To Appear In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2016, Denver, CO) 2016.

Estimating the Advantage of Age-Layering in Evolutionary Algorithms

Hormoz Shahrzad[1], Babak Hodjat[1], and Risto Miikkulainen[1,2]

[1]Sentient Technologies, Inc.
[2]The University of Texas at Austin
babak,hormoz,risto.miikkulainen@sentient.ai

ABSTRACT

In an age-layered evolutionary algorithm, candidates are evaluated on a small number of samples first; if they seem promising, they are evaluated with more samples, up to the entire training set. In this manner, weak candidates can be eliminated quickly, and evolution can proceed faster. In this paper, the fitness-level method is used to derive a theoretical upper bound for the runtime of $(k+1)$ age-layered evolutionary strategy, showing a significant potential speedup compared to a non-layered counterpart. The parameters of the upper bound are estimated experimentally in the 11-Multiplexer problem, verifying that the theory can be useful in configuring age layering for maximum advantage. The predictions are validated in a practical implementation of age layering, confirming that 60-fold speedups are possible with this technique.

Keywords

Fitness approximation; Performance Analysis; Age Layering

1. INTRODUCTION

The most computationally expensive part of running an evolutionary algorithm is the fitness evaluation. In many domains it requires extensive computations, perhaps running a simulation, or even interacting with a physical system. Moreover, most of such evaluations are spent on candidates that will be promptly discarded. In order for the evolutionary operators to be creative, i.e. to discover unexpected and novel solutions, they will have to generate many candidates that turn out to be evolutionary dead ends [23, 26]. If it was possible to identify and discard such candidates quickly, evolutionary algorithms could be much more efficient.

Many methods have been developed with this goal in mind. Most of them are based on approximating the fitness of a candidate quickly, and spending full fitness evaluations only on a few selected candidates (see [11] for a review). *Age layering* [8, 9] is a particularly simple and general such method that has been shown effective in particular in domains where fitness approximation is possible through an increasing number of samples. Each new candidate is first evaluated with a small set of samples. If its performance is good on that set, it will be evaluated on a further set of samples in the next layer; if not, it will be discarded. In this manner, bad candidates are eliminated quickly, speeding up evolution.

Age-layering has been found to work well experimentally [8, 21]. However, in order to build practical applications using it, and to improve it further, it is helpful to understand how and why it works, how its parameters affect its performance, and what its expected speedup would be on a given problem. Providing such guidelines for age layering is the goal of this paper.

First, an upper bound for the runtime of an age-layered evolutionary strategy will be derived theoretically, using the well-known fitness-level method of analysis [14, 25]. Second, the parameters of this bound will be approximated experimentally on an example problem: the 11-Multiplexer. The idea is that the theory can then inform how age-layering should be set up to obtain maximum benefit. This approach indeed leads to insights: Age-layering balances the savings of discarding bad candidates early with the cost of having to generate more good candidates (because sometimes good candidates are discarded prematurely). Furthermore, there is a sweet spot where these two factors are balanced and where the benefit from age layering is the greatest.

Third, these conclusions are validated empirically on a practical implementation of an age-layered evolutionary strategy [21]. The empirical results are consistent with the theory, confirming both the upper bound and the sweet spot, and suggesting that the theoretical results can be used as a guideline to get maximum benefit from age layering in practice. The paper thus confirms age layering as a robust and effective fitness-estimation technique, suggesting that 60-fold speedups are possible in practice.

Below, related work in fitness estimation methods is first briefly discussed. The fitness-level method for theoretical runtime analysis of evolutionary algorithms is reviewed, and the age-layering method defined. A theoretical runtime upper bound for $(k+1)$ age-layered evolutionary strategy is derived, and then approximated in order to make it possible to instantiate it with empirical parameter estimates from the 11-Multiplexer domain. The conclusions are then validated in a practical implementation of evolutionary strategy in this domain.

GECCO '16, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908812.2908911

2. BACKGROUND

Related work on efficient fitness estimation is first discussed and the fitness-level method of analyzing evolutionary algorithms reviewed, The age-layered algorithm is then outlined and the terminology and notation used throughout the paper presented.

