Evolved Parameterized Selection for Evolutionary Algorithms

Samuel N. Richter
Missouri University of Science and Technology
Rolla, Missouri, U. S. A.
snr359@mst.edu

Daniel R. Tauritz
Missouri University of Science and Technology
Rolla, Missouri, U. S. A.
dtauritz@acm.org

ABSTRACT

Selection functions enable Evolutionary Algorithms (EAs) to apply selection pressure to a population of individuals, by regulating the probability that an individual's genes survive, typically based on fitness. Various conventional fitness based selection functions exist, each providing a unique method of selecting individuals based on their fitness, fitness ranking within the population, and/or various other factors. However, the full space of selection algorithms is only limited by max algorithm size, and each possible selection algorithm is optimal for some EA configuration applied to a particular problem class. Therefore, improved performance is likely to be obtained by tuning an EA's selection algorithm to the problem at hand, rather than employing a conventional selection function. This paper details an investigation of the extent to which performance can be improved by tuning selection algorithm. We do this by employing a Hyper-heuristic to explore the space of algorithms which determine the methods used to select individuals from the population. We show, with both a conventional EA and a Covariance Matrix Adaptation Evolutionary Strategy, the increase in performance obtained with a tuned selection algorithm, versus conventional selection functions. Specifically, we measure performance on instances from several benchmark problem classes, including separate testing instances to show generalization of the improved performance.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Samuel N. Richter and Daniel R. Tauritz. 2019. Evolved Parameterized Selection for Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2019 (GECCO '19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Evolutionary Algorithms (EAs), as well as related stochastic metaheuristics, are applied to many real-world problems, including problems in the fields of optimization, modeling, and system design. Their success is due to a number of important factors. They make no assumptions about the optimality of any solution design, and they are blind to the preconceptions that human-built solutions may be constructed from. As such, they can generate solutions using approaches that a human might not invent. Additionally, the mechanisms of an EA allow it to avoid and escape local optima in the solution space, exploring a greater range of solutions in search of a better optimum. Their structure naturally lends itself to parallelism, as candidate solutions can be evaluated simultaneously on their own individual computing threads. EA's are relatively easy to implement, requiring, at a basic level, only a method of evaluating solutions and a method of building new solutions from old ones. However, the performance of an EA is highly sensitive to the parameters used to configure it [6]. The fields of automated algorithm configuration and Hyper-heuristics address this by exploring methods to remove the human bias in selection of algorithm parameters and search methods. Hyper-heuristics, in particular, are 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 probability that an individual's genes are selected for recombination and survival. Various conventional fitness-based selection functions exist, each providing a unique method of selecting individuals. These (potentially independent) 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 [28]. 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 software, to maximize the performance of an EA on a particular problem or problem 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 only limited by the maximum algorithm size, and so it is highly unlikely that any conventionally human-designed algorithm offers the optimal selection behavior for the EA. An implication of the "No Free Lunch" theorem [27], each possible selection algorithm is optimal for some EA configuration applied to a particular problem 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

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [30].

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 [16], 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.

The rest of this paper is organized as follows. Section 2 covers a review of the relevant literature concerning the use of Hyper-heuristics for the targeted improvement of search algorithms and EA components, including selection functions. Section 3 details the methodology of the meta-EA powering our Hyper-heuristic, including our customized representation of selection functions that defines the space we search with the Hyper-heuristic. Section 4 details the three experiments testing our methodology. In Section 5, we show and discuss the results obtained from our experiments. We conclude our findings in Section 6. In Section 7, we discuss how the assumptions made in our experimental setup may affect the validity of our conclusions. In Section 8, we discuss potential avenues of future work for this research.

2 LITERATURE REVIEW

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 [1]. 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 [18], Boolean Satisfiability solvers [15], local search heuristics [4], and iterative parse trees representing Black Box Search Algorithms [21]. 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 [5] and explored in a number of works [2, 3, 12].

Previous work has also focused on improvement of targeted components of EAs, including the evolution of new mutation operators [13, 31], mating preferences [9], genetic representation of individuals [25], and crossover operators [8]. 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 [30]. 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 [21]. 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 problem class [19]. The work described in this paper expands on these ideas with a new representation for selection algorithms: an encoding of the relationship between various factors, such as an individual's fitness, uniqueness, and the population size, and how desirable it is for an EA to select that individual, as well as the method used to ultimately select individuals based on how desirable they are.

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 [17], EA operator control [22], low-level heuristic management [23], and vehicle routing [7].

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 [16]. Rather than encoding entire programs within the GP-Tree, which could result in an infeasibly wide search space of selection algorithms [29], 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 possible operators and Table 2 for a description of the possible 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. Psuedocode for each of these selection methods may be found in [24]. 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. The figure 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 psuedocode for this method of selection is shown in Algorithm 1. With 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.

