Missouri S&T Library and Learning Resources

Scholars' Mine

Masters Theses

Student Theses and Dissertations

Spring 2012

Robust evolutionary algorithms

Brian Wesley Goldman

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

 Part of the Computer Sciences Commons

Department:

Recommended Citation

Goldman, Brian Wesley, "Robust evolutionary algorithms" (2012). *Masters Theses*. 5148.
https://scholarsmine.mst.edu/masters_theses/5148

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License.

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

ROBUST EVOLUTIONARY ALGORITHMS

by

BRIAN WESLEY GOLDMAN

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2012

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Fikret Erçal
Dr. Donald Wunsch

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Published journal articles retain their original copyrights.

Copyright 2012

BRIAN WESLEY GOLDMAN

All Rights Reserved

PUBLICATION THESIS OPTION

This thesis has been prepared in the style utilized by the Genetic and Evolutionary Computation Conference. Pages 4 – 29 have been accepted for publication and pages 31 – 53 have been submitted for publication.

ABSTRACT

Evolutionary Algorithms (EAs) have shown great potential to solve complex real world problems, but their dependence on problem specific configuration in order to obtain high quality performance prevents EAs from achieving widespread use. While it is widely accepted that statically configuring an EA is already a complex problem, dynamic configuration of an EA is a combinatorially harder problem. Evidence provided here supports the claim that EAs achieve the best results when using dynamic configurations. By designing methods that automatically configure parts of an EA or by changing how EAs work to avoid configurable aspects, EAs can be made more robust, allowing them better performance on a wider variety of problems with less requirements on the user.

Two methods are presented in this thesis to increase the robustness of EAs. The first is a novel algorithm designed to automatically configure and dynamically update the recombination method which is used by the EA to exploit known information to create new solutions. The techniques used by this algorithm can likely be applied to other aspects of an EA in the future, leading to even more robust EAs. The second is an existing set of algorithms which only require a single configurable parameter. The analysis of the existing set led to the creation of a new variation, as well as a better understanding of how these algorithms work. Both methods are able to outperform more traditional EAs while also making both easier to apply to new problems. By building upon these methods, and perhaps combining them, EAs can become even more robust and become more widely used.

ACKNOWLEDGMENT

I would like to thank my advisor, Dr. Daniel Tauritz, for all his help getting me this far. Without him I would not have produced nearly as much quality work. I would also like to thank Dr. Fikret Erçal and Dr. Donald Wunsch for agreeing to be on my thesis committee.

I would like to thank and apologize to my fiancée, Melissa Dobinsky, for putting up with how much time I have spent working. Finally, I want to thank my parents, Jeff and Jean Goldman, for giving me the support and freedom I needed to get here.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION

1. INTRODUCTION

Each year Evolutionary Computation (EC) produces new human competitive results on complex real world problems in an ever expanding array of fields. Often the limiting factor for an Evolutionary Algorithm (EA) is not its potential, but the difficulty of correct application for novice users or its time consuming nature for expert users. If EC can overcome these limitations, then its impact on real world problem solving will increase dramatically.

The basis for all EC is to perform population based heuristic search in which quality past solutions are used to create new solutions that are predicted to have an even higher quality. There are two hard requirements on applying EC to a new problem: a representation of possible problem solutions that allows for random generation and modification and some method to determine the relative quality of those solutions. From a biological perspective, these requirements can be restated as needing a method to perform genetic encoding and fitness calculation in order to allow for survival of the fittest with heritable traits.

While there are many varieties of EAs, the basic process involves incremental improvement of a population by allowing higher quality solutions (more fit individuals) more control over the generation of new solutions (offspring). First, some set of individuals from the population is chosen to act as the parents for a new offspring. The genetic material from these parents is recombined and mutated in such a way as to stochastically create a new solution with the hope that any successful traits that existed in the parents will pass on to the offspring. Offspring are then rated to

determine their quality (fitness). Once some number of offspring have been created (a generation is completed), survival selection is performed to weed out low quality solutions.

Most of the steps described above are controlled by configurable parameters and operators. The size of the population and the number of offspring per generation represent common configurable parameters, and the method of recombining parents to create quality offspring represents a common configurable operator. When applying EC to a new problem, it is often necessary to expertly configure the EA to achieve the best results.

The goal for generalized search techniques such as EAs is to find the highest quality results in the least amount of time on as wide a range of useful problems as possible. EAs, in particular, are interested in finding high quality solutions to highly complex real world problems. Many, if not most, complex real world problems require a significant amount of time to determine the fitness of a single solution. For example, when optimization is performed employing a simulation, a single fitness evaluation can take minutes or more to complete. Based on this feature, run time analysis for EAs is traditionally performed using the number of evaluations performed, as the processing required to generate new candidate solutions is eclipsed by the time it takes to evaluate those solutions. Furthermore, because real world problems can be difficult to reproduce and time consuming to use, the EC community commonly employs the use of benchmark functions designed to recreate the complexity of real world problems without the associated overhead.

The first paper performs novel analysis of not only the impact of correctly configuring an EA to each problem, but more specifically the impact of changing those parameters during a single run of an EA. The experimental results provided support the claim that to be maximally effective, the parameters of a common EA cannot be kept constant during a single run. This provides a further argument for the

difficulty of EA configuration as well as emphasizes the need to develop techniques that avoid or reduce the impact of configuration.

The second paper introduces a method to automatically discover and dynamically update how individuals are recombined during evolution. The proposed recombination method, Self-Configuring Crossover, evolves along side the solution, with more effective crossovers having a higher chance to propagate. This removes the need to choose the best method of recombination for each problem, as over time the recombination method will evolve to fit the problem in use. Furthermore, as supported by experimental evidence, this method allows the recombination operator to adjust dynamically to meet the changing needs of evolution.

