Is the Meta-EA a Viable Optimization Method?

Sean Luke
Department of Computer Science
George Mason University
Washington, DC
sean@cs.gmu.edu

AKM Khaled Ahsan Talukder
Department of Computer Science
George Mason University
Washington, DC
atalukde@cs.gmu.edu

ABSTRACT

Meta-evolutionary algorithms have long been proposed as an approach to automatically discover good parameter settings to use in later optimization runs. In this paper we instead ask whether a meta-evolutionary algorithm makes sense as an optimizer in its own right. That is, we're not interested in the resulting parameter settings, but only in the final result. As it so happens, this use of meta-EAs make sense in the context of large numbers of parallel runs, particularly in massive distributed scenarios. A primary issue facing meta-EAs is the stochastic nature of the meta-level fitness function. We consider whether this poses a challenge to establishing a gradient in the meta-level search space, and to what degree multiple tests are helpful in smoothing the noise. We discuss the nature of the meta-level search space and its impact on local optima, then examine the degree to which exploitation can be applied. We find that meta-EAs perform well as optimizers, and very surprisingly that they do best with only a single test. More exploitation appears to reduce performance, but only slightly.

Categories and Subject Descriptors

I.2.m [**Artificial Intelligence**]: Miscellaneous—*Evolutionary Computation*

Keywords

Meta-Evolutionary Algorithms; Parallel Algorithms

1. INTRODUCTION

A meta-optimization algorithm iteratively improves the parameter settings of a second optimization algorithm, assessesing their quality or fitness by using them to run the second algorithm and examining its performance. The method is related to the general family of hyperheuristics [31]. Since the 1970s [8, 27] this idea has been proposed as a mechanism for automating the tuning process of evolutionary algorithms and other metaheuristics.

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

The traditional reasoning behind meta-optimization goes like this: an experimenter typically performs up-front parameter tuning of an optimization algorithm, and this is essentially a crude form of hill-climbing. Why not take him out of the loop? If one were going to do a large number of optimization runs with the same parameters, it might be helpful to automatically find good parameter settings first.

This philosophy leads to the notion of meta-optimization with the goal of finding an optimal set of *parameter settings* to put to use at some point in the future. We think this is the primary impetus behind much meta-optimization research. But there is an alternative reason to perform meta-optimization: as an optimization algorithm in its own right. Here, the goal would be to directly find an optimal *solution.* At first glance one might be bewildered by this: after all, meta-optimization is extraordinarily expensive. Surely all those cycles could be put to better use? But we think this second use of meta-optimization in fact has a definite niche in massively parallel optimization.

In this paper we study meta-optimization with the goal of an optimal solution. In our study, both levels of optimization are EAs (hence a *meta-EA*), and we use them in the context of optimizing thousands of runs on a supercomputer cluster.

Scenario. Imagine that an experimenter has a difficult-to-optimize function and has decided to tackle it by doing some (very large) N runs and returning the best result. This is hardly an uncommon scenario: depending on the problem it is often a preferred approach over other forms of parallelism, such as island models or distributed evaluation.

In fact, there are some parallel architectures for which the maximum of N runs is the most attractive option. One example: Parabon Computing produces a variant of the ECJ evolutionary toolkit [26] called *Origin*, which runs on their *Frontier* massively parallel platform.[1] Parabon contracts with organizations to install software on their PCs which makes them available as computational units to Frontier when not being used otherwise. This makes up to millions of PCs available as computational units. But because of security concerns inherent in contracting for PCs, jobs running on Frontier PCs are sandboxed and are not permitted to communicate with other PCs. This lack of intercommunication means that island models do communication through the master server, which is unattractive because communication to the master is slow. Furthermore, this slow communication with the master server means that distributed evaluation is infeasible unless the evaluation time is very high per-individual, or we

[1] http://www.parabon.com/

are doing opportunistic evolution [34]. But Frontier is an excellent target for massively parallel runs.

If the experimenter has settled on doing N runs, he must decide two things. First, he must determine how many *evaluations* each of these runs will perform. Second, he must decide on the details of his optimization procedure: he may have to choose an optimization procedure among many; or just select parameter settings for a procedure if he has already decided on which one to use.

The first decision (number of evaluations) is something that the experimenter must make himself: he cannot delegate it to an optimizer. But the second decision (algorithmic details) can be delegated. If he is fortunate, the experimenter may already have a general idea of the good range for parameter settings through previous experience or the suggestions of the research literature. But he is unlikely to know the *precise* best parameter settings for this particular problem. He can take a stab at some parameter setting for all N runs, or perhaps try random settings within the range. In any case, it would be helpful if the algorithm optimized to good-performing settings mid-run, given the high cost of the run itself.

Let us consider using a meta-EA to optimize these run settings. We divide the N total runs into G generations of P runs each. We create a population of P individuals consisting of parameter settings, and test them on parallel machines. We set the fitness of an individual to the best-so-far fitness from a run using its parameters. We use an evolutionary algorithm to produce a second generation of P individuals, and repeat up to G times.