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.

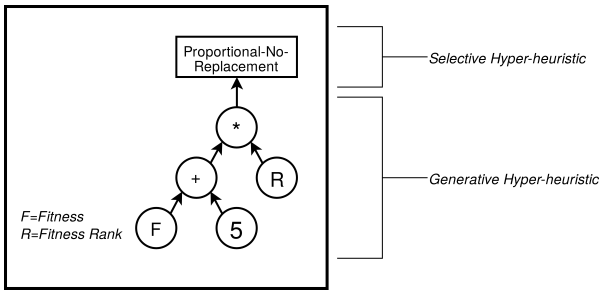

Figure 1: Example of a generated selection function.

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.

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

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.

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.

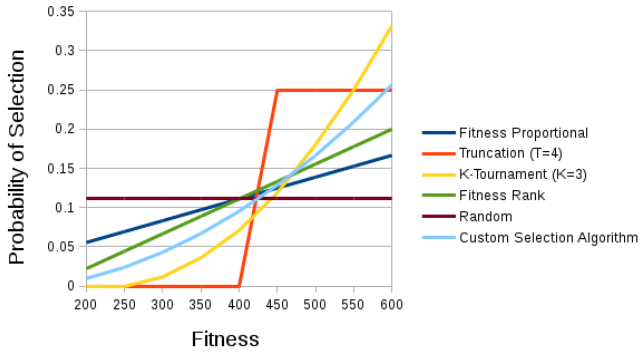

Figure 2: A comparison of the chances that each member of a sample population, with fitnesses as listed, will be selected, under each of the typical selection strategies listed, as well as the custom selection strategy shown in Figure 1.

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 on a suite of static training instances from a benchmark problem 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.

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 all the function's GP-Trees, 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

