and autonomously exchange genetic material to generate new candidate solutions that they test themselves;

Hybrid The two previous approaches can be combined: robots now maintain a self-sufficient evolutionary algorithm on-board, but they also exchange genetic material with their neighbours – like islands with migration in the well-known island model of parallel evolutionary algorithms.

The subject of this study is an algorithm that embraces the first approach: the $(\mu + 1)$ ON-LINE algorithm (introduced in detail in Sec. 3) encapsulates an evolutionary algorithm within each individual robot, thus enabling it to evolve its controller on-the-fly, independently from the others.

Rather than simply testing the algorithm and reporting on the outcomes, we adopt the scientific testing approach advocated by Hooker [19]. This approach aims at "learning about your algorithm", rather than "testing your algorithm" and emphasises the analysis of robustness and parameter interactions. These different aspects and their relevance to evolutionary robotics are discussed in Sec. 4. In our analysis of $(\mu + 1)$ ON-LINE we use an automated parameter tuning method called Bonesa that is specifically designed for this purpose. Its basic concepts are explained in Sec. 5.

Through scientific testing, we want to investigate which of $(\mu + 1)$ ON-LINE's parameters have the most profound impact on the robots' performance in a variety of tasks, which parameter settings work well in what kind of task and which settings (if any) work well overall.

The purpose of our tuning exercise is not simply to find the best parameter vector for a particular task and environment, but it is to learn about the algorithm's applicability in a range of circumstances, to identify parameters that require problem-specific tuning and to gain insights into algorithm behaviour that will allow for proper deployment in real hardware with either well-established, tuned, parameter values or with (as yet to be developed and validated) parameter control schemes.

Such a thorough exploration of the algorithm's parameter space requires a very large number – literally thousands – of experiments. For reasons of time, but also because of expected hardware failures due to wear and tear in so many runs, it is not possible to perform these experiments with real robots, and we have to turn to simulated trials for our analysis. Obviously, we cannot expect the performance of real robots to be the same as that of simulated robots (the reality gap), or even that parameter settings found to be optimal will be the best settings when deploying $(\mu + 1)$ ON-LINE in real robots. However, based on our extensive experience with evolutionary computing, we expect that such simulations do give good indications about algorithm behaviour such as sensitivity to parameter settings and identification of important parameters, allowing for focussed, more tractable numbers of experiments with real robots in subsequent research.

2 On-line, On-board Evolutionary Robotics

In their seminal paper, Watson, Ficici, and Pollack [40] coin the phrase *Embodied Evolution* for a system where "a population of physical robots [...] autonomously reproduce with one another while situated in their task environment." In other words, the controllers evolve on-line ("in their task environment") through the physical robots exchanging genetic material with one another. This groundbreaking example of the distributed approach introduces a fully autonomous scheme where the robots broadcast (mutated) genes on local-range communication channels at a rate proportional to their fitness (Probabilistic Gene Transfer Algorithm, PGTA). Also, robots resist 'infection' with genes broadcast by other proportionally to their fitness. Robots incorporate received genes into their own genome and so immediately update the controller and continually evaluate its performance. In this scheme, there is no central authority or global view of the population, but there is, in fact, one item of global information: the maximum possible fitness which is used to determine the broadcast and resistance rates. This seems to limit the applicability of PGTA, but it would seem reasonably straightforward to propagate at least the best achieved fitness through a gossiping-like algorithm as described in Jelasity, Montresor, and Babaoglu [20] and so avoid the need for a pre-specified maximum fitness. Nehmzow [28], Simões and Dimond [35], Wischmann, Stamm, and Wörgötter [41] describe similar distributed implementations of on-line evolutionary robotics.

Before focussing on encapsulated implementations, let us first note that evolutionary algorithms that embrace the encapsulated approach do not seem materially different from 'regular' evolutionary algorithms because the whole process runs on a single robot, just as it would run inside a single computer. There are, however, particularities of on-line evolution as we will see in Sec 3. One practicality of on-line evolution (and of the encapsulated variant in particular) is *time-sharing*; obviously, only a single controller can be active at any one time on a single robot. Therefore, the encapsulated algorithm can only evaluate a single individual (which defines a single controller) at a time. To evaluate multiple individuals – as any evolutionary process must – the algorithm has to try them in sequence, one after the other: a controller runs for a certain amount of time and the task performance over that period determines the controller's fitness. Then, the next controller is loaded and gets its chance to control the robot while fitness is measured, and so on. As a consequence, there can be no guarantee that two individuals are evaluated in the same or even similar circumstances: a controller's evaluation starts where the previous left off, and there is no re-initialisation, repositioning or other mechanism to ensure similar circumstances of evaluation. As Nordin and Banzhaf [31] note in their description of one of the earliest implementations on-line evolution of robot controllers:

> "Each individual is thus tested against a different real-time situation leading to a unique fitness case. This results in 'unfair' comparison where individuals have to navigate in situations with very different possible outcomes. However, our experiments show that over time averaging tendencies of this learning method will even out the random effects of probabilistic sampling and a set of good solutions will survive."

Floreano, Schoeni, Caprari, and Blynel [12] implement an encapsulated algorithm that evolves spiking neural networks for obstacle avoidance. This implementation illustrates two characteristics that are common to almost all encapsulated and hybrid implementations of on-line, on-board evolutionary algorithms. Firstly, the algorithm implements steady-state evolution: an individual is tested and possibly replaces an individual in the current population so that parents and offspring may exist side-by-side. In this case, the new individual replaces the worst in the population if it performs better. Secondly, the population on a single robot (consisting of only 6 individuals) is tiny compared to what is common in evolutionary computation; after all, it has to fit inside the robot's on-board memory.

Walker, Garrett, and Wilson [39] show that, in addition to providing hands-free adaptivity, on-line, on-board evolution can help overcome the 'reality gap' where a simulator irons out the wrinkles and warts of reality. Often, when robot controllers are developed using a simulator (which is typically cheaper and faster than using real robots), the controllers in reality perform worse than might be expected due to infidelities in the simulation and particularities of individual hardware [see 7]. Walker et al add a second stage to mainstream evolutionary robotics to tackle this issue. They first employ an off-line, centralised evolutionary algorithm to develop controllers in simulation (the 'training phase'). They then transfer the resulting controllers onto a real robot, and implement an on-line, on-board evolutionary algorithm that further refines the controller and adapts it to a changing environment. In their experiments, the robot has to avoid randomly placed obstacles while moving towards a goal. The environment changes by moving the obstacles and by varying the number if obstacles in the arena. The dynamics of the environment highlight the hands-free adaptivity that on-line, on-board evolution enables. Walker et al's on-line algorithm has a single champion genome that serves as parent to a challenger; the challenger replaces its parent if it proves to perform better. To this fairly common arrangement they add a buffer memory: a second child can replace the challenger (but not the parent directly) if it outperforms it. Every iteration, the second child is replaced with a newly mutated version of the parent, so the first child is the current

best challenger. This two-tiered approach was designed to overcome the problem mentioned above: as a result of sequential evaluation, the circumstances of evaluation could differ significantly from one individual to the next and 'a poor chromosome could perform uncharacteristically well and be rewarded and vice-versa.' The population of the on-line algorithm is – again – tiny: only three individuals are considered, in all. Walker et al mention that this carries the benefit of promoting rapid adaptation to changes in the environment: 'the smaller the population size, the faster each generation of chromosomes is evaluated and the sooner the effects of evolution manifest.'

Haroun Mahdavi and Bentley [16] describe experiments involving encapsulated on-line evolution in robots that use shape memory alloy actuators: metal filaments that change shape when a small current is applied. In one set of experiments, Haroun Mahdavi and Bentley use on-line evolution to let a snake-shaped robot learn to move forward. These experiments don't actually exhibit the level of autonomy suggested by on-line, on-board evolution because the fitness was not measured by the robot itself but by the experimenters (and therefore evolution took place, at least in part, off-board), but they, nonetheless, provide a striking example of one of the benefits of on-line adaptation: during one experiment, one of the filaments broke while the robot was evolving a gait. Normally, this would spell disaster and mean that the robot must be fixed and the experiment restarted. Haroun Mahdavi and Bentley, however, realised that this was an excellent opportunity to see the controller adapt to the new circumstances and continued the experiment with one broken actuator. In short order, the robot adapted its gait to move with one less actuator, showing how on-line evolution can help robots cope with hardware failure. Similar robustness under on-line, on-board adaptation was reported by Christensen, Spröwitz, and Ijspeert [8], who purposely introduced hardware faults – for instance, failing modules in a modular robot body – during experiments with an on-line, on-board stochastic adaptation method akin to Newton-Raphson minimisation methods.