Another approach to making EAs more robust is to redefine how EAs work in order to reduce the number of configurable parameters and operators. Often these methods overcome problem specific configuration by learning during evolution, similar to Self-Configuring Crossover. The Linkage Tree Genetic Algorithm (LTGA) is a state of the art algorithm which takes this approach, requiring only a single parameter and performing automatic detection of gene linkage. The third paper provides a detailed discussion of the strengths and weaknesses of this algorithm helps provide a basis for future work in making this easy to configure algorithm more widely applicable.

PAPER

1. META-EVOLVED EMPIRICAL EVIDENCE OF THE EFFECTIVENESS OF DYNAMIC PARAMETERS

ABSTRACT

Traditional evolutionary algorithms (EAs) are powerful problem solvers that have several fixed parameters which require tuning. An increasing body of evidence suggests that the optimal values of some, if not all, EA parameters change during the course of executing an evolutionary run. This paper investigates the potential benefits of dynamic parameters by applying a Meta-EA to evolving optimal dynamic parameter values for population size, offspring size, n in n-point crossover, Gaussian mutation's step size, bit flip mutation's mutation rate, parent selection tournament size, and survivor selection tournament size.

Each parameter was optimized both as the only dynamic parameter, and with all parameters dynamic. The most effective two parameters when acting independently were also allowed to optimize in tandem. The results were compared with a Meta-EA tuned EA using static parameters on the DTrap, NK, Rastrigin, and Rosenbrock benchmark problems. Results support that all tested parameters have the potential to improve solution fitness by changing dynamically, and using multiple dynamic parameters was more effective than using each independently.

1.1. INTRODUCTION

The already time consuming process of parameter tuning to receive optimal results in an Evolutionary Algorithm (EA) increases combinatorially when considering

dynamically changing parameter values. A number of efforts have been made to determine parameter values for each stage of an EA. Mutation has received the most attention [1], but work has also been done in population size [2, 3], offspring size [4], recombination [5], parent selection [6] and survival selection [3]. Some work has also been done controlling multiple parameters at once [7, 8]. In most instances, the focus of parameter control research has been to remove or reduce an EA's dependency on a priori tuning, or on creating algorithms to modify or predict parameter settings, not necessarily to determine optimal settings. In [2], a hand-tuned EA using a static population size was compared to a Meta-EA evolved dynamic population size. While the dynamic population size obtained significantly better results, the difference in tuning methods and the usage of generations instead of evaluations for tuning makes the implications unclear.

1.2. METHODOLOGY

The following operators were chosen to be made dynamic: population size, offspring size, n in n-point crossover, Gaussian mutation's step size, bit flip mutation's mutation rate, parent selection tournament size, and survivor selection tournament size. These operators were selected because they use parameters that allow for straightforward dynamic modification and they are commonly used in EAs. Each meta-individual's genome represented *key* parameter values to use at evenly spaced intervals of evaluations. To determine the value of a parameter between keys, linear interpolation was used. By using a small number of *key*s, the Meta-EA can achieve a significant decrease in search space complexity without a significant loss in expressive power. To ensure fair comparison, a Meta-EA using a single *key* for each parameter was used to determine the best static configuration and all non-dynamic parameters employed these values.

The fitness of a meta-individual was determined by the average final best fitness of the set of EAs using that meta-individual. To compensate for meta-evolving six different parameter combinations on four benchmark problems, the number of Meta-EA runs and evaluations used were low, and each meta-individual was given 30 runs of an EA using a maximum of 5,000 evaluations. The DTrap, NK, Rastrigin and Rosenbrock benchmark functions were employed to test the effectiveness of dynamic parameters. They were chosen to represent the binary separable, binary inseparable, real-valued separable multimodal, and real-valued inseparable classes of problems, respectively.

Every parameter was tested using both two and five *key*s, to allow for experiments with low search space versus high control. As a higher number of *key*s can replicate fewer *key*s – by using identical key values sequentially – the worst fitness any number of *key*s should be able to obtain is equal to the best value found by a lower number of *key*s. As such, if the best dynamic parameter settings received worse results than static, the reason is due to search space complexity, because in an exhaustive search dynamic would at least find a way to mimic static. Additional testing was also performed using the two best *key* settings in tandem, as well as using two *key*s for every parameter at once.

1.3. RESULTS & DISCUSSION

The average and standard deviation of the best final fitness found by each configuration on each problem is given in Table 1.1. Experiments are labeled using the name of the parameter made dynamic and the number of *key*s used, with *Combined* referring to the combination of the best two. Paired T-Tests were used to compare all experiments with using the best found static configuration. To ensure the quality of the static configuration, the hand-tuned parameters found in [2] were compared with

the Meta-EA tuned parameters found here, showing the Meta-EA's configuration to receive significantly better results in half as many evaluations.

On all four problems tested, using a single dynamic parameter was able to outperform the best found static configuration, in many cases to a statistically significant degree. The exceptions to statistical significance are NK and Rosenbrock, were the lack of time to converge is likely responsible for the high standard deviation. Despite the significantly larger search space, using two parameters together was able to outperform using either independently on DTrap, Rastrigin and Rosenbrock. Furthermore, allowing all parameters to change dynamically resulted in the best results of all experiments on Rosenbrock.

