Employing Supportive Coevolution for the Automated Design and Configuration of Evolutionary Algorithm Operators and Parameters

by

Nathaniel Kamrath

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 11, 2021

Keywords: evolutionary computing, coevolution, memetic algorithm, local optimization, genetic programming, hyper-heuristics, automatic design

Abstract

Evolutionary Algorithms (EAs) can be highly complex structures requiring many operators and parameters to function. Since EA performance can be highly dependent on these parameters and operators, creating automatic methods for EA parameter and operator selection/design could provide significant performance improvements while reducing the amount of manual configuration required to implement/configure an EA. EA design/configuration is further complicated by the fact that optimal parameter and operator settings can change throughout the run of an EA. Thus, a method that can also adapt parameters and operators on the fly during the evolutionary run could provide even further performance improvements.

The research reported in this thesis leverages a technique called Supportive Coevolution (SuCo) to automate the configuration and control of mutation step size, crossover operators, and local optimization operators. Several experiments focused on floating point optimization are presented which show promising results using SuCo to configure and control both mutation step size and crossover operators. SuCo is then used to evolve local optimization operators for the local learning step of a Memetic Algorithm (MA) in what is called SuCo-MA. SuCo-MA is then extended by employing a diffusion model to encourage the evolution of deme specific local optimization strategies in SuCo-Dif-MA. This method is then applied to the Traveling Thief Problem (TTP) as a final empirical study of its effectiveness on modern, complex problems. Experimental design and empirical results are presented for all algorithm configurations. Comparisons are made for performance analysis and some points of interest are also discussed. Conclusions are presented based on the observations made during experimentation. Empirical results from benchmark testing showed that SuCo can improve performance when compared to both static parameters and operators and self-adapted parameters. SuCo-MA showed promising results on a variety of floating point benchmark functions. It was also shown that SuCo-Dif-MA can be used to generate competitive results when compared to state of the art techniques on TTP.

Acknowledgments

Various portions of the work presented in this thesis were supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 570204.

I would like to thank my advisor, Dr. Daniel Tauritz, for all of the support and guidance he provided to me throughout this journey. I would also like to thank my other committee members, Dr. Gerry Dozier and Dr. Aaron Pope, for generously volunteering their time and support. I am also grateful for all of the guidance from Chris Rawlings as a collaborator at Los Alamos National Laboratory. I would also like to thank Deacon Seals, Michael Prince, and Braden Tisdale for their feedback and support as both friends and colleagues in the BONSAI Lab at Auburn University. Finally, I would like to thank my family for the continual love and support throughout this adventure.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of computer science contains many optimization problems defined by unique objective functions. An objective function provides the mapping between a given input solution and that solution's quality. Optimization problems with a small set of possible solutions (called the solution space) are typically trivial to solve since it is possible to evaluate all possible solutions. However, many complex optimization problems have a large solution space that makes this form of brute force search infeasible. Many optimization algorithms have been developed to improve the efficiency and effectiveness of large solution space traversal. These algorithms usually employ one or more techniques that exploit aspects of the solution space to achieve performance improvements when compared to more naive approaches such as random search. Most complex optimization problems are also multi-modal which is yet another aspect that must be considered in optimization algorithm design and selection.

Evolutionary Computing (EC) [8] is one example of a class of optimization algorithms that has shown promising performance when applied to complex, multi-modal optimization problems with large solution spaces. An evolutionary algorithm (EA) utilizes a population based approach to search the solution space by evolving solution candidates. One of the drawbacks of an EA is the need to properly configure and tune the algorithm. A number of operators and parameters must be configured by an implementer in order to utilize an EA. This task can be quite daunting and may even require problem domain expertise. Supportive Coevolution (SuCo) [13] is a technique that aims to not only automate parameter and operator selection, but also perform online optimization of parameters and operators which can improve performance. This thesis provides an in depth study of SuCo applied to various operators and parameters on several different benchmark functions from both the floating point optimization domain as well as the combinatorial optimization domain.

1.1 Evolutionary Algorithms

In an EA, the initial population of individuals containing candidate solutions are generated using one of a number of strategies, most commonly random initialization, and then evaluated by a fitness function (the target optimization problem's objective function, if necessary appropriately transformed to be a maximization problem). Each individual receives a fitness value which measures the quality of its encoded candidate solution when evaluated against the fitness function. New individuals are then generated from a set of parents typically by using a form of genetic recombination. This results in offspring that are composed of genetic material donated by its parents. Offspring, after typically being mutated in a stochastic manner, are then evaluated by the fitness function to measure their quality. Some or all individuals then compete for survival and the chance to reproduce. The selection pressure induced by the competitive aspects of selection and survival stochastically guide the population to higher fitness areas of the solution space.

1.2 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, multimodal 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 solution spaces, EAs tend to favor exploration over exploitation to avoid local optima. In particular, they can perform coarse grain searches very effectively, but they can sometimes struggle to fine tune solutions during the evolutionary process [21]. Tailoring an EA to favor exploitation typically results in pre-mature convergence of the population when a more optimal solution could have been found. However, since 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) [6]. This has proven to be a very effective method of improving EA performance.

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.

1.3 Genetic Programming

Genetic Programming (GP) is a type of EC where the trial solutions encoded in the individuals comprise of dynamic length, typically hierarchical, representations, for model identification, most popularly for automated programming. Meta-heuristics which automate the design of programs are known as hyper-heuristics, and GP is by far the most popular type of hyper-heuristic. In addition to complex representations and stochastic search components, they have in common that they generate heuristics (programs) to solve a given problem instead of evolving direct solutions.

1.3.1 PushGP

Genetic programs can be represented in a number of different ways with various trade-offs [16]. Push is a GP representation specifically designed for expressing evolved programs at a low level with minimal restrictions on program syntax and ordering [35, 37, 38]. 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.

1.4 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 [20].

One form of machine learning for automated algorithm selection is a method of search for generating heuristics called generative hyper-heuristics. In [26] 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 [25] 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 [33] and local optimization algorithms [23].

1.5 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 [7] 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) [12]. As a follow up to SCX, [18] 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 [42]. Selection operators were also evolved in [32] which employed a Hyper-heuristic approach to explore selection algorithm search space. The parent selection process was automated in [34] where mate selection operators were evolved in a GP style.

1.6 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 [13]. 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 coevolved 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 [22]. Second, it has been shown that optimal parameters and operators can change during the evolutionary process [11]. 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 [18].

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.

1.7 Diffusion Model EAs

Canonical EAs are panmictic; i.e., every individual in the population can mate with any other individual in the same population. A diffusion model EA employs a segmentation strategy to create logical divisions in the population which are referred to as demes. Evolutionary operations are restricted to operating on demes instead of the population as a whole which allows the demes to be more spread out around the solution space. Demes overlap which creates instances of cross-deme breeding, but it is generally more common for selection and mating to be performed within a deme. This technique of performing more localized selection and mating within demes has been shown to produce superior results when compared to unsegmented, population wide selection and mating [5]. The concept of structured populations has also been extended to produce new algorithmic models which have proven to be effective when solving complex problems [2].

1.8 Contributions

The content presented in Chapter 2 was published in the ECADA workshop track at GECCO 2013 [18]. It expands upon the original investigation into SuCo [13] and provides the foundation for evolving more complex support populations such as a crossover operator. It also investigates the evolution of multiple support populations for a single primary population.

Chapter 3 is based on an investigation that was published in the ECADA workshop at GECCO 2020 [19]. It provides implementation details and experimental results for a method of evolving local optimizers as a support population which are utilized by the primary population as the local learning operators of a MA called SuCo-MA.

Chapter 4 enhances SuCo-MA by adding a mapped diffusion model to the primary and support populations creating a method called SuCo-Dif-MA. The purpose of this extension is to encourage deme specific optimization strategies to evolve in the support population demes. Finally, Chapter 5 details an investigation which applies SuCo-Dif-MA to the Traveling Thief Problem.

Chapter 2

Using Supportive Coevolution to Evolve Self-Configuring Crossover

Creating an Evolutionary Algorithm (EA) which is capable of automatically configuring itself and dynamically controlling its parameters is a challenging problem. However, solving this problem can reduce the amount of manual configuration required to implement an EA, allow the EA to be more adaptable, and produce better results on a range of problems without requiring problem specific tuning. Using Supportive Coevolution (SuCo) to evolve Self-Configuring Crossover (SCX) combines the automatic configuration technique of multiple populations from SuCo with the dynamic crossover operator creation and evolution of SCX. This chapter reports an empirical comparison and analysis of several different combinations of mutation and crossover techniques including SuCo and SCX. The Rosenbrock, Rastrigin, and Shifted Rastrigin benchmark problems were selected for testing purposes. The benefits and drawbacks of self-adaptation and evolution of SCX are also discussed. SuCo of mutation step sizes and SCX operators produced results that were at least as good as previous work, and some experiments produced results that were significantly better.

2.1 Introduction

The performance of most Evolutionary Algorithms (EAs) is strongly correlated with the configuration of many parameters and operators [22]. The optimal configuration of an EA is typically problem specific and usually requires the knowledge and skills of an expert to obtain [34]. Furthermore, it is likely that the optimal parameter values and operators change throughout the run of an EA [11]. One solution to these problems is an EA which can configure itself and optimally adjust its parameters and operators throughout its run. This would help reduce the need for an expert while reducing the need for parameter tuning, giving the EA the potential to adapt to different problems and perform well without problem specific tuning. To address the previously stated issues, this chapter introduces a novel EA which employs Supportive Coevolution (SuCo) to evolve Self-Configuring Crossover (SCX). The evolution of SCX by means of SuCo creates an EA which can benefit from the strengths of both techniques. SuCo allows for the evolution of parameters and operators in support populations along with the primary solution population. The support populations supply the primary population with the best parameters and operators found at that point. When SCX is added to SuCo in a support population of crossover operators, more flexibility is added and the need for configuration is further ameliorated.

2.2 Background

Previous work has been performed on automatic parameter and operator creation and evolution. In [42], the automatic creation of Genetic Algorithm (GA) mutation operators was discussed. [29] implemented control methods for all the parameters in an EA. This method was more generally applicable than less adaptive EAs; however, it could not achieve the same solution quality as a correctly tuned EA on specific problems. Self-adaptation of EA parameters has also previously been investigated; in [14, 41] the offspring operators were controlled. However, self-adaptation has limitations. When a suboptimal mutation method randomly creates a single high quality individual, there is no guarantee that the self-adapted operator will create another high quality individual. This individual may have difficulty creating high-quality offspring of its own due to its self-adapted genes. Thus, a method is needed to separate the operator's fitness from the individual's fitness.

2.2.1 Supportive Coevolution

SuCo, thus far, has only been used to evolve mutation step sizes [13]. Mutation step size individuals were represented as lists of floating point numbers which contained the same number of genes as a primary individual. Then, when mutation was performed, each mutation step size gene was used to control a Gaussian mutation which was added to the corresponding locus in the primary individual's genes. SuCo has the potential to evolve more than a single parameter. Since it has been shown that evolving a parameter with SuCo produces promising results, this chapter describes research aimed at discovering if evolving multiple populations of support individuals by means of SuCo is a feasible approach to further automating EA configuration. Additionally, this research has another novel aspect. As the evolution of an entire operator through SuCo has yet to be explored, evolving crossover operators is also a novel extension of previous work.

2.2.2 Self-Configuring Crossover

With the ability of SuCo to evolve support populations of parameters and operators, it would be optimal to create support populations of individuals which are designed to be easily evolvable and highly expressive. The operator support individuals should be evolvable by standard EA techniques. With operators, this task is slightly more complicated. [7] employed a Meta-Genetic Program (Meta-GP) to evolve crossover operators. However, this method is unable to adapt within an EA run and is extremely computationally expensive. SCX is a method of automatic crossover creation and evolution which is computationally feasible and can be easily evolved by an EA [12].

SCX is encoded employing a linear structure similar to Linear Genetic Programming (LGP) and composed of a list of primitive functions. The first primitive function, Swap, was designed to represent crossovers that move genetic information between parents and between positions in a single parent such as n-point crossover, uniform crossover, and most permutation based crossovers. The second primitive function, Merge, was designed to represent crossovers that create genetic material by combining genes in a manner similar to arithmetic crossover. Both primitives use three parameters. The first two parameters correspond to the gene locations (in either parent) where the operation will occur. The final parameter depends on the primitive function. Swap's final parameter is a width which determines how many genes after each gene location should be swapped. This allows for blocks of consecutive genetic information to remain together. Merge's final parameter is a weight. This weight is used to combine the value at each selected location which is represented by the first two parameters in the Merge primitive. Equation 2.1 shows how Merge combines two values using a weighted average where $g(i)$ is the gene at the fist position, $g(j)$ is the gene at the second position, and α is the weight.

$$g(i) = \alpha \cdot g(i) + (1 - \alpha) \cdot g(j) \tag{2.1}$$

When using SCX to perform a crossover, the genes of both parents are first concatenated. Then, the copy of the genes is modified by applying each primitive in the SCX individual in sequence. After each primitive has been applied to the list of genes, half the combined genome is used to create an offspring. To allow for further dynamic behavior, each parameter in a primitive can be set in multiple ways. The first is the Number construct, which uses a static value set at primitive creation every time the primitive is used. The second is the Random construct which creates a new random number every time the primitive is used. The final construct is the Inline construct. Inline forces the genetic operations to be performed between the same gene locations in each parent. See Figure 2.1 for an example of applying SCX. In previous work, SCX was evolved using techniques similar to self-adaptation. This was achieved by giving every primary solution individual its own crossover operator. Thus, there was no separate population for the SCX operators.

SCX has several qualities which make it attractive as a candidate for an operator in SuCo. First, SCX reduces the search space of crossover operators by using a structure similar to LGP and by basing its functionality on primitives. Since the search space of crossover operators is very large and the operators need to be evolved within a single run of the EA, the reduction of search space is vital to the search for optimal crossover operators. Second, even with the reduced search space, SCX still has the potential to be extremely dynamic. Its size, primitive combination, parameters, and even the way the parameters are defined, are all dynamic properties of SCX. Lastly, SCX's structure makes it easy to evolve using standard EA operations. With these reasons in mind, and the fact that SCX has shown promising results [12], SCX has been chosen as the dynamic crossover operator to be evolved by SuCo in the research presented in this chapter.

