Noname manuscript No.
(will be inserted by the editor)

Evolving Mean-Update Selection Methods for CMA-ES

Samuel N. Richter · Michael Schoen · Daniel R. Tauritz

Received: date / Accepted: date

Abstract Selection functions enable Evolutionary Algorithms (EAs) to apply selection pressure to a population of individuals, whether by directly making decisions on whether an individual's genes survive, or by selecting the individuals used to update the internal variables of the EA and thus influencing its behavior. Various conventional selection functions exist, each providing a unique method of selecting individuals, typically based on fitness. However, the full space of selection functions is effectively unlimited in size, and each possible selection algorithm is optimal for some EA configuration applied to a particular function class. Therefore, improved performance can be obtained by tuning selection functions to the problem at hand, rather than employing conventional selection functions. This paper details an investigation of the extent to which performance can be improved for the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) by tuning the selection of individuals used for the mean-update algorithm. A hyper-heuristic is employed to explore the space of algorithms which select individuals from the population. We show the increase in performance obtained with a tuned selection algorithm, versus the unmodified CMA-ES mean-update algorithm. Specifically, we measure performance on instances from several real-valued benchmark function classes to demonstrate generalization of the improved performance.

Keywords First keyword · Second keyword · More

Samuel N. Richter
Missouri University of Science and Technology
Natural Computation Lab E-mail: snr359@mst.edu

Michael G. Schoen
Missouri University of Science and Technology
Natural Computation Lab E-mail: ms778@mst.edu

Daniel R. Tauritz
Missouri University of Science and Technology
Natural Computation Lab E-mail: dtauritz@acm.org

1 Introduction

The performance of an Evolutionary Algorithm (EA) is highly sensitive to the parameters used to configure it (Eiben et al., 1999). The field of automated algorithm design addresses this by exploring methods to remove the human bias in algorithm design and configuration. Hyper-heuristics, in particular, have been used to automate the generation of EA heuristics and algorithmic components, such as mutation operators, recombination operators, population sizing, and selection functions.

EAs employ selection functions to control the method by which an individual's genes are selected, for purposes such as, recombination, survival, or updating internal variables. Various conventional fitness-based selection functions exist, each providing a unique method of selecting individuals. These selected individuals may then undergo recombination and/or survival selection, or be used for some other update to the status of the population and internal variables of the EA. Often, the goal of these selection functions is to push the population of the EA towards an area of higher fitness, or to explore a region of the search space to find a potential growth direction. It follows that each selection algorithm plays a significant role in determining the behavior of the EA and the population, and thus, the average performance of the EA's search through the space of solutions (Woodward, 2010). Many selection algorithms are parameterized, allowing for further variance in the behavior they provide. In cases where parameterized selection algorithms are applied, the parameters can be carefully tuned, either manually or with tuning algorithms, to maximize the performance of an EA on a particular function or function class.

New selection algorithms can be designed in cases where the performance offered by existing algorithms is insufficient, even with well-tuned parameters. However, the full space of selection algorithms is effectively unlimited, and so it is highly unlikely that any conventionally human-designed algorithm offers the optimal selection behavior, given a specific problem. An implication of the "No Free Lunch" theorem (Wolpert et al., 1995) is that each possible selection algorithm is optimal for some EA configuration applied to a particular function class. Therefore, a performance gain is likely to be attained by exploring the space of selection algorithms to find one that offers better performance than any conventional selection algorithm. Previous work has confirmed this hypothesis, prompting our approach to use a hyper-heuristic and a custom representation of selection functions to explore the space of new selection functions (Woodward and Swan, 2011).

Our approach employs a hyper-heuristic, with both generative and perturbative elements, to explore the space of selection algorithms, with each search algorithm represented by two components. The first component is a Koza-style Genetic Programming (GP) tree (Koza, 1994), encoding a mathematical function that calculates how desirable an individual is at the current stage of evolution. The second component is a method of selecting individuals, based on how desirable they are calculated to be. We use this hyper-heuristic to

evolve a new scheme for selecting individuals in the mean update function of the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen and Ostermeier, 1996). The canonical CMA-ES has shown great success in solving a wide variety of problems, and given the improvement to a canonical EA via evolution of a new selection process shown in (Richter and Tauritz, 2018), we sought to apply the same process to CMA-ES, anticipating a similar improvement.

2 Related Work

The field of hyper-heuristics encompasses many different approaches for the automated design of new algorithms. Methods may utilize offline learning, in which computation is done *a priori* to develop a heuristic, or online learning, in which a heuristic is developed dynamically alongside a running problem. Hyper-heuristic searches can be perturbative, in which complete solutions are considered individually, or generative, in which solutions begin partially built and are extended iteratively (Burke et al., 2013). The hyper-heuristic presented in this paper is an offline-learning heuristic that combines these approaches, building one component of a selection algorithm with a generative method, and selecting another component in a perturbative manner.