Algorithm 1 An example of the psuedocode 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
```

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 $c = 0.005$, 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 $20 - 0.25 = 19.75$. For individuals whose genomes contain multiple selection functions, and thus, multiple GP-Trees, the sum of the sizes of all GP-Trees is used.

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 problem 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,

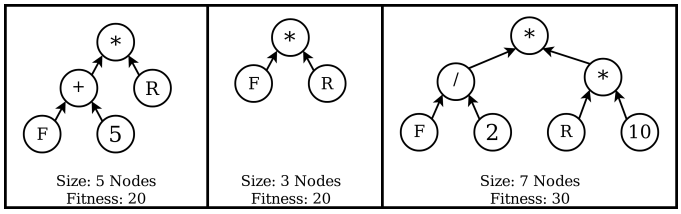

Figure 3: The GP-Trees for an example population of 3 individuals.

then we can say that the evolved selection function successfully generalized to the problem 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 problem 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 problem classes used for the underlying EA are selected from two sources. The first source of benchmark problems is the MK-Landscape class of optimization problems [26]. This problem set is a generalization of the NK-landscape problem class, first introduced in [14]. This benchmark was chosen because the properties of the fitness landscape can be easily controlled with the N, M, and K parameters, as well as the policy with which the fitness values of the loci are generated. With one set of parameters, a wide range of new MK-Landscapes can be easily generated, making it easy to provide both a variety of training landscapes to tune a selection algorithm to, and a variety of new, separate landscapes to use for testing the generalization of the selection algorithm.

As a second source of benchmark problems, we also select functions from the Comparing Continuous Optimizers (COCO) platform used for the GECCO Workshops on Real-Parameter Black-Box Optimization Benchmarking [10]. This benchmark set provides a suite of real-valued optimization problems that serve as a robust testbed for optimization algorithms, including EAs. These problems 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. Additionally, each problem class in the suite offers multiple problems, which allows us to both test the EA's performance on a variety of similar problems and test whether an increase in performance offered by a higher-quality selection function generalizes to other instances within the problem class.

4 EXPERIMENTS

We conducted three sets of experiments with our meta-EA methodology. In our first experiment, we evolved new parent and survival selection functions for a basic EA solving benchmark problems in the MK-Landscape problem class. This experiment is described further in Section 4.1.

In our second experiment, we used the same EA as in the first experiment, but instead solving for real-valued benchmark problems

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

selected from the COCO platform. This experiment is described further in Section 4.2.

In our third experiment, we evolved a new mean-update scheme for a covariance matrix adaptation evolution strategy (CMA-ES) [11], again solving for real-valued benchmark problems selected from the COCO platform. We select our benchmark problems from the same problem classes as in Experiment 2, but with different dimensionalities, in some cases. This experiment is described further in Section 4.3. 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.

4.1 Evolution of Parent and Survival Selection For A Basic EA Solving MK-Landscapes

For our first experiment, we constructed a basic EA, utilizing traditional parent selection, recombination, mutation, and survival selection. The target problem of the EA is an MK-Landscape problem, which is a generalization of the NK-Landscape problem [14, 26]. The EA has a number of parameters, including μ, the population size; λ, the offspring size; parent and survival selection schemes; and a mutation rate.

Each EA is assigned a single landscape to use when evaluating the population. When the EA is initialized, a set of μ bitstrings is randomly generated as the initial population. The fitness of each bitstring is measured against the MK-Landscape. 2λ individuals are selected from the population. The selected individuals are paired up, and one new individual is generated from each pair using per-bit crossover. Individuals are mutated at a configurable rate, and all of the new individuals are evaluated. The population of $\mu + \lambda$ individuals is then culled back down to μ individuals using survival selection. The process repeats until the best fitness of the population stops increasing, and the fitness of the best bitstring in the final population is taken as the final result of the EA run.