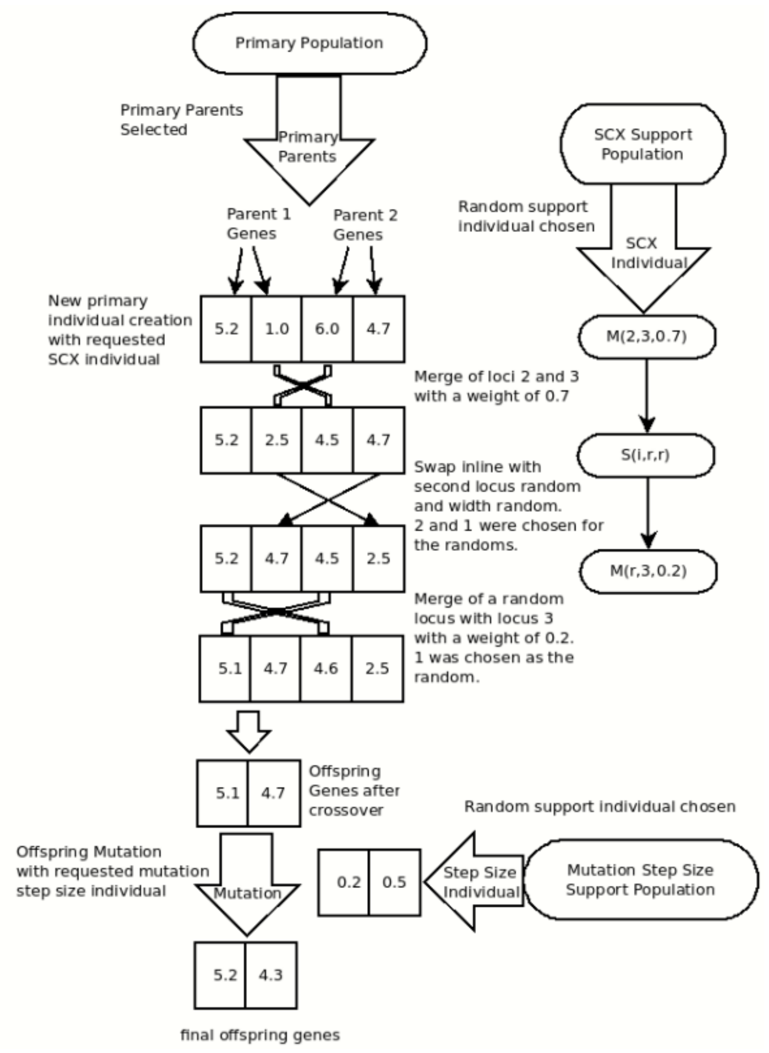

Primary Population
Primary Parents Selected
Primary Parents
SCX Support Population
Parent 1 Genes
Parent 2 Genes
Random support individual chosen
SCX Individual
New primary individual creation with requested SCX individual
5.2 1.0 6.0 4.7
M(2,3,0.7)
Merge of loci 2 and 3 with a weight of 0.7
5.2 2.5 4.5 4.7
S(i,r,r)
Swap inline with second locus random and width random. 2 and 1 were chosen for the randoms.
5.2 4.7 4.5 2.5
M(r,3,0.2)
Merge of a random locus with locus 3 with a weight of 0.2. 1 was chosen as the random.
5.1 4.7 4.6 2.5
5.1 4.7
Offspring Genes after crossover
Random support individual chosen
Offspring Mutation with requested mutation step size individual
Mutation
0.2 0.5
Step Size Individual
Mutation Step Size Support Population
5.2 4.3
final offspring genes

Figure 2.1: SCX and Mutation Support Individuals in Offspring Creation

2.3 Methodology

SuCo was implemented with two support populations: one for mutation step sizes and one for crossover operators as shown in Figure 2.2. The mutation step size individuals were represented using lists of floating point numbers. The crossover individuals were implemented using SCX. When the EA starts, each support population is randomly initialized along with the primary population. After initialization, the creation of each new individual requests a crossover operator and a mutation operator from the support populations. The SCX operator creates an offspring from two parents. Then, the offspring is mutated by applying the mutation step sizes from each of the mutation support individual genes to the offspring genes at the same locus. Support individuals are chosen randomly and the same support individual combinations

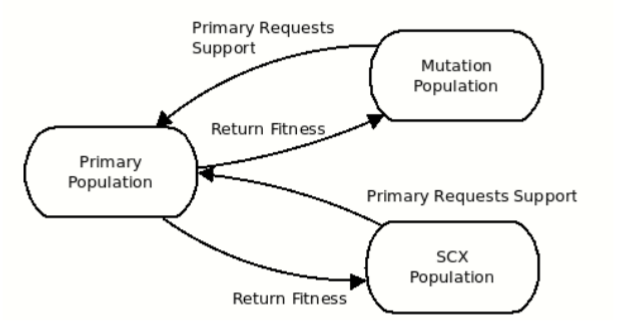


Figure 2.2: SuCo Primary and Support Population Interaction

are prevented to ensure different combinations of support individuals are evaluated. Figure 2.1 demonstrates the production of offspring using the described method.

Once the new primary individual is created and evaluated, the support individuals that were requested for its creation are assigned the product of Equation 2.2 and Equation 2.4 as fitness where Equation 2.3 is used to compute the distance between two individuals. This encourages genetic exploration while awarding fitness improvements. Each support individual can be evaluated multiple times to more accurately assess its true potential to generate quality offspring.

$$RelativeFitness = offspring - average(parents) \tag{2.2}$$

$$distance(a, b) = \sum_{i=1}^{n} (a_i - b_i)^2 \tag{2.3}$$

$$RelativeDistance = \frac{distance(o, p_1) \cdot distance(o, p_2)}{distance(p_1, p_2)} \tag{2.4}$$

Two adaptations were made in this research; the first to SuCo and the second to SCX. SuCo had to be adapted to allow one of its support populations to be configured for SCX individuals. Since SCX was originally self-adapted as part of other individuals, it was slightly modified to allow for evolutionary techniques to be executed on each individual.

The SuCo adaptations were required because a support population for a crossover operator had not been previously explored. A similar approach to the mutation step size population was adopted. Individuals were randomly initialized at the start and given to the primary population when requested by the EA throughout the run. k-tournament selection with replacement was used for parent selection and without replacement for survivor selection. The mutation and crossover methods for this support population were handled by standard SCX methodologies as specified in [12].

Since all SCX operators are being evolved in their own population, a way to perform crossover on two separate primary individuals with one crossover operator was devised. The functionality is very similar to the original SCX method. Both parents' genes are first copied and concatenated. Then, all the primitive functions of the SCX support individual are applied to the copied genes. Once this is complete, half the genetic material in the resulting list of genes is used to create an offspring. This is different from the standard SCX implementation in that it employs a single SCX individual from a separate population to perform the crossover operation.

Since both crossover and mutation are being evolved, it can become difficult to determine what to attribute the fitness of an individual to: crossover, mutation, its parents, or random chance. For many techniques, this is difficult to determine. However, SuCo explicitly attempts to separate these factors by normalizing for parent fitness, repeatedly using the same operator to adjust for noise, and using operators in different combinations to assign operator fitness independent of the other operators it is paired with. Also, since SCX is evolving in a support population as opposed to being self-adapted, there are several potential benefits as well as several potential disadvantages.

One possible disadvantage is that a crossover operator may work well with only one specific primary individual and perform poorly with all other primary individuals. This could potentially lead to the production of unfit individuals. However, evolving SCX in a support population allows the primary population to have access to fit crossover operators instead of just one crossover operator which it is attached to as in self-adapted SCX. Another benefit from SuCo of SCX is it allows the fitness of the operator to be decoupled from the primary individual's fitness. Thus, the operator can receive its own fitness which is indicative of that operator's ability to produce quality, genetically diverse offspring. This allows for better evolution of

Parameter	**Rastrigin**	**Shifted Rastrigin**	**Rosenbrock**
A	10	10	100
Genome Size	100	100	50
Evaluations	10,000	100,000	100,000
Population Size	712	418 (37)	355 (200)
Offspring Per Generation	5	6 (50)	2
Mutation Rate	0.009838	0.009738	0.213557
Mutation Step Size	1.47144	0.587875	0.033593
Parent Tournament	114	172 (35)	339 (139)
Survival Tournament	63	421 (27)	344 (190)
Evals Per Generatioin	9	1	10

Table 2.1: Experiment Parameters Used for Testing with Novel EA Parameters in Parenthesis

crossover individuals and a better SCX operator population from which the primary population can draw.

2.4 Experimental Setup

The main goal of this research is to determine if SuCo is a feasible way for EAs to perform self-configuration and evolve their own parameters and operators. Thus, results and analysis strictly based on fitness are not the only product of these tests. They will serve as a measure of the potential of SuCo when employed to evolve multiple support populations as an indication for future work.

The base settings for the parameters are shown in Table 2.1. These parameters were taken from [12] which implemented a meta-EA to search for optimal parameters. However, some basic hand tuning for Shifted Rastrigin and Rosenbrock was performed on the EA which evolves both mutation step sizes and SCX operators since this combination has never been tested before. This EA's parameters are shown in parentheses where different. All EAs used the same parameters for the Rastrigin problem. Since the goal of this research is to reduce the amount of parameter setting required by EAs, all support population parameters were identical to the primary population's parameters. Further analysis of the parameters used is available in Section 2.6.2.

Several benchmark problems were employed to test performance and produce results for analysis and comparison. The following test problems were selected to match previous work and allow for backward comparisons. For a real-valued, highly multimodal problem with no gene interdependence, the Rastrigin Function was chosen [15]. The Rastrigin Function is given in Equation 2.5:

$$An + \sum_{i=1}^{n} [x_i^2 - 10cos(2\pi x_i)] \; \forall x \; \in \; [-5.12, 5.12] \tag{2.5}$$

The Rosenbrock problem [17], as defined in Equation 2.6, was used to represent real-valued problems with gene interdependence.

$$\sum_{i=1}^{n-1} [A(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \; \forall x \; \in \; [-5, 10] \tag{2.6}$$

In both of these problems, the optimum value for each gene is the same. Thus, the Shifted Rastrigin problem, introduced in [12], was implemented as the final real-valued benchmark problem to test the EA's ability to solve problems where not every gene has the same optimum value. The Shifted Rastrigin problem is a specially designed modification of the Rastrigin problem which, at the start of every run, generates an Shifted vector O. Every element in O is randomly assigned a value from [-2.5, 2.5] at intervals of 0.5. In the Rastrigin function, if x_i is a local best, $x_i + .5$ is a local worst, and if x_i is near the global optimum, then $x_i + 1$ will be near a local optimum. This makes the problem much harder for SCX since simply swapping a gene which is optimal in one position with a gene in another position has a high probability of resulting in a poor fitness value.

Several combinations of mutation and crossover were implemented for testing. First, a static mutation step size combined with arithmetic crossover (static + arithmetic) was implemented as a base line. Next, the following combinations were implemented for empirical comparisons: static mutation step size with self-adapted SCX (static + SASCX), SuCo of mutation step sizes with arithmetic crossover (SuCo mut + arithmetic), static mutation step size with SuCo of SCX (static + SuCo SCX), SuCo of mutation step sizes with self-adapted SCX (SuCo mut + SA-SCX), and SuCo of both mutation step sizes and SCX (SuCo mut + SuCo SCX).

	Static Mutation	**SuCo Mutation**
Arithmetic	-55.925 (6.4379)	-61.249 (1.5353)
SA-SCX	-0.0072 (0.00465)	-0.01711 (0.00690)
SuCo SCX	-0.00051 (0.00233)	-0.00025 (0.00078)

Table 2.2: Rastrigin Results: mean final best fitness with standard deviation in parentheses

	Static Mutation	**SuCo Mutation**
Arithmetic	-0.12568 (0.18911)	-0.15197 (0.19091)
SA-SCX	-0.02718 (0.01818)	-0.02366 (0.01194)
SuCo SCX	-0.30794 (0.46271)	-0.02579 (0.01022)

Table 2.3: Shifted Rastrigin Results: mean final best fitness with standard deviation in parentheses

These combinations were picked to explore all possible combinations for static or evolved mutation step sizes with static, self-adapted, or evolved SCX operators.

Since none of the new techniques have significant asymptotic complexities, all experiments rely on counting evaluations as the measure of search efficiency. Experiments with SuCo count only evaluations of the primary population as the primary individuals are the only individuals who make calls to the fitness function. Support individuals are assigned a fitness value based on the results of the primary individual's fitness evaluation which means support individuals do not need to be directly evaluated.

2.5 Results

The results comparing the different combinations of mutation step size parameter and crossover operator implementation are presented in Table 2.2, Table 2.3, and Table 2.4. In these tables, each column represents the mutation method used (labelled by the first entry in the column)

	Static Mutation	**SuCo Mutation**
Arithmetic	-71.873 (39.489)	-85.468 (41.840)
SA-SCX	-28.999 (22.847)	-24.444 (23.446)
SuCo SCX	-22.638 (23.201)	-17.941 (22.824)

Table 2.4: Rosenbrock Results: mean final best fitness with standard deviation in parentheses

and each row represents the crossover method used (labelled by the first entry in each row). SA-SCX represents self-adapted SCX, SuCo SCX represents SuCo evolved SCX, and SuCo Mutation represents SuCo evolved mutation step sizes. Each experiment shows the mean final best fitness and standard deviation found over 30 runs. A series of Mann-Whitney U tests were performed comparing the results from SuCo mut + SuCo SCX to the results from all other combinations. SuCo mut + SuCo SCX was found to be the best combination on the Rastrigin and Rosenbrock problems. On the Shifted Rastrigin problem, there was no statistical difference between SuCo mut + SuCo SCX and SuCo mut + SA-SCX. A confidence level of $\alpha = 0.01$ was used in the statistical analysis.

2.6 Discussion

The results from the tests in Section 2.5 show that the SuCo mutation + SuCo SCX EA performed well on the Rastrigin and Rosenbrock problems. This was expected since allowing SCX to evolve in a support population allows for a better evaluation of SCX operators and gives the primary population access to higher quality crossover operators. However, it is interesting that the SuCo mutation + SuCo SCX EA did not perform as well on the Shifted Rastrigin problem. This could be because of its increased complexity which is difficult for SCX to overcome since different genes have different optimum values and there is no gene interdependence. Further explanation is provided in the following sections.