A major application of hyper-heuristics is the automated design of algorithmic components, which various algorithms have been shown to benefit from. Hyper-heuristics have been used to evolve new algorithms from components of existing algorithms for Ant Colony optimization algorithms (Lopez-Ibanez and Stutzle, 2012), Boolean Satisfiability solvers (KhudaBukhsh et al., 2009), local search heuristics (Burke et al., 2012), and iterative parse trees representing Black Box Search Algorithms (Martin and Tauritz, 2013). The research described in this paper applies the same concept to selection functions, employing a hyper-heuristic to build selection functions from smaller components to search the space of new selection functions. In particular, the practice of using GP as a hyper-heuristic has been discussed in (Burke et al., 2009) and explored in a number of works (Burke et al., 2010, 2006; Harris et al., 2015).

Previous work has also focused on improvement of targeted components of EAs, including the evolution of new mutation operators (Woodward and Swan, 2012; Hong et al., 2013), mating preferences (Guntly and Tauritz, 2011), genetic representation of individuals (Scott and Bassett, 2015), and crossover operators (Goldman and Tauritz, 2011). Methods for generating selection algorithms, in particular, have been investigated. A random walk through the space of register machines that compute and return a probability of selection for each individual showed that such custom-tuned selection algorithms can outperform typical selection algorithms (Woodward and Swan, 2011). A more informed search through the space of selection algorithms may yield an even greater benefit than a random search. In the previous work involving the evolution of Black Box Search Algorithms, the parse trees include evolved selection functions, although the selection functions are limited to two

conventional selection functions (k-tournament and truncation) with evolved parameters (Martin and Tauritz, 2013). An evolutionary search through selection functions developed with Grammatical Evolution showed that better selection functions can be developed using a hyper-heuristic, and that the performance of these selection functions can generalize to new instances within the same function class (Lourenço et al., 2013). A similar hyper-heuristic search employed GP to discover new selection algorithms, again showing a generalized increase in performance for the same EA running on the same problem set (Richter and Tauritz, 2018). The work described in this paper expands on these ideas by applying the hyper-heuristic search employing GP to the mean-update function of CMA-ES, changing the method used to select the population members used to update the mean in order to tune the performance of CMA-ES for a particular function class.

To search the space of selection algorithms defined by this representation, we employ a combination of perturbative and generative hyper-heuristics. Previous work has applied such combinations of hyper-heuristic types to memetic algorithms (Krasnogor and Gustafson, 2004), EA operator control (Maturana et al., 2010), low-level heuristic management (Remde et al., 2012), and vehicle routing (Garrido and Riff, 2010).

3 Methodology

Here we discuss the methodology of our hyper-heuristic, and the meta-EA powering it. We first outline the format we use to represent selection functions in the meta-EA. The selection functions are built by a combined generative and perturbative hyper-heuristic. We then discuss how we use the meta-EA to evaluate and search for new selection functions.

3.1 Encoding Selection Functions

Most typical selection functions are formatted as a series of algorithmic steps, which take as input a population of individuals and output a subset of the individuals, as selected by the algorithm. While we could explore the entire combinatorial space of algorithmic steps to find new selection functions, doing so would generate many algorithms which are not valid selection functions, or even functional algorithms. Therefore, we need a representation of selection functions that is both robust enough to represent a wide variety of selection functions, yet constrained enough that we can effectively search within it to find new, valid selection algorithms.

To this end, we developed a generalized format to represent a selection function, which can encode both a number of traditional selection functions as well as novel selection functions. The representation consists of two major parts. The first part is a binary Koza-style GP-Tree (Koza, 1994). Rather than encoding entire programs within the GP-Tree, which could result in an infeasibly wide search space of selection algorithms (Woodward and Bai, 2009), the

GP-Tree instead encodes a mathematical function. All of the function inputs (the terminals of the GP-Tree) are real-valued numbers, and all of the operators in the GP-Tree operate on, and return, real-valued numbers. The terminals of the GP-Tree include various factors pertinent to a single individual of the population, including the individual's fitness, the individual's fitness ranking among the population members, the uniqueness of the individual's genome, and the individual's age, in generations. The possible terminal inputs also include information pertinent to the evolution at large, including the total size of the population, the current generation, the maximum and minimum fitness values in the population, and the sum of the individuals' fitness values. Constants are also included, as well as random terminals, which return a random number within a (configurable) closed range. Binary operators in the GP-Tree include various arithmetic and other mathematic functions. Refer to Table 1 for a description of the operators and Table 2 for a description of the terminal nodes. When evaluated, the mathematical function encoded by the GP-Tree returns a single real-valued number, corresponding to the relative "desirability" of the individual whose data was input into the function. This GP-Tree is built by the generative part of the hyper-heuristic, and can encompass any valid parse tree built from the available operators and terminals.