None of the experiments run resulted in final best meta-individuals who mimicked simpler dynamic settings. Also, when using dynamic parameters in tandem, the best *key* settings found for each was different than using the parameters alone. This implies that the best methods for parameter control are highly dynamic and highly interdependent. Finally, each parameter tested received the best or second best results on at least one of the problems, meaning that to achieve best performance, all parameters may need to be allowed to change dynamically.

1.4. CONCLUSION & FUTURE WORK

When using a Meta-EA to evolve parameter settings, a single dynamic parameter was found to be more effective than using strictly static parameters and all of the tested dynamic parameters were more effective than static on at least one problem. Using two dynamic parameters was better than using one dynamic parameter on all problems except NK, and on Rosenbrock the most effective configuration allowed all parameters to change dynamically together. The more a parameter changed, the better fitness it was able to achieve. The optimal control trends also changed depending on how other parameters were changing concurrently.

Table 1.1: Fitness of dynamic parameters compared with static

Keys	Fitness	P-Value
4-Bit Concatenated DTrap, 100 Traps		
Combined	0.838 (0.013)	0
Parent Selection 5	0.837 (0.014)	0
Offspring 5	0.831 (0.012)	0
All	0.814 (0.008)	0.373
Static	0.813 (0.009)	1
NK Landscapes, N=20, K=3		
Population 2	0.766 (0.027)	0.325
Mutation Rate 5	0.765 (0.027)	0.383
Combined	0.764 (0.026)	0.464
Static	0.763 (0.024)	1.000
All	0.762 (0.026)	1.000
Rastrigin, N=20, A=10		
Combined	-8.278 (2.487)	0.002
Offspring 5	-8.816 (2.590)	0.011
Survival Selection 5	-9.538 (2.284)	0.063
Static	-10.769 (3.590)	1
All	-13.376 (4.471)	1
Rosenbrock, N=20, A=100		
All	-31.186 (31.716)	0.005
Combined	-39.435 (56.788)	0.023
Mutation Step Size 2	-55.181 (78.896)	0.138
Recombination 2	-63.263 (66.377)	0.217
Static	-79.545 (89.301)	1

While running a lower number of evaluations should be indicative of longer runs, experiments using more common evaluation counts for the EA will allow for more direct interpretation. Since the minimum quality for any dynamic configuration should be to at least mimic the best less dynamic configuration, enough meta-evolution to reach this point should be performed.

To increase generality of results, problems involving permutation based solutions and real world problems should be attempted. Further experimentation, including parameters using more *key*s, can determine if there are any common dynamic parameter trends that can be used as the basis for control strategies.

2. SELF-CONFIGURING CROSSOVER

ABSTRACT

Crossover is a core genetic operator in many evolutionary algorithms (EAs). The performance of such EAs on a given problem is dependent on properly configuring crossover. A small set of common crossover operators is used in the vast majority of EAs, typically fixed for the entire evolutionary run. Selecting which crossover operator to use and tuning its associated parameters to obtain acceptable performance on a specific problem often is a time consuming manual process. Even then a custom crossover operator may be required to achieve optimal performance. Finally, the best crossover configuration may be dependent on the state of the evolutionary run.

This paper introduces the Self-Configuring Crossover operator encoded with linear genetic programming which addresses these shortcomings while relieving the user from the burden of crossover configuration. To demonstrate its general applicability, the novel crossover operator was applied without any problem specific tuning. Results are presented showing it to outperform the traditional crossover operators *arithmetic crossover*, *uniform crossover*, and *n-point crossover* on the Rosenbrock, Rastrigin, Offset Rastrigin, DTrap, and NK Landscapes benchmark problems.

1.1. INTRODUCTION

When using an Evolutionary Algorithm (EA), choosing the crossover operator that best exploits known information can be difficult and is problem dependent [9] due to differences in gene interdependence and representation. Many of the traditionally used crossover operators require manual tuning to achieve the best results, beyond the requirement of choosing the one most effective for the target problem. Furthermore, this set of commonly used crossovers may not contain the operator best suited for

exploiting a specific problem's genetic representation. Finally, traditional crossover operators do not change during different stages of evolution.

Any method capable of automatically evolving the optimal crossover operator for a problem could help reduce development time and improve solution quality. If the EA can design a crossover operator during run time, practitioners will no longer need to configure that part of the EA. Also, evolved operators can discover ways of exploiting the search space that are unlikely to be discovered by traditional designs. These generated operators can also adapt to the changing needs of the evolutionary process.

A Meta-EA could be used to evolve the crossover used by another EA; however, in most real world applications of an EA, Meta-EAs require an infeasible amount of computing time. A multiple species cooperative coevolutionary algorithm could also be used. This method would require the creation of a second population to evolve crossover operators. As one of the goals of this research is to reduce the time required to use an EA, replacing the selection of the crossover operator with all of the tunable parameters of a second population is undesirable.

This paper introduces Self-Configuring Crossover (SCX) encoded with linear genetic programming (LGP), a nonbranching form of genetic programming (GP), and evolved using self-adaptation. When performing crossover, each individual uses its own operator to create a child. This child receives a version of the parent's operator as part of its genome. Quality crossover operators tend to create quality children, therefore increasing their ability to propagate. If the operator is ineffective, the offspring will tend to be poor, resulting in a lower likelihood of survival and propagation.

1.2. RELATED WORK

Network crossovers are specifically designed to exploit gene interdependence [10]. These operators use known problem specific gene linkage information to determine the most successful method for recombination, and are able to significantly outperform more traditional crossovers on problems with high gene interdependence. Unfortunately, network crossovers require a practitioner's knowledge of the search space to determine gene linkage, making them problem dependent and difficult to apply to black box optimization. While Linkage Learning [11] automatically determines gene interdependence, this method is designed around binary gene representations, limiting its general application.

