The Automated Design of Local Optimizers for Memetic Algorithms Employing Supportive Coevolution

Nathaniel R. Kamrath
BONSAI Lab
Auburn University
Auburn, Alabama, U.S.A.
kamrathn@acm.org

Aaron Scott Pope
Advanced Research in Cyber Systems
Los Alamos National Laboratory
Los Alamos, New Mexico, U.S.A.
aaronspope@acm.org

Daniel R. Tauritz
BONSAI Lab
Auburn University
Auburn, Alabama, U.S.A.
dtauritz@acm.org

ABSTRACT

One promising method of improving Evolutionary Algorithm (EA) performance is to improve its fine tuning capabilities by using an additional local optimization operator in the evolutionary cycle. This customization on the traditional EA is typically called a Memetic Algorithm (MA). Adding the appropriate local optimization algorithm can increase performance, while a poor choice can decrease performance. Thus, some knowledge is required to select the correct algorithm. In many cases the optimal algorithm selection is not known and it may not be static, but could change during evolution.

This investigation combines a method of local optimizer evolution using Push Genetic Programming with a method of automatic, self-configuration called Supportive Coevolution. This combination creates a novel MA that coevolves local optimization operators with target fitness function solution candidates. Implementation methodology is shown and experimentation details with corresponding results are presented. Some additional parameters that were discovered for performance tuning are discussed along with a study of their impact on the algorithm's performance. Discussion of some interesting insights followed by some suggestions for further investigation are also provided. Results show the proposed technique can improve the performance of an EA by providing automatically configured, coevolved local optimization operators to a MA.

CCS CONCEPTS

• **Theory of Computation** → **Bio-inspired optimization systems; Genetic Programming.**

KEYWORDS

memetic algorithm, local optimization, genetic programming, hyperheuristics, automatic design

ACM Reference Format:

Nathaniel R. Kamrath, Aaron Scott Pope, and Daniel R. Tauritz. 2020. The Automated Design of Local Optimizers for Memetic Algorithms Employing Supportive Coevolution. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion)*, July 8–12, 2020, Cancún, Mexico. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3377929.3398132>

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7127-8/20/07...\$15.00

<https://doi.org/10.1145/3377929.3398132>

1 INTRODUCTION

Evolutionary Algorithms (EAs) have been shown to be excellent problem solvers in a variety of domains. In particular, they can perform coarse grain searches very effectively, but they can sometimes struggle to fine tune solutions during the evolutionary process [10]. Because fine tuning a solution can result in large improvements in performance, methods of local learning or optimization have been designed to enhance EA capabilities with respect to fine grain searches. An EA which employs this type of local search algorithm as an additional evolutionary operator is typically referred to as a Memetic Algorithm (MA) [1].

There are many different learning techniques that can be leveraged within a MA. These algorithms are traditionally hand made, problem specific, and static. This means that expert knowledge or trial and error is required when selecting the optimization algorithm to use. It also means that different problems can require completely different optimization algorithms. If the local learning technique does not adapt to the state of the EA at run time, it is also impossible for the optimizer to consider current population metrics to make informed decisions that can result in better performance.

This work explores the use of generative hyper-heuristics in the form of stack-based Genetic Programming (GP) to evolve local optimization operators as a support population within a so-called Supportive Coevolution [6] MA. The local optimization operators are evolved in a separate population in parallel with solution candidates and are evaluated by executing the optimization operators on solution candidates and measuring the solution candidate's change in fitness. This novel combination allows for the automatic construction and evolution of optimization operators that can be used by the EA during the evolution of solution candidates to improve fine grain search capabilities.

2 RELATED WORK

Local optimization has been widely studied, especially in the field of artificial intelligence (AI). Various methods of hand modifying optimization algorithms have been investigated as well as ways of automatically selecting the optimal strategy when given several choices. Algorithms which can modify known optimization operators in an attempt to specialize them to a certain solution space or even to create a novel technique have also been studied. Evolution of EA operators and parameters such as mutation operators, mutation steps sizes, selection operators, and crossover operators has also been previously studied. Many of these methods employ some form of GP as the solution representation.