The primary challenge this approach faces is the stochasticity of the fitness assessment. Parameter settings are not assigned a deterministic fitness, but rather a fitness based on the vagaries of the underlying evolutionary algorithm which used that parameter setting. The high stochasticity of evolutionary algorithms leads to a high degree of noise in the meta-EA's fitness function, which in turn can make it difficult for the meta-EA to establish a gradient to optimize. The obvious way to deal with this is to assess each parameter setting using some T number of tests and taking the mean or median best-of-run performance as its fitness. As T increases, the noise ideally reduces, but at the high cost of eating into the optimization: after all, the total budget is $N = T \times G \times P$. Either G or P, or both, must be reduced if $T > 1$. If T is prohibitively high, more intelligent statistical methods may be used to reduce the total number of tests [33].

It is also not clear whether, ignoring noise, the meta-level parameter-setting fitness landscape is complex. Will a hill-climber suffice? Is the space complex enough to require a nontrivial global optimization algorithm, and if so, what should be the parameters for *that* algorithm? Is it, so to speak, "turtles all the way down"?

2. PREVIOUS WORK

The history of parameter-setting in evolutionary algorithms is a very long one and has garnered a large amount of literature [15]. Because of its importance, the question of whether a parameter setting could be *automated* has been provoking the research community for several decades, at least since the study of adaptive or self-adaptive parameter tuning in the first evolution strategies.

The earliest investigations into *meta-level* parameter optimization were probably [8, 27]. However, these methods didn't use an evolutionary algorithm per se at the meta level, but rather a kind of pattern-learning technique similar to that of [4]. Later this general idea of meta-level control of optimization was extended to the notion of the meta-EA: an evolutionary algorithm to optimize one or more other evolutionary algorithms [16, 18].

Meta-evolutionary algorithms are related to the family of *hyperheuristics* [10, 13, 31] The difference between the two is soft. While meta-EAs focus on optimizing parameters to an evolutionary algorithm, hyperheuristics instead usually try to select from a set of prespecified algorithms, along with optimizing their parameters.

The existing meta-optimization literature is largely concentrated on finding the optimal parameter settings for the underlying optimization algorithm, for example [25, 29, 32]. There is debate as to whether this is an effective method. In [9], it was reported that the parameters discovered can be unimpressive. However [22] reported substantial and consistent performance improvements in parameter settings for a variety of (non-EA) algorithms.

An obvious choice to test a meta-optimization algorithm is a massively parallel model, as the problem's inherent properties demand it. Early examples of parallel meta-EAs include [1, 2, 23], which essentially describe a master-slave model with asynchronous evolution where the primary goal of the investigation is to discover optimal parameter settings to achieve "maximum convergence velocity". There are also a number of implementations of parallel models focusing on different aspects of meta-optimization. For example, in [7] the main goal was to find the optimal parameter settings for the One-Max problem to compare with expected theoretical results. In [3], the authors investigated how parallel hyperheuristics (rather than meta-EAs) could be utilized to find an algorithm along with its optimal parameters to solve difficult numerical optimization problems. Similar analyses may be found in [28]. In our work we presume a fixed budget, but [5] has examined how to perform meta-optimization with a flexible budget.

The parameter-level fitness landscape of a meta-EA is noisy. However inferring the shape of the parameter landscape is critical to successful meta-optimization. Investigations into the stochastic parameter landscapes may be found in [17]. In [21], the authors build a gaussian process model to deal with stochasticity. In this paper we examine using multiple tests to filter noise, a notion further examined in [33].

3. METHOD

Our basic approach is a straightforward meta-EA applied in a parallel setting. A meta-EA has two levels of evolutionary algorithms. At the *meta level*, individuals are parameter settings. To test a meta-individual, we run some T *lower level* evolutionary runs using that individual's parameter settings. We then extract the T best-fitness-of-run results from each run and set the fitness of the meta-individual to their mean. Though we use the mean of the best-fitness-of-run results as fitness values at the meta level, this is not how we track the performance of the meta-level algorithm. For that, we are interested in the best result discovered so far by *any* lower-level evolutionary run.

The meta-EA is parallelized on a computer cluster using $S = 128$ slave processors and one master processor. The master runs the meta-level evolutionary algorithm. To test a meta-individual a single time, we assign a slave processor to run a lower-level evolutionary algorithm using its parameters.

Thus to assess the fitness of a meta-level individual, slave processors are employed T times in total. The evolutionary computation system we used was ECJ [26].

Though we applied various evolutionary methods at the lower-level, we always used a variant of the $(\mu + \lambda)$ evolutionary strategy at the meta-level, as follows. After fitness assessment, the fittest μ individuals were directly copied to the next generation. The remainder of the population was then filled with $P - \mu$ children, each produced by uniformly randomly selecting among the μ elites, then applying mutation. Though the population size varied in size (due to T), we kept the fraction of μ roughly constant by setting $\mu = \max(\lfloor P/8 \rfloor, 1)$, except in Experiment 3. Since the meta-level fitness function was noisy, elites were always re-assessed for fitness at each generation.

Much of the complexity of a meta-EA lies in how it represents meta-individuals. In our experiments, meta-level genomes were fixed-length vectors of parameter settings, where each parameter was of one of the following types:

- *Floating-point values.* For example, the parameter "mutation probability" might have values from 0.0 to 1.0. Values were initialized randomly within legal bounds. Mutation occurred with a per-gene probability of 0.25, using Gaussian convolution with $\sigma = 0.25$ except where indicated. The algorithm repeatedly tried to find a convolution which would result in a value within bounds.

- *Ordered integer values.* For example, the parameter "number of generations" might have values from 1 to 64. Values were initialized randomly within legal bounds. Mutation occurred with a per-gene probability of 0.25, using an integer random walk of probability w. The random walk was a loop: each iteration either 1 or -1 was added to the gene value, and then with probability $1 - w$ the loop was exited.