For the purposes of both determining the quality of evolved selection functions, and testing the generalization of their performance, a number of MK-Landscape problems are generated. Most of the generated problems are set aside for testing generalization, and the rest are used for training. Before the meta-EA is run, the optimal parameters for the EA are determined using the *irace* parameter optimization package [20]. *irace* is also used to determine schemes for traditional, non-evolved parent and survival selection, selecting from a number of typical parent and survival selection methods, such as truncation, fitness-proportional selection, and k-tournament selection. The parameters and selection schemes are tuned together, so the parameters are tuned for the selection scheme used. For the purposes of determining these parameters with *irace*, only the training problems are used, and not the testing problems. This way, we can compare the generalization of the selection functions picked by *irace* to the generalization of the selection functions generated by the meta-EA.

Every member of the meta-EA population consists of two selection functions, which are evolved together as one genome. The first selection function is used for parent selection, and the second function is used for survival selection. When the meta-EA is initialized, each selection function in the initial meta-EA population is measured by running the underlying EA on each of the training problems, using the parameters found by *irace* but substituting the evolved selection function for the *irace*-picked selection schemes. The final result of the EA is averaged across all the training problems, for multiple runs per problem, and this mean is assigned as the fitness of the evolved selection function. Once all of the evolved selection functions have an assigned fitness, the meta-EA performs recombination and mutation to generate the next generation of selection functions, which are evaluated in the same way. The meta-EA continues until a number of generations pass with no improvement in the quality of the selection functions, at which point it terminates.

When the meta-EA terminates, the selection function with the highest fitness is selected, and EAs using this selection function are run on the testing problem instances. The EA is also run on the same testing instances using the original *irace* parameters. The performance of the EA using the evolved selection function is then compared to the performance of the EA using the *irace*-chosen selection methods to test for generalization.

Table 5 shows the parameters used in the bottom-level EA and for generating the landscape functions. These parameters include the parent and survival selection functions chosen by *irace*; for evaluating an evolved selection function, these functions are replaced with the evolved functions.

4.2 Evolution of Parent and Survival Selection For A Basic EA Solving Real-Valued Benchmarks

Our second experiment utilized a similar approach to our first experiment, but the benchmarks problem targeted by our EA were real-valued functions chosen from the 24 noiseless benchmark test functions contained in the COCO benchmark set.

Table 5: Bottom-level EA Parameters For MK-Landscape

Parameter	Value
Population Size	91
Offspring Size	34
Genome Length (N)	40
Locus Length (K)	8
Number of Loci (M)	40
Locus Minimum Value	0
Locus Maximum Value	8
Parent Selection	Fitness Ranking With Replacement
Survival Selection	Fitness Ranking Without Replacement
Termination Criteria	Convergence
Generations to Convergence	25
Mutation	Random Bit Flip
Mutation Rate	0.0038
Crossover	Per-Bit Crossover

For each of the 24 problem classes, COCO offers multiple benchmark instances in each of several dimensionalities. For our experiment, we chose the 10-dimensional problems, and utilized all of the available instances for each of the 24 problem classes. For each of the problem classes, a number of the instances were chosen to be the training instances, used by the meta-EA to assign a fitness to an evolved selection function, and the rest of the instances were set aside as testing instances, to test for generalization.

As with the first experiment, each run of the underlying EA is initialized with a randomized population, but the population members are real-valued vectors initialized within a range recommended by COCO, rather than bitstrings. Parent selection, arithmetic recombination, mutation, and survival selection are used to advance the EA until the population fitness is no longer increasing, and the best fitness attained by the EA is used as a measure of the EA's performance.

For this experiment, *irace* was run for the training instances of each of the 24 problem classes individually, generating 24 sets of parameters for the EA. These parameters are listed in [24].

The meta-EA was run as it was for the first experiment, with each evolved member containing two selection functions: one to replace parent selection, and one to replace survival selection. At the meta-EA's conclusion, the best selection function evolved is pitted against the traditional selection functions chosen by *irace*, running on the testing problem instances to test generalization.

4.3 Evolution of Selection For CMA-ES Solving Real-Valued Benchmarks

In the third experiment, we replaced the simple EA with a more complex covariance-matrix-adaptation evolution strategy (CMA-ES). This method 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.