It is also interesting to note that one evaluation per generation was used for the Shifted Rastrigin problem. Normally, allowing for multiple evaluations per generation helps to determine the true quality of support individuals. This allows them to be evaluated in different combinations and with different primary individual parents to ensure that one bad combination does not cause a potentially fit support individual to be removed from the support population. However, it was found that performing one evaluation per generation was more beneficial on the Shifted Rastrigin problem. In order to determine why, data from the Shifted Rastrigin problem was further analyzed.

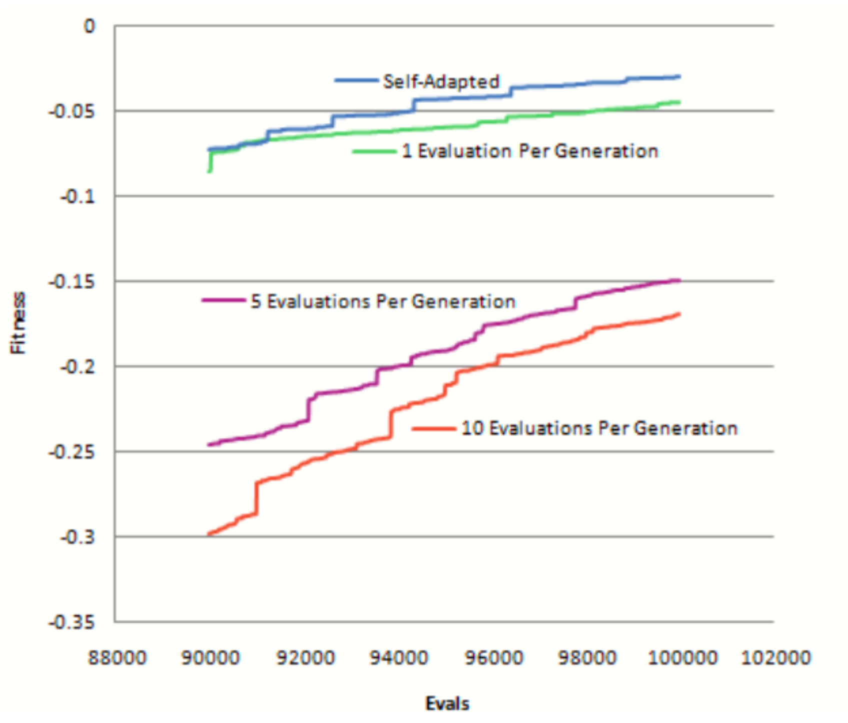


Figure 2.3: Fitness vs Evals for Different Numbers of Evaluations per Generation - 100,000 Evals

2.6.1 Evolution Versus Self Adaptation of SCX

Figure 2.3 shows mean best fitness versus the number of evaluations for the last 10,000 evaluations of several 100,000 evaluation runs. These values were obtained by averaging values from the Shifted Rastrigin problem over fifty runs for each different setting shown. The higher number of evaluations per generation produces large, sudden increases in fitness followed by periods of small improvement. The increases in fitness are more extreme than the graph may depict as the results are smoothed. The described changes in value indicate that either multiple runs all improve at identical evaluation numbers, or individual runs increased dramatically. As the number of evaluations per generation becomes smaller, the sudden increases in fitness happen more often, but have a smaller magnitude in change. As a result, support individuals are evolved more quickly and can adapt to the changing primary individuals faster. This creates the more constant, consistent fitness improvement curve observed with small numbers of evaluations per generation.

When observing the increase in fitness of the higher number of evaluations per generation, it would appear as though it would surpass the smaller number of evaluations per generation

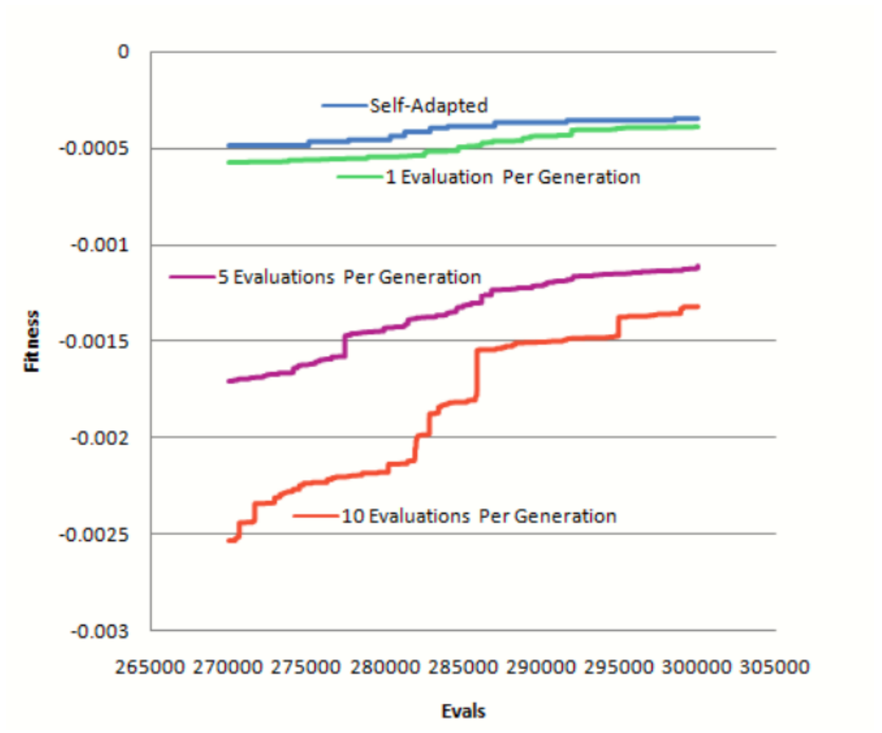


Figure 2.4: Fitness vs Evals for Different Numbers of Evaluations per Generation - 300,000 Evals

given more evaluations. However, Figure 2.4 shows the last 30,000 evaluations of a run which was allowed to continue to 300,000 evaluations. The data for Figure 2.4 was collected in the same way as the data for Figure 2.3, but with only 10 runs per experiment and the number of evaluations parameter varied. It is easy to see that the higher number of evaluations per generation still has not surpassed the lower evaluations per generation. This is because the large increases in fitness following an evolutionary cycle of the support populations become increasingly smaller and the period between evolutionary cycles of the support populations also yields significantly less improvements in fitness as the EA progresses.

As shown in Figure 2.3 and Figure 2.4, it was found that allowing for multiple evaluations per generation through SuCo mut + SA-SCX yielded only marginally different results from static + SA-SCX. It was also observed that SuCo mut + SA-SCX performed better than SuCo mut + SuCo SCX when using the same parameters and multiple evaluations per generation. However, when only one evaluation per generation was used, SuCo mut + SuCo SCX performance improved. This can be attributed to the fact that self-adapting SCX always uses one evaluation per generation. SCX is never explicitly evaluated when being self-adapted because each operator belongs to a solution individual. Since each solution individual is only evaluated once per generation, each SCX operator is as well. When being evolved and evaluated multiple times per generation, it takes more evaluations to produce the same number of new support individuals as compared to self-adaptation. On a more complex problem, such as Shifted Rastrigin, it may be easier to determine which SCX operators are more fit. Thus, one evaluation per generation of each support individual is sufficient to accurately determine the fitness of that support individual. It is also possible that one evaluation per generation allows the support individuals to evolve more quickly and continue to supply the primary population with new SCX operators. Both explanations could be reasons why SuCo performed better when performing only a single evaluation of each support individual per generation on Shifted Rastrigin.

This demonstrates one of the benefits of using SuCo to evolve SCX in a support population as opposed to selfadapting SCX. On problems where complexity requires many evolutionary cycles to produce crossover operators which are highly developed, evolving SCX can mimic self-adapting SCX, in both behavior and performance, by adjusting the evaluations per generation. This can be seen in Figure 2.3 where the results from one evaluation per generation are very similar to the results from self-adaptation. However, on simple problems where the complexity is not necessary, evolved SCX operators can be evaluated multiple times per generation to truly assess the quality of more simplistic operators and produce more fit individuals. This is evident in the Rosenbrock results presented in Table 2.4, and even more evident in the Rastrigin results presented in Table 2.2. Clearly, evolving SCX operators in support populations increases the adaptability of an EA and allows it to perform more optimally over a wider range of problems.

2.6.2 Population Size when Evolving Multiple Support Populations

Another interesting result of evolving multiple support populations is observed when considering the population sizes used on the Offset Rastrigin and Rosenbrock problems. When SuCo was used to evolve both mutation step sizes and SCX operators, a smaller population size produced better results. This could partially be due to the ability of SuCo to produce high quality individuals from limited genetic material and the potential of SuCo to explore genetically. Since

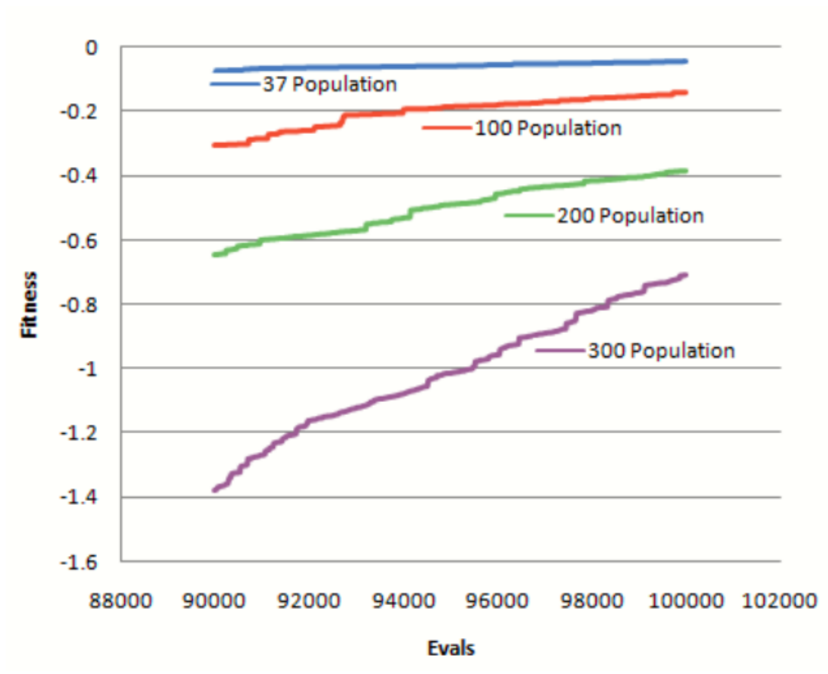


Figure 2.5: Fitness vs Evals for Different SuCo mut + SuCo SCX Population Sizes - 100,000 Evals

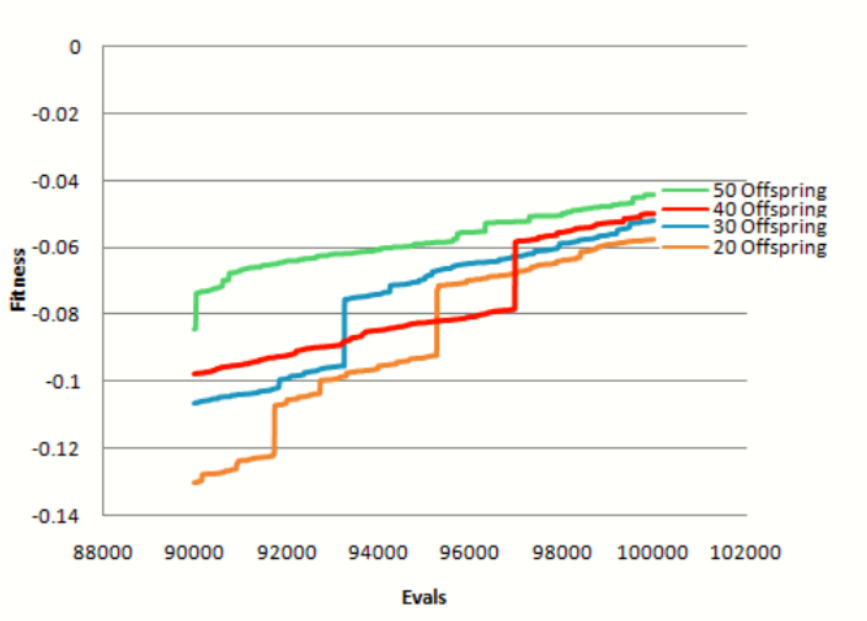


Figure 2.6: Fitness vs Evals for Different SuCo mut + SuCo SCX Offspring Sizes - 100,000 Evals

the parameters and operators are evolving to allow for a better search of the primary individual search space, less unique genetic material is needed in the primary population. The evolved parameters and operators can find new genetic material more easily which means unique genetic material does not have to be retained by individuals in the primary population. This was demonstrated by the configuration settings for the SuCo EA which evolved both mutation step sizes and SCX operators. Figure 2.5 shows the last 10,000 evaluations of several 100,000 evaluation runs where different population sizes were implemented. Each different population size experiment was performed fifty times and then averaged to produce the data shown in Figure 2.5.

Also, a higher number of offspring produced in each generation was preferred. This could be caused by the previously discussed effect which allows evolved parameters and operators to find new genetic material more easily. Another contributing factor could be that, since support individuals are encouraged to create genetic diversity, gene saturation induced by the large number of offspring produced is prevented. Figure 2.6 shows the last 10,000 evaluations of several 100,000 evaluation runs where different offspring sizes were implemented. The data for Figure 2.6 was collected in the same way as the data for Figure 2.5 with only the offspring size parameter being varied.

Another possible explanation for the small population size is the fact that the support populations all used parameters identical to the primary population. This method of setting parameters may not be optimal. The parameters observed could simply be the best parameters when considering all populations as a whole. If the support populations were each configured separately, different parameters may be found to be more optimal. This could also increase performance as the optimal configurations for each population could be found.

2.7 Conclusion

Proper EA configuration is a challenging task which often requires the experience of an expert. However, there are methods which allow an EA to configure itself optimally. An even more challenging problem is allowing an EA to reconfigure its parameters throughout its run. This ability allows the EA to adapt its parameters and operators in an attempt to find optimal settings.