2.1 Fitness Estimation

Several techniques have been developed to make fitness evaluation more efficient. Most of these are based on some way of approximating fitness in a more efficient manner [11]. Perhaps the most versatile of these, if knowledge intensive, is surrogate optimization: Instead of evaluating candidates on the actual fitness function, a surrogate function is used that is faster to compute [12]. The crucial candidates are then tested periodically for their actual fitness. Optimization of physical systems can often be done in this manner [5, 6, 24], and it is a crucial component of bilevel optimization as well [15, 22].

However, surrogate fitness functions may not always be available, or they may not be fast or accurate enough. A more general approach is to use the parent's performance as a surrogate: It is always available, fast, and usually a good model (since only the best candidates are selected as parents). A good strategy therefore is to generate many offspring and evaluate fully only those that behave similarly to their parents [16]. An obvious disadvantage is that sometimes also good candidates, which may be more innovative and thereby different from the parents, are thrown away prematurely.

A third approach is to estimate the fitness of a candidate based on fitnesses around it, that is, the fitness of its parents, associated candidates, or a small sample population [1, 17, 20]. It may also be possible to predict fitness based on fitness history and types of evolutionary operations used [7, 13]. While these methods are helpful in many cases, in more complex problems it is difficult to predict fitness in this manner [4].

This paper focuses on age layering [8, 9], a simple yet effective method where fitness estimation is based on partial computation of the fitness function itself. It is most appropriate where the fitness calculation is based on a number of samples, so that it can be estimated with few samples quickly. Age layers are thus defined on the number of samples over which the candidate has been evaluated. Note that this technique is distinctly different from similarly named Age-Layered Population Structure (ALPS) method [10]. ALPS partitions populations into layers according to generations, with the main goal of maintaining diversity. Age layering in this paper is more closely related to the Early Stopping method in evolutionary robotics, where a complex evaluation is terminated if it is guaranteed not to produce offspring even if evaluated fully [3, 18]. However, early stopping methods are specifically designed for a particular robot and evaluation task. Of the fitness estimation methods, age layering is therefore arguably the most straightforward and general: It is based simply on reliability of the fitness estimates, i.e. the number of samples used to evaluate the candidates. It will therefore be the focus of the analysis in this paper.

2.2 Fitness-Level Method

The *fitness-level method* [14, 25] for analyzing the runtime of evolutionary algorithms splits the search space of a given problem into m partitions $P_1, P_2, \ldots, P_m$ such that $P_i <_f P_{i+1}, \forall i \in [1; m-1]$, and P_m contains the global optima of the problem. That is, the fitness of the elements in P_i is lower than the fitness of all the elements belonging to the partitions $P_{i+1}, \ldots, P_m$.

Let us assume that the probability of generating an improvement from an element in partition P_i (that is, of generating an element in a partition P_h where $h \in [i+1..m]$), is bound from below by $s_i > 0$. Then, an upper bound on the expected number of evaluations required to reach a global optimum can be obtained [14, 25] as:

$$n = \sum_{i=1}^{m-1} \frac{1}{s_i}. \quad (1)$$

Note that the number of evaluations is used as proxy for the running time of the algorithm.

For simplicity, the fitness-level method is usually formulated for the $(1 + 1)$ Evolutionary Strategy [14, 25]. This way, the number of evaluations equals the number of generations. However, the same analysis extends to populations as well, and parallel islands of populations, as long as the strategy is elitist, i.e. the best fitness value in the population can only increase because the best candidates are retained in the population. In the following, such an extension is introduced where the population consists of a maximum of k elite candidates organized into age layers.

2.3 Age-Layering Method

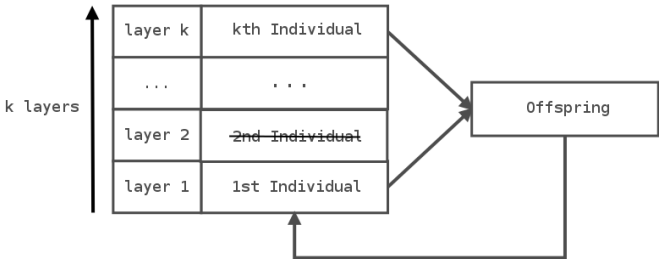
The age-layering method maintains layers with candidates evaluated on different numbers of samples. The general idea is to evaluate only promising candidates on all the available training data, and to discard poorly performing candidates early on. This way, these bad candidates are discarded after a few partial evaluations, and the total number of fitness evaluations is reduced.