For this experiment, we used the same meta-EA setup from previous experiments 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 unchanged CMA-ES.

To select benchmark functions, we tested the unmodified CMA-ES on the COCO functions. On many of the COCO functions, the solutions found by CMA-ES are close enough to the function's global optimum to meet the criteria set by COCO for solving the function. For each of the 24 function classes, we selected the lowest dimensionality for which CMA-ES was unable to find the global optimum (according to COCO's own criteria) at least half the time. We ignore the function classes for which CMA-ES is able to solve the instances more than half the time with $D \geq 20$. The functions chosen are detailed in Table 6. Unlike with the first two experiments, we did not use *irace* to find a parameter set for CMA-ES before running the meta-EA, as CMA-ES is much less dependent on initial parameters. We use $\lambda = 10 * D = 100$, $\mu = \lambda/2 = 50$, and $\sigma_i = 0.5$.

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 problem.

5 RESULTS AND DISCUSSION

Here, we present the results observed from the three experiments performed, as well as a discussion of our observations.

Table 6: COCO Benchmark Functions Chosen For Experiment 3

Function Index	Dimensionality
3	2
4	2
6	10
12	10
15	2
16	10
19	2
20	2
21	2
22	2
23	5
24	2

5.1 Results And Observations From First Experiment

For each of the problem instances in the testing set, we run the EA for 200 testing runs, both with the evolved selection functions and with the traditional selection functions. We then compare the final best fitnesses reached in the testing runs with a two-sided T-test to test for a significant difference. If the variances are shown to be unequal by Bartlett's test ($p \leq 0.05$), then the T-test is performed assuming unequal variances; otherwise, the T-test is performed assuming equal variances. We found that, in 45 out of 50 of the testing instances, the EA using the evolved selection function reached a significantly higher final best fitness than the EA without an evolved selection function ($p \leq 0.05$). In the remaining 5 testing instances, there was no statistical difference between the EA with and without the evolved selection function.

Figure 4 shows the parent selection stage of the best selection function evolved by the Hyper-heuristic. The survival selection was controlled by a separate, similar function.

Figure 5 shows the progression of the best population member's fitness through many runs of one of the testing functions, averaged for both the typical selection functions and the evolved selection functions. We see that, on average, the evolved selection function climbs more slowly towards a higher fitness before converging and terminating. Figure 6 shows a boxplot comparison of the final best fitnesses achieved on the same testing fitness function, showing that, for this particular function, while the best fitnesses achieved by the evolved selection functions are generally within the same range as those found by the unmodified EA, they tend to be more consistently within the higher end, by comparison.

The general trend that we observe, across all the testing problems, is that the EA using the evolved selection functions tends to trend more slowly toward an area of high fitness than the EA using the traditional selection functions. Although the evolved selection functions take more evaluations to converge, they often converge at a higher fitness.

Table 7: Experiment 2 Results

Problem Index (D=10)	Number of Instances Improved
F=1	0 / 12
F=2	1 / 12
F=3	2 / 12
F=4	2 / 12
F=5	5 / 12
F=6	3 / 12
F=7	7 / 12
F=8	2 / 12
F=9	12 / 12
F=10	1 / 12
F=11	7 / 12
F=12	0 / 12
F=13	2 / 12
F=14	0 / 12
F=15	0 / 12
F=16	0 / 12
F=17	4 / 12
F=18	4 / 12
F=19	12 / 12
F=20	0 / 12
F=21	7 / 12
F=22	6 / 12
F=23	0 / 12
F=24	0 / 12

5.2 Results And Observations From Second Experiment

For each of the problem instances in the testing set, we run the EA for 200 testing runs, both with the evolved selection functions and with the traditional selection functions. We then compare the final best fitnesses reached in the testing runs with a two-sided T-test to test for a significant difference. If the variances are shown to be unequal by Bartlett's test ($p \leq 0.05$), then the T-test is performed assuming unequal variances; otherwise, the T-test is performed assuming equal variances.. For each benchmark problem class in the COCO functions dataset, we count the number of instances for which we find a statistically significant difference between the final best fitness found by the EA with the best evolved selection function and the final best fitness found by the EA without one. These results are shown in Table 7.

We find that, for some problem classes, such as 7, 9, 11, 19, 21, and 22, the evolved selection function enabled the EA to perform significantly better in at least half of the testing instances used. For most of the testing cases, however, there were few or no testing instances for which the EA using the evolved selection function performed significantly better than the EA using only the parameters found by *irace*.