SuCo and SCX are both methods which allow for EA self-configuration and dynamic parameter control.

To build on previous work and determine if adding more parameters and operators to support populations in SuCo is beneficial, a new support population composed of SCX individuals was introduced to SuCo. It was determined that evolving SCX through SuCo as opposed to self-adaptation can further add flexibility to the EA. This is due to the fact that evolved SCX can operate in a mode functionally similar to self-adaptation or completely different depending on what is more optimal for the current problem. SuCo was shown to be not only a feasible way of evolving multiple parameters and operators, but also a promising way of doing so. Using SuCo to evolve multiple populations of parameters and operators has the potential to further reduce EA configuration needs, allow for dynamic parameter control, and produce high quality results.

Chapter 3

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

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.

3.1 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 [21]. 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) [6].

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 [13] 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.

3.2 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 3.1 shows the vector Push instructions implemented for local optimizer

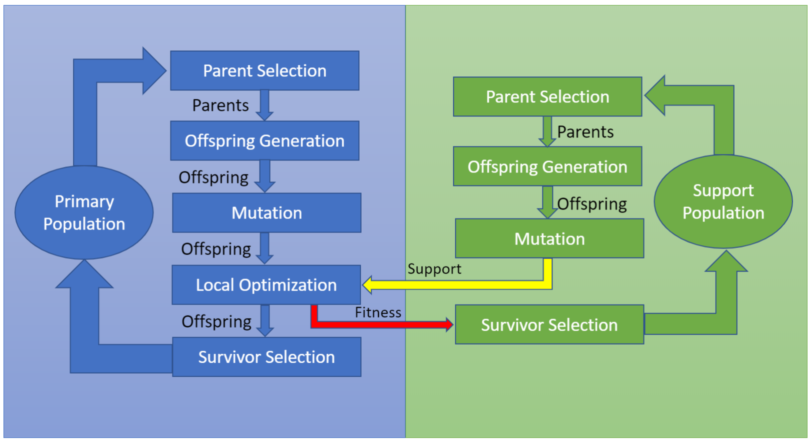


Figure 3.1: Structure of the proposed SuCo MA

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 3.1: Push Vector Instructions Implemented

construction and evolution. The push instruction set here is similar to the set used in previous work [23]. Figure 3.1 provides a visual representation of the structure implemented. The rest of this section provides low level details about the implementation of the SuCo MA.

3.2.1 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 3.5.1. 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.

3.2.2 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 3.5.2. After support has been requested (yellow arrow in Figure 3.1), 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 3.5.1 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 3.1).

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

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 3.2: Experiment Parameters

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 3.5.3.

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 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.

3.3 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 3.2. Parameters in parentheses are support population parameters that were different from the primary population. The Shifted Rosenbrock 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 3.2 were hand-tuned to help provide more stability in the support population.

The last two rows in Table 3.2 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 3.5.1. 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 3.5.2. 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 3.1. The vector z will be used to define the fitness functions used for experimentation and analysis.

$$z = x + o \ \ , x = [x_1, x_2, ...x_n], o = [o_1, o_2, ...o_n] \tag{3.1}$$

The Shifted Rastrigin function is given in Equation 3.2 and was selected to provide a highly multi-modal test problem.

$$\sum_{i=1}^{n} [z_i^2 - 10cos(2\pi z_i) + 10] \, \forall z \ \in \ [-5.12, 5.12] \tag{3.2}$$

Equation 3.3 defines the Shifted Rosenbrock function which was selected to represent a problem with gene interdependence.

$$\sum_{i=1}^{n-1} [100(z_{i+1} - z_i^2)^2 + (1 - z_i)^2] \, \forall z \ \in \ [-5, 10] \tag{3.3}$$

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 3.3: 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 3.4: Shifted Rosenbrock Results: mean final best fitness with standard deviation in parenthesis

3.4 Results

The results listed in Table 3.3 and Table 3.4 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	**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 3.5: Examples of evolved push programs

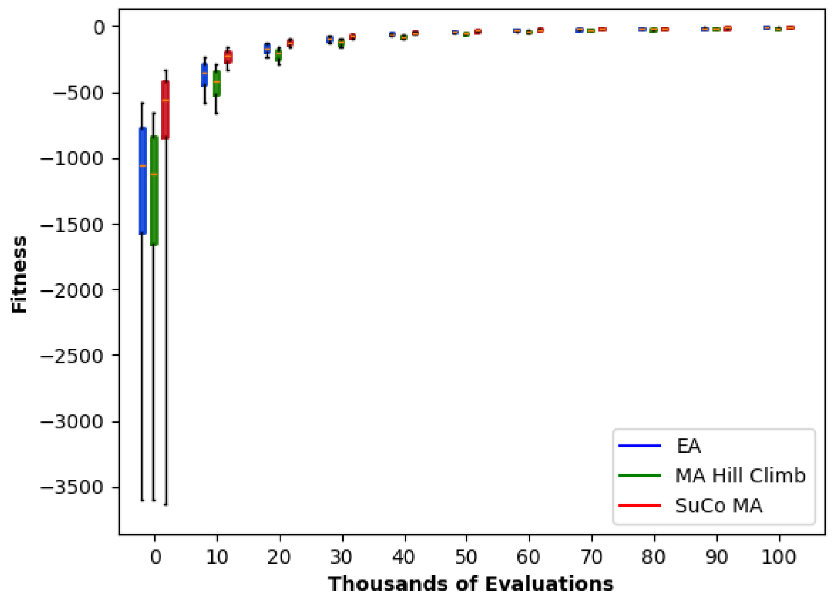


Figure 3.2: Fitness versus evaluations for Rastrigin 200 dimensions

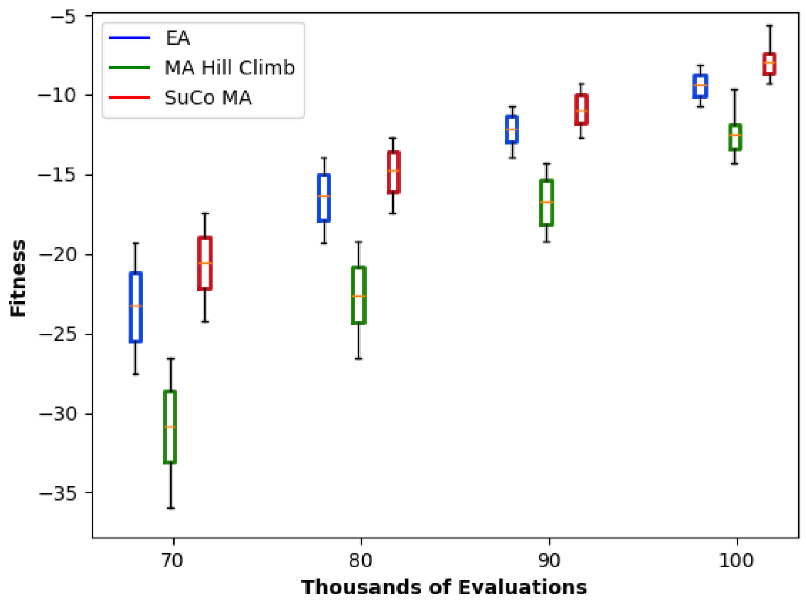


Figure 3.3: Fitness versus evaluations for Rastrigin 200 dimensions (final 30k evaluations)

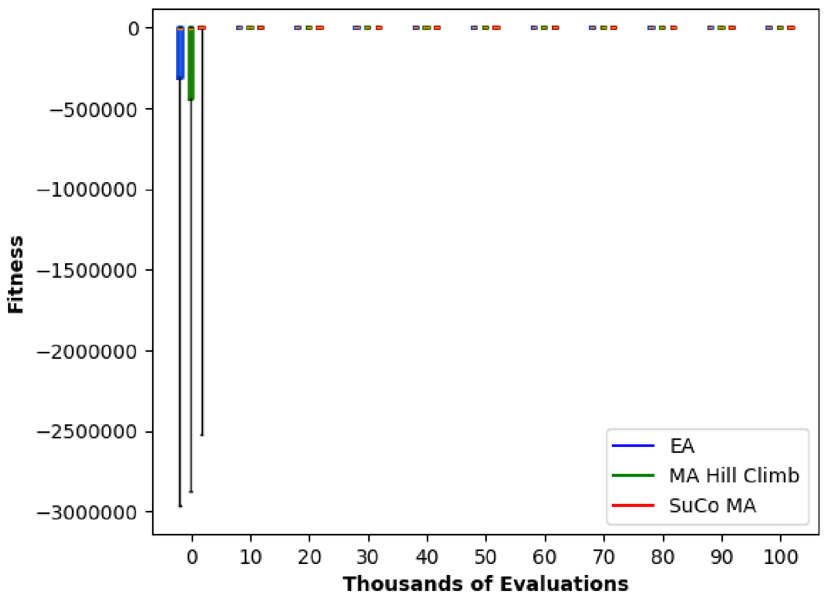


Figure 3.4: Fitness versus evaluations for Rosenbrock 50 dimensions

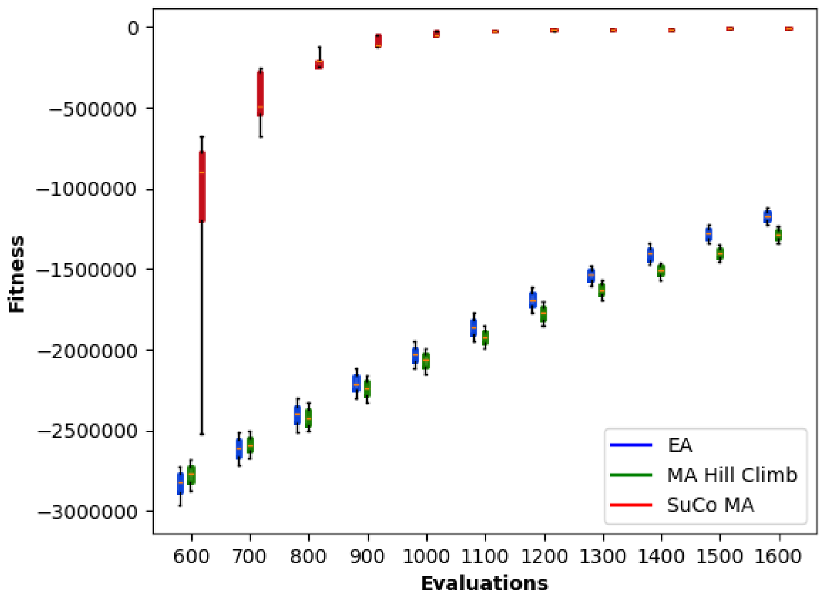


Figure 3.5: Fitness versus evaluations for Rosenbrock 50 dimensions (early in the run)

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 3.5 provides two simple examples of push programs evolved during experimental runs.

3.5 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 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 3.5.1.

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 3.5.2 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 3.5.3 discusses this topic further with some analysis of various methods.

Figure 3.2, Figure 3.3, Figure 3.4, and Figure 3.5 provide fitness versus evaluations results for each algorithm on different dimensions of both benchmark problems. Figure 3.3 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.

3.5.1 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

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

Table 3.6: SuCo MA results with different evaluations per local optimizer

50-dimensional Shifted Rosenbrock problems were used as the benchmark functions. Table 3.6 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.

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 3.7: SuCo MA results with different number of generations between optimization

3.5.2 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 3.7 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 3.7 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

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 3.8: SuCo MA results with different methods of handling optimization results

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.

3.5.3 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 3.8 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.

3.6 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 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.

Chapter 4

The Evolution of Deme Specific Local Optimizers in a Diffusion Memetic Algorithm Employing Supportive Coevolution

Memetic algorithms (MAs) are extensions of the classical evolutionary algorithm (EA) that add a local search operator to the evolutionary cycle. This local search operator can improve fine-tuning capabilities, which can improve overall performance. Implementing an MA requires the appropriate selection or design of a local search algorithm. Methods of algorithm evolution have been investigated that can reduce the burden of implementing an MA. SuCo-MA is a method of generalized local search operator evolution that employs a coevolutionary technique called supportive coevolution (SuCo).

This investigation extends SuCo-MA by implementing a diffusion model to create the novel Supportive Coevolution Diffusion Memetic Algorithm (SuCo-Dif-MA). The purpose of this combination is to create specialized local search algorithms by mapping demes of evolved local search operators to demes of solution candidates. This chapter presents the methodology used to create the SuCo-Dif-MA as well as the experimental design used to test it against competing algorithms on two benchmark problems. The results show that the proposed method improves performance on both benchmark problems. The SuCo-Dif-MA's ability to achieve higher performance is discussed, and areas for future work are provided.

4.1 Introduction

The canonical evolutionary algorithm (EA) can be enhanced by many methods. One example of such an enhancement is called the memetic algorithm (MA), which adds a local search operator to the evolutionary cycle [6]. Because EAs can struggle to fine tune solutions [21], MAs have the potential to achieve higher performance when an appropriate local search technique is employed.

When utilizing an MA, local search operator selection can be difficult due to the large number of methods available. Selecting a high-quality local search algorithm can be further complicated by problem-specific details as well as the solution representation. If there is no appropriate local search algorithm available, one must be designed by the implementer, which typically requires expert knowledge. Traditional local search algorithms are manually designed and remain static throughout the MA's evolutionary run.

This chapter describes an extension of a novel approach to automated local search algorithm design employing a generative hyper-heuristic technique. Employing genetic programming (GP) and supportive coevolution (SuCo), this technique evolves local search algorithms in parallel with target fitness function solution candidates (called the primary population). The evolved algorithms are applied to solution candidates as both a measure of fitness and a way of improving overall performance. This method, called a Supportive Coevolution Memetic Algorithm (SuCo-MA), eliminates the need for manual selection or design of a local search algorithm. SuCo-MAs also eliminate the traditionally static nature of local search algorithms by evolving the GP-encoded local optimizers in parallel with the primary population. This chapter extends the SuCo-MA method by adding a diffusion model to both the primary population and the GP local search population to create what is called SuCo-Dif-MA. Both populations are split into demes that are mapped onto each other such that local search algorithms are only applied to solution candidates from their corresponding deme. The goal of this method is to go beyond simply evolving the algorithms and randomly applying them to solution candidates by encouraging the local search algorithms to specialize to the current solution landscape location of their deme.