2.1 Push for Genetic Programming

Genetic programs can be represented in a number of different ways with various trade-offs [7]. Push is a GP representation specifically designed for expressing evolved programs at a low level with minimal restrictions on program syntax and ordering [17–19]. Each program is composed of a list of instructions with only parentheses providing structural organization for operations like branches. The instruction set can easily be modified to fit a wide variety of experimental needs.

Push programs are stack based which means program data is stored in a global set of typed stacks which instructions use to fetch inputs and store outputs. If an instruction needs an argument and none is available on the specified stack, the instruction does not perform any operation and the program continues. This means that generating and evolving valid programs is simple since strict ordering of instructions and requirements on input arguments are not necessary. The data types and number of stacks are easily modified for specific problems and applications as well. Instructions are usually low level with straightforward implementations. Simple instructions and a linear instruction format similar to most common programming languages also makes the programs easier to read and understand. The push program code itself can be considered a stack which provides programs with self modification capabilities if desired.

2.2 Automating Algorithm Design

As mentioned previously, the performance improvement gained in MAs by leveraging a local learning operator comes with the cost of selecting an appropriate optimization algorithm. A good selection can increase performance while a poor choice can hinder the EA. Algorithm selection can be a very difficult task given the large number of algorithms available and their different configurations and variations. A designer is sometimes required to understand the complexities of the problem (solution space) and solution representation for the target fitness function to select a good operator. Considering all of these factors makes selection very difficult. Thus, automated algorithm selection has been investigated using machine learning techniques [9].

One form of machine learning for automated algorithm selection is a method of search for generating heuristics called generative hyper-heuristics. In [12] it was shown that the automated design of black box search algorithms using GP could produce better results than known black box search algorithms. This method had some problems with over-specialization which caused issues with robustness. This specialization issue was addressed in [13] which improved robustness using a multi-sample approach. Hyper-heuristics have also been used to explore the search space of algorithms for functions such as selection algorithms [14] and local optimization algorithms [11].

2.3 Dynamic EA Operators

Previous investigations have been performed that showed dynamic EA operators can yield performance improvements. Dynamic operators can be implemented in a number of different ways. Some examples of dynamic EA operators range from parameter control or probabilistic operator selection methods, to self adaptive

schemes, to coevolution of operators encoded as genetic programs. In [3] GP was used to evolve crossover operators by implementing a Meta-Genetic Program to evolve the operators. While this method is unable to adapt within a single run of the EA and is extremely expensive computationally, it showed that GP encoded crossover operators can improve performance. GP was again used to encode crossover operators, but in a self adaptive scheme called Self-Configuring Crossover (SCX) [5]. As a follow up to SCX, [8] employed a coevolutionary strategy to evolve the crossover operators in a separate population along with the primary solution candidates.

The crossover operator is not the only EA operator which has been evolved using GP. Genetic Algorithm (GA) mutation operators were evolved in [20]. Selection operators were also evolved in [15] which employed a Hyper-heuristic approach to explore selection algorithm search space. The parent selection process was automated in [16] where mate selection operators were evolved in a GP style.

2.4 Supportive Coevolution

Supportive Coevolution (SuCo) is a technique that allows any number of EA components such as operators and parameters to be evolved in separate support populations in parallel with the primary population [6]. The solution candidates are evolved in what is called the primary solution population. These individuals are evaluated by the target fitness function and are the population of a traditional EA. All support operators and parameters are coevolved in separate populations called support populations. For example, an EA solving a numerical optimization problem with a solution representation of a vector would have a primary population consisting of vector individuals. These individuals would receive their fitness values by evaluating their solution on the numerical optimization problem. A number of support populations could then be added to the EA using SuCo which would aid the primary population throughout the evolutionary process. Support population individuals are then evaluated by how effective they were at aiding the primary population in the requested operation.

The primary goal of adding support populations is to increase performance by allowing the EA to explore not only the search space of the target fitness function, but also the search spaces of co-evolved support individuals. Performance improvements resulting from this method can be attributed to two complexities of EAs. First, the performance of most EAs is dependent on the configuration of many parameters and the design of operators [2]. Second, it has been shown that optimal parameters and operators can change during the evolutionary process [4]. SuCo addresses both issues by automatically creating and evolving support individuals along with the primary population. Previous work has evolved crossover operators and mutation step sizes using SuCo and it was shown that adding support populations can improve EA performance [8].