Let us define *age* as the number of fitness samples upon which a candidate has been validated. The population is structured into *layers* that correspond to different discrete ages. The result is a *layered population* in which candidates are sorted by the amount of data used for their evaluation. More specifically, the minimum number of samples for estimating fitness is a, also called the maturity age (note that generally $a > 30$ in order to obtain statistically reliable estimates). A candidate in layer k has thus been evaluated on $k \times a$ samples.

Figure 1 illustrates the simplest layered algorithm, a $(k+1)$ Layered Evolutionary Strategy, or $(k+1)$ LES, where a maximum of k candidates of the population are distributed into k layers. Informally, the process is initialized by setting the current population fitness to 0, generating the first candidate randomly, and placing it in the first age layer. In every generation thereafter, three steps take place:

Aging of the current population: Each candidate currently in the population in layers $1..j-1$ is evaluated with a new samples and its fitness is updated. If this fitness is worse than the current population fitness, the candidate is discarded; otherwise it is advanced to the next age layer.

Updating the population fitness: If there is a candidate at layer k and its fitness is greater than the current population fitness i, the population fitness is updated; if further that fitness is the optimal fitness, the



(*a*) $(k+1)$ LES Schematic Diagram

1. Set current population fitness level $i = 0$
2. Create initial candidate randomly and place in age layer 1
3. For age layers $j = 1$ to $k-1$
   If there is a previous candidate in layer j
   Evaluate it with a samples
   If candidate's fitness level $\geq i$
   Advance candidate from layer j to $j+1$
   else
   Remove the candidate
4. If a candidate was promoted to layer k
   Remove the previous candidate in layer k
   Update $i :=$ candidate's fitness level
   If $i = m$
   Return the candidate in layer k as the solution
5. Create new candidate from population and place in layer 1
6. Go to 3.

(*b*) $(k+1)$ LES Pseudocode

Figure 1: An illustration of the $(k+1)$ Layered Evolutionary Strategy (LES). In this minimal form of age layering, a maximum of k candidates in the population are divided into k age layers. At each generation, a single offspring is generated from the current population and entered in the first age layer. As candidates are evaluated with more samples, more confidence is gained in their fitness estimate, and they move up in layers, If a candidate reaches the top layer, we have its true fitness, which is then used to discard candidates whose estimate falls below the top fitness (like the hypothetical 2nd candidate in subfigure a). In this manner, bad candidates can be eliminated without spending much time evaluating them, speeding up evolution in general.

algorithm returns that candidate as the solution.

Generation of a new candidate: One new candidate is generated from the entire population and placed in the first age layer.

The set of a samples is chosen randomly among those that the candidate has not seen before. Thus, as candidates are promoted to upper layers, further partial evaluations are performed and, as a consequence, the fitness estimation noise is reduced. At layer k, the candidate has been evaluated on all samples and its fitness is the true fitness. A new candidate can be generated from the current population by any of the usual selection, crossover, and mutation methods. The population always has at least one member and can have as many as k members.

This algorithm can be extended to consider several candidates per layer and to distributed islands of populations. The following analysis, however, focuses on the $(k+1)$ LES for simplicity.

2.4 Terminology

The following terminology will be used throughout the paper:

- *Fitness level*: one of the buckets in the partitioning of fitness values into discrete ranges.
- *Population fitness*: fitness level of the current candidate in the top age layer k.
- *Sample*: a subset of the dataset $(X^1, Y^1), \ldots, (X^N, Y^N)$, where Y is the correct output for the input X.
- *Partial evaluation*: a fitness evaluation based on a set of samples smaller than the entire dataset.
- *Estimated fitness*: the fitness assigned to an candidate as a result of a partial evaluation.
- *True fitness*: the fitness of an candidate evaluated on all available samples.
- *Age*: number of data subsets upon which a given candidate has been evaluated.
- *Age layer*: set of candidates with the same age.
- *Good candidate*: one with true fitness at or above the current population fitness, i.e. one that should be advanced to the next age layer.
- *Bad candidate:* one with true fitness below the current population fitness, i.e. one that should be discarded.