The second part of the evolved selection function is a method of selecting individuals based on their desirabilities, as calculated by the mathematical function encoded by the GP-Tree. The possible selection methods are inspired by traditional selection functions, such as truncation, tournament selection, fitness-proportional selection, and stochastic universal sampling. Some selection methods will select with replacement, allowing a single individual to be selected more than once per generation. Other methods will not select with replacement; under these methods, an individual may only be selected once per generation. See Table 3 for a description of each selection method. Pseudocode for each of these selection methods may be found in (Richter, 2019). This component is the part of the selection function built by the perturbative hyper-heuristic, being exactly one choice from a set of pre-determined methods.

To perform selection on a population, the function encoded by the GP-Tree is evaluated once for each member of the population, using the data points for that individual (fitness value, fitness ranking, etc.) as inputs to the function. The number output by the function becomes the desirability score for each individual. Finally, the selection step is used to select individuals based on the individuals' desirability scores. The selected individuals can then be used for recombination, as the survivors for the next generation, or for any other update to the internal variables that depends on a chosen subset of the population, as pertinent to the evolutionary search strategy used.

An example of this representation is shown in Figure 1. It shows an example of a GP-Tree that represents the function evaluated for each individual, as well as a final selection method used. It also indicates which portions of the function are generated by the generative and perturbative logic of the hyper-heuristic. The pseudocode for this method of selection is shown in Algorithm 1. With

Table 1 Possible GP-Tree Operators

Operator	Operands	Description
+	2	Adds the left and right operands.
−	2	Subtracts the right operand from the left operand.
×	2	Multiplies the left and right operands.
/	2	Divides the left operand by the right operand. If the right operand is 0, the left operand is instead divided by a very small number, returning a large number while preserving the sign of the left operand.
Min	2	Returns the minimum of the left and right operands.
Max	2	Returns the maximum of the left and right operands.
Step	2	Returns 1 if the left operand is greater than or equal to the right operand, and 0 otherwise.
Absolute Value	1	Returns the absolute value of the operand.

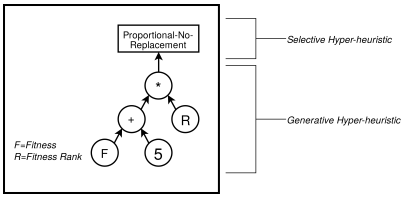

Fig. 1 Example of a generated selection function.

this selection function, the desirability of any individual is calculated as the individual's fitness rating plus 5, multiplied by the individual's ranking in the population ordered by fitness. The selection method used is Proportional-No-Replacement, so the probability of any individual being selected is directly proportional to its desirability score, and an individual cannot be selected more than once. Note that the GP-Tree is build by the generative component of the hyper-heuristic, and the selection method is chosen by the perturbative component of the hyper-heuristic.

Figure 2 shows how, for a hypothetical sample population of nine individuals, different selection functions will result in different probabilities of each individual being selected. The graph also includes the selection probabilities for the custom selection algorithm represented by the GP-Tree in Figure 1.

Table 2 Possible GP-Tree Terminals

Terminal	Description
Fitness	The individual's fitness value.
Fitness Rank	The individual's index in a list of the population members sorted by fitness, increasing.
Relative Fitness	The individual's fitness value divided by the sum of all fitness values in the population.
Birth Generation	The generation number that the individual first appeared in the population.
Relative Uniqueness	The Cartesian distance between the individual's genome and the centroid of all genomes in the population.
Population Size	The number of individuals in the population.
Min Fitness	The smallest fitness value in the population.
Max Fitness	The largest fitness value in the population.
Sum Fitness	The sum of all fitness values in the population.
Generation Number	The number of generations of individuals that have been evaluated since the beginning of evolution.
Constant	A constant number, which is generated from a uniform selection within a configured range when the selection function is generated and held constant for the entire lifetime of the selection function.
Random	A random number, which is generated from a uniform selection within a configured range every time selection is performed.

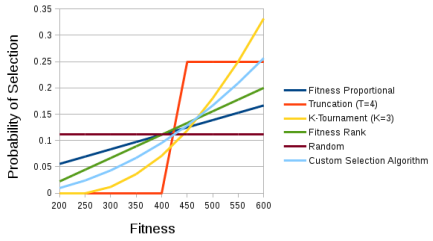

Fig. 2 A comparison of the chances that each member of a sample population, with fitnesses as listed, will be selected, under each of the typical selection strategies listed, as well as the custom selection strategy shown in Figure 1.

Table 3 Possible selection methods for evolved selection functions

Method	Description
Proportional-Replacement	A weighted random selection, with each individual's weight equal to its desirability score.
Proportional-No-Replacement	As with Proportional-Replacement, but an individual is removed from the selection pool after being selected.
k-Tournament-Replacement	A random subset of k individuals is considered, and the individual with the highest desirability score in the subset is selected.
k-Tournament-No-Replacement	As with k-Tournament-Replacement, but an individual is removed from the selection pool after being selected.
Truncation	Individuals with the highest desirability score are selected, with no individual being selected more than once.
Stochastic-Universal-Sampling	Individuals are chosen at evenly spaced intervals of their desirability scores.