4.2 Methodology

The SuCo-Dif-MA is designed with a single primary population and a single support population. Each support population individual is a local optimization operator encoded as a Push genetic program. The evolutionary cycle for each population is shown in Figure 4.1. The only

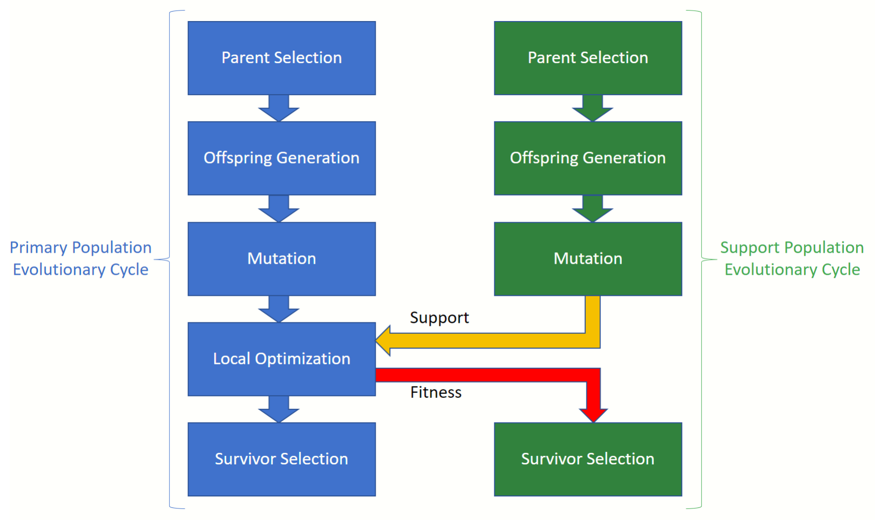


Figure 4.1: SuCo-Dif-MA Evolutionary Cycle

interaction between the two populations is shown by the two arrows connecting the pipelines. The yellow arrow labeled “support” represents the primary population requesting support in the form of operators to perform local optimization on primary individuals. The support population provides these operators, and a fitness value is returned to the support population indicating the performance of each support individual.

The diffusion model adds another layer to the overall design. In order to split the populations into demes, a parameter for the number of demes is selected and both populations are segmented into the same number of equal sized demes based on this parameter. Each deme is a two-dimensional grid composed of individuals. The population as a whole is also a two-dimensional grid composed of demes. Each individual is restricted to breeding with immediate neighbors. Due to the grid topology, this means that breeding between demes only occurs when a selected parent is on the edge of a deme. Each primary population deme only utilizes support individuals from their corresponding support deme. This feature encourages the development of specialization in the support population demes to better exploit the solution space of each primary population deme. The breeding limitations imposed by the diffusion model help to slow down the spread of genes throughout the population so that specialization has more time to occur. It also allows different demes to search different areas of the solution space without competing against individuals from other demes every generation.

Instruction	**Input**	**Output**	**Operation**
vector.+	vector, vector	vector	adds two vectors
vector.-	vector, vector	vector	subtracts two vectors
vector.*	vector, vector	vector	multiplies two vectors
vector./	vector, vector	vector	divides two vectors
vector.SCALE	vector, float	vector	scales a vector by a double
vector.INDEX+	vector, int, float	vector	adds to vector value at index
vector.INDEX-	vector, int, float	vector	subtracts from vector value at index
vector.INDEX*	vector, int, float	vector	multiplies vector value at index
vector.INDEX/	vector, int, float	vector	divides vector value at index
vector.DUP	vector	vector	duplicates the top vector
vector.RAND	none	vector	creates a random vector
vector.RAND_UNIT	none	vector	creates a random unit vector
vector.MAGNITUDE	vector	float	calculates the magnitude of a vector
vector.BETWEEN	vector, vector	vector	calculates the midpoint vector
vector.FROM_FLOAT	float	vector	creates a vector from a float
vector.AT_INDEX	vector, int	float	gets value at index
vector.MIX	vector, vector	vector	mixes two vectors together
vector.SIN	vector	vector	takes the sin of a vector by index
vector.COS	vector	vector	takes the cos of a vector by index

Table 4.1: Push Vector Instruction Set

Both primary and support populations utilize tournament selection for both parent and survivor selection. The primary population utilizes a single point crossover recombination operator and a Gaussian step mutation operator for offspring generation. The support population randomly selects between sub-tree mutation, deletion mutation, or sub-tree crossover, with equal probability for offspring generation.

4.2.1 Push Implementation

The Push implementation used was taken from [36]. Custom instructions specific to this investigation's vector benchmark functions were added to the Push interpreter. Table 4.1 provides a condensed list of vector instructions added. Other default instructions were also included but are not listed for the sake of brevity. For more information on default instructions provided by available implementations, see [36].

4.2.2 Support Individual Fitness Calculation

Support individual fitness is calculated based on the best fitness improvement found by a local optimizer after being executed on multiple primary individuals randomly selected from the appropriate primary deme. This method of fitness assignment produced on average the highest final primary fitness when compared to other methods. Section 4.5 provides further discussion related to this topic.

All support individuals are re-evaluated every support generation to ensure that support individual fitness values are representative of the current primary population's state. This technique of re-evaluation every generation is costly when considering total number of evaluations, but it prevents support individuals which were found to be high performance early in the evolutionary run from dominating the population for the rest of the run. Since large improvements can be easily found early in a run, higher fitness support individuals are common at the start of a run. Since the primary population is changing every generation and the support population fitness is based on each local optimizer's ability to improve a primary individual, re-evaluating each support individual every generation is one way to ensure that support fitness values accurately reflect each support individual's performance.

4.3 Experimental Setup

The floating point benchmark functions Schwefel and Rosenbrock were selected for empirical analysis. Both benchmark problems were implemented as shifted variants by generating random shift vectors each run in order to move the global optimum away from values that can be trivial to discover. Random shifts also ensure that not every gene has the same global optimum value. This prevents certain operations such as crossover from trivializing the benchmark problem since copying high-quality genes from one index to another does not necessarily create a higher performing individual. The benchmark problems are shifted by applying the random vector to each individual immediately before evaluation. Equation 4.1 demonstrates

Parameter Name	**Shifted Schwefel**	**Shifted Rosenbrock**
Evaluations	25,000,000	25,000,000
Population Size	405 (144)	355 (81)
Number of Demes	9	9
Offspring Per Generation	2 (9)	2 (9)
Mutation Rate	0.081	0.213557
Mutation Step Size	1.09	0.033593
Parent Tournament Size	381 (134)	339 (71)
Survival Tournament Size	392 (138)	344 (74)
Evals Per Support Individual	4	4
Optimization Frequency	0.1	0.1

Table 4.2: Experiment Parameters

how the shift vector is applied to an individual, where the vector x represents the solution vector encoded by a solution candidate and the vector o represents the random shift vector. This operation creates the vector z which is passed to the fitness functions.

$$z = x + o \ \ , x = [x_1, x_2, ...x_n], o = [o_1, o_2, ...o_n] \tag{4.1}$$

The Shifted Schwefel function is given in Equation 4.2 and was selected to provide a test problem with weak global structure.

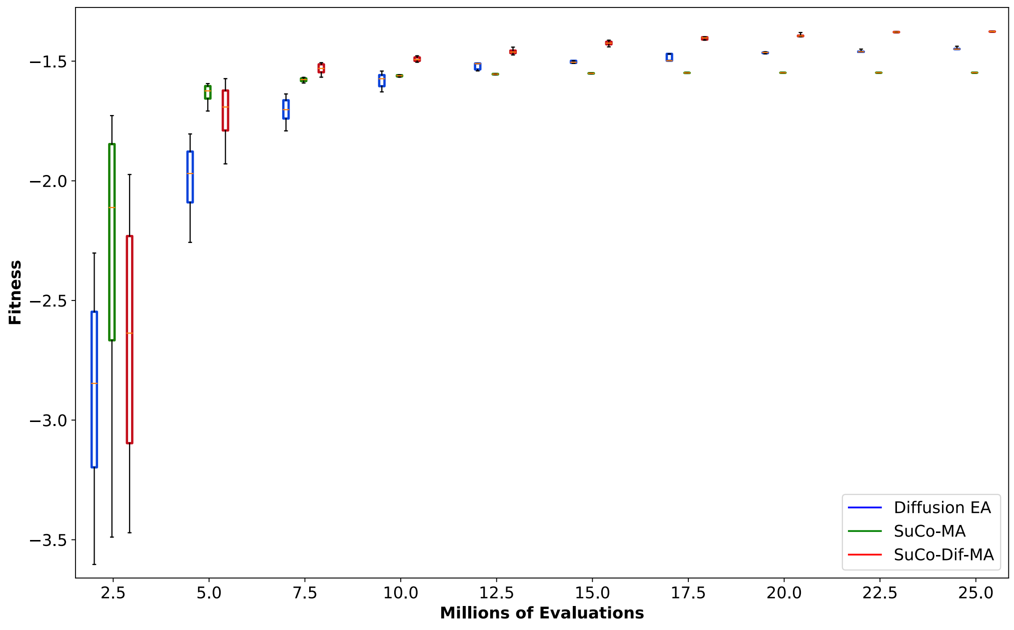
$$-\frac{1}{100n}\sum_{i=1}^{n} z_i \sin(\sqrt{|z_i|}) + 4.189828872724339 \; \forall z \ \in \ [-500, 500] \tag{4.2}$$

Equation 4.3 defines the Shifted Rosenbrock function which was selected to represent a problem with gene interdependence.

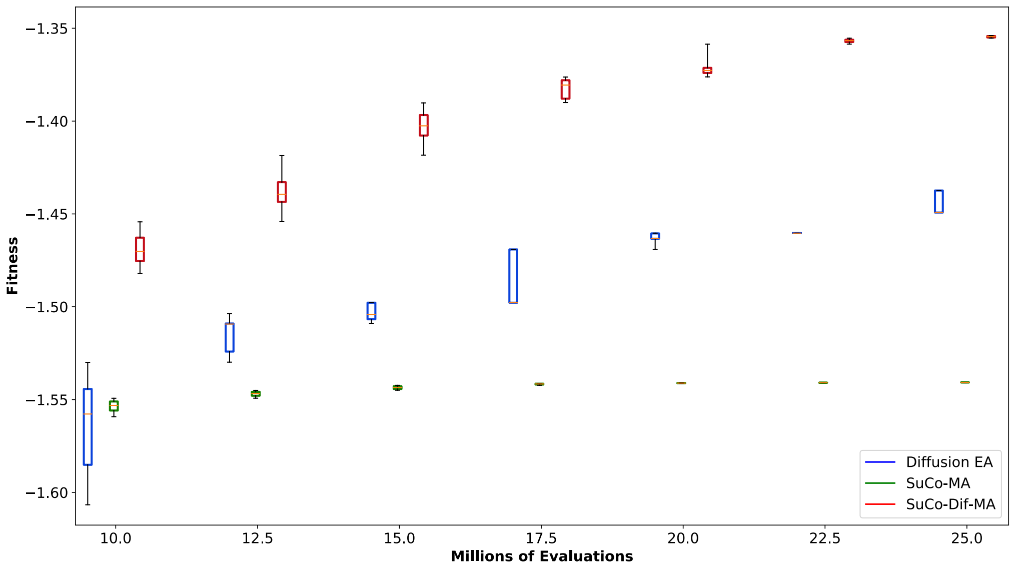
$$\sum_{i=1}^{n-1} \ [100(z_{i+1} - z_i^2)^2 + (1 - z_i)^2] \; \forall z \ \in \ [-5, 10] \tag{4.3}$$

4.4 Results

Results are shown in Table 4.3 with bold entries indicating the highest performance. Two different dimensions of both benchmark problems were tested. Each result listed in the results table represents the mean final best fitness over thirty runs for each configuration. The standard