The notation used in the analysis is adopted from earlier work [14], and augmented with age-layering parameters:

- m is the number of partitions or fitness levels.
- s_i is the lower-bound probability of finding an improvement from a candidate in partition P_i.
- n is the number of fitness evaluations required to reach a global optimum of the problem.
- N is the number of samples in the dataset.
- k is the number of age layers in which the population of the algorithm is divided.
- a is the number of samples used at each partial evaluation, i.e. the maturity age.
- p is the upper bound of the probability of a partially evaluated bad candidate advancing to the next age layer.
- c is the upper bound of the probability of a partially evaluated good candidate advancing to the next age layer.

3. ANALYSIS OF AGE LAYERING

In this section, a theoretical upper bound for the runtime of $(k+1)$ LES is derived. Conclusions about how, why, and when it works well are then drawn in the 11-Multiplexer domain. The results are validated in comparison with experimental runs of a practical implementation of LES on this domain.

3.1 An Upper Bound for (k+1) LES

First, let us count how many evaluations are needed without age layering, or equivalently, when each candidate is evaluated at all k layers. At each layer, each candidate is

evaluated with a samples (called maturity age), and thus all $N = ak$ fitness samples in total. At each fitness level, $1/s_i - 1$ bad candidates need to be generated in average to find one good candidate that advances to a higher level. The number of evaluations is therefore bound by

$$n < \sum_{i=1}^{m-1} \frac{ak}{s_i}. \quad (2)$$

This is an upper bound estimate of the number of evaluations needed to advance the first candidate into the top fitness level in an evolutionary process that does not take advantage of age-layering.

In the age-layered algorithm, however, the candidates are not always evaluated with all ak examples: if a partial evaluation suggests that a candidate has a low fitness, it is not evaluated further. Discarding bad candidates early is desirable because it saves evaluations. On the other hand, a small number of good candidates are also likely to be discarded in this manner because they look bad (so they could be called "ugly" candidates). Therefore, in order to detect one good candidate, on average it is necessary to generate

$$g_i = \frac{1}{c_i s_i} \quad (3)$$

candidates, where c_i is the probability that an actual good candidate will advance to the next level. It is a product of probabilities that a good candidate advances through all age layers, i.e.

$$c_i = c_{i,1} c_{i,2} ... c_{i,k-1}. \quad (4)$$

Let $p_{i,j}$ be the probability that a bad candidate at level i is advanced from layer j to layer $j+1$. The expected number of evaluations spent on this candidate is

$$n_{ib} = a + ap_{i,1} + ap_{i,1}p_{i,2} + ... + ap_{i,1}p_{i,2}...p_{i,k-1} = a\hat{k}_{ib}. \quad (5)$$

Similarly, the number of evaluations spent on a good candidate is

$$n_{ig} = a + ac_{i,1} + ac_{i,1}c_{i,2} + ... + ac_{i,1}c_{i,2}...c_{i,k-1} = a\hat{k}_{ig}. \quad (6)$$

The ratio of good/bad candidates is at least $s_i/(1 - s_i)$, so the number of evaluations at level i is

$$n_i < \frac{a\hat{k}_{ib}(1 - s_i) + a\hat{k}_{ig}s_i}{c_i s_i}, \quad (7)$$

and the total number of evaluations is bound by

$$\begin{aligned} n &< \sum_{i=1}^{m-1} \frac{a\hat{k}_{ib}(1 - s_i) + a\hat{k}_{ig}s_i}{c_i s_i} \\ &= \sum_{i=1}^{m-1} \frac{a\hat{k}_{ib}(1 - s_i)}{c_i s_i} + \sum_{i=1}^{m-1} \frac{a\hat{k}_{ig}}{c_i}. \end{aligned} \quad (8)$$

Thus, the number of evaluations spent to find a good candidate is the sum of those for the bad and the ugly.