3.2 Search Methodology

To develop high-quality selection functions, we need a method to search through the space of selection functions defined by the representation described in Section 3.1. We use a meta-EA to develop the selection functions, treating each complete selection function as a member of a higher-order population. After generating an initial pool of randomly constructed selection functions, the quality of each complete selection function is determined, and well-performing selection functions are chosen to recombine and mutate into new candidate selection functions.

The quality of each selection function is determined by running an underlying EA (CMA-ES) on a suite of static training instances from a benchmark function class. The underlying EA utilizes the selection function in question, and keeps all other parameters constant. The performance of the underlying EA is averaged over multiple runs, and is used to determine the quality of a selection function; selection functions that enable the EA to perform better, with all other parameters constant, are considered to be "higher-quality" selection functions. This information is fed back into the meta-EA to generate the next set of candidate selection functions. Selection functions that perform extremely poorly, because they fail to provide any selection pressure toward areas of high fitness, are pruned out of the population and never used to generate new selection functions.

Algorithm 1 An example of the pseudocode for a generated selection function. The function takes as input a population P of individuals, and a number of individuals m to be selected. Each individual p in P has member elements $p.Fitness$ and $p.FitnessRank$, encoding the individual's fitness and fitness ranking, respectively. Other generated selection functions may use additional information (see Table 2). The function returns a set of selected individuals. Note the weight calculation performed on Line 4, which is controlled by the GP-Tree encoded in the selection function. Also note that $\emptyset$ is used to denote the empty set.

```
1: function ExampleSelection(P, m)
2:     W(p) ← 0, ∀p ∈ P
3:     for all p ∈ P do
4:         W(p) ← (p.Fitness + 5) * p.FitnessRank
5:     end for
6:     w_min ← minimum(W)
7:     s ← 0
8:     for all w ∈ W do
9:         if w_min < 0 then
10:            s ← s + (w − w_min)
11:        else
12:            s ← s + w
13:        end if
14:    end for
15:    selected ← ∅
16:    for j ← 1, m do
17:        r ← random(0, s)
18:        i ← 1
19:        p_s ← P(i)
20:        while r > W(p_s) do
21:            r ← r − W(p_s)
22:            i ← i + 1
23:            p_s ← P(i)
24:        end while
25:        selected ← selected ∪ p_s
26:        P ← P − p_s
27:    end for
28:    return selected
29: end function
```

To prevent the size of the GP-Trees from growing too large, parsimony pressure is applied to the selection functions in the following manner: after the entire generation of selection functions is rated, the fitness assigned to each selection function is reduced by an amount equal to the number of nodes in the function's GP-Tree, times a parsimony coefficient c, times the difference in fitness between the best and worst selection functions in the population (after the extremely poor selection functions are removed). For example, suppose we have a population of three selection functions with the GP-Trees shown in Figure 3, and fitness assignments of 20, 20, and 30, respectively. For the first tree, the penalty due to parsimony pressure is calculated as $(best\ fitness - worst\ fitness) * c * size$. If we assume a parsimony coefficient of $c = 0.005$,

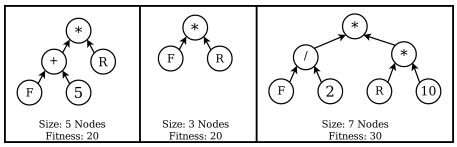

Fig. 3 The GP-Trees for an example population of 3 individuals.

then the fitness penalty is equal to $(30 - 20) * 0.005 * 5 = 10 * 0.025 = 0.25$, and the fitness assigned to the individual is $20 - 0.25 = 19.75$.

When the meta-EA concludes, the bottom-level EA utilizing the best selection function from the meta-EA is run on a set of separate testing instances from the same function class to test the generalization of the selection function's performance. If the EA utilizing the best selection function performs significantly better on the testing instances than the same EA using a standard selection function, then we can say that the evolved selection function successfully generalized to the function class of interest.

By using a meta-EA to evolve selection functions with *a priori* computation, we are utilizing an offline hyper-heuristic, with the goal being to evolve a selection function that offers better generalized performance on all instances of a particular function class. For this initial exploration into evolving selection functions, we decided to employ an offline hyper-heuristic in order to better estimate the performance upper bound without the overhead potentially incurred by an online hyper-heuristic.