3 The $(\mu+1)$ ON-LINE Evolutionary Algorithm

The $(\mu+1)$ ON-LINE algorithm as described by Haasdijk, Eiben, and Karafotias [13] and Bredeche, Haasdijk, and Eiben [6] was devised specifically to provide autonomous adaptation through on-line, on-board evolution. It is an encapsulated evolutionary algorithm: a robot maintains a self-sufficient evolutionary process on-board, without recourse to other robots or to some central overseer. It is inspired by evolution strategies [1, 34]; these provide a likely basis for $(\mu+1)$ ON-LINE because the the robot controllers' parameters in our experiments can be represented by a vector of real-valued numbers and evolution strategies have a very good reputation as evolutionary solvers of numerical optimisation problems [1]. Other encapsulated algorithms for on-line adaptation that share this basis are described by Haroun Mahdavi and Bentley [16], Nehmzow [28], Walker et al [39].

The main features of $(\mu+1)$ ON-LINE algorithm are the following: 1) it relies on a population of μ individuals that produce one child per generation, 2) it uses a time-sharing scheme to evaluate the fitness of individuals (robot controllers), 3) it allows for re-evaluation to reduce noise, 4) it can self-adapt the mutation parameters and the length of the evaluation period. The exact details are given in the pseudo code in Algorithm 1. The main design decisions and the underlying rationale are discussed in the following subsections.

3.1 Evolutionary operators

In typical applications of evolutionary algorithms, the overall success is determined by the best individual at termination. The quality of the other individuals considered during evolution is of no importance as they will be discarded when the champion is deployed. Things are very different for on-line evolution as we envisage here: because the controllers evolve as the robots go about their tasks the real performance of the robots is determined by the quality of *all* individuals (controllers) they evaluate. While a robot evaluates poor controllers, that robot's *actual* performance will be inadequate, no matter how good the best known individuals as archived in the population.

In evolution strategies, it is common to describe algorithms using a pair μ,λ where μ indicates the size of the population and λ indicates the number of offspring generated at each iteration. The '$+$' indicates that the algorithm uses an elitist replacement strategy where new individuals are only introduced into the population if they outperform current members of the population. Thus, $(\mu+1)$ ON-LINE generates $\lambda=1$ child per cycle, which then may replace the worst in the population if it achieves higher fitness. Normally in evolution strategies $\frac{\lambda}{\mu}$ is usually between 4 and 8, but the cost of evaluating poor individuals (about which more below) is so high that we choose to set $\lambda=1$. To promote rapid convergence we diverge from the common practice of uniform random parent selection in evolution strategies and use binary tournament parent selection, increasing the selective pressure. Also, we use recombination to promote convergence. Thus, each new individual is created from two parents, each of which is selected with a binary tournament. When the representation allows (it does in all the tests we conducted), we apply the evolution strategy standard of gaussian mutation with self-adaptation of the mutation step-sizes. We consider various self-adaptation schemes as described in Sec. 6.

3.2 Re-evaluation to combat noise

In on-line evolution, fitness must be evaluated *in vivo*: the quality of any given controller can only be determined by actually giving that controller the run of the robot and see how well the robot performs its tasks. Whatever the details of the evolutionary mechanism, different controllers will be evaluated under different circumstances: any controller's evaluation will start wherever and in whatever state the previous evaluation left the robot. The very dissimilar evaluation conditions caused by one –possibly very poor– individual setting the scene for the evaluation of another individual result in very noisy ("unfair" in the words of Nordin and Banzhaf [31]) fitness assessments.

To level the playing field among genomes, $(\mu+1)$ ON-LINE re-evaluates genomes in the population with a given probability ρ. This means that every evolutionary cycle one of two things happens: either a new individual is generated and evaluated (with probability $1-\rho$), or an existing individual is re-evaluated (with probability ρ). To ensure that re-evaluation efforts are spent more or less on those individuals that actually become parents, the individual to be re-evaluated is chosen by a binary tournament from the whole population. The fitness values from subsequent (re-)evaluations of any given individual are combined using an exponential moving average as advocated by Bredeche et al [6]; this emphasises newer performance measurements and so is expected to promote adaptivity in changing environments. Bredeche et al also show that using an exponential moving average outperforms alternative averaging strategies.

Re-evaluating in this manner is similar to a resampling strategy to deal with noisy fitness evaluations as advocated by Beyer [4], although the moving average introduces a temporal aspect and it makes convergence to the true fitness less likely (although, in a dynamic environment, that would be problematic in any case).

3.3 Racing to shorten fitness evaluations

An often heard criticism of stochastic search methods in general and evolutionary algorithms in particular is that they are computationally inefficient because they spend so much time evaluating poor candidate solutions. This is especially painful in the context of on-line evolution, because the actual performance of the evolving system (the robot's controller, in this case) drops when these sub-optimal solutions are evaluated.

To minimise the amount of time spent trying less than promising individuals, Haasdijk, ul Qayyum, and Eiben [14] suggested adding *racing* to $(\mu+1)$ ON-LINE. This entails that during a new individual's evaluation, intermediate results are compared to the fitnesses of individuals already in the population to estimate the likelihood that the challenger is good enough to beat the worst individual in the population and replace it. If it is fairly certain that this challenger is going to turn out worse than the worst in the current population, further evaluation is most likely a waste of time with an elitist replacement scheme such as $(\mu+1)$ ON-LINE uses. What exactly 'fairly certain' means is determined by two parameters α and β as described below. At intermediate time steps during a challenger's evaluation, its intermediate fitness $F_{challenger}$ is compared to a lower bound which depends on the current worst fitness in the population F_{worst}. If the performance drops below this lower bound, the evaluation is aborted and a new iteration of the algorithm commences, otherwise, evaluation continues—at least until the next comparison. To estimate the likelihood that the challenger is at least as good as the current worst in the population, we use a modified version of the Hoeffding inequality [18]:

$$F_{challenger} \geq F_{worst} - 2\xi(t) \tag{1}$$

with $\xi(t)$ calculated as follows:

$$\xi(t) = \sqrt{\frac{(F_a - F_b)^2 log(2/\alpha)}{\beta t}} \tag{2}$$

with F_a and F_b the best and the worst fitness values of the population, respectively; α is the significance level of the comparison. The β parameter, introduced by Haasdijk et al [14], allows more robust tuning of the comparison's strictness than the original bounds, where it has a fixed value of 2. Using formulas 1 and 2, an individual's evaluation has a large likelihood of continuing in the early stages of evaluation (high values of ξ lower the bar), but the pressure increases as time passes.

Putting all this together – and assuming real-valued vectors as genotypes representing robot controllers – we obtain an evolutionary algorithm with the following main components and parameters:

4 Scientific Testing

For many years, so called *horse-race-papers* [22] have dominated the field of evolutionary computing. Researchers showed how their algorithm outperformed some other algorithm on some test-suite. These papers shed light on the question 'whether a certain algorithm instance can outperform another algorithm instance on a particular problem'. They do not, however, indicate when or why this is the case, nor do they give an indication of an algorithm's performance on other problems.

Hart and Belew [17] noted that the fact that an algorithm performs well on a certain problem is no guarantee that it also works on a different one. Even worse, the fact that an algorithm with a particular set of parameter values works

Algorithm 1: The $(\mu + 1)$ ON-LINE evolutionary algorithm.

```
for i ← 1 to μ do                                                    // Initialisation
    population[i] ← CreateRandomGenome();
    population[i].σ ← σ_initial;
    population[i].Fitness ← RunAndEvaluate(population[i]);
end for
Sort(population);
for ever do                                                          // Continuous adaptation
    if random() < ρ then                                  // Don't create offspring, but re-evaluate
        Evaluatee ← BinaryTournament(population);
        Evaluatee.Fitness ← (Evaluatee.Fitness + RunAndEvaluate(Evaluatee)) / 2;    // Combine
        re-evaluation results
    else                                                  // Create offspring and evaluate that as challenger
        Challenger ← BinaryTournament(population);
        if random() < P_c then
            ParentB ← BinaryTournament(population - Challenger);
            AveragingCrossover(Challenger, ParentB);
        end if
        Mutate(Challenger);                    // Also updates σs depending on adaptation scheme
        a ← population[1].Fitness;
        b ← population[μ].Fitness;
        for n ← 1 to τ do                      // Racing: monitor evaluation, abort if challenger fails
            Challenger.Fitness ← RunAndEvaluateForOneTimeStep(Challenger);
            ξ ← sqrt((a−b)^2 log(2/α) / βt) ;
            if Challenger.Fitness < population[μ].Fitness - 2ξ then
                abort evaluation;
            end if
        end for
        if Challenger.Fitness > population[μ].Fitness then          // Replace last (i.e. worst) individual
            population[μ] ← Challenger;
            population[μ].Fitness ← Challenger.Fitness;
        end if
    end if
    Sort(population);
end for
```

Evolutionary Algorithm Components	
Representation	Real valued vectors
Mutation	Adding Gaussian noise and self-adapting σ's
Crossover	Averaging parents
Parent selection	Binary tournament
Survivor selection	Replace worst on improvement
Evolutionary Algorithm Parameters	
Population size	μ
Crossover rate	p_c
Evaluation period	τ
Re-evaluation rate	ρ
Significance level for racing	α
Strictness level for racing	β

Table 1 Main components and parameters of $(\mu + 1)$ ON-LINE

well on a certain problem gives no indication of its performance on any other function. This is especially painful because common parameter settings – for instance, a population size of 100 – tend to be much less robust than assumed. In practice, parameter values are mostly selected by conventions (mutation rate should be low), ad hoc choices (let's use uniform crossover), and experimental comparisons on a limited scale (testing combinations of three different crossover rates and three different mutation rates) rather than based on solid foundations. Until recently, there were not many workable alternatives for such manual tuning.