Very little work has been done on automatically creating crossover operators. [12] used a Meta-GP to evolve crossover operators. To perform crossover, GP was applied to each locus independently, such that all genes used the same method to recombine and each gene could only recombine with another gene at the same locus. The result of the EA's run was used as the fitness for the crossover operator. On the problems tested, the resulting operators performed comparably to the most appropriate traditional crossover operator. While such novel crossover operator creation is of theoretical interest, the amount of a priori computation involved with meta evolution is not justified, unless these novel operators are significantly less problem dependent than traditional ones, which was not reported. If the benefits of dynamic parameters seen in mutation step size [13] and population size [2] hold true for crossover, then the crossovers designed in [12] are also limited by their inability to change during a single run of the EA. The requirement of treating all loci in the same manner and no allowance for crossover with genes in different loci may also have hindered this method.

1.3. BACKGROUND

LGP was first introduced in [14], where a language was created to evolve programs using a list of integers to represent a list of functions. Each of these functions performed operations on, and stored results to, memory locations. The list of functions is evaluated sequentially, and the final result is chosen from one of the memory locations. The exclusion of any program branching limits this form of GP, but also greatly reduces the search complexity.

Self-adaptation has been used extensively for mutation step size [15], and has been applied to crossover by having individuals encode probabilities used to determine which human created operator to use [5], or controlling the balance between crossover and mutation [8]. While those methods were shown to work about as well as choosing the correct operator from the start, they are highly limited by the operators they are provided, which may not be the best operators possible for the target problem. Some of their success may be based on the crossover operator's ability to change dynamically during the run, but no information was presented testing this possibility.

GP has been employed to encode self-adapted operators in the past. In one instance the mate selection operator was automated by embedding a GP tree structure into each individual, which was used to choose a mate [6]. This study found that this system obtains results similar to EAs that required manually created mate selection on the Onemax, D-TRAP, and SAT test problems.

1.4. METHODOLOGY

1.4.1. Crossover Encoding and Application. Self-Configuring Crossover (SCX) is encoded using a linear structure similar to that of LGP [14] to reduce the search space, as operators must be evolved during the evolution of a single EA. A sample SCX and example application are shown in Figure 1.1. A crossover operator

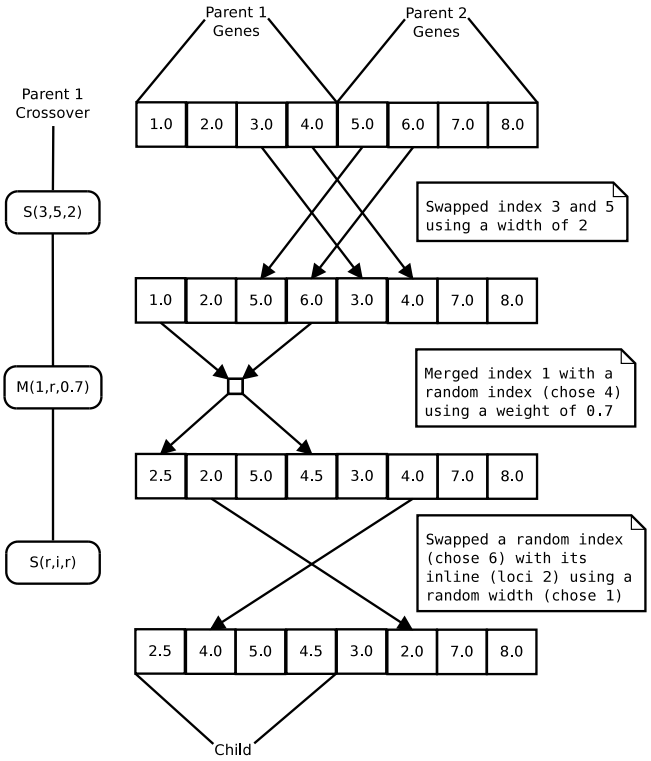

Figure 1.1: Sample Self-Configuring Crossover (left) and example application (right)

is represented by a list of primitive functions. To perform a crossover, the genes of both parents are first concatenated. Next, each primitive in the first parent's list is applied in sequence to the copy of the genes, modifying it. In Figure 1.1 each new row of genes represents a step between each primitive function's application. Once the entire list has been applied, half of the combined genome is used to create an offspring.

As evolution of useful operators needs to occur during the run of a single EA, search space minimization was a key factor in primitive design. The primitive set was designed using the well explored functionality of traditional crossover operators. While this biases encoded crossovers towards existing crossover operators, the vast majority of encoded crossovers will be novel, while retaining the ability to replicate traditional crossover operators when appropriate. *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. *Merge* was designed to represent crossovers that create genetic material by combining genes in a manner similar to arithmetic crossover. Both primitives have three parameters. The first two control the gene positions effected by the primitives. These positions can either both be located in a single parent or be located within two different parents. The final parameter is primitive dependent. In Figure 1.1, primitives are represented by a character identifier – S for *Swap* and M for *Merge* – and their list of three parameters.

The *Swap* primitive's third parameter is width. To perform a *Swap*, genetic material starting at the position indicated by the first position parameter swaps location with genetic material starting at the second position. The amount of material moved is equal to the width of the *Swap*. This method allows blocks of information to be kept together. The first primitive applied in Figure 1.1 is a *Swap*. Data is moved from the first position, 3, to the second position, 5, at a width of 2.