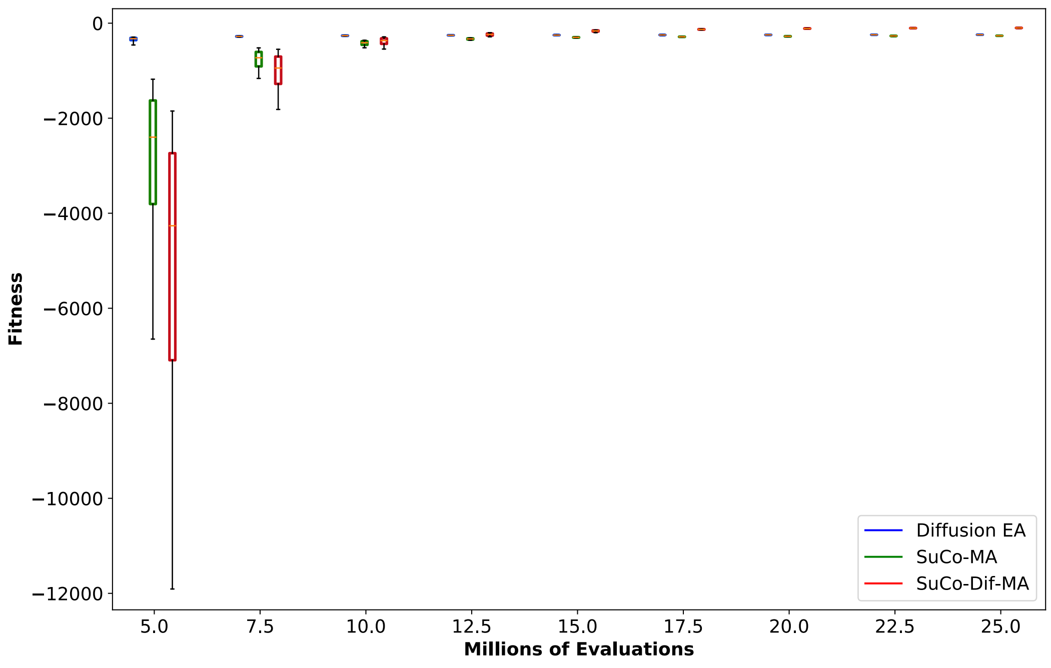


(a) Schwefel 200D

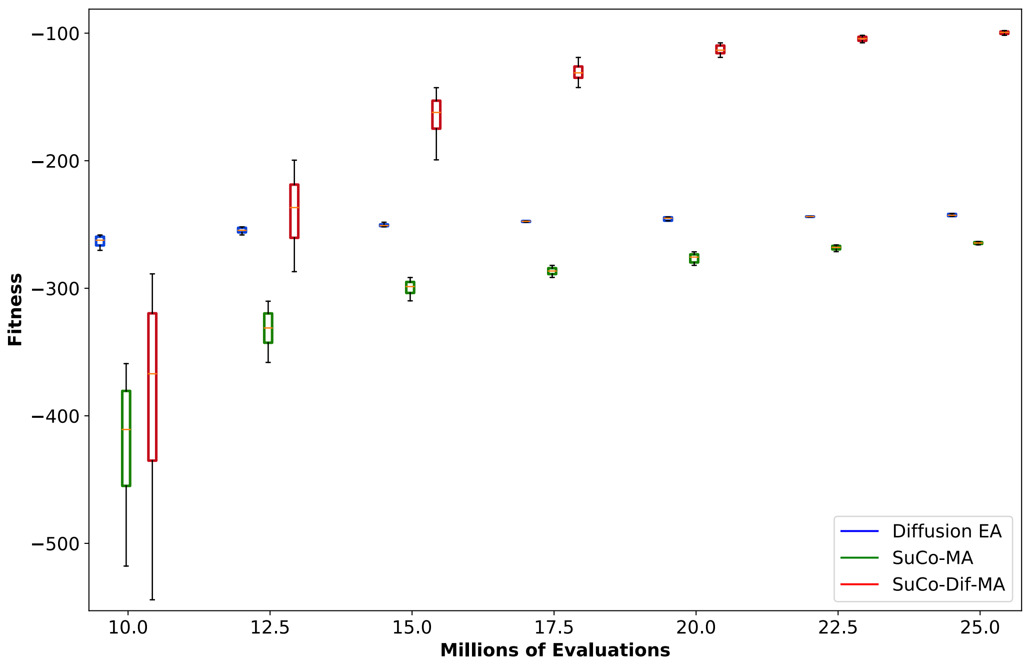


(b) Schwefel 200D end

Figure 4.2: Fitness versus evaluations Schwefel 200D



(a) Rosenbrock 200D



(b) Rosenbrock 200D end

Figure 4.3: Fitness versus evaluations Rosenbrock 200D

	Schwefel 50D	Schwefel 200D	Rosenbrock 50D	Rosenbrock 200D
Diffusion EA	**-0.87 (0.17)**	-1.48 (0.08)	-41.54 (5.57)	-241.60 (25.10)
SuCo-MA	-1.25 (0.16)	-1.55 (0.15)	-39.09 (17.26)	-262.32 (79.70)
SuCo-Dif-MA	**-0.83 (0.14)**	**-1.38 (0.07)**	**-10.33 (16.51)**	**-91.10 (67.26)**
T-test p-values vs. SuCo-Dif-MA				
Diffusion EA	0.211	5.92E-4	1.88E-11	1.29e-10
SuCo-MA	2.46E-04	1.72E-05	3.38E-06	5.56E-11

Table 4.3: Mean final best fitness with standard deviation in parenthesis

deviation of each experiment is also presented in parenthesis. A set of f-tests and t-tests with a confidence level of $\alpha = 0.05$ was performed which showed that the SuCo-Dif-MA was statistically different from the other algorithms on all configurations of both benchmark problems except Schwefel in 50 dimensions. The results of the t-tests can be found in the last two rows of Table 4.3. Box plots comparing the fitness versus evaluations for all three algorithms can be seen in Figure 4.2 and Figure 4.3.

4.5 Discussion

This section presents additional data that was observed during experimentation which merits further discussion. First we take a closer look at how SuCo-Dif-MA is able to achieve higher performance when compared to the other techniques shown in this investigation. Next, we provide details related to support individual fitness calculation. Finally, we provide more detail on the optimization frequency parameter. Previous work with SuCo-MA contained a parameter which controlled the frequency of local optimization of the primary population. This parameter had a significant impact on the overall performance of the support population and we close by discussing it further.

4.5.1 Local Optimizer Performance Improvements

The results presented in Table 4.3 show that the SuCo-Dif-MA is able to find higher performance individuals than the baseline methods on all problem configurations except Schwefel in 50 dimensions. In order to better demonstrate the cause of this performance improvement, data from a single deme over one run of the Rosenbrock benchmark problem in 200 dimensions is presented in Figure 4.4. Each sub-plot presents data at a different level of granularity. The top sub-plot shows nearly the entire run omitting only a small portion of the beginning of the run to prevent the very poor early fitness values from dominating the y-axis. The middle sub-plot shows data from a point near the end of a run where a high-performance local optimizer appears to have been found. To show the impact of the high-performance local optimizer more clearly, the bottom sub-plot shows a very small portion of the run where the best primary fitness is greatly improved by a support individual. During this portion of the evolutionary run, a series of high-performance local optimizers are able to find ways to make significant improvements to the best fitness of the deme. In many cases the optimizers are able to take an individual that is not the best, and modify it in a way that causes it to become the highest fitness individual. This behavior can be observed by the blue and orange dots which represent the pre-optimization and post-optimization primary fitness values respectively. Data was also collected which showed that the highest fitness primary individual in a deme is typically the offspring of an individual that has been optimized recently. This is interesting because it demonstrates how evolution can use an individual that was recently optimized to create the highest performance individual in a deme.

4.5.2 Support Individual Fitness

The results shown in Table 4.3 were obtained when using a support fitness calculation based on the best improvement found by any local optimization run. This value was calculated as the change to a primary individual's fitness resulting from the optimization operation. Since each support individual is executed on a number of primary individuals, the maximum primary fitness improvement found among all trials was used as the support individual's fitness. This method provided the best performance across all experiments.

Another method of support fitness calculation explored was taking the average primary fitness change over all trials during the current generation for the local optimizer. However, this method did not prove to be as effective in improving the final best fitness. Since this method gives the optimizer an average score over multiple trials, it seems to hinder the specialization of local optimizers to specific demes. The goal of this work was not to create local optimizers

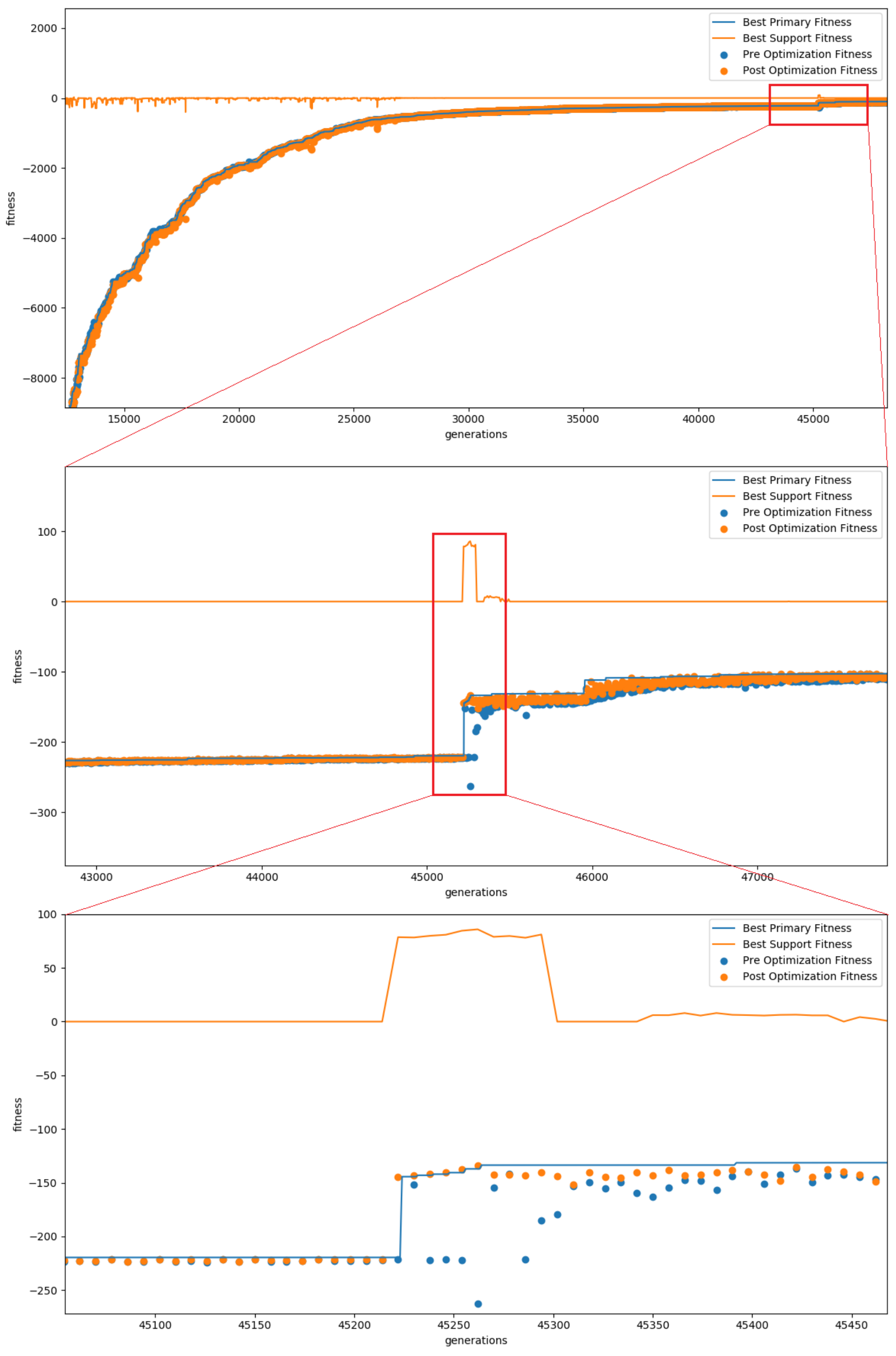


Figure 4.4: SuCo-Dif-MA run zoomed on a high-performance optimizer

that worked well on multiple individuals, but to create the highest performance local optimizers possible for any given individual in the corresponding primary deme. Thus it was better for final best performance to value a local optimizer which found a single large improvement on one individual over a local optimizer that found lesser improvements across multiple individuals.

4.5.3 Optimization Frequency

Since the primary population does not require optimization every generation, it is possible to skip the execution of local optimization for an arbitrary number of generations. The frequency of optimization is a parameter which can be adjusted to change the ratio of evaluations spent on evolutionary exploration to local search exploitation. The frequency parameter is calculated by dividing the number of local optimization generations by the number of primary population generations per run. Previous work has demonstrated there are optimal settings for this parameter, but the overall impact on performance was marginal [19]. Table 4.4 contains the results of an experiment which measured SuCo-Dif-MA performance over different optimization frequency settings with the best setting in bold. We observed that the optimization frequency parameter has a larger impact on SuCo-Dif-MA performance. We also observed that a higher frequency setting was more optimal than in previous work. The reason for this can be attributed to how many evaluations are required for the support population to discover high-performance, specialized local optimizers. It could be the case that the discovery of specialized operators requires more evaluations than the discovery of operators that generalize better. Alternatively it can be attributed to demes being smaller than the single population used in previous work, which reduces the probability of any given local optimizer finding some set of genes that can generally be optimized well. More work is required to fully understand this parameter and its impact on performance.

4.6 Conclusion

This investigation explored the extension of SuCo-MA by adding a diffusion model to both the primary and support populations to create SuCo-Dif-MA. It was shown that this technique has the ability to improve performance when compared to both SuCo-MA and an EA employing

Optimization Frequency	**Rosenbrock 200D**
0.2	-168.37 (55.66)
0.125	-94.64 (62.87)
0.1	**-91.09 (67.26)**
0.0833	-98.17 (76.66)
0.0667	-130.91 (85.04)

Table 4.4: Rosenbrock performance with different optimization frequency settings

only a diffusion model. SuCo-Dif-MA has demonstrated one way to evolve high-performance local optimization operators that are tailored for a specific portion of the solution space. This specialization allows the local optimizers to exploit regions of the solution space around the primary population to achieve higher performance.

Chapter 5

Solving the Traveling Thief Problem with a Diffusion Memetic Algorithm Employing Supportive Coevolution

Benchmark functions are a common way by which EA techniques are compared; they often exhibit certain characteristics that are of interest such as being highly multi-modal or deceptive. Because of this, it is common to use benchmark function results to analyze a particular technique's performance with respect to a problem class that is represented by that benchmark function. In this way, benchmark performance is commonly used as an estimation of how a given approach will perform on real-world problems. However, many benchmark functions in use today may not adequately represent the complexity of modern real-world problems.

The Traveling Thief Problem (TTP) is a modern benchmarking function that has been shown to be a better estimator of performance on modern real-world problems. With this in mind, this chapter provides further analysis of Supportive Coevolution Diffusion Memetic Algorithm (SuCo-Dif-MA) by utilizing TTP as the benchmark function. The methodology and experimentation details are provided. Results and analysis of SuCo-Dif-MA's performance are discussed along with conclusions that can be drawn from the observations during experimentation.

5.1 Introduction

In this chapter, SuCo-Dif-MA is applied to the Traveling Thief Problem (TTP). A new set of primitives is introduced for local optimization operator evolution to utilize during program evolution. A state of the art, hand designed method called MA2B is selected from literature as a comparison for SuCo-Dif-MA. Results on a representative set of TTP instances are given for both SuCo-Dif-MA and MA2B. While SuCo-Dif-MA is not able to outperform MA2B on any instance, promise is shown in that SuCo-Dif-MA is a fully evolved solution technique which can match the performance of a hand crafted method on several TTP instances. Some discussion around possible improvements and conclusions on SuCo-Dif-MA applied to TTP are also provided.

5.2 Background

SuCo-Dif-MA has shown promising results in the field of floating point optimization. In order to better understand the general applicability of SuCo-Dif-MA, this work investigates its application to combinatorial optimization. The Traveling Thief Problem (TTP) [3] is a well known, modern benchmark problem in the field of combinatorial optimization. One major attraction of TTP is its ability to better simulate the complexity and difficulty of real world problems. The remainder of this section provides more detailed background information on SuCo-Dif-MA and TTP.