- *Ordered set values.* These were strings with a specific order among them. For example, the parameter "population size" might have values of "32", "64", "128", "256", or "512". These values were treated as ordered integer values, namely 0, 1, 2, 3, and 4 respectively.

- *Unordered set values.* These were strings with no order among them. For example, the parameter "crossover type" might have values of "one-point", "two-point", and "uniform". These values were treated as integers (in this example, 0, 1, 2), and were initialized randomly within legal bounds (0–2). Mutation occurred with a per-gene probability of 0.25, and simply randomized the gene value. Boolean values were considered unordered sets consisting of the strings "true" and "false".

4. BENCHMARK PROBLEMS

To examine the generality of the technique across various optimization methods, we tested it on three different kinds of evolutionary algorithms (GA, ES, and DE) at the lower-level, each using a different objective function.

The objective functions were chosen for three qualities. First, they are fairly difficult, so as to make multiple parallel runs reasonable. In no case would we expect the optimum to be discovered. Second, like many objective functions, they are sensitive to different evolutionary parameter settings, making them amenable to meta-evolution. And third but far from least, they are fast. This third characteristic was crucial given the 14.2 million evolutionary runs performed. The three benchmark problems were:

A Genetic Algorithm (GA) applied to the Leading-Ones Blocks Problem. Leading-Ones Blocks is a stairstep variation of the Leading-Ones problem with tunable difficulty. Leading-Ones Blocks takes a boolean vector, and a tuning parameter b, then counts the number of successive strings of 1s of length b in the vector, starting from the beginning, until the first 0 is encountered. If b is small the problem is trivial, but if b is large the problem can be prohibitively difficult. We may formally define Leading-Ones Blocks as:

$$f(x_1, ..., x_n) = \left\lfloor \frac{1}{b} \sum_{i=1}^{n} \prod_{j=1}^{i} x_j \right\rfloor$$

We set the genome size to 400, and b to 20, meaning that the optimal fitness was 50 (higher is better).

The genetic algorithm worked as follows. Initial individuals were generated randomly, rejecting duplicate individuals. After fitness evaluation, given a population size P and a proportion $0.0 \leq e \leq 1.0$ of elites, some number $\lfloor e \times P \rfloor$ of elites were directly copied to the next generation. The remainder of the next generation population was filled by iteratively selecting parents in pairs, crossing them over, and mutating them. An evolutionary run lasted for 65,536 fitness evaluations.

At the meta-level, we optimized the following parameters:

- *Population Size (P): 32, 64, 128, 256, 512, 1024, 2048* Since the evaluations were fixed, this in turn would affect the generation size (2048, 1024, 512, 256, 128, 64, or 32 respectively). At the meta-level, these values were treated as ordered set values.

- *Elitism Proportion (e): [0.0...1.0]*

- *Tournament Size (t): [1.0...20.0]* Note that this is a floating-point number. With probability $t - \lfloor t \rfloor$ we select with tournament size $\lceil t \rceil$, else with tournament size $\lfloor t \rfloor$.

- *Per-gene Mutation Probability: [0.0...1.0]* Mutation was done by flipping the gene's bit.

- *Crossover Type: one-point, two-point, uniform* One-point crossover, two-point crossover, or uniform crossover with a per-gene probability c. At the meta-level, these were unordered set values.

- *Per-gene Crossover Probability: [0.0...0.5]* This was only relevant to uniform crossover.

A (μ, λ) Evolution Strategy (ES) applied to the Rotated Griewank Problem. The *non*-rotated Griewank problem is floating-point and ranges with $x_i \in [-600, 600]$:

$$f(x_1, ..., x_n) = 1 + \frac{1}{4000} \left(\sum_{i=1}^{n} x_i^2 \right) + \prod_{i=1}^{n} \cos \left(\frac{x_i}{\sqrt{i}} \right)$$

Let $\vec{x} = \langle x_1, ..., x_n \rangle$. We rotated Griewank with a rotation matrix M, generated randomly once per lower-level evolutionary run, which defined a revised fitness function $r(\vec{x}) = f(M\vec{x})$. M was produced using Gram-Schmidt orthonormalization, following the procedure described in [19]. The genome size was 100. Lower fitness values were better.

The (μ, λ) ES worked as follows. We defined a population size λ and a specified proportion $0.0 \leq m \leq 1.0$ of parents, where μ was then set to $\max(\lfloor m\lambda \rfloor, 1)$. Initial individuals were created by randomizing each gene uniformly from [0.0...1.0], not permitting duplicate individuals. After fitness evaluation, we gathered the μ best members of the population as parents. Each parent generated λ/μ children via mutation, which were then added to the next-generation population. An evolutionary run would last for no more than 16,384 fitness evaluations.