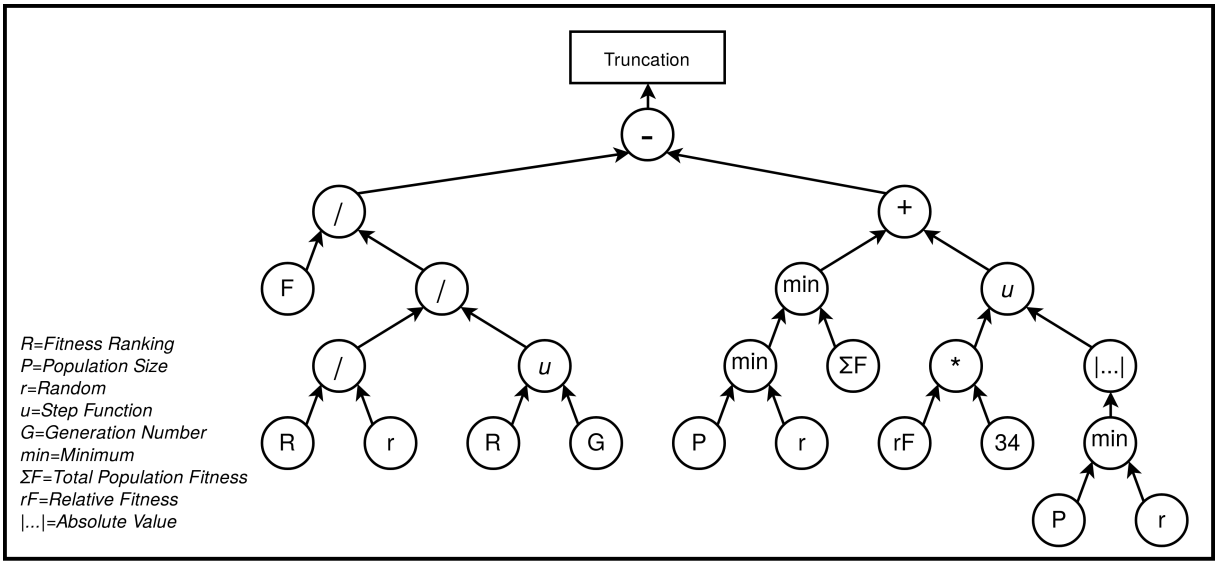

Figure 4: The function that controlled parent selection in the best evolved selection function.

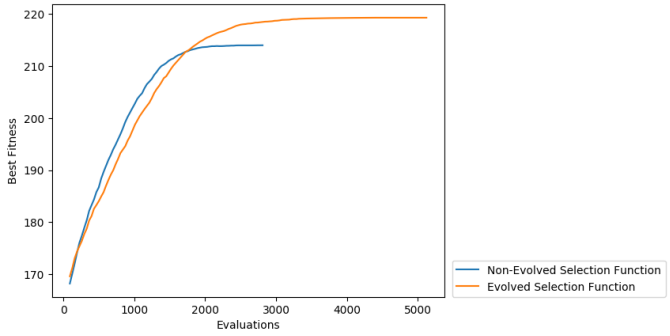

Figure 5: A plot of the best population fitness against the number of EA evaluations, achieved by both the typical selection function and the evolved selection function, averaged over all runs, for one of the testing functions of Experiment 1.

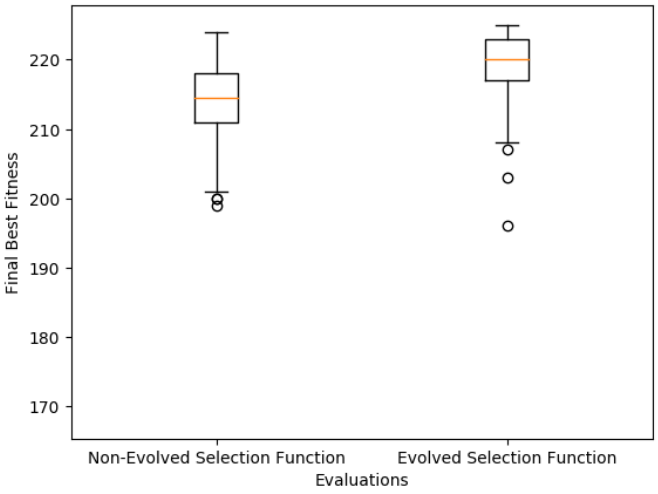

Figure 6: A box plot comparing the final best fitnesses achieved on one of the testing functions of Experiment 1, by both the typical selection functions and the evolved selection functions.

5.3 Results And Observations From Third Experiment

For each problem 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. For each testing instance, the percentage of runs solved by the modified and unmodified CMA-ES is shown in Table 8.

For the problem classes 4, 6, 12, 19, 20, and 21, the CMA-ES modified with the evolved selection function solved the problem instance more often than the unmodified CMA-ES. For the functions 4, 19, 20, and 21, the success rate of CMA-ES increased by 20-30 percent when modified with the evolved selection function. For functions 6 and 12, the effect is much more dramatic: on function 6, the success rate increased from 0 percent to around 96 percent, and on function 12, the success rate increased from 18-67 percent, varying across the function instances, to 100 percent for all function instances.

For the function indices 3, 15, 16, 23, and 24, there was no major difference between the success rate of the modified and unmodified CMA-ES.

Table 8: Experiment 3 Results

Problem Instance	F=3,D=2	F=4,D=2	F=6,D=10	F=12,D=10
0	36.0% / 34.5%	34.0% / 4.0%	96.5% / 0.0%	100.0% / 41.0%
1	32.5% / 33.5%	36.0% / 1.5%	96.0% / 0.0%	100.0% / 40.0%
2	38.5% / 38.0%	26.5% / 2.0%	97.5% / 0.0%	100.0% / 55.5%
3	37.5% / 30.5%	34.5% / 3.5%	94.5% / 0.0%	100.0% / 52.0%
4	34.0% / 35.0%	38.0% / 3.5%	93.5% / 0.0%	100.0% / 51.5%
5	35.0% / 32.5%	9.0% / 4.0%	95.0% / 0.0%	100.0% / 48.0%
6	33.5% / 35.0%	36.5% / 4.5%	95.0% / 0.0%	100.0% / 18.0%
7	38.0% / 37.5%	37.5% / 2.0%	98.0% / 0.0%	100.0% / 31.0%
8	35.0% / 33.5%	36.0% / 3.5%	97.5% / 0.0%	100.0% / 36.0%
9	36.5% / 32.0%	33.5% / 4.5%	96.5% / 0.0%	100.0% / 67.5%
Problem Instance	F=15,D=2	F=16,D=10	F=19,D=2	F=20,D=2
0	40.0% / 45.5%	39.5% / 37.0%	48.0% / 38.0%	45.0% / 25.0%
1	43.0% / 47.0%	42.5% / 41.5%	53.0% / 38.0%	46.0% / 22.5%
2	47.0% / 45.5%	40.0% / 41.0%	58.0% / 36.5%	54.0% / 24.5%
3	39.5% / 40.0%	43.0% / 38.5%	55.0% / 30.0%	44.5% / 22.5%
4	40.0% / 40.0%	42.0% / 47.0%	54.0% / 34.0%	46.0% / 26.0%
5	41.0% / 45.0%	32.5% / 38.0%	56.5% / 40.0%	45.5% / 26.5%
6	38.0% / 32.0%	45.5% / 40.5%	55.5% / 38.0%	50.0% / 27.0%
7	42.0% / 42.5%	41.5% / 42.0%	61.0% / 32.0%	48.5% / 23.5%
8	43.0% / 43.0%	37.5% / 37.5%	53.0% / 34.0%	54.5% / 24.5%
9	41.0% / 39.0%	36.5% / 34.0%	56.5% / 40.5%	49.5% / 26.0%
Problem Instance	F=21,D=2	F=23,D=5	F=24,D=2	
0	61.0% / 37.5%	25.0% / 39.0%	1.0% / 3.0%	
1	36.0% / 19.5%	26.5% / 28.0%	0.5% / 1.0%	
2	59.5% / 28.0%	29.5% / 30.0%	0.5% / 1.5%	
3	66.5% / 37.0%	37.5% / 35.0%	3.5% / 1.0%	
4	38.5% / 18.5%	31.5% / 29.0%	2.5% / 0.5%	
5	34.5% / 16.5%	30.0% / 33.0%	1.0% / 2.0%	
6	63.0% / 24.5%	33.0% / 33.5%	1.0% / 0.5%	
7	56.0% / 28.5%	35.5% / 30.0%	3.5% / 2.0%	
8	76.5% / 33.0%	35.5% / 26.5%	1.0% / 2.0%	
9	76.5% / 33.5%	36.5% / 28.0%	1.5% / 1.0%	

5.4 Discussion of Results

In our first experiment, we observed that, for an EA with a given set of parameters, an evolved selection function can lead to a significant increase in performance that successfully generalizes to other similar instances in the same problem class. The behavior that we see in the EAs using the evolved selection functions, where the fitness grows more slowly and converges later, at a higher fitness, suggests that the meta-EA evolved toward selection schemes with less selection pressure on the population, allowing them to explore further to find better optima. Even though the candidate parameters for the EA, as selected by *irace*, included several selections of varying selection pressures (including k-tournament with configurable k), the evolved selection functions still seemed to find a better balance of exploration versus exploitation.

In our second experiment, we observed varying results in the ability for the EA with the evolved selection function to outperform the EA without one. In some cases, a large improvement was seen by applying the evolved selection function, resulting in the EA finding significantly higher fitnesses in at least half the testing instances. In many cases, applying the evolved selection function results in significantly higher fitnesses on fewer than half of the test cases, and in some cases, there is no observed significant improvement whatsoever. This deficiency could be due to a number of factors. It is possible that, within the meta-EA, the evolved selection functions over-specialized to the training functions provided, and sacrificed the ability to generalize for increased improvement on the training functions. It is also possible that some of the functions selected from the COCO benchmarks are sufficiently complex that the EA setup we used required a more sophisticated improvement than a new selection function to gain an appreciable benefit to solving these functions. Some functions chosen may also be so simple that the EA already performs well enough that a change in selection function has a negligible impact on performance, as long as it exerts sufficient selection pressure. We attempted to remedy the issue of function difficulty in our third experiment by carefully selecting