Because each SuCo support population carries out its own evolutionary cycle, parameters and operators are also necessary for each population. Since one goal of SuCo is to reduce the need to select operators and tune parameters, support populations traditionally utilize the same operators and parameters as the primary population with only slight modifications if necessary. This method of copying as much of the primary population configuration as

possible for support populations has shown high performance in previous work and reduces EA design time.

2.5 Memetic Algorithms

Creating high performance solutions to complex problems can be very challenging in part due to solution space size and shape. EAs provide a general mechanism for searching large, multi-modal solution landscapes very effectively by attempting to combine different components of known good solutions to create even higher quality solutions. Because complex problems typically have very deceptive solutions spaces, EAs tend to favor exploration over exploitation to avoid local optima. Tailoring an EA to favor exploitation typically results in pre-mature convergence of the population when a more optimal solution could have been found. MAs provide an enhancement to the classical EA by adding an additional local learning operator to the evolutionary cycle which improves the exploitation capabilities of the EA without sacrificing exploration [1]. This has proven to be a very effective method of improving EA performance.

3 METHODOLOGY

In this work, a single support population of local optimization operators is created using SuCo. The local optimization operators are encoded as Push genetic programs which allows them to be fully evolved. Table 1 shows the vector Push instructions implemented for local optimizer construction and evolution. The push instruction set here is similar to the set used in previous work [11]. Figure 1 provides a visual representation of the structure implemented. The rest of this section provides low level details about the implementation of the SuCo MA.

3.1 Initialization

Upon EA initialization, the primary population is first created and initialized. The support population is then randomly generated. Random generation is achieved by first generating a bounded random number for the size of a program in instructions. Once the size is known, random instructions are generated from the pool of available instructions until the randomized size is achieved. At this point, the support individual is ready for evaluation. Normally, the local optimization support individuals would be evaluated on primary offspring. During initialization there are no offspring available as the evolutionary process has not yet begun. As a work around, random individuals from the primary population are used instead. The number of primary individuals to sample per local optimizer when calculating support fitness is a configurable parameter which is discussed in Section 6.1. Once the configured number of primary solution candidates have been optimized by the local optimizer, a fitness value is computed. Once every support individual in the initial support population has a fitness value, the support population is ready for the evolutionary cycle.

3.2 Evolutionary Cycle

The primary population experiences a traditional evolutionary cycle with the addition of the local optimizer application to offspring. Parent selection, recombination, and mutation are performed using the configured operators and parameters. Support is then requested from the local optimizer population to optimize the current set of

primary offspring. Local optimization can be selectively performed by the support population to improve performance which is discussed in Section 6.2. After support has been requested (yellow arrow in Figure 1), survivor selection is performed which is the last step in the primary population's evolutionary cycle.

The evolutionary cycle for the support population is identical to a traditional EA cycle. First, parents are selected using the configured parent selection operator. Then, recombination is performed by randomly combining sub-programs of two parents. Mutation is then performed via randomized gene changes. This operation simply replaces an instruction in the push program with a new randomly generated instruction pulled from the set of available instructions.

After mutation, the support individuals are ready for evaluation. Evaluation for the local optimizers was performed at a specific point during the primary population's evolutionary cycle, namely immediately after offspring mutation. This point was selected because a new generation is about to enter the primary population and it means the local optimizers are the last operator to manipulate a primary solution candidate before survivor selection. Evaluation of a local optimizer requires one or more primary solution candidates. The primary solution candidate genes are copied and pushed onto the vector stack. The length of the vector as well as the original fitness of the primary individual are also pushed to the integer and float stacks respectively as inputs for the push program. The local optimizer is then executed and the vector on top of the vector stack after execution is treated as the output optimized genes. If no value is available on the vector stack, a fitness penalty is applied to the local optimizer individual. Assuming there is a vector on the vector stack, the new genes are then evaluated on the target fitness function so that the post optimization primary fitness can be used in support individual fitness calculation. The stacks are then cleared and the process is repeated until the configured number of primary solution candidates have been sampled (see Section 6.1 for discussion on primary population sampling configuration). Support individual fitness is then calculated as the average of all post optimization fitness values and returned to the support population (red arrow in Figure 1).