The *Merge* primitive's third parameter is weight. To perform a *Merge*, the values contained at each position are combined using a weighted average as shown in Equation 1.1, where *g(i)* is the gene at the first position, *g(j)* is the gene at the second position, and α is the weight, identical to arithmetic crossover's α. The process is repeated to find the new value for the second position. When performing a *Merge* on discrete problems, values are rounded to the nearest valid value. For example, if on a binary problem *g(i)*=0, *g(j)*=1, and α=0.6, the resulting value of *g(i)* would be 0.

$$g(i) = \alpha \cdot g(i) + (1 - \alpha) \cdot g(j) \tag{1.1}$$

To increase flexibility, the parameter for each 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. Next, the *Random* construct creates a

new random number every time the primitive is used. The purpose of *Random* is to allow SCX to exhibit a stochastic behavior similar to almost all human created crossover operators. The final construct *Inline* is only used for position parameters. *Inline* ensures that the primitive is performing all of its actions between the same loci in both parents. In Figure 1.1, r is for primitive parameters using the *Random* construct, and i is for using the *Inline* construct.

1.4.2. Evolving Crossovers. When using self-adaptation, the common practice is to evolve the operator before using it to produce a child [16]. This allows the operator to evolve more freely, but if an optimal operator is developed, it will not be reused. Another option is to evolve the parent's operator after use, allowing the same operator to be reused. Preliminary experimentation indicated that evolving the crossover before use obtained better results. This method was used in all further experiments.

Since the crossover being evolved is not designed to handle GP, a method is needed to evolve the encoded crossover operator. When performing crossover on the encoded crossover operator, a random point is chosen in each parent's list of primitives. All of the genetic information before the randomly chosen point in the first parent and all of the information after the point chosen in the second parent is concatenated. This allows the lists to easily change in size, due to omitted or duplicated loci usage, with children having between one primitive and all of the primitives of both parents. On average this style will only change the length of the crossover if there is selective pressure to change the size. Empirical evidence shows no need for parsimony pressure as the crossover lengths converged to problem dependent sizes.

Mutation of crossovers is done by randomly generating the primitive chosen for mutation, using the same number of expected mutations per primitive as the primary

genome. These methods for crossover and mutation have both been used successfully in LGP [14].

1.4.3. Added Complexity. As one of the goals of this method is to reduce the need to configure and tune an EA, any parameters added by using SCX need to be checked for their impact on solution quality. The initial length of the crossovers in the population is an unavoidable parameter, unique to SCX. To examine the impact of this parameter, tests were performed as described in Section 1.5 to measure how effective SCX is with widely varying initial crossover lengths. The results of these tests were compared with the best found value for this parameter as described in Section 1.7. If optimization is found to produce no significant change in results, the impact of this parameter can be considered insignificant.

A final consideration for this technique is how it effects the overall convergence time of an EA. As this method does not use any added a priori tuning, and does not require any additional evaluations of the fitness function, the only increase in runtime is due to the application and evolution of crossovers. As each crossover is linearly encoded, and each primitive performs at most a linear operation on the constant size of the genes, the algorithmic and space complexities added by SCX are negligible.

1.5. EXPERIMENTAL SETUP

To test the effectiveness of SCX, a set of three real-valued benchmark problems and two binary benchmark problems was chosen. The Rosenbrock problem [17], as defined in Equation 1.2, was used as representative of real-valued problems with gene interdependence.

$$\sum_{i=1}^{n-1} \left[(1 - x_i)^2 + A(x_{i+1} - x_i^2)^2\right] \forall x \in [-5, 10] \tag{1.2}$$

Table 1.1: Base parameters used in Self-Configuring Crossover testing on real valued problems

Parameter	**Rosenbrock**	**Rastrigin**	**Offset Rastrigin**
A/K	100	10	10
Genome Size	50	100	100
Evaluations	100,000	10,000	100,000
Population	355	172	418
Offspring	2	5	6
Mutation Rate	0.213557	0.009838	0.009738
Mutation Step	0.033593	1.47144	0.587874
Parent Tournament	339	114	172
Survival Tournament	344	63	421
Mutation Type	Creep	Creep	Creep
Crossover	Arithmetic	Arithmetic	Arithmetic

This function contains a valley which is easy to find, but once found it can be difficult to converge to the global optimum. To represent the class of highly multimodal functions, the Rastrigin problem [18] was chosen, shown in Equation 1.3.

$$An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i) \right] \forall x \in [-5.12, 5.12] \tag{1.3}$$

In both Rosenbrock and Rastrigin, the optimum value for each gene is identical. To test SCX's ability to handle loci with different optima, the final real-valued equation was chosen as a specially designed modification of the Rastrigin equation, here called Offset Rastrigin. Before the first evaluation, an offset vector O is generated, where O_i is randomly chosen from $[-2.5, 2.5]$ at intervals of *0.5*. For example, if three dimensions are used, some possible offset vectors are $\{2, -0.5, 1.5\}$ and $\{0, -2.5, 1\}$. In all subsequent evaluations, the fitness is calculated on an individual's modified genes by adding this offset vector. Intervals of *0.5* were chosen because in the Rastrigin function, if x_i is a local best, $x_i + 0.5$ is a local worst. If x_i is near the global optimum, then $x_i + 1$ will be near a local optimum.