However, developments over the last couple of years have resulted in a number of automated tuning methods and corresponding software packages that enable evolutionary computation practitioners to perform a more detailed analysis of the algorithm, and the appropriate parameter values, without much effort.

Performance is the most straightforward and most commonly used measure of algorithm quality. Because performance varies markedly with parameter settings, with the problem type and even with the inherent stochasticity of

evolutionary algorithms, it is often better to consider the robustness of algorithms.

There are different interpretations of the notion of robustness in the literature. The existing (informal) definitions do have a common feature: robustness is related to the variance of algorithm performance across some dimension, but they differ in what this dimension is. Below, we take a closer look at different type of robustness considered in literature.

Problem Sensitivity In the simplest case, we test an evolutionary algorithm A on a single problem f. Then, the performance of an instance of A (with a specific parameter vector) can be measured as the performance of A on f. It might be the case that A is very good at solving f, however, there can be no claims or indications regarding its performance on other problem instances. This can be a satisfactory result if one is only interested in solving f. However, algorithm designers in general, and evolutionary computing experts in particular, are often interested in (instances of) evolutionary algorithms that work well on many different problems. In such cases, tests need to be performed on a test suite consisting of many different problems $f_1, \ldots, f_n$.

Parameter Influence Another popular interpretation of algorithm robustness is related to performance variations caused by different parameter settings. Most algorithms are sensitive to parameter settings, which means that the parameters need to be set carefully. An ill-chosen parameter value may cause a huge drop in performance. Such behaviour is often undesirable, especially if the algorithm is also sensitive to changes of the problem. In that case, a lot of effort is needed to identify the correct parameter values for each situation. It is then important to know how sensitive the algorithm is to each of the parameters, in order to focus the search for good settings. If, on the other hand, the algorithm is quite robust against changes in the problem space, a sensitivity in the parameter space may not be a big issue. The only requirement is that such an algorithm is carefully tuned beforehand to identify the "sweet spot" for parameter values; these settings will then work well for a large range of problems.

Variance across different random seeds Evolutionary algorithms are inherently stochastic. Therefore, all experimental investigations should be statistically sound, requiring a number of independent repetitions of a run with the same setup, but with different random seeds. This yields information about the third kind of robustness. To what extent an algorithm's performance varies from one instance to the next, varying only the random seed, of course greatly influences its applicability.

5 Bonesa

Bonesa[37] is an iterative model-based procedure to search noisy response surfaces where evaluations can be very expensive. Like other search methods using surrogate models [11, 21], it is based on an intertwined searching and learning loop. Bonesa has been developed for automated parameter tuning of (evolutionary) algorithms, based on ideas behind REVAC [27] and Sequential Parameter Optimisation [3].

The pseudo code in Algorithm 2 describes the initialization, the two loops and their interaction. First, M random parameter vectors are generated and tested to obtain an initial model of the landscape for each of the problems at hand. Then, the search and learning loops start.

The search loop is a generate-and-test procedure that iteratively generates K new parameter vectors, pre-assesses their quality using the information gained in the learning loop, and finally tests the most promising vector by executing a run with these specific parameter values. In its turn, the learning loop uses the information obtained about the quality of the tested parameter vector and all previously tested ones to develop a model of the performance surfaces. These performance surface models can then be used to pre-assess the K generated parameter vectors in the next search loop. Notice the two-way interaction between the two loops: information generated by the search loop is used to update the model, while estimations based on the model direct the search.

Pre-assessing the vectors with the model rather than blindly testing them greatly reduces the computational costs of tuning parameters. However, establishing the performance of the parameter vectors that do pass the model-based test is still computationally expensive. Namely, the result of a single run is often not representative due to the stochasticity of the evolutionary algorithm. Therefore, several expensive runs of the evolutionary algorithm have to be executed in order to establish the real performance of a certain parameter vector; this still causes high tuning costs.

To reduce the noise caused by the stochasticity with the least computational costs, Bonesa uses a kernel filter. This is a common method of reducing noise in spatial data that does not depend on repetitive testing of the same point but uses proximity data [5, 15] to smooth out the noise. This allows Bonesa to test every parameter vector only once and still produce statistically sound results.

Compared to other currently known iterative model-based search procedures, Bonesa distinguishes itself by its multi-objective approach. Rather than optimising an algorithm's parameters for a single problem or on a weighted sum of performance values [33, 36], Bonesa uses the Pareto-strength approach from SPEA2 by Zitzler, Laumanns, and Thiele [42] to optimise parameters for a whole range of problems in one go. Therefore, one is ultimately able to se-

Algorithm 2: The Bonesa algorithm.

```
for i ← 1 to M do                                                    // Initialisation
    archive[i] ← CreateRandomParameterVectors();
    archive[i].RawQualities[] ← RunAndEvaluate(archive[i], problems);
end for
i ← M ;
while maximum budget not reached do                    // Intertwined Searching and Learning
    i ← i + 1;
    archive.PredictedQualities[] ← KernelFilterMean(archive, problems);
    archive.PredictedVariances[] ← KernelFilterVariance(archive, problems);
    for j ← 1 to K do                                                // Generate Candidates
        candidates[j] ← DrawFromArchiveDensity(archive) ;
        candidates[j].PredictedQualities[] ← KernelFilterMean(candidates[j], problems);
        candidates[j].PredictedVariances[] ← KernelFilterVariance(candidates[j], problems);
        candidates[j].PredictedParetoRank ← CalculateParetoRank(candidates[j], archive);
    end for
    Sort(candidates, candidates.PredictedParetoRank);
    archive[i] ← candidates[1] ;
    archive[i].RawQualities[] ← RunAndEvaluate(archive[i], problems);
end while
for j ← 1 to i do                                                // Create termination set
    archive[j].ParetoRank ← CalculateParetoRank(archive[j], archive) ;
    if archive[j].ParetoRank == 0 then
        terminationset ← { terminationset, archive[j] };
    end if
end for
```

lect not only the best parameter vector for a single problem, but also for a class of problems. In order to determine dominance, one important change has been made with respect to SPEA2: since the distributions of performance need to be compared rather than a single value, the calculation of dominance is based on significance testing. A certain parameter vector $\overline{z}$ dominates another parameter vector $\overline{y}$ if and only if its performance is significantly better (using a Welch-T test) on at least one problem, and not significantly worse on the others. This means that, also in case of single-problem tuning, the result is a set of 'best' vectors rather than a single one.

For a more detailed description of the Bonesa algorithm and toolbox we refer to Smit and Eiben [37].

The most important feature of model-based tuners such as Bonesa is that rather than delivering only the 'perfect' parameter settings for each (set of) problem(s), they can analyse the complete spectrum of performance, robustness and parameter interactions, allowing for much deeper insights into the algorithm and its behaviour.

6 Experimental Set-up

We use Bonesa to optimise the following $(\mu + 1)$ ON-LINE parameters:

- α The significance level of the comparison for racing (see Sec. 3.3). α can range from 0 to 1, with 0 disabling racing altogether.
- β The second parameter that regulates the strictness of the racing comparisons. This can range from 0.5 to 2.0.
- σ Or, rather, the choice of σ adaptation scheme. In $(\mu + 1)$ ON-LINE, new individuals are mutated using a gaussian mutation with step-size σ. We consider a number of ways to update σ: two 'standard' self-adaptive schemes, where either a single σ for the whole genome or multiple σs, one for each real-valued gene in the genome, evolve as additional part of the genome as described by Eiben and Smith [9, pp. 75–77]. The third scheme updates σ using the derandomised approach proposed by Ostermeier, Gawelczyk, and Hansen [32], who specifically deem this method useful with small populations. In the result plots, these schemes are identified as 'ss', 'ms' and 'dr', respectively.
- ρ The probability that an existing individual is re-evaluated rather than that a new individual is generated and tested. ρ can range from 0 to 0.6.
- τ The length of an episode during which a controller is given the run of the robot to evaluate it. To (re-)evaluate an individual, the genome is expressed in the controller (the weights of the neural net are set to those specified in the genome), which then determines the robot's actions over a certain number – τ – of discrete time-steps, unless evaluation is aborted by the racing procedure. τ can range from 100 to 1200.