5.2.1 Traveling Thief Problem

TTP is a combination of the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP). An instance of TTP contains a number of cities, n, with a number of items, m, distributed among the cities. Each item k has an associated profit, p_k, and weight, w_k. The knapsack has a maximum capacity denoted W that can not be exceeded. In a valid TTP solution, the thief must visit every city exactly one time while filling the knapsack without exceeding the capacity, and then travel back to the starting city. The velocity of the thief at city x_i is determined by the current weight of the knapsack w_{x_i} using Equation 5.1 where v_{min} and v_{max} represent the minimum and maximum velocity of the thief respectively.

$$v_{x_i} = v_{max} - \frac{v_{max} - v_{min}}{W} \cdot w_{x_i} \tag{5.1}$$

The parameter R is called the renting rate and determines the cost the thief must pay per unit time to rent the knapsack. Given a knapsack packing plan z where z_i represents the packing plan choice for item at index i, Equation 5.2 is the total profit of all items in the knapsack. Equation 5.3 calculates the total travel time of the given TSP tour x and KP packing plan z. Using these terms, the goal is to optimized the objective function defined in Equation 5.4.

$$g(z) = \sum_{k=1}^{m} p_k \cdot z_k \tag{5.2}$$

$$f(x, z) = \frac{d_{x_n x_1}}{v_{x_n}} + \sum_{i=1}^{n-1} \frac{d_{x_i x_{i+1}}}{v_{x_i}} \tag{5.3}$$

$$G(x, z) = g(z) - R \cdot f(x, z) \tag{5.4}$$

A TTP solution is encoded as two distinct vectors. The first is an ordered vector of cities encoding the TSP tour referred to as x. The second is a binary vector, z, where a value of 1 indicates the thief steals the item associated with that index, and a value of 0 the opposite. For example, consider the following TTP example problem which is shown by Figure 5.1. Since the start point is fixed and does not contain any items, every city but the first contains at least one item. For this example problem let the knapsack capacity $W = 3$, renting rate $R = 1$, $v_{max} = 1$, and $v_{min} = 0.1$. Consider the TTP solution composed of tour $x = (1, 2, 4, 3)$ and packing plan $z = (0, 0, 0, 1, 1, 0)$. For this solution, the objective value given by Equation 5.4 produces the value 50. This can be calculated by performing the following steps. First, we must sum the value of the items in the knapsack using the packing plan z: $g(z) = (20 \cdot 0) + (30 \cdot 0) + (100 \cdot 0) + (40 \cdot 1) + (40 \cdot 1) + (20 \cdot 0) = 80$. Next the trip duration must be calculated. Since the thief does not steal any items until the final city in the tour, the thief's velocity is 1 for the entire trip except for while traveling back to the starting city. This means the travel time up to the last segment is 15. The weight during the thief's return journey is 2 (since 2 items of weight 1 have been stolen) which means the return trip time is $\frac{6}{1 - \frac{1-0.1}{3} \cdot 2} = 15$. Thus, using Equation 5.4, the final objective value of this example solution is $80 - (1 \cdot 30) = 50$

The interdependence between the two sub-problems of TTP has been well studied [4, 27]. It has been shown that an optimal solution to either the TSP or KP sub-problem does not

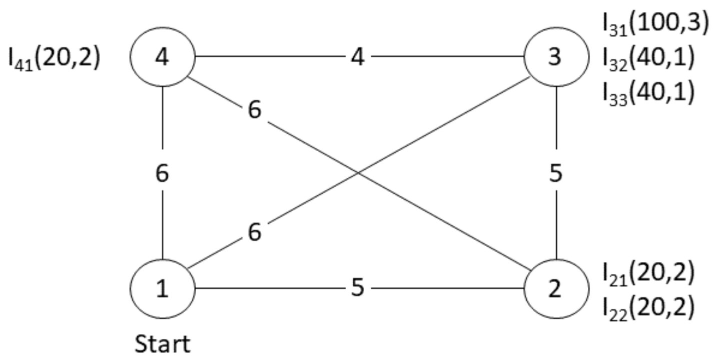


Figure 5.1: Example TTP Instance

guarantee a high quality TSP solution. This means that a high performance solution technique must consider both the TSP and KP sub-problems simultaneously.

5.2.2 MA2B Memetic Algorithm

MA2B is a population based memetic algorithm that uses 2-OPT and bit-flip mutation combined with MPX crossover to evolve TTP solutions [9]. In MA2B, the TSP tour is initialized by generating a random tour and applying the Lin Kernighan heuristic for a number of iterations. A greedy algorithm is then applied to create the packing plan. A local search employing 2-OPT to improve the TSP tour and bit-flip to improve the packing plan is then applied. MA2B utilizes tournament selection and Maximal Preservative Crossover [28] to select and recombine individuals.

5.2.3 Supportive Coevolution Diffusion Memetic Algorithm

A Supportive Coevolution Diffusion Memetic Algorithm (SuCo-Dif-MA) is a special EA variant that utilizes Genetic Programming (GP) to evolve local optimization operators that are applied to solution candidates in order to increase performance. This is accomplished by employing a technique known as Supportive Coevolution [13] to evolve the local optimization operators in parallel with the target fitness function solution candidates. Local optimization

Instruction	**Input**	**Output**
num_cities		int
num_items		int
shuffle_tour	tour	tour
shuffle_packing_plan	packing_plan	packing_plan
random_swap_tour	tour	tour
random_swap_packing_plan	packing_plan	packing_plan
add_item	packing_plan, item	packing_plan
remove_item	packing_plan, item	packing_plan
get_item_from_city	tour, packing_plan, int	item
get_city_by_index	int	city
get_next_city_in_tour	tour, city	tour, city
get_city_of_item	item	city
make_weight	packing_plan	packing_plan
swap_cities	city, city, tour	tour
add_random_item	packing_plan	packing_plan
get_random_city		city
get_random_item		item
get_starting_city		city
get_item_weight	item	int
get_item_profit	item	int
get_city_distance	city, city	int
apply_lin_kernighan	tour	tour

Table 5.1: Push TTP Instruction Set

operators are encoded as PushGP [35] programs. The technique used in this chapter was originally introduced in Chapter 4. Some changes were made to the original formulation so that TTP could be used for generating benchmark results. Section 5.3 describes the changes in detail.

5.3 Methodology

The SuCo-Dif-MA is designed with a single primary population and a single support population. Each support population individual is a local optimization operator encoded as a Push genetic program. The Push implementation used here was taken from [36]. Table 5.1 shows the TTP Push instructions implemented for local optimizer construction and evolution. Many default instructions were also included but are not listed for the sake of brevity. For more information on default instructions provided by available implementations, see [36].

The evolutionary cycle for each population is shown in Figure 4.1. The diffusion model used here is identical to the configuration used in Section 4.2. Both primary and support populations utilize tournament selection for both parent and survivor selection. The primary population utilizes partially mapped crossover (PMX) [10] for recombination. For mutation, tours are mutated with random swaps while packing plans are mutated with a bit flip mutation. The support population randomly selects between sub-tree mutation, deletion mutation, or alternation crossover, with equal probability for offspring generation.

Upon EA initialization, the primary population is first created and randomly initialized. TSP tour initialization is augmented by a limited number of Lin Kernighan heuristic iterations to be consistent with prior work. The support population is then randomly generated over the space of push programs. 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. To address this issue, random individuals from the primary population are used instead. Once every support individual in the initial support population has a fitness value, the support population is ready for the evolutionary cycle. The evolutionary cycle used here is the same cycle explained in Subsection 3.2.2.

5.4 Experimental Setup

Much work has been done to study the design of TTP instances for the purposes of benchmarking [4, 30]. Since TTP is composed of two popular problems, most instances utilize well known TSP instances combined with an established method of KP generation. For TSP instances, the TSP Library from Reinelt is traditionally used [31, 39]. Drawing on prior work with KP from [1, 24], there are traditionally three methods for generating item weights and profits: uncorrelated, uncorrelated with similar weights, and bounded strongly correlated. Each approach generates a different knapsack type based on recommendations from [24].

Instance	MA2B	SuCo-Dif-MA
eil76	4201 (2.32)	3997 (2.04)
u159	4575 (2.27)	4317 (2.02)
a280	18413 (0.74)	16822 (1.68)
u574	22648 (1.76)	21003 (1.54)
dsj1000	**77942** (0.47)	72111 (1.78)
fl1577	**92801** (0.38)	83011 (1.02)
pcb3038	151339 (1.19)	146904 (1.08)
pla7397	368985 (0.82)	342756 (1.01)
usa13509	**772507** (0.62)	716548 (1.21)
pla33810	**1763677** (0.59)	1592501 (0.98)

Table 5.2: TTP Experimental Results for instances from Category 1

It is important to note that in this work, no new TTP instances were generated. Every instance utilized for benchmarking purposes was taken from prior work by [30] which established a benchmark set of TTP instances. This work only tests a subset of the available benchmark instances and all TTP instance data can be found at [40].

As a representative benchmark set, we have selected the following instances: eil76, u159, a280, u574, dsj1000, fl1577, pcb3038, pla7397, usa13509, and pla33810. This set spans from small instances up to some of the largest.

Each instance can be configured with various different KP parameters. This chapter studies the three KP generation methods listed below.

- Category 1 - 1 item per city, bounded strongly correlated, small knapsack capacity
- Category 2 - 5 items per city, uncorrelated similar weights, average knapsack capacity
- Category 3 - 10 items per city, uncorrelated, large knapsack capacity

5.5 Results

Results from experiments comparing MA2B against SuCo-Dif-MA across the selected benchmark set for each of the three KP categories are presented in Table 5.2, Table 5.3, and Table 5.4. Each cell contains the mean best fitness over thirty runs with the relative standard deviation in

Instance	**MA2B**	**SuCo-Dif-MA**
eil76	30117 (0.72)	29001 (1.01)
u159	58754 (0.57)	57237 (0.89)
a280	112534 (0.62)	109657 (0.86)
u574	253576 (0.76)	250092 (1.12)
dsj1000	339318 (1.44)	330872 (1.71)
fl1577	651565 (1.07)	639255 (1.21)
pcb3038	**1191961** (0.58)	1062091 (1.34)
pla7397	**4020075** (2.0)	3679271 (1.66)
usa13509	**7556691** (1.35)	6821558 (2.12)
pla33810	**14594137** (1.01)	10051312 (2.09)

Table 5.3: TTP Experimental Results for instances from Category 2

Instance	**MA2B**	**SuCo-Dif-MA**
eil76	108037 (0.05)	100213 (0.18)
u159	244544 (0.36)	241989 (0.21)
a280	436932 (0.17)	430002 (0.20)
u574	965753 (0.26)	957245 (0.25)
dsj1000	1466106 (0.35)	1410276 (0.39)
fl1577	2589346 (0.71)	2526212 (0.49)
pcb3038	**4627657** (0.56)	4023778 (0.89)
pla7397	**13612483** (1.39)	12098128 (1.01)
usa13509	**25180776** (0.43)	20187221 (0.87)
pla33810	**55622060** (0.29)	45539817 (0.56)

Table 5.4: TTP Experimental Results for instances from Category 3

parenthesis. Rows with entries listed in bold represent a higher performance result while rows with no bold entry means the results were not statistically different.

5.6 Discussion

Experimentation with SuCo-Dif-MA on TTP produced some interesting results with respect to optimal parameter values that conflicted with previous work. The parameter used to control the frequency of optimization, called selective optimization, was found to follow a somewhat expected trend from previous work and is discussed in Subsection 5.6.1. The evaluations per local optimizer during local optimizer fitness calculation also had a different optimal setting in this work and is investigated more in Subsection 5.6.2. Finally, some interesting general observations regarding online program synthesis utilizing SuCo for local optimization algorithms are presented in Subsection 5.6.3.

5.6.1 Selective Optimization in TTP

Previous work with SuCo-MA and SuCo-Dif-MA has shown that adjusting the ratio of primary generations to support generations such that the primary population receives more evaluations positively impacts performance. This work saw an even larger improvement in performance by setting this ratio in a similar way, but by executing several support generations in a row when optimization was performed. This essentially freezes the primary population so the support population can operate on the same set of individuals for a number of generations. This could be because TTP is a much more complex problem than the floating point benchmark functions explored previously. Performing several support generations with the primary population frozen could allow the support population more time to explore the current state of the primary population and adjust the optimization strategies accordingly which yielded higher performance.

5.6.2 Evals Per Local Optimizer on TTP

When a local optimizer is being evaluated, it can be executed on any number of primary population solutions. Previous investigations into SuCo-MA have explored the impact of this parameter on performance. This work also explored this parameter and found results that differ from previous investigations. It was found that performance on TTP is dramatically improved when this parameter is set such that each support individual is evaluated on seven or eight primary individuals (depending on the problem instance). This finding is different from the findings of previous work which showed that the best performance was achieved when local optimizers were executed on only one support individual.

This result is interesting since the only major change in experimentation was changing the benchmark function to TTP. This suggests that the difficulty of determining a local optimizer's fitness is related to the difficulty of the benchmark problem. This connection is not fully understood yet, and more work is needed to fully analyze the results to strengthen this conclusion or propose alternative causes.

5.6.3 General Observations on Program Synthesis

During experimentation with SuCo-Dif-MA applied to TTP, many interesting observations were made. A general, high level observation about program synthesis for local optimizers in this work that sums up many of the observations is the following: evolving local optimizers is chaotic and unstable. In EC, chaos is a benefit as it enables the effective search of a multimodal solution landscape. However, the programs representing the local optimizers are very sensitive to small changes. To make matters worse, the evolution in the primary population further compounds the fragility of the local optimizers because strategies that worked in previous generations may be completely invalid due to the primary population traversing the solution space.

All of these observations lead to the need for better mechanisms to improve the program synthesis methods utilized for evolving the local optimizers. Methods which aid the support population in performing more informed offspring generation and mutation could provide large improvements in stability and performance by reducing the fragility of evolution in the support population.