At this point, there are a few different methods of handling the optimized genes and resulting fitness value. One option is to discard both the optimized genes and the optimized fitness value, allowing the support population to remain completely isolated from the primary population. However, the primary population does not benefit from the work being done to design the local optimizers which makes this method only viable as a source of comparison for experimentation. Another option is to replace the original fitness value with the optimized fitness value, but leave the genes in their pre-optimization form which applies the Baldwin effect. Yet another option is to replace the original individuals' genes and fitness value according to Lamarckian learning. A final option is to leave the original optimization target individual as is and create a new individual with the optimized genes and fitness value as an additional offspring before survivor selection. Experiments with each configuration are performed and presented with some insights in Section 6.3.

It is important to note that support individual fitness calculation requires extra primary population evaluations in order to determine the post optimization fitness. Since one of the goals of this

Figure 1: Structure of the proposed SuCo MA

Push Instruction	Inputs	Output	Operation
vector.add	vector, vector	vector	Vector addition
vector.subtract	vector, vector	vector	Vector subtraction
vector.multiply	vector, vector	vector	Vector multiplication
vector.scale	vector, float	vector	Scale a vector by a float value
vector.index_add	vector, int, float	vector	Add a value to vector member at index
vector.index_subtract	vector, int, float	vector	Subtract a value from vector member at index
vector.index_multiply	vector, int, float	vector	Multiply vector member at index by float value
vector.index_divide	vector, int, float	vector	Divide vector member at index by float value
vector.duplicate	vector	vector	Duplicates the top vector item
vector.random		vector	Creates a vector of random floats within a specified range
vector.random_unit		vector	Creates a random unit vector
vector.magnitude	vector	float	Computes the magnitude of a vector
vector.between	vector, vector	vector	Creates a vector between two vectors
vector.reverse	vector	vector	Reverses a vector
vector.length	vector	float	Gets the length of a vector
vector.constant		vector	Creates a constant vector
vector.from_float	float	vector	Creates a vector of where every element is the input value
vector.at_index	vector	float	Gets an element from a vector

Table 1: Push Vector Instructions Implemented

investigation is to show that adding a support population of local optimization operators improves performance, it is critical that the comparison between a traditional EA and the SuCo MA be as fair as possible. To accomplish this, all target fitness function evaluations count against the total number of evaluations for the EA when considering EA termination criteria (specifically, maximum evaluations). This means that every support individual evaluation costs at least one extra evaluation when compared to a traditional EA which reduces the total number of evaluations available to the primary population's evolutionary process (less primary population generations will be executed). For the SuCo MA to be competitive in performance with this method of counting evaluations, the evaluations spent by the support individual fitness function must result in, on average, better performance than the traditional evolutionary cycle alone.

4 EXPERIMENTAL SETUP

The primary focus of this research is to investigate the use of evolved local optimizers as a support population in a SuCo MA where the support individuals are used as the memetic operator. To this end, experiments were conducted comparing a traditional EA with no local optimization operator, a MA with a simple hill climber, and the proposed SuCo MA. Since all additional evaluations performed by the local optimizers in the SuCo MA are counted in the total number of evaluations, halting experiments at the same number of total evaluations should create a fair experimental setup for comparison. Further still, the parameters used for all experiments were the same even though they were found by tuning the traditional EA (no tuning was performed for the SuCo MA primary population). Experimental parameters used can be seen in Table 2. Parameters in parentheses are support population parameters that were different from the primary population. The Shifted Rosenbrock

Parameter Name	Shifted Rastrigin	Shifted Rosenbrock
Evaluations	100,000	100,000
Population Size	418 (355)	355
Offspring Per Generation	6 (2)	2
Mutation Rate	0.009738 (0.10)	0.213557
Mutation Step Size	0.89	0.033593
Parent Tournament	172 (339)	339
Survival Tournament	421 (344)	344
Evals Per Support Individual	1	1
Generations Between Optimization	12	16

Table 2: Experiment Parameters

primary population parameters were used as the support population parameters in both problems because they resulted in higher support population fitness values. The support population mutation rate on the Shifted Rastrigin problem is the only parameter that was adjusted by hand. The values shown in Table 2 were hand-tuned to help provide more stability in the support population.