P_c The crossover rate. The likelihood of performing recombination of two parents when producing a new individual. Ranges from 0 to 1.

The population size μ has been fixed at a value of 10 in these experiments. There are two reasons not to include μ as a tunable parameter: firstly, μ cannot be very large: in the kind of robotic hardware we consider, memory is at a premium, so there is simply no room for large populations. Moreover, a large population would necessitate a longer episode of testing the initial population, which is not desirable in the kind of on-line adaptation we envisage because it would mean very poor performance for a fairly long period with all its attendant risks. Secondly, we have seen that, within these limitations, μ seems not to be a very influential parameter [13].

6.1 Four Tasks

To understand how $(\mu+1)$ ON-LINE behaves across a range of problems, we test it on four tasks, all commonly found in evolutionary robotics literature: phototaxis, fast forward (or obstacle avoidance), collective patrolling and locomotion.

Even with Bonesa's optimisations to reduce the number of tests, this type of tuning is not feasible with real robots: it still requires hundreds of experiments to generate a reliable surrogate model. Therefore, we conducted our experiments in simulation using the Webots simulator[1]. Note that we do not advocate tuning as extensively as this for every deployment of an evolutionary algorithm, but we use tuning to help us understand algorithm behaviour. This understanding will allow for more focussed tuning for specific deployment or guide the development of parameter control algorithms to adapt parameters on-line.

In the phototaxis, fast-forward and patrolling experiments, the robots are simulated e-pucks, controlled by simple perceptron neural networks and the evolutionary algorithms determine the weights of the connections between the neurons. The perceptrons use a *tanh* activation function and receive inputs from light, distance and pheromone sensors (depending on the task) and have two output neurons that drive the wheels. All inputs are normalised in the $[0,1]$ interval before being fed to the neurons. The outputs governing the e-puck wheel speeds are interpreted as fraction of the full speed for the motors. In each of these experiments, we simulate 10 robots in a single arena. In the phototaxis task, the robots do not interact in any way. In the fast-forward task, they pose additional, moving, obstacles for each other. The patrolling experiment adds communication between robots by spreading pheromones.[2]

The locomotion experiment is based on that described by Christensen et al [8]; it simulates 5 roombot modules that are linked to form a quadruped shape. This task requires the most intricate level of collaboration: the modules are physically linked and have to synchronise their movements to obtain an effective gait. In this experiment, the controller is not a neural network, but a central pattern generator.

In all cases, a single trial runs for 10,000 seconds of simulated time; we use time rather than number of evaluations or generations because we are interested in the performance of the robots in real time, regardless of how that is achieved by the evolutionary algorithm. As mentioned, τ is measured in discrete time steps; in fact, these time steps are defined by the simulator calling the robots controller at periodic intervals. The length of a single time step (in simulated time) is 50 ms for the the phototaxis, fast-forward and patrolling experiments. The more intricate physics of the locomotion simulations require a lower settings of 16 ms. Note that the settings for τ, α and β influence the number of evaluations performed per timespan: lower values of τ obviously increase the number of evaluations per timespan, just as increasing the strictness of racing comparisons (by increasing α and β). We are primarily interested in the actual performance of robots, not in the performance of the best individual in the population at any given time.

Actual performance is measured as the average performance during a time-span, irrespective of how many controllers may have been activate during that time. In these experiments we measure actual performance over the last 6,000 seconds of simulated time. For the phototaxis, fast-forward and patrolling experiments, this yields ten measurements for each trial (one per robot) as input for Bonesa. For the locomotion experiment, only a single value is reported because the distance travelled by the compound robot already combines the information for all constituent modules. The exact fitness functions are obviously task dependent and are described with each task, below.

Bonesa can be used for both single-problem tuning and multi-problem tuning. When doing multi-problem tuning, it uses the Pareto-strength approach from SPEA2[42] to optimize on a whole range of different problems in one go. Since such a method terminates with a whole set of parameter-vectors, an experimenter can choose which of those suits his needs best. For example, certain parameter vectors could be well-suited for solving a subset of problems, rather than only a single problem. Most of these "silver bullets", would not have been found with a single-problem approach, since they are often outperformed by a true "specialist" on the prob-

[1] `http://www.cyberbotics.com/`

[2] Source code for the algorithm as well as Webots configuration files for the experiments described here is available at `http://www.few.vu.nl/~ehaasdi/papers/MuPlusOne-2012`

lem. Often, "silver bullets" are much more interesting than true specialist because they lead to much more robust algorithm behaviour. Finally, such a multi-problem approach can be used to identify similar problems. If we assume that problems that can be solved with the same parameter-values can be regarded as "similar", we can try to determine if there are any characteristics that they have in common. Such knowledge can then be reused when facing new problems.

Phototaxis Seeking out or tracking a light source is a very straightforward task that has been addressed by many researchers in evolutionary robotics. The task is frequently combined with other tasks such as goal homing [38] and flocking [2]. In our comparison, we use the simplest version of phototaxis: robots only have to move towards a stationary light source and then remain as close to it as possible. In the phototaxis task, the robots use eight light sensors to detect light intensity and base their behaviour on that. The fitness function simply rewards intensity of received light:

$$f = \sum_{t=0}^{\tau} \max_{i=1}^{8}(lightSensor_i) \tag{3}$$

where $lightSensor_i$ is the normalised input from a light sensor between 0 (no light) and 1 (brightest light).

The arena is an empty (apart from the ten robots) square with a light source in the middle: we ignore collisions between robots in these experiments, so the robots' distance sensors can be ignored.

Experiment details	
Task	phototaxis
Robot group size	10
Simulation length	10,000 seconds (simulation time)
Controller details	
Controller	Perceptron neural net
Input nodes	8 light sensors + bias
Output nodes	2 (left and right motor values)
Evolution details	
Representation	real valued vectors with $-4 \leq x_i \leq 4$
Chromosome length	18

Table 2 Phototaxis set-up

Fast Forward Moving in as straight line as possible as fast as possible while avoiding obstacles – also known as obstacle avoidance – is maybe the most commonly tackled task in evolutionary robotics research. In a confined environment with obstacles this task implies a trade-off between avoiding obstacles and maintaining speed and forward movement. The fitness function we use is based on one described by Nolfi and Floreano [30]; it favours robots that are fast and go straight ahead:

$$f = \sum_{t=0}^{\tau}(v_{trans} \cdot (1 - v_{rot}) \cdot (1 - d)) \tag{4}$$

where v_{trans} and v_{rot} are the translational and the rotational speed, respectively. v_{trans} is normalised between -1 (full speed reverse) and 1 (full speed forward), v_{rot} between 0 (movement in a straight line) and 1 (maximum rotation); d indicates the distance to the nearest obstacle and is normalised between 0 (no obstacle in sight) and 1 (touching an obstacle).

Intermediate results sum the reward from $t = 0$ to the current time step; they can then be compared with complete evaluation results after simply dividing both results by the number of time steps used to calculate them.

Although fast forward is considered a trivial task, we added some extra difficulty by using a complicated maze-like arena (Figure 1(a)) with tight corners and narrow corridors that fit only a single robot and sometimes lead to dead ends. This arena structure, combined with the fact that multiple robots will be simultaneously deployed, makes the task considerably harder than most commonly seen instances. This additional complexity is confirmed by results of baseline trials in this arena that Karafotias, Haasdijk, and Eiben [24] reported: the baseline Braitenberg controllers invariably get stuck after a while.

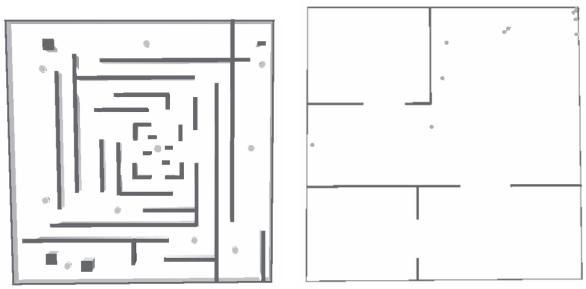
Experiment details	
Task	fast forward
Robot group size	10
Simulation length	10,000 seconds (simulation time)
Controller details	
Controller	Perceptron neural net
Input nodes	8 distance sensors + bias
Output nodes	2 (left and right motor values)
Evolution details	
Representation	real valued vectors with $-4 \leq x_i \leq 4$
Chromosome length	18

Table 3 Fast-forward set-up

Collective Patrolling An obvious real-world task for a group of autonomous mobile robots is that of a distributed sensory network, where the robots have to patrol an area as a group and detect events that occur periodically. It differs from the previous tasks since it requires some level of coordination: the success of the group depends not only on the efficient movement of the individual robots but also on the spread of the group across the arena to maximise the

probability of detecting events. Somehow, robots need to liaise so as not to patrol the same areas. To this end, they are equipped with a pheromone system: robots continuously drop pheromones (this is a fixed behaviour and not controlled by the evolved controller) while sensors detect the local pheromone levels. The collective patrolling task is described by Martinoli [26] where controllers evolve off-line, although in that work the approach to events is more complicated and the robots use other sensory inputs.