The benchmark function classes used for the underlying EA are selected from the Comparing Continuous Optimizers (COCO) platform used for the GECCO Workshops on Real-Parameter Black-Box Optimization Benchmarking (Hansen et al., 2016). This benchmark set provides a suite of real-valued optimization problems that serve as a robust testbed for optimization algorithms, including EAs. These functions are well-suited to measuring the strength of an EA and how well it performs under various conditions, which makes it an excellent choice for measuring how different selection functions impact the performance of an EA. Each function class is offered in multiple dimensionalities, and for each dimension, multiple unique instances of the function class are present. By using several different instances of a given problem for a given dimensionality, we can test whether an increase in performance offered by a higher-quality selection function generalizes to other instances within the function class. We do this by using only some of the instances during the meta-EA to evaluate the quality of selection functions, then run the bottom-level EA utilizing the evolved selection function on the unused, unseen instances. We select our benchmark problems from the 24 noiseless test function classes offered by the COCO platform.

Table 4 Meta-EA Parameters

Parameter	Value
Population Size	40
Offspring Size	40
Evaluation Count	4000
Max GP-Tree Initialization Depth	4
Parent Selection	k-tournament, k=4
Survival Selection	Truncation
Mutation	Subtree Regeneration
Crossover	Subtree Crossover
Parsimony Pressure Coefficient	0.0005
Mutation Rate	0.25
Range for Constant Terminals	[-100, 100]
Range for Random Terminals	[-100, 100]
Number of Runs (Training)	5
Number of Runs (Testing)	200

4 Experimental Setup

The parameters for the meta-EA used in each experiment are shown in Table 4. These parameters were manually tuned to allow for a population with high explorative potential while keeping the total computation time manageable.

To test our methodology, we target the CMA-ES algorithm for improvement. This algorithm repeatedly samples λ points in a space around a mean, and uses a weighted combination of the high-value points to update the mean, as well as the parameters that control the shape of the space to be sampled in the next generation. In its traditional form, CMA-ES uses the μ highest-fitness points to update the mean; from an EA-perspective, this is akin to truncation selection.

We used the meta-EA to evolve a new method of selecting which of the sampled points to use for recalculating the mean. The individuals of the meta-EA each encode a single selection function in their genome, which modifies the mean-update functionality of CMA-ES. Rather than selecting the μ highest-fitness points, which is done in standard CMA-ES, the encoded selection function selects a subset of all the sampled points, which are then ordered by fitness and used to update the mean. Once the mean is updated, all other state variables, such as the covariance matrix, are updated using the same methods as the unmodified CMA-ES.

To select benchmark functions for the CMA-ES, we use the 24 noiseless function classes in the COCO dataset. We use dimensionalities of 2, 3, 5, and 10, excluding 20 and 40 to save computational resources. The meta-EA is run separately for each combination of function class and dimensionality, using 6 of the instances for that function class and dimensionality to train the meta-EA and the rest of the available instances for testing generalization of the evolved selection function. For the parameters of CMA-ES, we use $\lambda = 10 \times D$, $\mu = \lambda/2$, and $\sigma_i = 0.5$. The CMA-ES terminates on population convergence,

Table 5 Percentage of Runs Solved By Modified CMA-ES/Unmodified CMA-ES, averaged over all instances

COCO Function Class	D=2	D=3	D=5	D=10
1	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
2	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
3	35.65% / 34.2%	22.0% / 21.35%	7.6% / 6.6%	1.2% / 1.35%
4	32.15% / 3.3%	0.15% / 0.25%	0.0% / 0.0%	0.0% / 0.0%
5	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
6	100.0% / 100.0%	100.0% / 100.0%	99.1% / 100.0%	96.0% / 0.0%
7	98.05% / 94.35%	100.0% / 97.65%	99.35% / 99.4%	100.0% / 99.95%
8	99.85% / 100.0%	100.0% / 100.0%	99.9% / 99.5%	99.8% / 100.0%
9	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
10	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
11	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
12	99.55% / 100.0%	99.5% / 100.0%	99.85% / 100.0%	100.0% / 44.05%
13	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
14	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%	100.0% / 100.0%
15	41.45% / 41.95%	32.85% / 32.45%	24.55% / 24.0%	11.95% / 16.85%
16	65.7% / 65.65%	59.95% / 64.0%	58.2% / 57.1%	40.05% / 39.7%
17	96.05% / 75.0%	96.15% / 81.9%	96.7% / 94.6%	99.2% / 99.2%
18	96.85% / 64.05%	97.6% / 79.25%	93.5% / 91.75%	93.9% / 94.9%
19	55.05% / 36.1%	20.1% / 20.9%	6.85% / 5.8%	6.6% / 0.45%
20	48.35% / 24.8%	19.35% / 12.15%	2.25% / 1.5%	0.0% / 0.0%
21	56.8% / 27.65%	55.4% / 28.7%	36.05% / 23.3%	28.6% / 36.0%
22	47.15% / 35.25%	25.8% / 27.85%	14.2% / 12.75%	5.95% / 0.4%
23	70.35% / 73.3%	54.7% / 52.3%	32.05% / 31.2%	7.35% / 8.05%
24	1.6% / 1.45%	0.1% / 0.3%	0.0% / 0.0%	0.0% / 0.0%

after $10000 \times D$ evaluations, or after finding the solution to a function instance, as specified by the COCO benchmark platform.