functions for which the unmodified CMA-ES performed moderately, but not exceptionally, well.

For our third experiment, 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. 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%. 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.

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 problem 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 problem, that can enable an EA to significantly outperform conventional selection functions on those problems. Thus, we show that, in order to discover the optimum selection method for an EA operating on a particular problem, 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 problems in the same problem class. Therefore, if one expects to run the same EA on many problems from the same problem 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 problem class, which would then enable an EA utilizing that selection function to perform better on other instances of that problem 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 problem class in question.

7 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.

In Experiment 1 and Experiment 2, we used a very rudimentary EA, and it is highly unlikely that such an EA would be the best practical approach for a given problem, given the countless advancements in the field of evolutionary computing. We chose such a simple EA to illustrate our point that the selection function, while deceptively simple to configure by selecting a conventional selection function with proven effectiveness, has a large impact on the performance of an EA.

For our third experiment, 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.

Some of the terminals used by the GP-Trees enable selection functions to have access to some information that conventional selection functions such as fitness proportional selection and truncation selection do not have. This information includes generation number and genome uniqueness, which can allow the selection function to account for other factors besides the fitness of the individual and the population at large. This might make the comparison to these functions in experiments 1 and 2 unfair. We chose the set of conventional selection functions available to *irace* in experiments 1 and 2 as the functions that are commonly used in straightfoward EAs, in order to show that an evolved selection function with access to additional information may be worth the *a priori* computation cost if the alternative is to simply use a conventional selection function selected from the most common choices that would work acceptably well for most EAs. To investigate whether the meta-EA still performs well, even with a limited set of terminals, we ran

the experimental setup discussed in Experiment 3, but with a more limited set of terminals. This is discussed in [24].

8 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 problems from the same problem 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 EA and CMA-ES in our experiments were tuned to increase performance on the MK-Landscape problem class and the COCO benchmark problem classes. While these problems 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 problem class. While the MK-Landscape problem class and related problems are used in several real-world fields, 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 problems from the same general problem 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 [19] and register machines [30], it remains to be seen which cases each method is more effective for, and a direct comparison of the methods on the same benchmark problems may yield more insight into which offers better performance benefits under certain conditions.

In the case of Experiment 1 and Experiment 2, the evolved selection functions were only tested against basic selection functions that lacked any self-adaptability. More work will need to be done to determine whether the best evolved selection functions can stand up to more dynamic, adaptable selection methods, if EAs such as the one used in Experiments 1 and 2 are targeted for improvement.

For the third experiment, 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. 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 problem classes. 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 increase the performance of the meta-EA.

REFERENCES

[1] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: A Survey of the State of the Art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724.