The experiments for this task take place in the arena shown in Figure 1(b).



(a) Fast forward arena (b) Patrolling arena

Fig. 1 Arenas for two tasks; the circles represent the robots to scale.

Pheromones are simulated as follows: the $5m \times 5m$ arena is divided into 500×500 cells, each with a pheromone level between $[0,2]$. Every second, each robots drops 1 unit of pheromones at the cell the robot is currently in, and a linearly decreasing amount in nearby cells up to a range of $R_p = 0.07m$. Pheromone levels decay over time at a rate of $R_c = -0.024/s$ in each cell.

As sensory input to the controller, 4 pheromone sensors are placed at the periphery of the circular body at $\frac{\pi}{4}$, $\frac{3\pi}{4}$, $\frac{5\pi}{4}$ and $\frac{7\pi}{4}$. Each sensor detects the accumulated pheromone levels of all cells in a range of $0.05m$; detected levels decrease linearly with distance from the sensor. For fitness calculation only, a similar sensor is positioned in the centre of the robot.

The fitness function penalises pheromones presence (detected by the central pheromone sensor) and proximity to obstacles:

$$f = \sum_{t=0}^{\tau} ((1-p) \cdot (1-d)) \quad (5)$$

where p indicates pheromone presence between 0 (no pheromones) and 1 (strongest pheromone level) at the current location and d indicates the distance to the nearest obstacle and is normalised between 0 (no obstacle in sight) and 1 (touching an obstacle).

Although movement is not explicitly rewarded, it should emerge due to the continuous dropping of pheromones and the deleterious effect of staying in a place where the robot just dropped them. Just as in the fast forward scenario, intermediate results sum the reward from $t = 0$ to the current time step; they can then be compared with complete evaluation results after simply dividing both results by the number of time steps used to calculate them.

Experiment details	
Task	collective patrolling
Robot group size	10
Simulation length	10,000 seconds (simulation time)
Controller details	
Controller	Perceptron neural net
Input nodes	8 distance sensors + 4 pheromone sensors + bias
Output nodes	2 (left and right motor values)
Evolution details	
Representation	real valued vectors with $-4 \leq x_i \leq 4$
Chromosome length	26
Mutation	Gaussian $N(0,\sigma)$
Crossover	averaging

Table 4 Patrolling set-up

Locomotion The locomotion task differs from the preceding three: it concerns individual robots that are physically linked together to form an *organism* that to all intents and purposes looks like a single entity. In fact, five Roombots robots form a four-legged organism as shown in Fig. 2. Another difference is that the controllers are not neural networks but oscillators that operate the roombots' three degrees of freedom: rotating the two halves of each sphere and rotating the con-

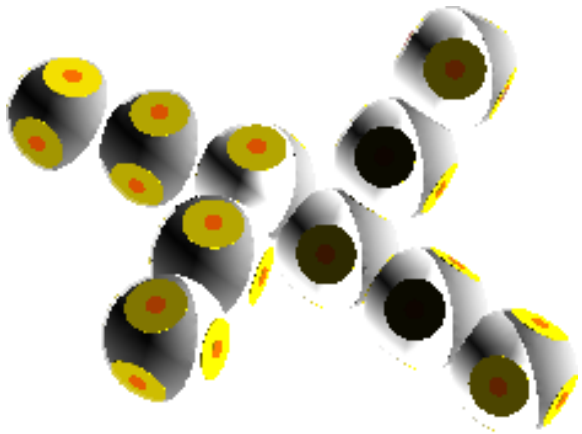


Fig. 2 Organism for the locomotion task. Each of the five consituent modules consists of two spheres. Each module has three degrees of freedom: it can rotate the two (dark and light) halves of each sphere and it can rotate the connection between the spheres.

nection between the two spheres. Each module's controller runs with only local communication.

These oscillators are parameterised by a common base frequency and, for each of the three actuators, a phase shift, amplitude and offset that determine the actual movement: ten real values in total. Our choice for the genetic representation of these parameters is obvious: a vector of these ten real values. Each module runs its own, independent, $(\mu+1)$ ON-LINE instance to optimise the oscillation parameters. This implies co-evolution between the modules' controllers: they must evolve to some common ground to synchronise their oscillations.

The fitness function is very straightforward: it simply sums the Euclidean distance travelled per time-step. As before, intermediate results can be compared with complete evaluation results by simply dividing both results by the number of time steps used to obtain them.

This experiment was first described by Christensen et al [8], who used a non-evolutionary stochastic approximation method, SPSA, to implement on-line adaptation. We refer to that publication for a more detailed description of the robots and the coupled oscillator controller.

Experiment details	
Task	locomotion
Robot group size	5
Simulation length	10,000 seconds (simulation time)
Controller details	
Controller	Central Pattern Generator
Oscillator parameters	phase shift, amplitude, offset for each actuator, one common base frequency
Evolution details	
Representation	real valued vectors with $0 \leq x_i \leq 1$
Chromosome length	10

Table 5 Locomotion set-up

7 Results

We review the experimental results in the light of three aspects of $(\mu+1)$ ON-LINE's parameters: how problem-specific are good parameter settings, which parameters influence task performance most and how do the parameters interact?

7.1 Generalism vs Specialism

The object of this paper is not to prove that tuning leads to good results – that has been amply illustrated by, for instance, Eiben and Smit [10]. Nevertheless, we do look at the performance of the tuned instances of $(\mu+1)$ ON-LINE to compare the results of tuning for a specific task with those of the multi-objective approach. This provides insight into the performance penalty of using generalist parameter settings (the result of multi-problem tuning) as opposed to specialist settings that result from tuning specifically for each problem.

From Fig. 3, we can see that for three of the four tasks – fast forward, patrolling and locomotion – multi-objective tuning carries no performance penalty. In fact, the locomotion performance of the multi-objective runs is often better than that of tuning specifically for locomotion. For phototaxis, however, the performance is slightly worse when tuning for all these four tasks at once. This seems to indicate that phototaxis requires parameter settings that conflict with the other tasks, something we will see more proof of later. We also see that parameter vectors that work well for the fast forward task also work well on locomotion and/or patrolling, but that the converse is not necessarily true: the highest performance for patrolling is achieved by parameter vectors with indifferent performance on the fast forward task.

7.2 Parameter Tolerance

An important insight from tuning exercises like these is what parameters matter most for algorithm performance; for an analysis of this parameter tolerance, we turn to Table 6. It shows, per problem, the variance of the best performance over the range of each parameter. For each of the three σ-strategies, for example, the best performance achieved with that specific strategy is noted for each problem; the variance of these performances is divided by the difference between the best and worst achieved performance on that problem and then listed. Numerical parameters were binned into 100 intervals prior to this calculation. Large values therefore indicate a parameter that causes large fluctuation in performance – that has a large impact. Using the terminology introduced by Eiben and Smit [10], this is a measure for both the tunability and the tolerance of a parameter.

It is clear that τ is the most sensitive parameter, since for all of the problems it has the highest or second-highest variance. This indicates that it needs to be set very carefully because it has the biggest influence on the performance. ρ comes in second place, and P_c on the third place. The σ-adaptation strategy, interestingly, has least effect on performance, except in the case of patrolling.