When evaluating the performance of an evolved selection function to assign a fitness value to it, the fitness is taken as the proportion of the runs in which the modified CMA-ES reaches the global optimum, or moves close enough to it to meet the criteria to solve the function.

For each function instance in the testing set, we run the CMA-ES for 200 testing runs, both using the evolved selection function and unmodified. We then measure, for each testing instance, the proportion of runs which solved the function, in each case.

5 Results and Discussion

For each testing instance, the percentage of runs solved by the modified and unmodified CMA-ES is shown in Table 5.

For the function classes 4, 6, 12, 17, 18, 19, 20, and 21, the CMA-ES modified with the evolved selection function solved the function instances at least 20 percent more often than the unmodified CMA-ES for at least one dimensionality. For the function classes 4 and 19, the success rate of CMA-ES increased by 20-30 percent when modified with the evolved selection function at $D = 2$,

but performed similarly to the unmodified CMA-ES at other dimensionalities. For function classes 20 and 21, a performance increase is seen on dimensionalities $D = 2, 3$, and 5, but not $D = 10$; curiously, the modified CMA-ES performs worse on function class 21 when $D = 10$. For function classes 17 and 18, performance increases are seen for $D = 2$ and $D = 3$, with negligible performance differences on the other dimensionalities. Function class 22 sees about a 12 percent increase in solution rate on $D = 2$, but negligible improvement for other dimensionalities. For function classes 6 and 12, performance is similar for $D = 2, 3$, and 5, but for $D = 10$, there is a significant performance increase: on function 6, the success rate increased from 0 percent to around 96 percent, and on function class 12, the success rate increased from 18-67 percent, varying across the function instances, to 100 percent for all function instances.

For the other function classes, there was no major difference between the success rate of the modified and unmodified CMA-ES. In a few cases, the modified CMA-ES performs marginally worse than the unmodified CMA-ES, but the difference in solution rate usually no more than 5 percent, and only 7.4 percent at most.

5.1 Discussion of Results

We observed that evolving a new selection function for CMA-ES increased its solution quality on 6 of the 11 functions tested. In particular, we observed two cases with high dramatic improvements: the tests for COCO function classes 6 and 12 for dimensionality $D = 10$. In the case of function class 6, the unmodified CMA-ES was unable to solve any of the testing instances, but the CMA-ES using the evolved selection solved these instances nearly 100% of the time. In the case of function class 12, the success rate of the unmodified CMA-ES varied strongly between instances, ranging from 18 to 67%, with an average success rate of 44.05% across all instances. The modified CMA-ES, however, solved every test instance, reaching the global best fitness in each of 200 runs, on each of the testing instances, achieving a 100% success rate. By observing the increases in success rate for some of the functions, it is clear that evolving a new selection scheme for CMA-ES provides a substantial benefit in some cases.

The five cases where no improvement was observed involved functions that were highly multimodal. Three of these function were variants of the Rastrigin function, a highly multimodal function and the other two—the Weierstrass Function and the Katsuura Function—are highly rugged. It is likely, in these cases, that CMA-ES requires some other improvement aside from a new selection scheme to better learn and traverse the global structures of these functions. Because we only replaced the selection scheme of CMA-ES, we only changed how it internally updates the mean. Improving the performance of CMA-ES on these functions likely requires more intelligent updating

of the other internal variables of CMA-ES, such as the evolution paths, the covariance matrix, and the step size.

$F = 21$, $D = 10$ is the one case tested where CMA-ES performed markedly worse than unmodified CMA-ES. This effect is likely due to overspecialization to the set of training instances.

6 Conclusions

We hypothesized that a hyper-heuristic search through the space of selection functions for EAs could improve the performance of an EA on a particular function class by discovering a specialized selection function. We developed a representation of selection functions that uses a Koza-style GP-Tree to relate an individual's fitness value and fitness ranking to its relative probability of selection, and used a meta-EA to search through the space of selection functions in this representation.

With this meta-EA, we have shown that it is possible to generate new selection functions, tuned to a particular benchmark function, that can enable an EA to significantly outperform conventional selection functions on those functions. Thus, we show that, in order to discover the optimum selection method for an EA operating on a particular function, it may not be sufficient to use any of the static conventional selection functions tested. We have also shown that, in some cases, this performance increase from a custom selection algorithm will generalize to similar functions in the same function class. Therefore, if one expects to run the same EA on many functions from the same function class, one might expect to gain a performance increase by doing some *a priori* calculation to develop a specialized selection algorithm trained on instances of that function class, which would then enable an EA utilizing that selection function to perform better on other instances of that function class. However, our experiments have also shown that, for certain functions, replacing only the selection function may not yield significant performance improvements, depending on the behavior of the search strategy and the nature of the function being optimized by the EA. Careful consideration must be given to determine what the effect of tuning the selection scheme of a given EA will be on the performance of that EA, and whether such tuning will cause the EA to have an appreciable performance increase on the function class in question.