The last two rows in Table 2 show parameters which were unique to the SuCo MA. Evals per support individual represents the number of primary population evaluations expended per support individual fitness calculation. More details on this parameter are provided in Section 6.1. Generations between optimization is a parameter which controls the ratio of primary population to local optimizer support population generations. For example, on the Shifted Rastrigin problem a value of twelve was used. This means that every twelve primary population generations, one generation of the support population was executed. It also means that local optimization is only applied to the primary population every twelve generations. Further discussion on this parameter can be found in Section 6.2. The number of generations between optimization parameter was experimentally tuned and a similar generational gap tuning was applied in the traditional MA to help provide a fair comparison.

Several benchmark functions were selected in order to test the proposed method and to produce results for comparison and analysis. Each benchmark problem was implemented as a shifted variant using a random shift vector that was generated on a per run basis. The shift vector was added to the solution candidates gene by gene immediately before fitness calculation. This makes the benchmark problems much more difficult when compared to their original formulations for two reasons. First, it moves the global optimum away from values that are easy to reach such as zero. With a random shift, the global optimum will usually be at a location that is much more difficult to find. A random shift per gene also ensures that each gene has a different global optimum value. This makes the problem harder to solve because simply copying good gene values from one index to another does not necessarily create a high quality individual. Generating the offset randomly per run over sufficiently large experiments also helps guarantee fairness for comparisons and can also yield some insights into performance in some situations.

In order to precisely define the benchmark problems, let the vectors x represent the solution vector encoded by solution candidate, o represent the random shift vector for the problem instance, and z represent the shifted gene values calculated for fitness evaluation

using a solution candidate and the shift vector as shown in Equation 1. The vector z will be used to define the fitness functions used for experimentation and analysis.

$$z = x + o, x = [x_1, x_2, \dots, x_n], o = [o_1, o_2, \dots, o_n] \quad (1)$$

The Shifted Rastrigin function is given in Equation 2 and was selected to provide a highly multi-modal test problem.

$$\sum_{i=1}^n [z_i^2 - 10\cos(2\pi z_i) + 10] \quad \forall z \in [-5.12, 5.12] \quad (2)$$

Equation 3 defines the Shifted Rosenbrock function which was selected to represent a problem with gene interdependence.

$$\sum_{i=1}^{n-1} [100(z_{i+1} - z_i^2)^2 + (1 - z_i)^2] \quad \forall z \in [-5, 10] \quad (3)$$

5 RESULTS

The results listed in Table 3 and Table 4 show the performance of each EA on the Shifted Rastrigin and Shifted Rosenbrock benchmark problems respectively across different problem dimensions with bold text representing the highest performance. Each experiment consisted of thirty runs and the results shown in each cell list the mean final best fitness for each experimental configuration with the corresponding standard deviation in parenthesis. A set of Mann-Whitney U tests with a confidence level of $\alpha = 0.01$ were performed comparing the results of the SuCo MA to the traditional EA and MA. The SuCo MA was shown to have the highest performance on all problem configurations except the 50-dimensional Shifted Rastrigin problem. As seen in the results, the performance on the 50-dimensional Shifted Rastrigin problem was very similar between the traditional EA and the SuCo MA and the Mann-Whitney U test confirmed that there was no statistical difference between the two. Table 5 provides two simple examples of push programs evolved during experimental runs.

6 DISCUSSION

During the course of the investigation, the sensitivity of some parameters unique to the SuCo MA were studied. These parameters influenced the performance of the algorithm which inspired further experiments to understand their impact. The first parameter investigated was the number of primary population individuals used in each support individual's evaluation. Initially, it was expected that

Algorithm	50D	100D	200D
Traditional EA	-0.071 (0.025)	-0.48 (0.11)	-8.14 (1.14)
MA with Hill Climb	-0.085 (0.022)	-0.47 (0.12)	-8.18 (1.65)
MA with SuCo	-0.075 (0.022)	-0.38 (0.080)	-7.05 (1.02)

Table 3: Shifted Rastrigin Results: mean final best fitness with standard deviation in parenthesis