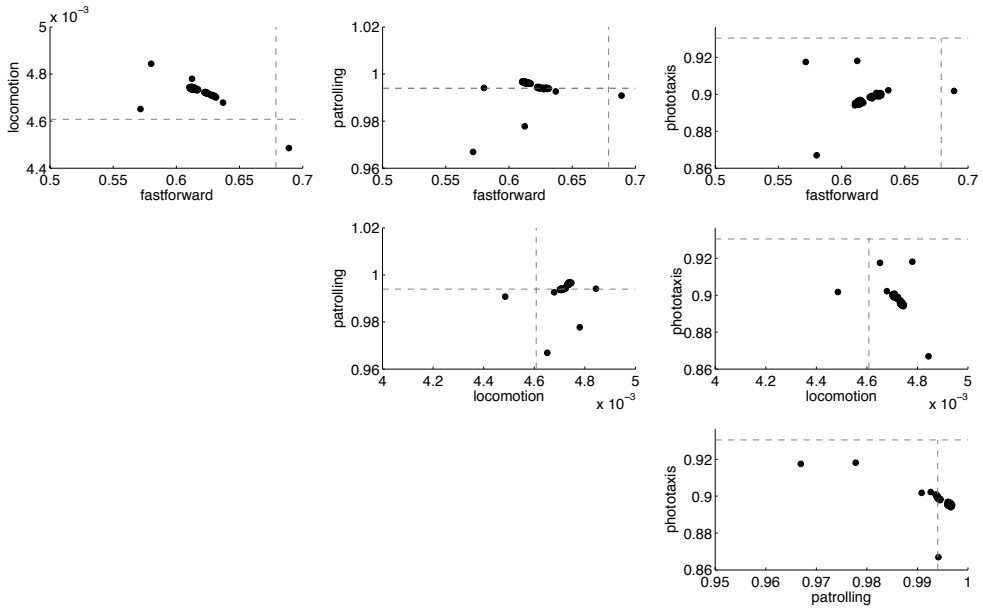


Fig. 3 2-dimensional projections of the Pareto-front. Each plot shows the performance of all non-dominated parameter vectors from the multi-objective run on two tasks. The horizontal and vertical dotted lines indicate the maximum performance reached when tuning for that specific task only. Thus, points on or above/to the right of dotted lines indicate parameter vectors found with multi-objective tuning that match or outperform the best parameter vector found with task-specific tuning.

task	τ	σ	ρ	P_c	α	β
fastforward	**0.0262110**	0.0000177	0.0168660	<u>0.0253330</u>	0.0002089	0.0001982
phototaxis	<u>0.0000622</u>	0.0000010	**0.0002482**	0.0001162	0.0000287	0.0000371
patrolling	**0.0029074**	0.0000082	<u>0.0027625</u>	0.0000057	0.0000076	0.0000088
locomotion	**0.0000463**	0.0000009	<u>0.0000069</u>	0.0000025	0.0000025	0.0000029

Table 6 The variance of the best performance across possible values for each of the parameters. The parameter with the highest variance for each problem is shown in bold, the second highest is underlined

7.3 Parameter Interaction

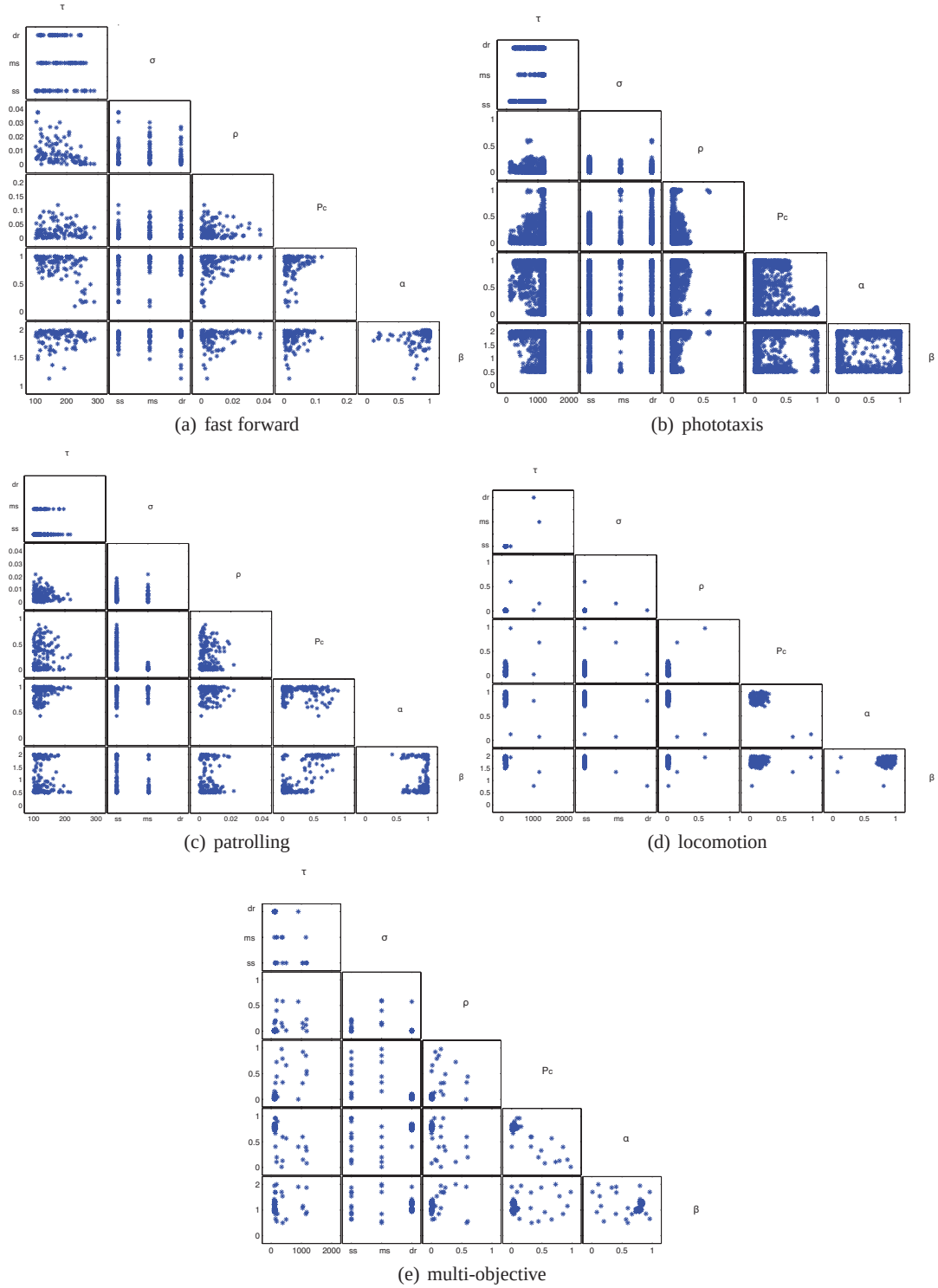
Figure 4 shows the make-up of non-dominated parameter vectors in pairs of parameter values. For single-task experiments (plots 4(a) to 4(d)), non-domination is defined as not performing statistically worse (using a Welch-T test with $\alpha = 5\%$) than the best parameter vector found. For the multi-objective results in Fig. 4(e), Pareto dominance is used, in which "better" and "worse" is replaced with "significantly better" and "significantly worse".

Each dot represents a pair of values in a non-dominated vector; the top-left plot in each matrix, for instance, shows combinations of σ and τ; the dots' vertical position denotes the σ control scheme (single-step self-adaptive (*ss*), multi-step self-adaptive (*ms*) or the derandomised approach (*dr*)), their horizontal position shows the τ value.

As an example, consider Fig. 4(d). In the cell that shows combinations of τ and σ, we see that $\sigma = 0$ is much more prevalent than other values and that the non-dominated vectors combine that value with low values of τ. This means that, although it hardly influences the best possible performance level that can be reached (cf. Table 6), it does influence the sensitivity of the other parameters.

Fig. 4(d)'s plot for P_c vs α shows a rectangular region that contains the majority of dots; this indicates that no interaction between these two parameters was found and they can therefore be set to optimal values independently. This contrasts with a cell like that of ρ and α in Fig. 4(a), which shows interaction: there, if ρ is low, α may have any value and vice versa.

The results from the fast forward task in Fig. 4(a) show interaction between ρ and the racing parameters α and β: in those vectors where racing is less strict (low α and β), re-evaluation is all but turned off (although ρ is never very high for this task). There are similar interactions between racing and crossover with low values of α and β occurring only when P_c is almost zero. For this task, the strategy for updating mutation step-size seems immaterial. Note that the val-



(a) fast forward (b) phototaxis

(c) patrolling (d) locomotion

(e) multi-objective

Fig. 4 Non-dominated parameter vectors matrices. Each matrix details the non-dominated parameter vectors for an experiment. A cell in one of the five matrices shows two entries of each non-dominated parameter vector, for instance τ vs. ρ in the first column, third row. Each dot indicates a combination of values that occurs in the non-dominated set.

ues for τ are very low: even the highest values are less than 300, while the full range extends to 1200. Within this relatively small range of τ values, there is an interesting pattern of interaction with ρ: the triangular shape of the region containing the value pairs seems to indicate a minimum number of new evaluations: if τ is lightly higher, ρ is less as if to compensate and vice versa.

Phototaxis (Fig. 4(b)) is by far the easiest task that the robots have to learn as borne out by the number of different non-dominated vectors: many parameter settings lead to near-optimal behaviour. It is the only task that seems to require – or should we say allow – lengthy evaluations (note that the scales differ from problem to problem). Again, there is evidence of interaction between crossover and racing: racing may be almost off (low α), but only when P_c is high.