The NK-Landscapes (NK) problem [19] was chosen to represent binary problems with gene interdependence. In this problem, N genes are dependent on K other

Table 1.2: Base parameters used in Self-Configuring Crossover testing on binary problems

Parameter	**NK**	**DTrap I**	**DTrap II**
A/K	7	4	4
Genome Size	50	400	400
Evaluations	100,000	50,000	50,000
Population	283	498	95
Offspring	148	489	135
Mutation Rate	0.055115	0.000001	0.001429
Mutation Step	N/A	N/A	N/A
Parent Tournament	1	1	1
Survival Tournament	374	658	42
Mutation Type	Bit Flip	Bit Flip	Bit Flip
Crossover	16 Point	11 Point	SCX Only

genes for their fitness. As gene dependence is not related to their position in the genome, many crossover operators have difficulty determining what blocks of information to keep together, making this an ideal candidate for testing SCX's ability to overcome the limitations of traditional crossovers. This method relies on randomly generated fitness landscapes, so a common set was used by all experiments.

For comparison, 4-bit concatenated DTrap [20] is included. This binary problem consists of a series of 4-bit traps. Each trap's fitness is equal to 4 if all of the bits are set, and equal to the number of unset bits otherwise. Equation 1.4 shows the fitness for a single trap where t is the number of set bits. Unlike NK, DTrap has very well defined groupings of genes. DTrap also has easy to define fitness bounds, allowing for the fitness to be expressed as a percentage of optimal.

$$f(t) = \begin{cases} 3 - t, & t < 4 \\ 4, & t = 4 \end{cases} \tag{1.4}$$

To test the quality of SCX on these benchmark problems, comparisons were performed with arithmetic crossover (AX), n-point crossover (NPX), and uniform

crossover (UX), which were the basis for SCX's primitives, and represent the most widely used classes of crossover.

Since mutation works in tandem with crossover, a great deal of tuning was done to choose this operator. For real-valued problems, a commonly used mutation operator is Gaussian. In preliminary testing, this operator showed some difficulty scaling to the larger genome sizes desired. So a modification was made, borrowing some techniques from the traditional integer mutation operator Creep. Instead of adding a normally distributed random number to all genes, this real-valued Creep mutation has a probability to mutate each gene, and each mutation is performed by adding a normally distributed random number. This allows real-valued Creep mutation to make minor changes to a solution independent of the number of genes, by modifying a percentage of the total number of genes and still using an appropriate step size. This mutation was used for all real-valued problems. All binary problems used Bit Flip mutation.

A common problem when comparing EAs is the question of parameter quality. Hand tuning parameters can often lead to configurations far worse than many more advanced tuning techniques [21]. In many cases lack of rigorous tuning can lead to questions of if an algorithm succeeded due to its own quality, or due to the optimality of the parameters used for each algorithm tested. In an attempt to overcome this issue, a Meta-EA was used to determine the parameters for the comparison EAs. An EA using each of the comparison crossover operators was given to a Meta-EA capable of modifying all of its parameters. This Meta-EA was run extensively using each crossover operator for each problem to find high quality parameter settings. AX obtained the best results on all three real-valued problems and NPX obtained the best on the two binary problems. UX was not found to be the best traditional crossover for any test problem.

Whichever traditional operator obtained the best results after tuning was chosen to compete with SCX in final experimentation. To minimize the difference between the comparison operators and SCX, all parameter settings were kept identical. This method sets the quality of the traditional operator as high as possible, while forcing SCX to use possibly suboptimal tuning. The configuration settings are presented in Table 1.1 and Table 1.2. The only exception to this method was DTrap, as the optimal settings for NPX were too detrimental to SCX. To provide a fair comparison, NPX was run using its best found configuration (DTrap I), while SCX was run using UX's optimum configuration (DTrap II). UX's configuration was used because UX achieved the second best results of the traditional crossovers. As SCX's parameter of initial crossover length could not be used directly from the comparison operator's tuning, an arbitrary value of 5 was used for all problems. All experiments were run 100 times to gather statistically significant results.

Additional testing was also done to determine the sensitivity of solution quality to initial crossover length. To do so, SCX was run on each problem using initial crossover lengths in the range [10,200] at steps of 10. The edge case of 1 was also tested.

1.6. RESULTS

The results comparing the best traditional crossover found with SCX are presented in Table 1.3. Each experiment shows the mean and standard deviation of the final best over 100 runs for each crossover. A T-test was performed to determine if SCX's results were statistical better than the traditional crossover operator. All tests had statistically significant differences with a confidence level better than α=0.001.

Figure 1.2 and Figure 1.3 show fitness plots for NK and Rastrigin respectively. These figures are representative of all of the problems tested, both average and best. They indicate that SCX improved faster than the best comparison crossover, as well

Table 1.3: Mean final best fitnesses achieved with standard deviation in parentheses

Problem	Comparison	SCX
Rosenbrock	-86.94 (54.54)	-26.47 (23.33)
Rastrigin	-59.20 (6.998)	-0.0088 (0.021)
Offset Rastrigin	-0.1175 (0.116)	-0.0300 (0.028)
NK	0.7710 (0.011)	0.8016 (0.013)
DTrap	0.9782 (0.005)	0.9925 (0.021)