Algorithm	50D	100D	200D
Traditional EA	-75.57 (39.43)	-179.37 (66.48)	-389.26 (98.51)
MA with Hill Climb	-119.29 (71.77)	-292.77 (125.95)	-538.73 (142.05)
MA with SuCo	-45.99 (13.68)	-123.02 (41.52)	-268.47 (56.50)

Table 4: Shifted Rosenbrock Results: mean final best fitness with standard deviation in parenthesis

Problem	Push Program
Rastrigin 200D	(vector.constant vector.between vector.at_index integer.duplicate float.add vector.random_unit vector.at_index vector.between vector.at_index)
Rosenbrock 50D	(float.random vector.at_index float.sin 26 vector.random vector.duplicate vector.divide vector.index_divide vector.multiply float.duplicate float.random)

Table 5: Examples of evolved push programs **Figure 2: Fitness versus evaluations**

multiple primary individuals would be required to adequately assess the support individual's fitness. However, this was not the case as a single primary individual evaluation per support individual was found to be optimal. This is discussed further in Section 6.1.

It was also determined that applying local optimization every primary population generation is sub-optimal in the selected benchmark problems: a generational gap between local optimization applications is beneficial. Section 6.2 provides results and some analysis of this parameter. As previously mentioned, there are a few ways to deal with the results of local optimizer execution. Section 6.3 discusses this topic further with some analysis of various methods.

Figure 2 provides fitness versus evaluations results for each algorithm on different dimensions of both benchmark problems. Figure 2c demonstrates that more runs could have improved the performance of all three algorithms on the Shifted Rastrigin benchmark problem since none of the populations had converged at 100,000 evaluations.

6.1 Evaluations Per Local Optimizer

When a local optimizer is being evaluated, it can be executed on any number of primary population solutions. In order to understand the impact of this parameter, a set of experiments was conducted in which the parameter was varied. The 200-dimensional Shifted Rastrigin and the 50-dimensional Shifted Rosenbrock problems were used as the benchmark functions. Table 6 shows the results of the experiments using the values one through five. It can be seen that the best performance for both problems was observed when only a single primary individual was used per support individual evaluation. Every additional primary individual evaluation per support individual decreased performance. This could be attributed to a few different factors.

It could be the case that applying a local optimizer to a single individual is a good enough measure of support fitness and spending additional evaluations on the same support individual is sub-optimal. This could be caused by poor genetic diversity among the primary population offspring. If the offspring were all genetically similar, a local optimizer could have roughly the same performance on each individual which would mean one trial would be a good approximation of the support individual's true fitness for the current set of primary offspring. It could also be the result of optimization operators that tend to either generalize well, or perform very poorly overall. If an optimizer generalized well, it would have roughly the same good performance over a number of trials. Conversely, if an operator was low quality it would have on average the same poor performance over a number of trials. Thus, spending more than one evaluation results in a waste which reduces performance by consuming valuable evaluations that could have been more optimally spent elsewhere. It is also possible that limiting each support individual to one primary population individual per evaluation helps to prevent spending excess evaluations on primary offspring that have poor fitness and can not be improved by the optimizers.

6.2 Selective Optimization

Since the local optimization step in the SuCo MA is not required to generate primary population offspring, it is possible to selectively perform local optimization. One method of selecting when to

perform optimization is simply choosing a number of generations which skip the optimization step followed by a generation where it is executed. For example, every other generation, every third generation, etc. To better understand the impact of this parameter on the performance of the SuCo MA, a set of experiments were conducted where the number of generations between optimization application was varied. Table 7 shows the results of applying different generational gaps between generations that receive optimization on both benchmark problems. Two important insights can be drawn from these results. First, it can be sub-optimal to apply SuCo local optimization to every generation. Second, the optimal generational gap parameter can be different for different problems.

One possible reason for the performance improvement is generational gaps reduce the number of evaluations consumed by the local optimizer support population which allows the primary population to experience a larger number of generations. This implies that spending evaluations on further evolution of the primary population is more effective at solving the problem than applying local optimization. However, it can be seen from the results in Table 7 that spending the correct number of evaluations on local optimization can result in slightly improved performance. This indicates that there are certain points in the evolutionary run where further evolving the primary population is optimal and there are other points in the same run where spending evaluations on local optimization is more optimal.