The patrolling task (Fig. 4(c)) is the first to show a clear preference when it comes to σ-adaptation schemes: the derandomised approach does not occur in the non-dominated set. In the cases where multi-step adaptation is selected, the crossover rate is very low. The results for this task show an interaction between τ and ρ similar to that for fast forward: if one is high (within the small range that occurs in the non-dominated vectors), the other is low.

The locomotion task is the hardest of the four tasks and it is hard to distinguish any interactions from Fig. 4(d). It is clear, though, that τ and ρ are mostly low, that racing is quite strict and that the preferred σ-adatation scheme is single-step self-adaptation.

Figure 4(e) shows the non-dominated parameter vectors when optimising settings for all four tasks at the same time. Here, we see a strong preference for strict racing, but only if P_c is low. Although there are far fewer points with higher crossover probabilities, there seems to be a trend to lower values for α and β as P_c increases. All three σ-adaptation schemes occur, but the derandomised scheme seems to require very specific settings for the remaining parameters: the dots for *dr* are very closely clustered around specific values for the other parameters.

8 Discussion

For all tasks except the easy phototaxis task, the best parameter settings tend towards low values for the re-evaluation rate ρ and evaluation length τ and towards settings that imply strict comparisons for racing (high α and β). These settings all indicate that it is preferable to try many points in the search space and not dwell too much on the effects of noisy or unfair evaluations.

From all this, one might conclude that re-evaluation is not a worthwhile feature of $(\mu+1)$ ON-LINE. This seems at odds with the findings presented by Bredeche et al [6], where re-evaluation was shown to be beneficial in e-pucks evolving controllers for the fast-forward task. However, the tests there were conducted with $\mu = 1$, so the population could not as easily recover from genomes that were unluckily evaluated as very poor. Also, our experiments did not consider dynamic environments where higher population diversity could be essential - which in turn might imply a need for higher re-evaluation rates.

It is possible that unfair evaluations pose less of a problem in our experiments because the influence of an unrealistically high evaluation is limited: an individual that actually performs quite poorly may become part of the population, but enough properly good individuals remain to make sure that sufficient good offspring is generated. If this 'lucky' individual is selected as parent, its offspring will be cut short by racing (hence the high values for α and β) and so not even have lasting impact on the robot performance. This may also explain why P_c is set to low values: crossover increases the likelihood of such a bad individual contaminating the offspring of good individuals. Moreover, crossover acts as a macro-mutation [23], so low P_c values lead to a lower population diversity as illustrated in Fig. 5. This increases the likelihood that when a good individual is replaced unfairly with a bad one, a copy or at least a very similar individual remains to continue the bloodline.

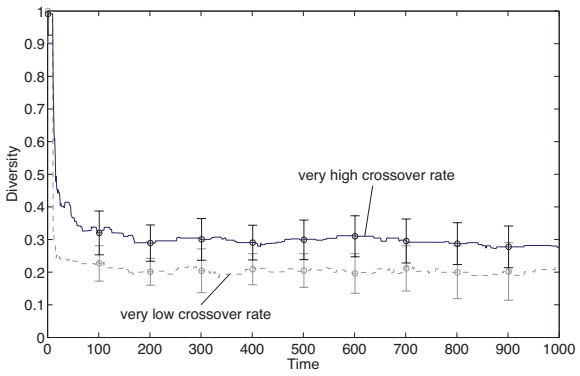


Fig. 5 Average population diversity over time for 10 runs of the fast forward task with low and with high P_c. Although the difference is not spectacular, a high crossover rate seems to increase diversity, suggesting macro-mutation effects.

We have seen that the phototaxis task requires parameter settings that conflict with those for the other tasks. We can only hypothesise what makes phototaxis so different – possibly the fact that it is so easy to solve. Be that as it may, we can certainly conclude that there is no 'silver bullet' parameter vector that yields optimal $(\mu+1)$ ON-LINE performance for all tasks.

Let us restate the vision that underlies our research into on-line, on-board evolutionary algorithms such as $(\mu+1)$ ON-LINE: it is that of robots autonomously adapting to any circumstances, particularly ones that we cannot foresee. Without a 'silver bullet,' this requires robust control schemes that allow $(\mu+1)$ ON-LINE or similar algorithms to change their parameters on the fly and so ensure universal adaptability.

9 Conclusion

In this paper we investigated an evolutionary algorithm for developing robot controllers on-the-fly. Our study offered innovations in two different dimensions, algorithmically as well as methodologically. The $(\mu + 1)$ ON-LINE evolutionary algorithm is encapsulated within each individual robot, thus enabling the robots to evolve their controllers independently from each other. $(\mu + 1)$ ON-LINE has a number of essential properties: 1) it relies on a population of μ individuals that produce one child per generation, 2) it uses a time-sharing scheme to evaluate the fitness of individuals (robot controllers), 3) it allows for re-evaluation to reduce noise, 4) it can self-adapt the mutation parameters and the length of the evaluation period. Our experiments showed that this algorithm can make the robots develop appropriate controllers in a number of tasks, ranging from 'simple' individual tasks (obstacle avoidance) through swarm tasks with light coupling between robots (patrolling) up to a task where robots are very strongly coupled (organism locomotion).

Our experimental analysis of $(\mu + 1)$ ON-LINE showed that:

- It seems preferable to try many alternative solutions and spend little effort on refining possibly faulty assessments;
- There is no single combination of parameters that performs well on all problem instances, indicating a need for on-line parameter control schemes to achieve robust autonomous adaptivity;
- The most influential parameter of this algorithm – and therefore the prime candidate for a control scheme – is the evaluation length τ.

These conclusions identify issues that warrant further research: firstly, there is an apparent need for control schemes, in particular for the very influential τ parameter which defines the length of evaluations. Secondly, we hypothesise that re-evaluation is not needed with even as small a population size as 10; this should be investigated further, especially in circumstances where the environment and/or task is dynamic.

As we already noted in Section 1, we cannot expect the results we find in our simulated trials to persist unaltered when $(\mu + 1)$ ON-LINE is applied in real robots. In that light, our results serve as a precursor for more focussed experiments to be conducted with real robots in subsequent research.

Methodologically, we deliberately deviated from the conventional horse-race approach: rather than just testing the algorithm and reporting the performances (perhaps comparing them to some benchmark), we adopted the scientific testing approach. This approach aims at "learning about your algorithm", rather than "testing your algorithm" and emphasises the analysis of robustness and parameter interactions. To this end, we used an automated parameter tuning method called Bonesa that is specifically designed for this purpose. Through scientific testing, we obtained valuable insights into $(\mu + 1)$ ON-LINE: which parameters have the most profound impact on the robots' performance in a variety of tasks, which parameter settings work well in what kind of task and which settings (if any) work well overall.

Maybe the most important conclusion from this research is a methodological one: analysing the make-up of the non-dominated parameter vectors as provided by Bonesa yields tremendous insight into the tuned algorithm. This insight can help us formulate new research questions and identify possibilities for improvement of the algorithm. These results of tuning are much more valuable than merely finding the best possible setting for some parameter on a particular problem or even on a range of problems.

Acknowledgements The authors gratefully acknowledge D.J. Christensen's providing the code on which we based our own locomotion experiments. Giorgos Karafotias was very helpful in setting up the other experiments. The authors thank the reviewers for their extensive and insightful comments; this has helped us produce a much better paper.

References

1. Bäck T (1996) Evolutionary Algorithms in Theory and Practice. Oxford University Press, Oxford, UK
2. Baldassarre G, Nolfi S, Parisi D (2002) Evolving mobile robots able to display collective behaviours. Artificial Life 9:255–267
3. Bartz-Beielstein T, Lasarczyk C, Preuss M (2005) Sequential parameter optimization. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation IEEE Congress on Evolutionary Computation, IEEE Press, Edinburgh, UK, vol 1, pp 773–780 Vol.1, DOI 10.1109/CEC.2005.1554761
4. Beyer HG (2000) Evolutionary algorithms in noisy environments: theoretical issues and guidelines for practice. Computer Methods in Applied Mechanics and Engineering 186(2-4):239 – 267, DOI DOI:10.1016/S0045-7825(99)00386-2, URL `http://www.sciencedirect.com/science/article/B6V29-40CRYKF-7/2/d4ea05ab01cc73ab36dbdf737c752439`
5. Branke J, Schmidt C, Schmec H (2001) Efficient fitness estimation in noisy environments. In: Proceedings of Genetic and Evolutionary Computation, pp 243–250
6. Bredeche N, Haasdijk E, Eiben A (2009) On-line, on-board evolution of robot controllers. In: Collet P, Monmarché N, Legrand P, Schoenauer M, Lutton E (eds) Artificial Evolution, Springer, Lecture Notes in Computer Science, pp 110–121,