At the meta-level, we optimized the following parameters:

- *Population Size (λ): 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048* Since the evaluations were fixed, this in turn would affect the generation size (8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, or 8 respectively). At the meta-level, these were ordered set values.
- *m (μ as a Proportion of λ): [0.0...0.5]*
- *Per-gene Mutation Probability: [0.0...1.0]*
- *Mutation Type: Reset, Gaussian, Polynomial* At the meta-level, these were unordered set values. Reset mutation simply randomized the gene. Gaussian mutation added random gaussian noise of standard deviation σ to the gene until the result. Polynomial mutation applied random noise to the gene under the polynomial distribution of index value i. Two variants of polynomial mutation were used as described next.
- *Alternative Polynomial Version: true, false* We know of at least four different versions of polynomial mutation in the literature. ECJ provides two of these variants, which it calls the "standard" variant, as defined in [11], and the "alternative" variant, as defined in [12]. At the meta-level, these were unordered set values.
- *Polynomial Distribution Index (i): [0...40] (integers)* This was only relevant to polynomial mutation.
- *Gaussian Standard Deviation (σ): [0.0...1.0]* This was only relevant to gaussian mutation.

Both Gaussian and Polynomial Mutation can produce results outside of valid gene bounds. In both cases we would retry up to 100 times to find a valid in-bounds mutation, after which the gene would remain un-mutated.

Differential Evolution (DE) applied to the Lennard-Jones Problem. Here the goal is to find the global minimum of a potential energy function over a cluster of atoms which depends on their relative positions [36]. The potential energy E of a Lennard-Jones cluster is calculated as:

$$E = 4\epsilon \sum_i \sum_{j>i} \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right]$$

r_{ij} is the Euclidian distance between atoms i and j respectively. We define atom positions in three dimensions, and so $r_{ij} = \sqrt{(i_x - j_x)^2 + (i_y - j_y)^2 + (i_z - j_z)^2}$. The genome has three numbers (x, y, z) for each atom, hence $i_x, i_y, i_z, j_x, j_y, j_z, k_x, k_y, k_z, ...$ etc. Like much of the literature [6], we set the ϵ and σ constants to 1. We used ten atoms, hence a vector of size 30. We also converted the problem to maximization by negating the fitness function.

We performed standard Differential Evolution (DE) with three common DE breeding operators: DE/rand/1/bin, DE/best/1/bin, and DE/rand/1/either-or. Differential Evolution is too complex a technique to describe fully here: instead, consult [30]. We deviated from DE tradition in one respect: we guaranteed that the operators produced children within valid gene bounds. We did this by trying up to 5 times to generate a valid child; otherwise we simply copied its primary parent instead. An evolutionary run would last for no more than 65,536 fitness evaluations.

At the meta-level, we optimized the following parameters:

- *Population Size (P): 32, 64, 128, 256, 512, 1024, 2048* Since the evaluations were fixed, this in turn would affect the generation size (2048, 1024, 512, 256, 128, 64, or 32 respectively). At the meta-level, these were ordered set values.
- *Breeding Operator: DE/rand/1/bin, DE/best/1/bin, DE/rand/1/either-or* At the meta-level, these were unordered set values.
- *Per-gene DE Crossover Probability (C_r): [0.0...1.0]* A standard DE parameter.
- *DE Mutation Scale Factor (FNOISE): [0.0...0.05]* A standard DE parameter used only by the DE/best/1/bin operator. FNOISE's range is so small that we modified its meta-level standard deviation, setting $\sigma = 0.001$.
- *DE Either-Or Probability (PF): [0.0...1.0]* This is a standard DE parameter used only by the DE/rand/1/either-or operator.

5. FIRST EXPERIMENT

We began by asking: how many tests per meta-level fitness evaluation are worthwhile? Let us assume that we had S processors at our disposal and were permitted to run each for some E evolutionary runs, for a total budget of $N = E \times S$ runs. In this arrangement, if we applied a single test per meta-level fitness evaluation, our population size P would be constrained to be $\geq S$. We chose the obvious setting ($P = S = 128$).

To examine the effect of tests, we must decide how increasing tests will impact on the budget. Tests could reduce the population size, or the number of generations, or both. For this experiment we chose to reduce the population size. Thus given T tests per meta-level fitness evaluation, $P \times T = S$ ($= 128$). This also determines the number of generations G in our meta-level algorithm, specifically, $G = E$.

We tried 1, 2, 4, 8, and 16 tests, with corresponding meta-level population sizes of 128, 64, 32, 16, and 8, and μ values of 16, 8, 4, 2, and 1 respectively. We performed 50 independent runs of each test size, and also performed 50 runs of entirely random search at the meta-level, where the parameter settings tried were randomly chosen from among their legal gene values. Random search provided our lower bound for comparison. Finally, we identified the single most successful parameter setting discovered from all meta-level runs, and performed 50 meta-level runs using just that parameter setting. This setting provided us with an approximate upper-bound. In all cases, meta-level runs lasted 64 generations, totaling 8192 fitness evaluations (hence lower-level evolutionary runs) per meta-level run.

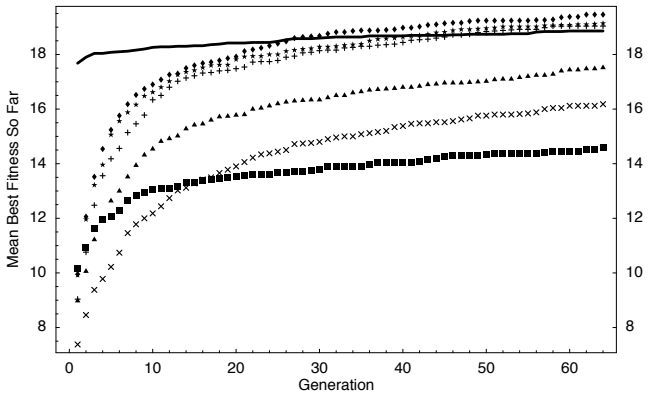


Figure 1: Meta-level mean best-fitness-so-far for the Leading-Ones-Blocks problem (GA, maximization). ♦ 1 Test ★ 2 Tests + 4 Tests ▲ 8 Tests × 16 Tests ■ Random Search — Best Discovered Setting