6.1 Threats To Validity

Although we strove to ensure the robustness of our methodology and experimental setup, there are some careful considerations that must be made when interpreting these results which may threaten the validity of our claims.

Of obvious note is the fact that, while we have shown improved performance on the benchmark functions tested, we still have yet to test this method on

EAs run in real-world scenarios, and we do not yet know whether the supposed performance benefit is, in practice, enough to warrant the *a priori* computation necessitated by our methods.

We chose CMA-ES as a testing target for extending our method to a more state-of-the-art method. However, the implementation of CMA-ES we used was fairly basic, lacking restarts in particular, and much research has been done on improved and alternative variants of CMA-ES. For certain problems, these newer variants may offer the same, or greater, performance increases as an evolved selection function. However, if these newer variants also use some selection function to update internal variables, as the basic implementation does, then evolving a new selection function for these newer variants could generate an even greater increase in performance; this opens up an avenue for future research.

6.2 Future Work

The work presented in this paper opens a number of potential avenues for future research. Of primary concern is the fact that the meta-EA presented in this paper requires a large amount of *a priori* computation to generate a high-quality selection function. While this computational cost may be worth it for EAs that will run on functions from the same function class many times, a more efficient method of finding good selection functions has a much greater potential to benefit EAs in general. Exploring a method of online learning could allow for the elimination of the expensive *a priori* computation time, allowing specialized selection functions to be generated during the evolution.

The CMA-ES in our experiments was tuned to increase performance on the COCO benchmark function classes. While these functions are difficult and non-trivial to optimize for, they are entirely artificial, and the meta-EA has not yet been used to tune an EA for solving a real-world function class. A major next step for this work is to apply the meta-EA to real-world EAs that could benefit from new, specialized selection functions. Of particular interest are EAs that are expected to be run many times on functions from the same general function class, which would, over many runs, amortize the *a priori* computation time required to tune the selection function.

Because the objective of this paper is similar to the work done to develop selection algorithms via Grammatical Evolution (Lourenço et al., 2013) and register machines (Woodward and Swan, 2011), it remains to be seen which cases each method is more effective for, and a direct comparison of the methods on the same benchmark functions may yield more insight into which offers better performance benefits under certain conditions.

The framework of CMA-ES used was fairly basic, lacking features such as restarts. In addition, many techniques have been developed to improve the performance of CMA-ES on functions where it may be deficient, such as in $(1 + 1)$-CMA-ES (Igel et al., 2006) and the Active CMA-ES (Jastrebski and Arnold, 2006). Evolving a specialized selection function for these new forms

of CMA-ES may lead to even greater performance gains, or may not even be necessary for particular es. Additionally, as we noted in our conclusions, we only made efforts to improve the selection of points used in the mean-update step of CMA-ES, which allowed to it gain increased performance in some, but not all, of the problem cases tested. Further experiments to tune the selection of points used for updating the other internal variables, such as evolution path, covariance matrix, and step-size, may lead to greater changes in the behavior of CMA-ES, and thus, greater changes in performance.

For our meta-EA, the GP-Trees utilized several terminals related to the individual, such as fitness, fitness ranking, uniqueness of genome, and birth generation, as well as terminals related to the evolution at large, such as population size and generation number. Adding additional available terminals increases the information available to the meta-EA, allowing it to potentially evolve more intelligent selection functions. In particular, there are no terminals that allow the selection function to have any internal persisting memory from generation to generation, such as the best fitness of past generations or which individuals have been selected previously. Additionally for EA's with sexual reproduction, or selection schemes involving individuals selected in pairs or groups, there is no terminal that measures two individuals relative to each other. This prevents the meta-EA from developing a selection function based on any information about an individual's mate, such as fitness, genome, etc. To add these terminals to the meta-EA, the selection function representation would need to be updated such that, when determining the desirability of an individual, the GP-Tree would take as input the information of some other individual to be selected for a similar purpose, such as recombination of the same child (for an EA with two-parent recombination) or selection for update of the same internal variable (for an Evolution Strategy such as CMA-ES).

The parameters at the meta-EA level were manually tuned to allow for a high degree of exploration. A sensitivity analysis of these parameters, as well an investigation of parameter tuning/control, could lead to increased performance of the meta-EA.