URL http://few.vu.nl/~ehaasdi/papers/EA2009selfadapt1robot.pdf
7. Brooks RA (1992) Artificial life and real robots. In: Varela F, Bourgine P (eds) Toward a Practice of Autonomous Systems: Proceedings of the 1st European Conference on Artificial Life, MIT Press, Cambridge, MA, USA, pp 3–10, URL http://people.csail.mit.edu/brooks/papers/real-robots.pdf
8. Christensen DJ, Spröwitz A, Ijspeert AJ (2010) Distributed online learning of central pattern generators in modular robots. In: Doncieux S, Girard B, Guillot A, Hallam J, Meyer JA, Mouret JB (eds) From Animals to Animats 11, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol 6226, pp 402–412
9. Eiben A, Smith J (2003) Introduction to Evolutionary Computing. Springer, Berlin Heidelberg
10. Eiben AE, Smit SK (2011) Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation 1(1):19–31
11. El-Beltagy M, Nair P, Keane A (1999) Metamodeling techniques for evolutionary optimization of computationally expensive problems: Promises and limitations. In: Banzhaf W, Daida J, Eiben A, Garzon M, Honavar V, Jakiela M, Smith R (eds) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999), Morgan Kaufmann, San Francisco, pp 196–203
12. Floreano D, Schoeni N, Caprari G, Blynel J (2002) Evolutionary bits'n'spikes. In: Standish RK, Bedau MA, Abbass HA (eds) Artificial Life VIII : Proceedings of the eighth International Conference on Artificial Life, MIT Press, Cambridge, MA, USA, pp 335–344
13. Haasdijk E, Eiben AE, Karafotias G (2010) On-line evolution of robot controllers by an encapsulated evolution strategy. In: Proceedings of the 2010 IEEE Congress on Evolutionary Computation, IEEE Computational Intelligence Society, IEEE Press, Barcelona, Spain, URL http://www.few.vu.nl/~ehaasdi/papers/cec-2010-MuPlusOne.pdf
14. Haasdijk E, ul Qayyum AA, Eiben A (2011) Racing to improve on-line, on-board evolutionary robotics. In: [25], pp 187–194, URL http://www.cs.vu.nl/ci/pubs/2011/MuPlusOneAndRacing.pdf
15. Haralick R, Shapiro L (1992) Computer and Robot Vision, vol 1, Addison-Wesley Publishing Company, chap 7
16. Haroun Mahdavi S, Bentley PJ (2006) Innately adaptive robotics through embodied evolution. Auton Robots 20(2):149–163, DOI http://dx.doi.org/10.1007/s10514-006-5941-6
17. Hart W, Belew R (1991) Optimizing an arbitrary function is hard for the genetic algorithm. In: Belew RK, Booker LB (eds) Proceedings of theFourthInternational Conference on Genetic Algorithms(ICGA '91), Morgan Kaufmann Publishers, Inc., San Mateo CA, pp 190–195, URL citeseer.ist.psu.edu/hart91optimizing.html
18. Hoeffding W (1963) Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301):13–30
19. Hooker J (1995) Testing heuristics: We have it all wrong. Journal of Heuristics 1:33–42, URL http://dx.doi.org/10.1007/BF02430364, 10.1007/BF02430364
20. Jelasity M, Montresor A, Babaoglu O (2005) Gossip-based aggregation in large dynamic networks. ACM Trans Comput Syst 23:219–252, DOI http://doi.acm.org/10.1145/1082469.1082470, URL http://doi.acm.org/10.1145/1082469.1082470
21. Jin Y (2005) A comprehensive survey of fitness approximation in evolutionary computation. Soft Computing 9(1):3–12
22. Johnson DS (2002) A theoretician's guide to the experimental analysis of algorithms. In: Goldwasser MH, Johnson DS, McGeoch CC (eds) Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges, American Mathematical Society, Providence, pp 215–250
23. Jones T (1995) Crossover, macromutation, and population-based search. In: Eshelman L (ed) Proceedings of the 6th International Conference on Genetic Algorithms, Morgan Kaufmann, San Francisco, pp 73–80
24. Karafotias G, Haasdijk E, Eiben A (2011) An algorithm for distributed on-line, on-board evolutionary robotics. In: [25], pp 171–178, URL http://www.cs.vu.nl/ci/pubs/2011/Encapsulated-vs-Distributed.pdf
25. Krasnogor N, Lanzi PL, Engelbrecht A, Pelta D, Gershenson C, Squillero G, Freitas A, Ritchie M, Preuss M, Gagne C, Ong YS, Raidl G, Gallager M, Lozano J, Coello-Coello C, Silva DL, Hansen N, Meyer-Nieberg S, Smith J, Eiben G, Bernado-Mansilla E, Browne W, Spector L, Yu T, Clune J, Hornby G, Wong ML, Collet P, Gustafson S, Watson JP, Sipper M, Poulding S, Ochoa G, Schoenauer M, Witt C, Auger A (eds) (2011) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2011), ACM, Dublin, Ireland
26. Martinoli A (1999) Swarm intelligence in autonomous collective robotics from tools to the analysis and synthesis of distributed control strategies. PhD thesis, Ecole Polytechnique Fédérale de Lausanne
27. Nannen V, Eiben A (2007) Relevance estimation and value calibration of evolutionary algorithm parameters. In: Proceedings of the International Joint Conference

on Atrificial Intelligence (IJCAI'07), International Joint Conferences on Artificial Intelligence, AAAI Press, pp 975–980
28. Nehmzow U (2002) Physically embedded genetic algorithm learning in multi-robot scenarios: The pega algorithm. In: Prince C, Demiris Y, Marom Y, Kozima H, Balkenius C (eds) Proceedings of The Second International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems, LUCS, Edinburgh, UK, no. 94 in Lund University Cognitive Studies
29. Nelson AL, Barlow GJ, Doitsidis L (2009) Fitness functions in evolutionary robotics: A survey and analysis. Robotics and Autonomous Systems 57(4):345 – 370, DOI DOI:10.1016/j.robot.2008.09.009, URL http://www.sciencedirect.com/science/article/B6V16-4TTMJV3-1/2/2549524d8e0f3982730659e49ad3fa75
30. Nolfi S, Floreano D (2000) Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines. MIT Press, Cambridge, MA
31. Nordin P, Banzhaf W (1997) An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. Adaptive Behavior 5:107–140
32. Ostermeier A, Gawelczyk A, Hansen N (1994) A derandomized approach to self adaptation of evolution strategies. Evolutionary Computation 2(4):369–380, URL http://dx.doi.org/10.1162/evco.1994.2.4.369
33. Pedersen MEH, et al (2008) Parameter tuning versus adaptation: proof of principle study on differential evolution. Tech. rep., Hvass Laboratories, URL http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.149.6020&rep=rep1&type=pdf
34. Schwefel HP (1995) Evolution and Optimum Seeking. Wiley, New York
35. Simões EDV, Dimond KR (2001) Embedding a distributed evolutionary system into population of autonomous mobile robots. In: Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference, URL citeseer.ist.psu.edu/simoes01embedding.html
36. Smit S, Eiben A (2010) Parameter tuning of evolutionary algorithms: Generalist vs. specialist. In: et al CDC (ed) Applications of Evolutionary Computation, Springer, Lecture Notes in Computer Science, vol 6024, pp 542–551, URL http://www.cs.vu.nl/~gusz/papers/2010-EVOSTAR-GeneralistSpecialist.pdf
37. Smit S, Eiben A (2011) Multi-problem parameter tuning using bonesa. In: Hao JK, Legrand P, Collet P, Monmarché N, Lutton E, Schoenauer M (eds) Proceedings of Artificial Evolution, 10th International Conference, Evolution Artificielle (EA 2011), Springer, Lecture Notes in Computer Science, pp 222–233
38. Tuci E, Quinn M, Harvey I (2002) Evolving fixed-weight networks for learning robots. In: CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress, IEEE Computer Society, pp 1970–1975
39. Walker JH, Garrett SM, Wilson MS (2006) The balance between initial training and lifelong adaptation in evolving robot controllers. IEEE Transactions on Systems, Man, and Cybernetics, Part B 36(2):423–432
40. Watson RA, Ficici SG, Pollack JB (2002) Embodied evolution: Distributing an evolutionary algorithm in a population of robots. Robotics and Autonomous Systems 39(1):1–18, URL http://eprints.ecs.soton.ac.uk/10620/
41. Wischmann S, Stamm K, Wörgötter F (2007) Embodied evolution and learning: The neglected timing of maturation. In: Almeida e Costa F (ed) Advances in Artificial Life: 9th European Conference on Artificial Life, Lecture Notes in Artificial Intelligence, vol 4648, Springer-Verlag, Lisbon, Portugal, pp 284–293
42. Zitzler E, Laumanns M, Thiele L (2001) SPEA2: Improving the strength pareto evolutionary algorithm. Tech. Rep. 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zürich, Gloriastrasse 35, CH-8092 Zürich, Switzerland