3.2 Theoretical Advantage

In order to validate the bound it needs to be turned into a form where its parameters $p_{i,j}$, $c_{i,j}$, and s_i can be estimated experimentally. The first insight is that $p_{i,j}$ and $c_{i,j}$ derive from the accuracy of estimation, and therefore they can be assumed to be the same at all levels i. Thereby the first subscript can be dropped, and p_j and c_j estimated in a Monte Carlo simulation instead of having to run evolution to create fitness levels. That is,

$$\hat{c} = c_1 c_2 ... c_{k-1}, \quad (9)$$

$$\hat{k}_b = 1 + p_1 + p_1 p_2 + ... p_1 p_2 ... p_{k-1}, \quad (10)$$

$$\hat{k}_g = 1 + c_1 + c_1 c_2 + ... c_1 c_2 ... c_{k-1}. \quad (11)$$

In such a simulation, first a large number of candidates are generated randomly; then, for different values of k, the probability that candidate fitness is overestimated at different age layers is measured as p_j. Assuming fitness is scaled between [0..1], overestimation occurs when

$$\lfloor kf_{\mathrm{E}} \rfloor > \lfloor kf_{\mathrm{T}} \rfloor, \quad (12)$$

where f_{E} is the estimated fitness and f_{T} the true fitness of the candidate. Note that estimating p_j in this manner gives us an upper bound: It is based on overestimation of at least one level, whereas in actual evolution, candidates generated at level i may have fitness lower than i, requiring overestimation of more than one level. Similarly, the probability that a candidate fitness is underestimated is $1 - c_j$, and an upper bound for it obtained analogously to Equation 12.

In general it is difficult to estimate the s_i. However (after first applying equations 9, 10, and 11), equation 8 can be rewritten as

$$n < \frac{a\hat{k}_b}{\hat{c}} \sum_{i=1}^{m-1} \frac{1}{s_i} + (m-1)\frac{a(\hat{k}_g - \hat{k}_b)}{\hat{c}}. \quad (13)$$

The $\sum_{i=1}^{m-1} \frac{1}{s_i}$ can then be estimated from converged non-layered evolutionary runs by

$$\sum_{i=1}^{m-1} \frac{1}{s_i} < g_n/N, \quad (14)$$

where g_n is the number of candidates generated in that run, and N is the number of fitness levels (which is equivalent to the number of samples).

To test these ideas, the 11-Multiplexer problem was used as the test domain. Multiplexer functions have long been used to evaluate machine-learning methods because they are difficult to learn but easy to check. In general, the input to the multiplexer function consists of u address bits A_v and 2^u data bits D_v, i.e. it is a string of length $u + 2^u$ of the form $A_{u-1}...A_1A_0D_{2^u-1}...D_1D_0$. The value of the multiplexer function is the value (0 or 1) of the particular data bit that is singled out by the u address bits. For example, for the 11-Multiplexer, where $u = 3$, if the three address bits $A_2A_1A_0$ are 110, then the multiplexer singles out data bit number 6 (i.e. D_6) to be its output.

A Boolean function with $u+2^u$ arguments has 2^{u+2^u} rows in its truth table. Thus, the sample space for the Boolean multiplexer is of size 2^{u+2^u}. When $u = 3$, the search space is of size $2^{2^{11}} = 2^{2048} \approxeq 10^{616}$. However, since evolution can also generate redundant expressions that are all logically equal, the real size of the search space can be much larger, depending on the representation.