[2] Edmund K Burke, Matthew Hyde, Graham Kendall, and John Woodward. 2010. A Genetic Programming Hyper-heuristic Approach for Evolving 2-d Strip Packing Heuristics. *IEEE Transactions on Evolutionary Computation* 14, 6 (2010), 942–958.

[3] Edmund K Burke, Matthew R Hyde, and Graham Kendall. 2006. Evolving Bin Packing Heuristics with Genetic Programming. In *Parallel Problem Solving From Nature-ppsn Ix*. Springer, 860–869.

[4] Edmund K Burke, Matthew R Hyde, and Graham Kendall. 2012. Grammatical Evolution of Local Search Heuristics. *IEEE Transactions on Evolutionary Computation* 16, 3 (2012), 406–417.

[5] Edmund K Burke, Mathew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R Woodward. 2009. Exploring Hyper-heuristic Methodologies with Genetic Programming. In *Computational Intelligence*. Springer, 177–201.

[6] Ágoston E Eiben, Robert Hinterding, and Zbigniew Michalewicz. 1999. Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 3, 2 (1999), 124–141.

[7] Pablo Garrido and María Cristina Riff. 2010. DVRP: a Hard Dynamic Combinatorial Optimisation Problem Tackled By An Evolutionary Hyper-heuristic. *Journal of Heuristics* 16, 6 (2010), 795–834.

[8] Brian W Goldman and Daniel R Tauritz. 2011. Self-configuring Crossover. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. ACM, 575–582.

[9] Lisa M Guntly and Daniel R Tauritz. 2011. Learning Individual Mating Preferences. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM, 1069–1076.

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

[11] Nikolaus Hansen and Andreas Ostermeier. 1996. Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*. IEEE, 312–317.

[12] Sean Harris, Travis Bueter, and Daniel R Tauritz. 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, 1043–1050.

[13] Libin Hong, John Woodward, Jingpeng Li, and Ender Özcan. 2013. Automated Design of Probability Distributions as Mutation Operators for Evolutionary

Programming Using Genetic Programming. In *European Conference on Genetic Programming*. Springer, 85–96.
[14] SA Kaufmann. 1993. The Origins of Order. (1993).
[15] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2009. SATenstein: Automatically Building Local Search SAT Solvers from Components". In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*. 517–524.
[16] John R Koza. 1994. Genetic Programming as a Means for Programming Computers By Natural Selection. *Statistics and Computing* 4, 2 (1994), 87–112.
[17] Natalio Krasnogor and Steven Gustafson. 2004. A Study on the Use of "Self-Generation" in Memetic Algorithms. *Natural Computing* 3, 1 (2004), 53–76.
[18] Manuel Lopez-Ibanez and Thomas Stutzle. 2012. The Automatic Design of Multi-objective Ant Colony Optimization Algorithms. *IEEE Transactions on Evolutionary Computation* 16, 6 (2012), 861–875.
[19] Nuno Lourenço, Francisco Pereira, and Ernesto Costa. 2013. Learning Selection Strategies for Evolutionary Algorithms. In *International Conference on Artificial Evolution (evolution Artificielle)*. Springer, 197–208.
[20] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. 2016. The irace package: Iterated Racing for Automatic Algorithm Configuration. *Operations Research Perspectives* 3 (2016), 43–58. https://doi.org/10.1016/j.orp.2016.09.002
[21] Matthew A Martin and Daniel R Tauritz. 2013. Evolving Black-box Search Algorithms Employing Genetic Programming. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. ACM, 1497–1504.
[22] Jorge Maturana, Frédéric Lardeux, and Frédéric Saubion. 2010. Autonomous Operator Management for Evolutionary Algorithms. *Journal of Heuristics* 16, 6 (2010), 881–909.
[23] Stephen Remde, Peter Cowling, Keshav Dahal, Nic Colledge, and Evgeny Selensky. 2012. An Empirical Study of Hyperheuristics for Managing Very Large Sets of Low Level Heuristics. *Journal of the Operational Research Society* 63, 3 (2012), 392–405.
[24] Samuel Richter. [n. d.]. Evolved Parameterized Selection for Evolutionary Algoruthms. ([n. d.]).
[25] Eric O Scott and Jeffrey K Bassett. 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, 1075–1082.
[26] L Darrell Whitley, Francisco Chicano, and Brian W Goldman. 2016. Gray Box Optimization for Mk Landscapes (NK Landscapes and MAX-kSAT). *Evolutionary Computation* 24, 3 (2016), 491–519.
[27] David H Wolpert, William G Macready, et al. 1995. *No Free Lunch Theorems for Search*. Technical Report. Technical Report SFI-TR-95-02-010, Santa Fe Institute.
[28] John R Woodward. 2010. The Necessity of Meta Bias in Search Algorithms. In *Computational Intelligence and Software Engineering (cise), 2010 International Conference On*. IEEE, 1–4.
[29] John R Woodward and Ruibin Bai. 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, 593–600.
[30] John Robert Woodward and Jerry Swan. 2011. Automatically Designing Selection Heuristics. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. ACM, 583–590.
[31] John R Woodward and Jerry Swan. 2012. The Automatic Generation of Mutation Operators for Genetic Algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. ACM, 67–74.