Another possible explanation is that the primary population can require some amount of time after optimization is applied to offspring before new offspring worth optimizing are produced. In essence, this allows the evolutionary process to explore more of the solution space between optimizations. This could save evaluations that would have been spent optimizing offspring that are too genetically similar to previously optimized individuals which prevents a meaningful increase in fitness.

6.3 Handling Optimization Results

As previously mentioned, there are a few ways of handling an individual post optimization. The optimization target individual could have only its fitness updated with the optimized fitness value but without changing the genes. The optimization target could be updated with both the optimized genes and the optimized fitness value. Yet another approach is to create a new offspring with the optimized genes and fitness value and leave the original individual unchanged. Table 8 shows the results of several experimental runs on both the 200-dimensional Shifted Rastrigin and 50-dimensional Shifted Rosenbrock problems. Each column lists the results of a different method of handling the output of individual optimization.

It is interesting to note that only replacing the fitness value dramatically reduced performance. The difference between creating a new offspring and updating the target individual's genes and fitness value was small, but creating a new offspring does improve performance slightly.

7 CONCLUSION

There are many methods by which EA performance can be improved. One promising approach that has been studied is applying local learning to individuals during the evolutionary cycle. While

Problem	1 Eval	2 Evals	3 Evals	4 Evals	5 Evals
Shifted Rastrigin 200D	-7.05 (1.02)	-8.33 (1.84)	-8.36 (1.35)	-9.11 (1.29)	-9.69 (1.91)
Shifted Rosenbrock 50D	-45.99 (13.68)	-50.42 (12.84)	-52.18 (26.08)	-53.54 (29.80)	-54.11 (29.91)

Table 6: SuCo MA results with different evaluations per local optimizer

Generations	Shifted Rastrigin 200D	Shifted Rosenbrock 50D
0	-11.97 (1.84)	-55.18 (28.15)
4	-7.74 (1.32)	-54.59 (22.81)
8	-7.21 (1.10)	-54.76 (27.11)
12	-7.05 (1.02)	-51.95 (20.76)
16	-7.08 (1.68)	-45.99 (13.68)
20	-7.72 (1.62)	-54.72 (24.17)
24	-7.73 (1.51)	-53.72 (22.46)
28	-7.60 (1.19)	-59.04 (31.38)

Table 7: SuCo MA results with different number of generations between optimization

Problem	Fitness	Fitness and Genes	New Offspring
Shifted Rastrigin 200D	-2163.83 (46.16)	-7.44 (1.63)	-7.05 (1.02)
Shifted Rosenbrock 50D	-27619.38 (1051.33)	-49.95 (13.81)	-45.99 (13.68)

Table 8: SuCo MA results with different methods of handling optimization results

the right local optimization operator can improve performance, a poor choice can also decrease performance. The optimal choice may be hard to find and may not be static. This investigation explored the combination of Push GP local optimizers and SuCo to create a MA with a support population of local optimizers as another method of increasing EA performance. It was shown that EA performance can be improved using the proposed method with the selection of a few parameters. These parameters are relatively easy to find and it was shown that the performance sensitivity to these parameters is small.

8 FUTURE WORK

The results shown here are promising, but there are a number of additional investigations that need to be performed to further explore the potential of using SuCo to evolve local optimizers as the memetic operators within a MA. Further investigation should be performed which compares this work to other local optimization methods in state of the art MAs to better understand its capabilities.

The local optimizers evolved here are typically very simple vector operations which often seem to resemble vector mutations. While this seems to perform well, there could be more advanced operations that may perform even better. Some work needs to be done to determine if more advanced push GP local optimizers can have higher performance. If this is found to be the case, the support population EA could be improved to produce more advanced, higher performance optimizers which could then lead to better overall performance. It may be the case that a more advanced implementation is required to produce more complex operators. Different primitives or Push instructions could also help to improve performance as the

set presented here is relatively small and somewhat basic. Applying local optimizers to random offspring could also be responsible for the mutation like operations. A more complex scheme which relates primary offspring to local optimizers could help produce more informed optimization algorithms.