References

Burke EK, Hyde MR, Kendall G (2006) Evolving Bin Packing Heuristics with Genetic Programming. In: Parallel Problem Solving From Nature-ppsn Ix, Springer, pp 860–869

Burke EK, Hyde MR, Kendall G, Ochoa G, Ozcan E, Woodward JR (2009) Exploring Hyper-heuristic Methodologies with Genetic Programming. In: Computational Intelligence, Springer, pp 177–201

Burke EK, Hyde M, Kendall G, Woodward J (2010) A Genetic Programming Hyper-heuristic Approach for Evolving 2-d Strip Packing Heuristics. IEEE Transactions on Evolutionary Computation 14(6):942–958

Burke EK, Hyde MR, Kendall G (2012) Grammatical Evolution of Local Search Heuristics. IEEE Transactions on Evolutionary Computation 16(3):406–417

Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: A Survey of the State of the Art. Journal of the Operational Research Society 64(12):1695–1724

Eiben ÁE, Hinterding R, Michalewicz Z (1999) Parameter Control in Evolutionary Algorithms. IEEE Transactions on Evolutionary Computation 3(2):124–141

Garrido P, Riff MC (2010) DVRP: a Hard Dynamic Combinatorial Optimisation Problem Tackled By An Evolutionary Hyper-heuristic. Journal of Heuristics 16(6):795–834

Goldman BW, Tauritz DR (2011) Self-configuring Crossover. In: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, pp 575–582

Guntly LM, Tauritz DR (2011) Learning Individual Mating Preferences. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, ACM, pp 1069–1076

Hansen N, Ostermeier A (1996) Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation. In: Evolutionary Computation, 1996., Proceedings of IEEE International Conference on, IEEE, pp 312–317

Hansen N, Auger A, Mersmann O, Tušar T, Brockhoff D (2016) COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. ArXiv e-prints arXiv:1603.08785

Harris S, Bueter T, Tauritz DR (2015) A Comparison of Genetic Programming Variants for Hyper-heuristics. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, pp 1043–1050

Hong L, Woodward J, Li J, Özcan E (2013) Automated Design of Probability Distributions as Mutation Operators for Evolutionary Programming Using Genetic Programming. In: European Conference on Genetic Programming, Springer, pp 85–96

Igel C, Suttorp T, Hansen N (2006) A Computational Efficient Covariance Matrix Update and a (1+ 1)-CMA For Evolution Strategies. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM, pp 453–460

Jastrebski GA, Arnold DV (2006) Improving Evolution Strategies Through Active Covariance Matrix Adaptation. In: Evolutionary Computation, 2006. CEC 2006. IEEE Congress on, IEEE, pp 2814–2821

KhudaBukhsh AR, Xu L, Hoos HH, Leyton-Brown K (2009) SATenstein: Automatically Building Local Search SAT Solvers from Components". In: Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09), pp 517–524

Koza JR (1994) Genetic Programming as a Means for Programming Computers By Natural Selection. Statistics and Computing 4(2):87–112

Krasnogor N, Gustafson S (2004) A Study on the Use of "Self-Generation" in Memetic Algorithms. Natural Computing 3(1):53–76

Lopez-Ibanez M, Stutzle T (2012) The Automatic Design of Multiobjective Ant Colony Optimization Algorithms. IEEE Transactions on Evolutionary Computation 16(6):861–875

Lourenço N, Pereira F, Costa E (2013) Learning Selection Strategies for Evolutionary Algorithms. In: International Conference on Artificial Evolution (evolution Artificielle), Springer, pp 197–208

Martin MA, Tauritz DR (2013) Evolving Black-box Search Algorithms Employing Genetic Programming. In: Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, pp 1497–1504

Maturana J, Lardeux F, Saubion F (2010) Autonomous Operator Management for Evolutionary Algorithms. Journal of Heuristics 16(6):881–909

Remde S, Cowling P, Dahal K, Colledge N, Selensky E (2012) An Empirical Study of Hyperheuristics for Managing Very Large Sets of Low Level Heuristics. Journal of the Operational Research Society 63(3):392–405

Richter S (2019) Evolved Parameterized Selection for Evolutionary Algorithms. Master's thesis, Missouri University of Science and Technology

Richter SN, Tauritz DR (2018) The Automated Design of Probabilistic Selection Methods for Evolutionary Algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, ACM, pp 1545–1552

Scott EO, Bassett JK (2015) Learning Genetic Representations for Classes of Real-valued Optimization Problems. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, pp 1075–1082

Wolpert DH, Macready WG, et al. (1995) No Free Lunch Theorems for Search. Tech. rep., Technical Report SFI-TR-95-02-010, Santa Fe Institute

Woodward JR (2010) The Necessity of Meta Bias in Search Algorithms. In: Computational Intelligence and Software Engineering (cise), 2010 International Conference On, IEEE, pp 1–4

Woodward JR, Bai R (2009) Why Evolution Is Not a Good Paradigm for Program Induction: A Critique of Genetic Programming. In: Proceedings of the First Acm/sigevo Summit on Genetic and Evolutionary Computation, ACM, pp 593–600

Woodward JR, Swan J (2011) Automatically Designing Selection Heuristics. In: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, pp 583–590

Woodward JR, Swan J (2012) The Automatic Generation of Mutation Operators for Genetic Algorithms. In: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, pp 67–74