5.7 Conclusion

In this chapter it was shown that SuCo-Dif-MA can be applied to TTP. A new instruction set was proposed for PushGP that enables local optimization programs to operate on TTP solutions. A set of representative benchmark instances from the TTP community were selected to test the proposed method. A competing MA approach from literature was selected for comparison. It was shown that SuCo-Dif-MA can evolve local learning strategies that yield competitive results when compared to the hand crafted MA approach. Some interesting insights in to SuCo-Dif-MA performance on TTP are shared, and some areas of further investigation are also detailed.

Chapter 6

Conclusions

In this thesis, SuCo was explored as a method of automated parameter control and also as a technique for operator evolution. Mutation step size, crossover operators, and local optimization operators in an MA were successfully evolved in support populations. It was also shown that multiple support populations can be effectively utilized when it is necessary to evolve more than one parameter or operator. Empirical results from benchmark testing showed that SuCo can improve performance when compared to both static parameters and operators, and to self-adapted parameters. It was also shown that SuCo-Dif-MA can be used to generate competitive results with state of the art techniques on TTP.

Throughout this investigation, some interesting characteristics of local optimizer evolution were observed. Most evolved local optimizers relied heavily on stochastic operations rather than deterministic operations. The exact cause of this is not fully understood, but some preliminary experimentation has shown that controlling the access to stochastic instructions can modify this behavior. However, in preliminary tests this has typically reduced performance. This is most likely due to the fact that simple stochastic operations are easy to evolve and produce statistically similar results when applied to many solutions. In contrast, high quality, deterministic optimization strategies can be very complex and may not work at all depending on the input. Therefore, it is much easier to evolve and continue to see improvements when utilizing local optimizers that rely heavily on stochastic operations. This is somewhat problematic because the theoretical improvements that can be obtained by high quality, deterministic operators are much larger than those that can be made by their stochastic counter parts (especially when considering computation time, e.g., repeated stochastic manipulation vs. deterministic improvement). It is also important to remember that the purpose of a local optimizer is to perform exploitation, not exploration (since the EA is a much better agent for exploration). Therefore, it is also important that local optimizers only use stochastic operations for exploitation. Some amount of stochastic behavior is more than likely necessary (and healthy) for local optimizer evolution, but the balance is a critical component.

It is also suspected that deterministic operations are harder to evolve and maintain because they typically rely on multiple instructions to make decisions. Program evolution is quite chaotic and can be very destructive when multiple instructions are necessary in a specific ordering to achieve an operation. This observation points to the need for a mechanism which can help inform the evolution of local optimizers such that sequences of instructions that produce good behavior are less likely to be broken up. This is a difficult task because identifying these sequences is not trivial. Identification must also be done at run time as it is very likely that the optimal sequence of instructions will be both problem specific and depend on the current state of the evolutionary search. If such a method were added to the support population of SuCo-Dif-MA, it could improve performance by encouraging the development of deterministic local optimization strategies.

6.1 Future Work

While this thesis has shown several methods by which SuCo can be used to improve EA performance, there are still many areas that require more study. SuCo has been used to evolve crossover operators and local optimization operators, but no investigations into evolved selection operators employing SuCo have been performed. SuCo has been used to evolve two support populations simultaneously with successful results. Additional research could be performed in which additional support populations are added to study the performance impact of having many SuCo populations.

SuCo-Dif-MA showed that deme specific evolution can further improve performance via the addition of a diffusion model. Deme specific evolution of other operators and parameters should be investigated to understand if this improvement will work in general, or is exclusive to local optimization operators in SuCo-Dif-MA. A more detailed study of the genetic diversity both within demes and globally could help provide more insights into some of the long periods of stagnation that precede the discovery of high performance optimizers. Another investigation of interest would be to observe the longevity of local optimization operators which could lead to further discoveries of techniques that encourage high performance local optimizer discovery.

More work needs to be done to better understand the complexities of online local optimizer evolution using SuCo and GP. There were many observations throughout the investigations of this thesis which seem to point at the instability of program evolution causing destructive behavior during the evolutionary runs. It also seems that SuCo-MA and SuCo-Dif-MA are much more likely to evolve highly stochastic local optimization operators and that the discovery of high performance operators can require long runs. Mechanisms that can provide informed choices for primitive selection and discover primitive relationships that benefit local optimizer performance could encourage the development of more interesting local optimizers. Such a method could increase the stability of fitness by identifying and increasing the speed at which high performance GP gene segments are discovered. However, such a technique must limit the impact to premature convergence and allow the support population to explore the solution space without introducing too much of a bias in any direction. One possible approach that could be studied is to seed the local optimizer support population with known, high-quality optimization techniques encoded in PushGP. This could address some of the stability issues observed and might encourage the local optimizer population to explore more interesting areas of the solution space instead of getting trapped by highly stochastic operators that rely heavily on luck. However, this approach must also carefully consider any biases that might be introduced. A more in depth study of evolved local optimizer behaviors could also provide some insights into how local optimizer evolution can be improved to develop more deterministic optimization strategies. It could also be possible that the highly stochastic nature of evolved local optimizers is highly optimal when compared to deterministic approaches for some unknown reason which could also be revealed by a careful behavioral analysis.

Bibliography

[1] *Advanced Generator for 0-1 Knapsack Problems*. URL: http://hjemmesider.diku.dk/~pisinger/codes.html (visited on). (accessed: 10.06.2021).

[2] Enrique Alba and Bernabe Dorronsoro. "Cellular Genetic Algorithms". In: Springer Publishing Company, Incorporated, 2008. ISBN: 978-0-387-77609-5.

[3] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. "The travelling thief problem: The first step in the transition from theoretical problems to realistic problems". In: *2013 IEEE Congress on Evolutionary Computation*. 2013, pp. 1037–1044. DOI: 10.1109/CEC.2013.6557681.

[4] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. "The travelling thief problem: The first step in the transition from theoretical problems to realistic problems". In: *2013 IEEE Congress on Evolutionary Computation*. 2013, pp. 1037–1044. DOI: 10.1109/CEC.2013.6557681.

[5] R. Collins and D. Jefferson. "Selection in massively parallel genetic algorithms". In: *Proceedings of the Fourth Intl. Conf. on Genetic Algorithms, ICGA-91*. 1991, pp. 249–256.

[6] Carlos Cotta, Luke Mathieson, and Pablo Moscato. "Memetic Algorithms". In: *Handbook of Heuristics*. Ed. by Rafael Martí, Pardalos Panos, and Mauricio G. C. Resende. Springer International Publishing, 2017, pp. 1–32. ISBN: 978-3-319-07153-4. DOI: 10.1007/978-3-319-07153-4_29-1. URL: https://doi.org/10.1007/978-3-319-07153-4_29-1.

[7] Diaşan and M. Oltean. “Evolving crossover operators for function optimization”. In: *Proceedings of the 9th European Conference on Genetic Programming*. Vol. 3905. Lecture Notes in Computer Science. Budapest, Hungary: Springer, 2006, pp. 97–108.

[8] AE Eiben and James E Smith. *Introduction to Evolutionary Computing*. Springer-Verlag Berlin Heidelberg, 2017. ISBN: 978-3-662-05094-1. DOI: 10.1007/978-3-319-07153-4_29-1.

[9] Mohamed El Yafrani and Belaïd Ahiod. “Population-Based vs. Single-Solution Heuristics for the Travelling Thief Problem”. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO ’16. Denver, Colorado, USA: Association for Computing Machinery, 2016, pp. 317–324. ISBN: 9781450342063. DOI: 10.1145/2908812.2908847. URL: https://doi.org/10.1145/2908812.2908847.

[10] David E. Goldberg and Robert Lingle. “Alleles, Loci, and the Traveling Salesman Problem”. In: *Proceedings of the 1st International Conference on Genetic Algorithms*. USA: L. Erlbaum Associates Inc., 1985, pp. 154–159. ISBN: 0805804269.

[11] Brian W. Goldman and Daniel R. Tauritz. “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, 2011, pp. 155–156.

[12] Brian W. Goldman and Daniel R. Tauritz. “Self-Configuring Crossover”. In: *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO ’11)*. Dublin, Ireland, 2011, pp. 575–582.

[13] Brian W. Goldman and Daniel R. Tauritz. “Supportive Coevolution”. In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO ’12)*. Philadelphia, Pennsylvania, USA, 2012, pp. 59–66.

[14] Jonatan Gomez. “Self adaptation of operator rates in evolutionary algorithms”. In: *in Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004*. 2004.

[15] D. Schomisch H. Mühlenbein and J. Born. “The Parallel Genetic Algorithm as Function Optimizer”. In: *Parallel Computing* 17.6–7 (1991), pp. 619–632.

[16] Sean Harris, Travis Bueter, and Daniel R. Tauritz. “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. Madrid, Spain: Association for Computing Machinery, 2015, pp. 1043–1050. ISBN: 9781450334884. DOI: 10.1145/2739482.2768456. URL: https://doi.org/10.1145/2739482.2768456.

[17] Kenneth A. De Jong. “An Analysis of the Behavior of a Class of Genetic Adaptive Systems”. PhD thesis. University of Michigan, 1975.

[18] Nathaniel R. Kamrath, Brian W. Goldman, and Daniel R. Tauritz. “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, 2013, pp. 1489–1496.

[19] Nathaniel R. Kamrath, Aaron Scott Pope, and Daniel R. Tauritz. “The Automated Design of Local Optimizers for Memetic Algorithms Employing Supportive Coevolution”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO ’20 Companion)*. Cancún, Mexico, 2020, pp. 1889–1897.

[20] Pascal Kerschke et al. “Automated Algorithm Selection: Survey and Perspectives”. In: *Evolutionary Computation* 27.1 (2019), pp. 3–45.

[21] Mark William Shannon Land. “Evolutionary algorithms with local search for combinatorial optimization”. PhD thesis. University of California, San Diego, 1998.

[22] A. E. W. de Landgraaf and V. Nannen. “Parameter Calibration Using Meta-Algorithms”. In: *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC’07)*. Singapore, 2007, pp. 71–78.

[23] Michael A. Lones. “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, 2019, pp. 1487–1494.

[24] Silvano Martello, David Pisinger, and Paolo Toth. “Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem”. In: *Manage. Sci.* 45.3 (Mar. 1999), pp. 414–424. ISSN: 0025-1909.

[25] Matthew A. Martin and Daniel R. Tauritz. “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. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 1389–1396. ISBN: 9781450328814. DOI: 10.1145/2598394.2609872. URL: https://doi.org/10.1145/2598394.2609872.

[26] Matthew A. Martin and Daniel R. Tauritz. “Evolving Black-Box Search Algorithms Employing Genetic Programming”. In: *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO ’13 Companion. Amsterdam, The Netherlands: Association for Computing Machinery, 2013, pp. 1497–1504. ISBN: 9781450319645. DOI: 10.1145/2464576.2482728. URL: https://doi.org/10.1145/2464576.2482728.

[27] Yi Mei, Xiaodong Li, and Xin Yao. “On Investigation of Interdependence Between Subproblems of the Travelling Thief Problem”. In: *Soft Computing* 20 (Oct. 2014). DOI: 10.1007/s00500-014-1487-2.

[28] Heinz Mühlenbein. “Evolution in time and space - the parallel genetic algorithm”. In: *FOUNDATIONS OF GENETIC ALGORITHMS*. Morgan Kaufmann, 1991, pp. 316–337.

[29] Gregor Papa. “Parameter-Less Evolutionary Search”. In: *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’08. Atlanta, GA, USA: Association for Computing Machinery, 2008, pp. 1133–1134. ISBN: 9781605581309. DOI: 10.1145/1389095.1389314. URL: https://doi.org/10.1145/1389095.1389314.

[30] Sergey Polyakovskiy et al. "A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem". In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. GECCO '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 477–484. ISBN: 9781450326629. DOI: `10.1145/2576768.2598249`. URL: `https://doi.org/10.1145/2576768.2598249`.

[31] Gerhard Reinelt. "TSPLIB - A Traveling Salesman Problem Library". In: *ORSA Journal On Computing* 20 (1991). DOI: `10.1287/ijoc.3.4.376`.

[32] Samuel N Richter and Daniel R. Tauritz. "The automated design of probabilistic selection methods for evolutionary algorithms". In: *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion*. Kyoto, Japan, 2018, pp. 1545–1552.

[33] Samuel N. Richter, Michael G. Schoen, and Daniel R. Tauritz. "Evolving Mean-Update Selection Methods for CMA-ES". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 1513–1517. ISBN: 9781450367486. DOI: `10.1145/3319619.3326827`. URL: `https://doi.org/10.1145/3319619.3326827`.

[34] Ekaterina A. Smorodkina and Daniel R. Tauritz. "Toward automating EA configuration: the parent selection stage". In: *Proceedings of CEC 2007 - IEEE Congress on Evolutionary Computation*. Singapore, 2007, pp. 63–70.

[35] Lee Spector. "Autoconstructive Evolution: Push, PushGP, and Pushpop". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. 2001, pp. 137–146.

[36] Lee Spector. *Push, PushGP, and Pushpop*. URL: `https://faculty.hampshire.edu/lspector/push.html` (visited on). (accessed: 11.11.2020).

[37] Lee Spector, Jon Klein, and Maarten Keijzer. "The Push3 Execution Stack and the Evolution of Control". In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. GECCO '05. Washington DC, USA: Association for Computing

Machinery, 2005, pp. 1689–1696. ISBN: 1595930108. DOI: 10.1145/1068009.1068292. URL: https://doi.org/10.1145/1068009.1068292.

[38] Lee Spector and Alan Robinson. "Genetic Programming and Autoconstructive Evolution with the Push Programming Language". In: *Genetic Programming and Evolvable Machines* 3 (Mar. 2002), pp. 7–40. DOI: 10.1023/A:1014538503543.

[39] *TSP Test Data*. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html (visited on). (accessed: 10.06.2021).

[40] *TTP Benchmark Problem Instances*. URL: https://cs.adelaide.edu.au/~optlog/CEC2014COMP_InstancesNew/ (visited on). (accessed: 10.06.2021).

[41] Fatemeh Vafaee et al. "Adaptively Evolving Probabilities of Genetic Operators". In: *2008 Seventh International Conference on Machine Learning and Applications*. 2008, pp. 292–299. DOI: 10.1109/ICMLA.2008.45.

[42] John R. Woodward and Jerry Swan. "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, 2012, pp. 67–74.