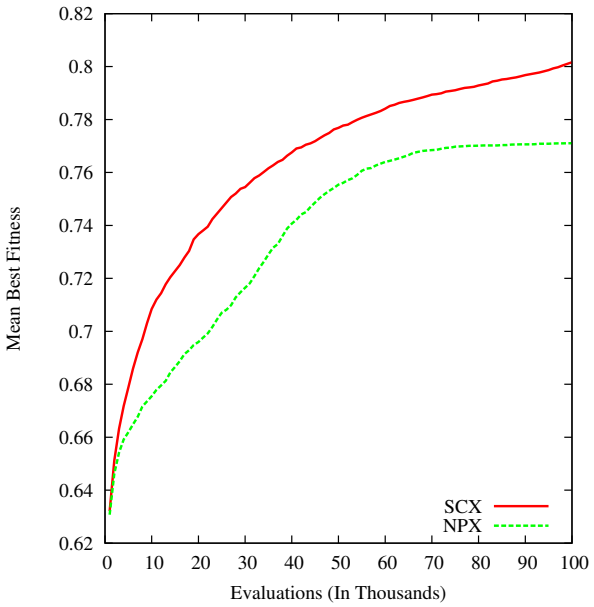

Figure 1.2: Mean best fitness on NK

as reaching the optimum sooner, or if the optimum was not reach, continuing to improve after the comparison operator converged. Figures 1.4, 1.5, and 1.6 provide insight into the internals of SCX. These show what percentage of the total number of primitives and parameter constructs are in use by SCX over the course of runs on different problems.

Sensitivity analysis of initial crossover length employed ANOVA and protected T-tests, with results shown in Table 1.4. ANOVA was used to determine if any initial length obtained significantly different final results from any other initial length. In any test where ANOVA reported a value higher than the critical value, pairwise T-tests were performed. Each run using a different initial crossover length was then compared

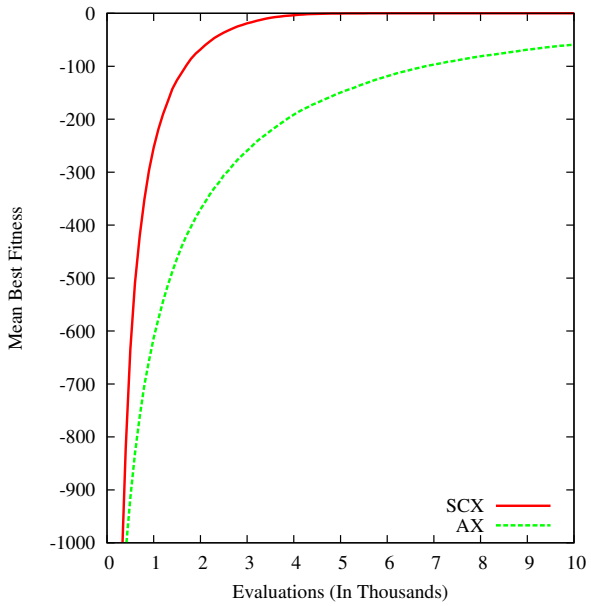

Figure 1.3: Mean best fitness on Rastrigin

with the run obtaining the highest fitness, resulting in a P-value. This value indicates the minimum confidence required to state that the test value was significantly worse than the best found. Figure 1.7 shows the effects of initial crossover length on the actual crossover length during the run.

1.7. DISCUSSION

For all five benchmark problems SCX obtained statistically better final results than AX, NPX, and UX. These results are reported in Table 1.3. Every problem except DTrap used the exact same configuration information for SCX as for the best traditional crossover. On DTrap, the hindrance of using a configuration optimal for NPX but not for SCX proved to be too significant, so SCX was allowed to use a different configuration setup. This difficulty is likely due to the strangely low mutation rate, shown in Table 1.2, used in the best NPX configuration found.

1.7.1. Adaptations to Each Fitness Function. On Rosenbrock, SCX was likely able to outperform AX due to its ability to better handle gene interdependence

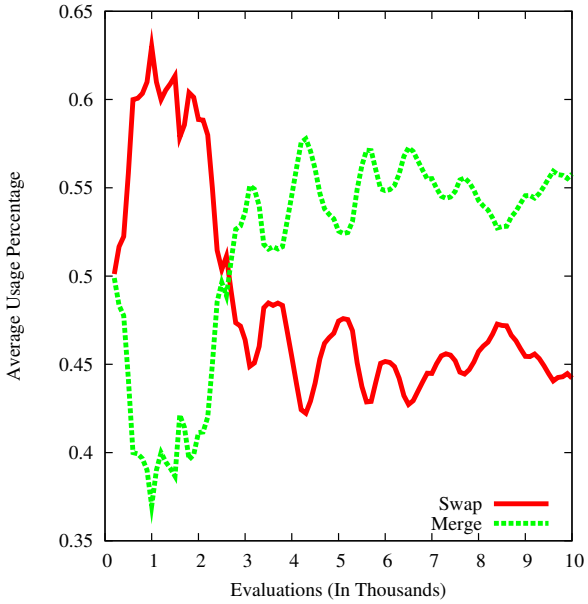

Figure 1.4: Primitive usage on Rastrigin

and to prevent gene stagnation. Due to SCX's ability to control how information is moved, it is possible that it could more easily escape local optima by only modifying specific genes. An even more likely possibility is that SCX's ability to move genetic information between loci allowed it to move optimized values in one gene location to another.