Following prior work on the 11-Multiplexer problem [21], a rule-based representation was used where each candidate specifies a set of rules of the type

$$< \text{rule} > ::= < \text{conditions} > \rightarrow < \text{action} > .$$

The conditions specify values on the bit string and the action identifies the index of the bit whose value is then output. For

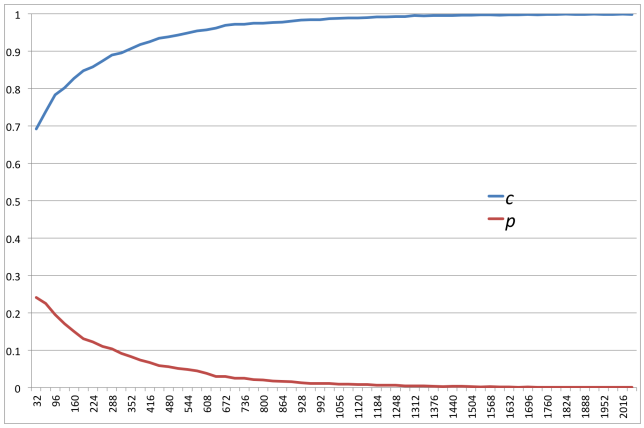


Figure 2: Monte Carlo estimation of p_j and c_j with $a = 32$ and $k = 64$ in the 11-Multiplexer problem, at different age layers; the number of samples in each layer is shown in the x-axis. Over- and underestimation is likely in the early age layers, but becomes negligible at higher layers.

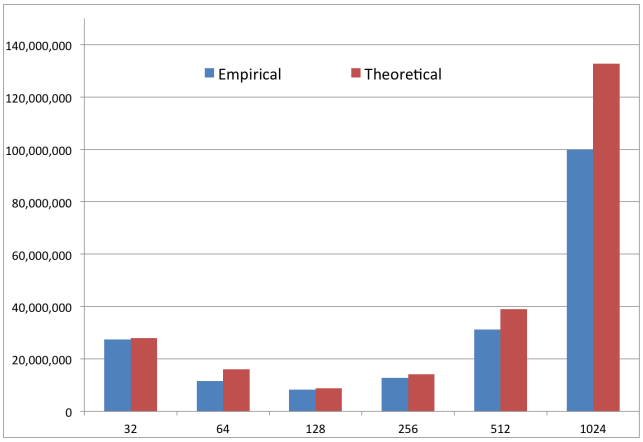


Figure 3: Number of evaluations needed to solve the 11-Multiplexer problem using LES, estimated both theoretically and empirically. The experimental results are averages over 10 runs; the number of samples in each age layer is shown in the x-axis. A non-age-layered ES (not shown) requires over $522,649,600$ evaluations; age layering is thus a significant advantage in all cases. The best theoretical value, $8,804,207$ is obtained when $a = 128$, resulting in a 60-fold speedup. The experimental results are based on a practical implementation of LES; the upper bound is still valid (and tight), and the same $a = 128$ results in best performance of $8,294,400$ and a 63-fold speedup. These results suggest that the theory is indeed useful in configuring LES in practice.

instance, the following rule outputs the value of data bit 6 when the first three bits are 110:

$$< A_0 = 0 \ \& \ A_1 = 1 \ \& \ !A_2 = 0 > \rightarrow \ D_6.$$

These rules are evolved through the usual genetic operators in genetic programming [2].

To estimate the parameters of equation 13, 4000 independent candidates were created for the 11-Multiplexer problem. Their fitness was estimated at different layers (with $N = 2048$, $a = 32$ and $k = 64$) by drawing independent samples with replacement among the 2048 different cases and seeing how often they gave the correct value as output. The resulting estimates of p_j and c_j at different age layers are shown in Figure 2. They are midrange with a small number of samples, but become more extreme very quickly, and are near zero and one at the highest layers. Similarly, $\sum_{i=1}^{m-1} \frac{1}{s_i}$ was estimated by averaging equation 14 over 10 successful non-layered evolutionary runs, obtaining an estimate of 255,200.

Figure 3 shows (in red bars) the theoretical upper bound thus estimated for different values of a. These bounds are much lower than the corresponding upper bound for non-layered evolution with the same s_i, which (following equation 2) is $2048 \times 255{,}200 = 522{,}649{,}600$. The best advantage occurs in the midrange of a, i.e. at 128. This result suggest that if a is chosen properly, significant speedups are possible in theory. The next question is: Does this prediction hold in practice?