	1 Test	*2 Tests*	*4 Tests*	*8 Tests*	*16 Tests*
> Random	**~1.0**	**~1.0**	**~1.0**	**0.9999**	**0.9999**
< Best	**0.9996**(>)	0.9399(>)	0.6979(>)	**0.9993**	**~1.0**
< 1 Test	——	0.8271	0.9524	**0.9999**	**~1.0**

Table 1: P-values of differences between methods on the GA Leading Ones Blocks problem. Bold values indicate statistically significant results. > means "is better than" and < means "is worse than". (<) means that the result is not better (the default) but in fact worse. Likewise (>) means that the result is not worse (the default) but in fact better.

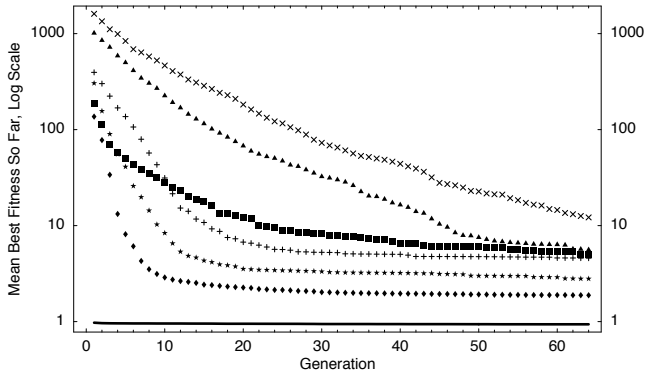


Figure 2: Meta-level mean best-fitness-so-far for the Rotated Griewank problem (ES, minimization). Log_{10} scale shown. ♦ 1 Test ★ 2 Tests + 4 Tests ▲ 8 Tests × 16 Tests ■ Random Search — Best Discovered Setting

	1 Test	*2 Tests*	*4 Tests*	*8 Tests*	*16 Tests*
> Random	**~1.0**	**~1.0**	**0.9999**	**0.9999**(<)	0.5774(<)
< Best	**~1.0**	**~1.0**	**~1.0**	**~1.0**	**~1.0**
< 1 Test	——	0.9979	**0.9999**	**0.9999**	**~1.0**

Table 2: P-values of differences between methods on the ES Rotated Griewank problem. Bold values indicate statistically significant results. > means "is better than" and < means "is worse than". (<) means that the result is not better (the default) but in fact worse. Likewise (>) means that the result is not worse (the default) but in fact better.

Results. We verified statistical significance using a nonparametric ranked two-tailed T-test. Experiments in the paper total to 67 comparisons, so we used a Bonferroni correction, adjusting the p value from 0.95 to 0.99925.

The results are shown in Figures 1 through 3. Additionally, Tables 1 through 3 show the p-values for comparison between various numbers of tests and three different runs: "Random" (random search), "Best" (the fittest parameter settings discovered in any run), and "1 Test" (since 1-Test was the best performing). As can be seen from the tables, comparisons generally produced quite high p-values. In many cases the p-value was "~1.0", meaning that the two compared distributions were disjoint when their samples were ranked.

Conclusions. How many tests was a good trade-off between noisy fitness and population size? The answer was very surprising: **1 test!** More tests yielded worse performance.

With a few exceptions, the multi-test runs were bounded between random runs and the best-of-all-runs parameter setting. In all cases, 1-, 2-, and 4-test arrangements proved statistically significantly superior to doing random runs. This confirmed our hypothesis that if you didn't really know what settings to use for a difficult problem, a meta-EA would probably be a better option to simply trying arbitrary settings within reasonable ranges.

In fact in several cases 1- and 2-test arrangements were statistically superior to the best-of-all-runs setting! This seemingly impossible event is easily explained: the best-of-all-runs setting is strongly susceptible to being a random outlier, and in practice it may actually underperform other top-ranked settings in the average. We believe that this provides more evidence for the claim made in [9]: that doing meta-evolution to produce an optimal parameter setting (as opposed to a final result) may not produce particularly good results due to such lucky outliers.

6. SECOND EXPERIMENT

In the first experiment, we examined trading off tests for population size. Another approach is to trade off tests for generations. However, if the meta-level evolutionary algorithm is slow to converge to a decent set of parameters, we imagined that cutting generations short could have a detrimental impact on the meta-evolutionary algorithm.

To test this, we once again used $S = 128$ processors, and fixed the population size $P = S = 128$. We varied the number of meta-level tests T per fitness assessment to 1, 2, 4, 8, and 16. To maintain a constant number of evaluations, we also varied the meta-level number of generations G to 64, 32, 16, 8, and 4 respectively. Thus more tests per individual resulted in fewer generations at the meta-level.