A study to understand the difference between local optimizer generalization versus local optimizer specialization could also yield some critical insights. Allowing local optimizers to specialize would mean allowing the local optimizers to specialize for a certain primary individual or area of the solution space. One way in which this could be investigated is by dividing the primary population into neighborhoods or demes using a diffusion model EA. The support population would be divided in the same way so that a deme of support individuals is mapped directly to a deme of primary individuals. This mapping would be fixed so that a support deme could specialize to its assigned primary population deme. This could allow the local optimizers to better learn targeted areas of the solution space around primary population individuals which could help them further improve performance.

Another area that requires further investigation is the impact of applying local optimization at select primary generation intervals. The root cause of this has not been fully explored and only a few guesses have been made as to why this improves performance. If the reason for this is discovered, it could lead to improvements that yield higher performance when compared to this work.

ACKNOWLEDGEMENTS

This work was supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 570204.

REFERENCES

- [1] Carlos Cotta, Luke Mathieson, and Pablo Moscato. 2017. *Memetic Algorithms*. Springer International Publishing, Cham, 1–32. https://doi.org/10.1007/978-3-319-07153-4_29-1
- [2] A. E. W. de Landgraaf and V. Nannen. 2007. Parameter Calibration Using Meta-Algorithms. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC'07)*. Singapore, 71–78.
- [3] Diaşan and M. Oltean. 2006. Evolving crossover operators for function optimization. In *Proceedings of the 9th European Conference on Genetic Programming (Lecture Notes in Computer Science)*, Vol. 3905. Springer, Budapest, Hungary, 97–108.
- [4] Brian W. Goldman and Daniel R. Tauritz. 2011. Meta-Evolved Empirical Evidence of the Effectiveness of Dynamic Parameters. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '11)*. Dublin, Ireland, 155–156.
- [5] Brian W. Goldman and Daniel R. Tauritz. 2011. Self-Configuring Crossover. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '11)*. Dublin, Ireland, 575–582.
- [6] Brian W. Goldman and Daniel R. Tauritz. 2012. Supportive Coevolution. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12)*. Philadelphia, Pennsylvania, USA, 59–66.
- [7] 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 (GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 1043–1050. <https://doi.org/10.1145/2739482.2768456>
- [8] Nathaniel R. Kamrath, Brian W. Goldman, and Daniel R. Tauritz. 2013. Using Supportive Coevolution to Evolve Self-Configuring Crossover. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '13)*. Amsterdam, The Netherlands, 1489–1496.
- [9] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. 2019. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation* 27, 1 (2019), 3–45.
- [10] Mark William Shannon Land. 1998. *Evolutionary algorithms with local search for combinatorial optimization*. Ph.D. Dissertation. University of California, San Diego.
- [11] Michael A. Lones. 2019. Instruction-Level Design of Local Optimisers using Push GP. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion)*. Prague, Czech Republic, 1487–1494.
- [12] 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 (GECCO '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1497–1504. <https://doi.org/10.1145/2464576.2482728>
- [13] Matthew A. Martin and Daniel R. Tauritz. 2014. A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO Comp '14)*. Association for Computing Machinery, New York, NY, USA, 1389–1396. <https://doi.org/10.1145/2598394.2609872>
- [14] Samuel N. Richter, Michael G. Schoen, and Daniel R. Tauritz. 2019. Evolving Mean-Update Selection Methods for CMA-ES. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1513–1517. <https://doi.org/10.1145/3319619.3326827>
- [15] Samuel N Richter and Daniel R. Tauritz. 2018. The automated design of probabilistic selection methods for evolutionary algorithms. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion*. Kyoto, Japan, 1545–1552.
- [16] Ekaterina A. Smorodkina and Daniel R. Tauritz. 2007. Toward automating EA configuration: the parent selection stage. In *Proceedings of CEC 2007 - IEEE Congress on Evolutionary Computation*. Singapore, 63–70.
- [17] Lee Spector. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. 137–146.
- [18] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 Execution Stack and the Evolution of Control. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO '05)*. Association for Computing Machinery, New York, NY, USA, 1689–1696. <https://doi.org/10.1145/1068009.1068292>
- [19] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3 (03 2002), 7–40. <https://doi.org/10.1023/A:1014538503543>
- [20] 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 (GECCO '12)*. Philadelphia, Pennsylvania, USA, 67–74.