3.3 Empirical Advantage

An interesting validation of the theory is to compare its predictions with results with a practical implementation of LES, such as the EC-Star platform for evolutionary computation [9, 19, 21]). In its most general form, EC-Star implements a distributed version of LES on multiple population islands. For this comparison, a single-population version of LES was created in it. This algorithm differs from the theoretical $(k + 1)$ LES in three ways: First, it utilizes a larger population of candidates: Instead of one offspring, q_1 offspring are generated for Layer 1 in Steps 2 and 5 of the algorithm (Figure 1*b*), and instead of one candidate in each age layer, an elite population of $q_2 < q_1$ candidates are maintained in total over all layers. Second, instead of aging candidates using a previously unseen samples, those samples are selected randomly (with replacement) from the entire dataset for each generation. Third, instead of a global fitness level i, separate fitness levels are maintained for each age layer; the process stops when fitness for layer $k = m$. These three features were found experimentally to be useful in EC-Star and were thus retained in the comparison. An interesting question is: Does the upper bound still hold, and can it be used to determine good settings for the parameters such as a (or equivalently, k)?

In the experiments, this practical implementation of LES was run with parameters found to be effective in prior work with EC-Star [21]: The population size was 4000. In each generation, the top 8% of the population (based on estimated fitness in all age layers) was passed on to the next generation. The top 20% of the population was used to generate 89% of the offspring, with 11% of it generated randomly. The population was aged once in each such generation.

Figure 3 shows (in blue bars) the number of evaluations for such evolutionary runs on the 11-Multiplexer problem, averaged over 10 runs. These experimental results validate the theory: The upper bound applies to the experimental runs and is quite tight. The best performance is obtained

with $a = 128$, resulting in a 60-fold speedup over non-age-layered evolution, as predicted by the theory. The results thus suggest that the theory is useful in configuring practical implementations of LES, and that the EC-Star platform is a good way to implement it.

4. FUTURE WORK

It is possible to extend this work in several ways. First, it may be possible to extend the theory to take more of the practical constraints into account. For instance, population size limitations make it more difficult for new candidates to enter the population. Evolutionary selection biases new candidates towards higher fitness, which should be taken into account in the Monte Carlo estimates. Such extensions should make the bounds even more tight, but they should also lead to practical guidelines on how those parameters should be set for best performance.

Another interesting direction is to extend the theory to distributed implementations of evolutionary computation such as the distributed version of EC-Star [19]. Based on the fitness-level method, upper bounds have been derived to several problems and parallel topologies [14]; the age-layered extension could be applied to those as well. Assuming a fully connected topology, the idealized version developed in this paper is straightforward to extend, but the practical constraints of managing the layers in the server are more challenging.

Third, it would be interesting to apply the techniques developed in this paper to other problems. They should work as is on many problems where fitness is based on sampling, such as the medical informatics task where age-layering was first developed [8, 9]. An interesting issue is how well the Monte Carlo simulation will work on problems with much larger and more structured search space. If such a simulation is available, the payoffs can be significant.

5. CONCLUSION

Age layering is a simple and general method for improving the runtime of evolutionary algorithms. It allows allocating evaluation cycles where it matters, by making it possible to identify and discard bad candidates quickly. This paper gives theoretical insight into this process, showing that the method balances the savings on bad candidates with the cost of having to generate more good ones. Moreover, a sweet spot exists between these factors where the age layering performs the best. These factors can be characterized by estimating the parameters of the theory experimentally, and using the theory to draw conclusions. A comparison with a practical implementation of layered evolutionary strategy demonstrates that these conclusions are valid, and 60-fold speedups are possible in practice. Age layering is thus a promising approach to improving performance of evolutionary algorithms.

References

[1] M.-R. Akbarzadeh-T, I. Mosavat, and S. Abbasi. Friendship modeling for cooperative co-evolutionary fuzzy systems: A hybrid GA-GP algorithm. In *Proceedings of the 22nd International Conference of North American Fuzzy Information Processing Society*, pages 61–66, 2003.

[2] F. J. Berlanga, A. Rivera, M. J. del Jesús, and F. Herrera. Gp-coach: Genetic programming-based learning of compact and accurate fuzzy rule-based classification systems for high-dimensional problems. *Information Sciences*, 180(8):1183–1200, 2010.