We restricted ourselves to the DE Lennard-Jones problem. While we included random search as a lower bound for reference, we did not include an upper bound, as we had already compared with it: we were more interested in how fewer generations impacted on performance. Besides P, T,

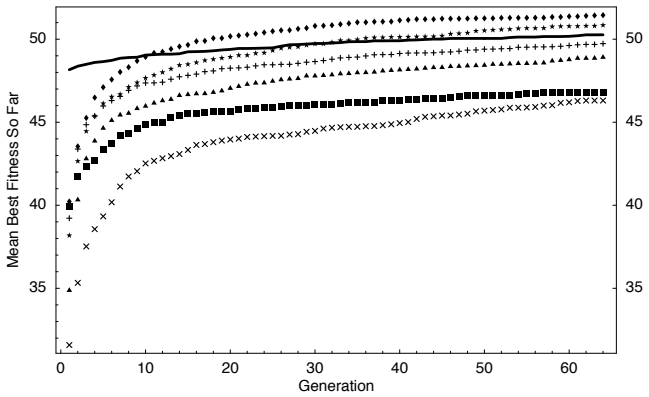


Figure 3: Meta-level mean best-fitness-so-far for the Lennard-Jones problem (DE, maximization). ⧫ 1 Test ★ 2 Tests + 4 Tests ▲ 8 Tests × 16 Tests ■ Random Search — Best Discovered Setting

	1 Test	*2 Tests*	*4 Tests*	*8 Tests*	*16 Tests*
> Random	**~1.0**	**~1.0**	**0.9999**	**0.9999**(<)	0.9544(<)
< Best	**0.9999**(>)	**0.9990**(>)	0.275	0.9006	**~1.0**
< 1 Test	——	**0.9997**	**0.9999**	**0.9999**	**~1.0**

Table 3: P-values of differences between methods on the DE Lennard-Jones problem. Bold values indicate statistically significant results. > means "is better than" and < means "is worse than". (<) means that the result is not better (the default) but in fact worse. Likewise (>) means that the result is not worse (the default) but in fact better.

and G, we used the same meta-level parameters and number of runs as in the first experiment.

Results. We again verified statistical significance using a nonparametric ranked two-tailed T-test with a Bonferroni correction, adjusting the p value to 0.99925. The results are shown in Figure 4 and summarized in Table 4.

Conclusions. Once again, results worsened monotonically with increasing tests. In all cases, T tests was statistically significantly better than $> 2T$ tests, and to random search.[2] Again we have no reason to suggest using anything other than a single test!

7. A DIGRESSION

At this point it's worth discussing the nature of the space of parameters in an evolutionary algorithm and its impact on local optima. In the parameter optimization literature, the classical presentation of parameters has been as a vector of values. But in fact these parameters may be better thought of as a *tree*. The reason for this is because the presence of a certain parameter A will demand the existence of another

[2]We informally let this experiment run to twice as many generations to see if this effect diminished. It does. The results were still monotonic, but the p-values dropped below significance except for 1 > 4 and all > Random Search. We think this suggests that the benefits of the meta-EA might lie largely in the first 32 or so generations.

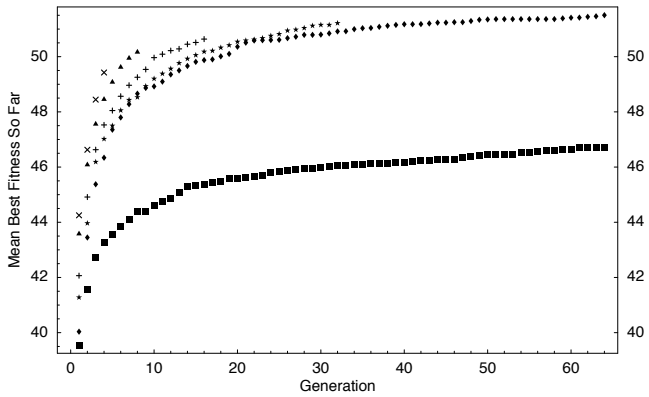
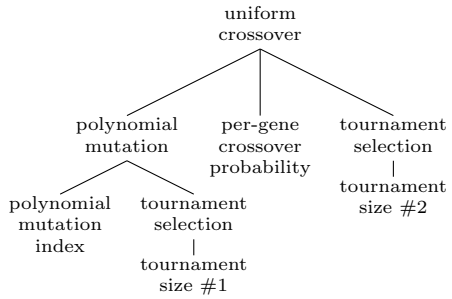


Figure 4: Meta-level mean best-fitness-so-far for the Lennard-Jones problem (DE, maximization) with generations traded for tests. ⧫ 1 Test ★ 2 Tests + 4 Tests ▲ 8 Tests × 16 Tests ■ Random Search

	2 Tests	*4 Tests*	*8 Tests*	*16 Tests*	*Random Search*
< 1 Test	0.9157	**0.9999**	**0.9999**	**0.9999**	**~1.0**
< 2 Tests	——	0.9775	**0.9999**	**0.9999**	**~1.0**
< 4 Tests	——	——	0.9430	**0.9999**	**~1.0**
< 8 Tests	——	——	——	0.9807	**~1.0**
< 16 Tests	——	——	——	——	**0.9999**

Table 4: P-values of differences between different numbers of tests on the DE Lennard-Jones problem. Bold values indicate statistically significant results. < means "is worse than".

parameter B; or the absence of A will make B moot. For example, there's no reason to have a "per-gene crossover probability" parameter unless the "crossover type" parameter has been set to "uniform". Additionally, the same parameter may have different values in different contexts in the same evolutionary algorithm. This is particularly common when parameters define the breeding pipeline for the algorithm. For example, consider a pipeline where tournament selection feeds into polynomial mutation, which provides one child for uniform crossover, the other child being provided by *another* tournament selection (with a different tournament size). Such a parameter tree might look like:



This tree structure is, we believe, a major reason why many evolutionary computation implementations define their parameters using a hierarchical rather than flat namespace.

We will not here attempt the daunting task of evolving tree-structured parameters, but we bring this up to point out that, when flattened into vectors, the naturally tree-

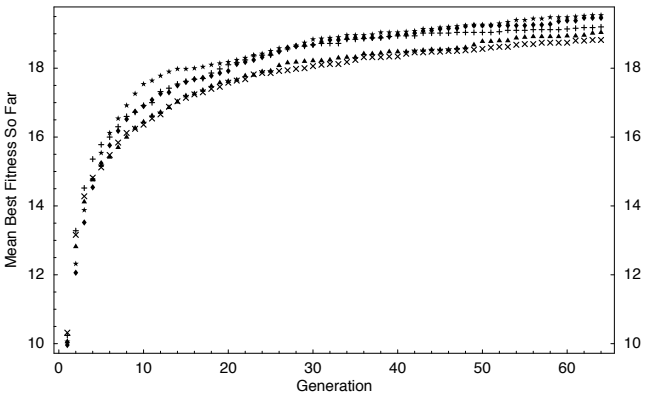


Figure 5: Meta-level mean best-fitness-so-far for the Leading Ones Blocks Problem (GA, maximization), with various levels of μ. ♦ $\mu = 16$ ★ $\mu = 8$ + $\mu = 4$ ▲ $\mu = 2$ × $\mu = 1$ Note that "♦ $\mu = 16$" is the same as "♦ 1 Test" from Figure 1.

	$\mu = 8$	$\mu = 4$	$\mu = 2$	$\mu = 1$
$< \mu = 16$	0.7424(>)	0.9799	**0.9998**	**0.9993**
$< \mu = 8$	——	0.7085	0.9925	0.9708
$< \mu = 4$	——	——	0.9490	0.8135
$< \mu = 2$	——	——	——	0.4470

Table 5: P-values of differences between different levels of meta-level μ on the GA Leading Ones Blocks problem. Bold values indicate statistically significant results. < means "is worse than". (>) means that the result is not worse (the default) but is in fact better. Note that "$\mu = 16$" is the same as "1 Test" from Table 1.

structured parameter space produces genes with a very high degree of epistasis, as some parameters are only active if other parameter are set to the right values.[3]

Our benchmark problems exhibited this several times: the GA's per-gene mutation probability; ES's alternative polynomial version, polynomial distribution index, and gaussian standard deviation; and DE's FNOISE and PF parameters. Such epistasis is one source of "weak" local optima in the parameter space. For example, uniform mutation with the right probability setting might be globally optimal, but if the genome is saddled with a poor per-gene probability setting, one- or two-point crossover will perform better. We have observed the meta-EA getting trapped in such local optima.

8. THIRD EXPERIMENT

We know that each of the parameter spaces has *at least* a few local optima due to the high epistasis among the parameters. Further, the space is quite noisy. However we still supposed that the underlying space was fundamentally simpler in nature than the complex test functions we chose. If so, it might be susceptible to a more exploitative hill-climber than the 1-Test result, which had used a population size of 128 and $\mu = 16$. We could adjust exploitation by changing the selection pressure, or the mutation rate, or both. Because the parameter genes are heterogeneous it is not simple to vary their mutation rates in a principled fashion, but we can easily change the selection pressure, in the form of μ. We compared 1-Test at $\mu = 16$ against the same approach at $\mu = 8, 4, 2$, clear down to 1, the most exploitative. We selected the GA Leading Ones Blocks benchmark as our target problem as it had the fewest parameters (1) in epistasis.

Results. We again verified statistical significance using a nonparametric ranked two-tailed T-test with a Bonferroni correction, adjusting the p value from 0.95 to 0.99925.

The results are shown in Figure 5, and Table 5 shows the p-values for comparison between various sizes of μ.

Conclusions. We had supposed that, assuming the space was relatively simple, a more exploitative technique would perform better. But the trend seems to be the opposite: as μ decreases, so does performance. This could be either due to a more complex space than we had expected, or (we think more likely) the high noise in the function. Even so, the difference is surprisingly small, and only $\mu = 16$ is statistically significantly different from $\mu = 2$ and $\mu = 1$.

9. CONCLUSIONS AND FUTURE WORK

The goal of much meta-evolutionary algorithm research has been to find optimal parameters: but we have identified a common situation, in massively parallel computing, where a meta-evolutionary algorithm is helpful to find an optimal solution directly.

One commonly assumed problem with meta-EAs is their noisy fitness functions. We tested the approach on three benchmark problems and varying numbers of tests to reduce the noise, and found that in fact no noise reduction works best. We also noted that meta-EAs have high epistasis between certain genes because of the nature of the parameter search space in modern evolutionary algorithms. This will produce certain known local optima, but despite this we presumed that the meta-EA search space is generally simpler than the underlying lower-level EA search space. Under this assumption, we tested whether a more exploitative approach might perform better, but found that in fact the opposite was true, thus suggesting that the search space may hold more surprises yet.

As future work, we hope to perform a full sweep of the meta-level search space, though we have so far found ourselves stymied by the noise and sheer number of evaluations involved. Additionally, we would like to extend our analysis of the search space to ask whether meta-meta-EAs and meta-meta-metaEAs etc. continue to face complex spaces (a question also asked, but not answered, in [24]). That is, whether the meta-EA scenario really is, to use the famous phrase, "turtles all the way down".