[3] J. C. Bongard and G. S. Hornby. Guarding against premature convergence while accelerating evolutionary search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2010.

[4] E. Ducheyne, B. De Baets, and R. deWulf. Is fitness inheritance useful for real-world applications? In *Evolutionary Multi-Criterion Optimization*, volume LNCS 2631, pages 31–42. Springer, Berlin, 2003.

[5] D. Floreano and J. Urzelai. Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13:431–4434, 2000.

[6] A. I. Forrester and A. J. Keane. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45:50–79, 2009.

[7] A. Gaspar-Cunha and A. Vieira. A multi-objective evolutionary algorithm using neural networks to approximate fitness evaluations. *International Journal of Computers, Systems and Signals*, 6:18–36, 2005.

[8] E. Hemberg, M. Wagy, F. Dernoncourt, K. Veeramachaneni, and U. M. O'Reilly. Imprecise selection and fitness approximation in a large-scale evolutionary rule based system for blood pressure prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2013.

[9] B. Hodjat and H. Shahrzad. Introducing an age-varying fitness estimation function. In R. Riolo, E. Vladislavleva, M. D. Ritchie, and J. H. Moore, editors, *Genetic Programming Theory and Practice X*, pages 59–71. Springer, New York, 2013.

[10] G. S. Hornby. ALPS: The age-layered population structure for reducing the problem of premature convergence. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 815–822, 2006.

[11] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computation*, 9:3–12, 2005.

[12] Y. Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1:61–70, 2011.

[13] A. Kosorukoff. Using incremental evaluation and adaptive choice of operators in a genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 688–694, 2002.

[14] J. Lässig and D. Sudholt. General upper bounds on the runtime of parallel evolutionary algorithms. *Evolutionary Computation*, 22:405–437, 2013.

[15] J. Z. Liang and R. Miikkulainen. Evolutionary bilevel optimization for complex control tasks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2015)*, Madrid, Spain, July 2015.

[16] P. McQuesten and R. Miikkulainen. Culling and teaching in neuro-evolution. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA-97, East Lansing, MI)*, pages 760–767. San Francisco: Morgan Kaufmann, 1997.

[17] R. Myers and D. Montgomery. *Response Surface Methodology*. Wiley, New York, 1995.

[18] S. Nolfi and D. Floreano. *Evolutionary Robotics*. MIT Press, Cambridge, 2000.

[19] U.-M. O'Reilly, M. Wagy, and B. Hodjat. EC-Star: A massive-scale, hub and spoke, distributed genetic programming system. In R. Riolo, E. Vladislavleva, M. D. Ritchie, and J. H. Moore, editors, *Genetic Programming Theory and Practice X*, pages 73–85. Springer, New York, 2013.

[20] M. Salami and T. Hendtlass. A fast evaluation strategy for evolutionary algorithms. *Applied Soft Computing*, 2:156–173, 2003.

[21] H. Shahrzad and B. Hodjat. Tackling the Boolean multiplexer function using a highly distributed genetic programming system. In R. Riolo, W. P. Worzel, and M. Kotanchek, editors, *Genetic Programming Theory and Practice XII*, pages 167–179. Springer, New York, 2015.

[22] A. Sinha, P. Malo, P. Xu, and K. Deb. A bilevel optimization approach to automated parameter tuning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014)*, Vancouver, BC, Canada, July 2014.

[23] K. O. Stanley and J. Lehman. *Why Greatness Cannot Be Planned: The Myth of the Objective*. Springer, Berlin, 2015.

[24] C. C. Tutum, K. Deb, and I. Baran. Constrained efficient global optimization for pultrusion process. *Materials and Manufacturing Processes*, 30(4):538–551, 2015.

[25] I. Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. In *Evolutionary Optimization*, volume 48 of *International Series in Operations Research and Management Science*, pages 349–369. Springer, New York, 2002.

[26] D. Whitley, S. Dominic, and R. Das. Genetic reinforcement learning with multilayer neural networks. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 562–569. San Francisco: Morgan Kaufmann, 1991.