10. ACKNOWLEDGMENTS

This work was funded by NSF Grant 0916870.

11. REFERENCES

[1] B. Adenso-Diaz and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Oper. Res.*, 54(1):99–114, Jan. 2006.

[2] T. Bäck. Parallel optimization of evolutionary algorithms. In *PPSN*, pages 418–427, 1994.

[3] Genetic programming is the obvious way to evolve tree-structured parameters. But we are not aware of an attempt to tackle entire parameter sets with GP, though there is work in using GP to adapt new breeding operators [14, 35, 20], usually through co-evolution.

[3] M. Biazzini, B. Banhelyi, A. Montresor, and M. Jelasity. Distributed hyper-heuristics for real parameter optimization. In *GECCO*, pages 1339–1346, 2009.

[4] W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Eastern Joint IRE-AIEE-ACM Computer Conference*, pages 225–232, 1959.

[5] J. Branke and J. A. Elomari. Meta-optimization for parameter tuning with a flexible computing budget. In *GECCO*, pages 1245–1252, 2012.

[6] J. Brest, S. Greiner, B. Boskovic, M. Mernik, and V. Zumer. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE Transactions on Evolutionary Computation*, 10(6):646–657, 2006.

[7] C. Candan, A. Goeffon, F. Lardeux, and F. Saubion. A dynamic island model for adaptive operator selection. In *GECCO*, pages 1253–1260, 2012.

[8] D. J. Cavicchio, Jr. *Adaptive Search Using Simulated Evolution*. PhD thesis, Computer and Communication Sciences Department, University of Michigan, 1970.

[9] J. Clune, S. Goings, B. Punch, and E. Goodman. Investigations in meta-GAs: panaceas or pipe dreams? In *GECCO*, pages 235–241, 2005.

[10] P. I. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *Third International Conference on Practice and Theory of Automated Timetabling*, pages 176–190, 2001.

[11] K. Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, 2001.

[12] K. Deb and S. Agrawal. A niched-penalty approach for constraint handling in genetic algorithms. In *International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)*, pages 235–243, 1999.

[13] J. Denzinger, M. Fuchs, and M. Fuchs. High performance atp systems by combining several ai methods. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 102–107, 1997.

[14] B. Edmonds. Meta-genetic programming: Co-evolving the operators of variation. *Turkish Journal of Electrical Engineering and Computer Sciences*, 9:13–29, 2001.

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

[16] D. Fogel, L. Fogel, and J. Atmar. Meta-evolutionary programming. In *Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 540–545, 1991.

[17] O. Francois and C. Lavergne. Design of evolutionary algorithms: A statistical perspective. *IEEE Transactions on Evolutionary Computation*, 5(2):129–148, 2001.

[18] J. Grefenstette. Optimization of control parameters for genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):122–128, 1986.

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

[20] K. Harrington, L. Spector, J. Pollack, and U.-M. O'Reilly. Autoconstructive evolution for structural problems. In *Workshop for the Automated Design of Algorithms, GECCO*, 2012.

[21] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In *GECCO*, pages 271–278, 2009.

[22] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *J. Artif. Int. Res.*, 36(1):267–306, Sept. 2009.

[23] D. Izzo, M. Rucinski, and C. Ampatzis. Parallel global optimisation meta-heuristics using an asynchronous island-model. In *CEC*, pages 2301–2308, 2009.

[24] K. D. Jong. Parameter Setting in EAs: a 30 Year Perspective. In F. G. Lobo, C. F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, pages 1–18. Springer, 2007.

[25] A. J. Keane. Genetic algorithm optimization of multi-peak problems: studies in convergence and robustness. *Artificial Intelligence in Engineering*, 9(2):75–83, 1995.

[26] S. Luke. ECJ 20: An Evolutionary Computation Library in Java. Open source software. Available at http://cs.gmu.edu/~eclab/projects/ecj/, 2012.

[27] R. Mercer and J. Sampson. Adaptive search using a reproductive metaplan. *Kybernetes*, 7:215–228, 1978.

[28] D. Ouelhadj, S. Petrovic, and E. Ozcan. A multi-level search framework for asynchronous cooperation of multiple hyper-heuristics. In *GECCO Late-Breaking Papers*, pages 2193–2196, 2009.

[29] M. E. H. Pedersen and A. J. Chipperfield. Simplifying particle swarm optimization. *Applied Soft Computing*, 10(2):618–628, 2010.

[30] K. Price, R. Storn, and J. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer, 2005.

[31] P. Ross. Hyper-heuristics. In E. K. Burke and G. Kendall, editors, *Search Methodologies*, pages 529–556. Springer, 2005.

[32] S. Smit and A. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *CEC*, pages 399–406, 2009.

[33] F. Stonedahl and S. H. Stonedahl. Heuristics for sampling repetitions in noisy landscapes with fitness caching. In *GECCO*, pages 273–280, 2010.

[34] K. Sullivan, S. Luke, C. Larock, S. Cier, and S. Armentrout. Opportunistic evolution: efficient evolutionary computation on large-scale computational grids. In *GECCO Conference Companion*, pages 2227–2232, 2008.

[35] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In *Advances in Genetic Programming 2*, pages 45–68, 1996.

[36] D. J. Wales and J. P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms. *Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.