Rastrigin obtained some of the most striking results in favor of SCX. After just 10,000 evaluations, SCX was within a marginal distance of the global optimum, while AX was still far from the optimum, as shown in Table 1.3. On Offset Rastrigin, which is effectively the same problem for AX, after 10 times more evaluations, AX was still significantly further from the global optimum than SCX. Again, SCX's ability to mix genetic material between gene locations is a likely explanation for some of its success. Another possible reason can be seen in Figure 1.4. Over the course of an average run, SCX switches from being predominantly composed of *Swap*s to using mostly *Merge*s. The ability to change dynamically over time is not a feature of traditional crossovers. It is likely that, in the early evaluations, *Swap*s are able to preserve genes in local optima, while *Merge*s could combine two local bests to create a local worst. Once

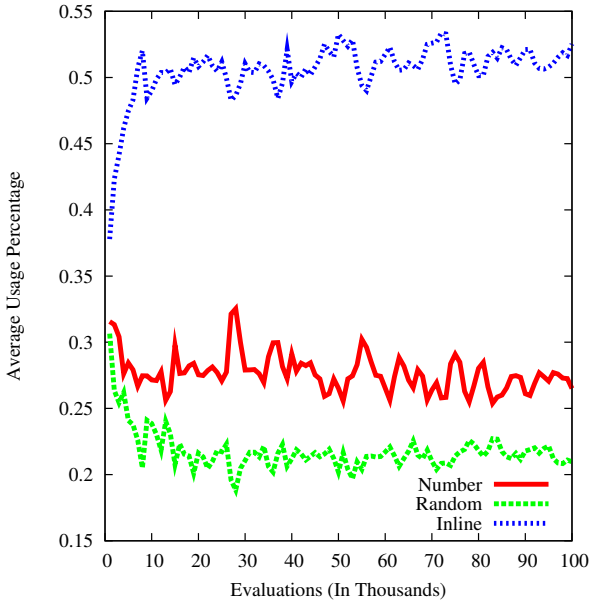

Figure 1.5: Primitive parameter usage on Offset Rastrigin

the search has reached a point where most genes are inside the peak where the global optimum is located, *Merge* can help create genetic material that is guaranteed to be better than the worst value of the two parents.

Offset Rastrigin was designed to counteract SCX's ability to move genes from one locus to another. As such, SCX had a significantly harder time solving this problem than normal Rastrigin. Even so, it resulted in statistically better results than AX. Figure 1.5 shows SCX's ability to adapt to the problem at hand. In a very small number of evaluations, approximately 50% of all constructs were *Inline*, meaning that almost all primitives were acting between identical loci.

The same operator that was able to outperform all of the traditional operators on real-valued problems was also able to outperform them on binary problems without any modification. Figure 1.2 shows SCX greatly outperforming NPX on NK, in that NPX appears to have stagnated while SCX is still improving. The most likely explanation is that SCX was better able to overcome gene interdependence than NPX. Another likely benefit is the reduction of gene stagnation. In NPX, if no member of the population has a particular allele at a particular locus, then only mutation can

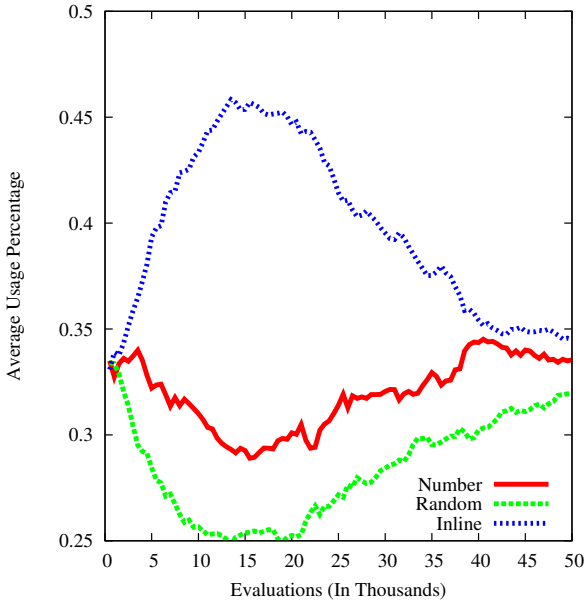

Figure 1.6: Primitive parameter usage on DTrap

reintroduce that allele. Thus, local optima can spread a specific individual through the population, reducing genetic diversity. In SCX, as long as an allele exists in any locus, SCX is able to spread that allele to all other genes.

The final test problem was DTrap. SCX's improvement over traditional crossovers was the least significant on this problem, and required a different configuration to achieve statistically significantly better results than NPX. Figure 1.6 shows SCX biasing towards *Inline* for a short time, but then returning to even usage, likely because SCX is exploiting the similarity in the search space. In the beginning it focuses on optimizing individual traps. Once this is done, it moves genetic material between traps. Since the optimal value is to have all bits set, and the next best is to have no bits set, this inter-gene mixing can help SCX escape local optima. A likely reason SCX had difficulty on DTrap as opposed to the other test problems is its genome size.

1.7.2. Initial Crossover Length Sensitivity. The significance of the initial crossover length parameter is reported in Table 1.4. All tests for Rastrigin and Offset Rastrigin obtained no statistical difference when using initial values ranging from 1 to 200, indicating this parameter has little to no effect on these problems.

Figure 1.7 shows the average crossover length quickly converging to a common value, which is likely responsible for how little effect the initial length has on final solution quality. Also, the convergence value changes throughout the run, indicating a need for the length to change dynamically.

When using a confidence level of α=*0.05* on DTrap, only the three smallest tests obtained statistically worse results than the optimal value found. Also, setting the parameter anywhere between 70 and 200 causes no significant change to the final results, and higher values are likely to continue this trend. Therefore, the final result should be considered very insensitive to this new parameter.

While Rosenbrock had more sensitivity to initial crossover length than Rastrigin, Offset Rastrigin, and DTrap, it was not very significant. Approximately 52% of the tests obtained statistically worse results than the optimal value found. All initial crossover length tests using a value greater than 120 obtained indistinguishable results at this confidence level.

NK experienced the greatest impact of initial crossover length on final solution quality, with 57% of tests receiving statistically worse results than optimal settings. NK was also the only problem to show improvement by having smaller initial length, as 10 through 90 obtained results of similar quality to the optimal value. A likely explanation for NK's sensitivity is shown in Figure 1.8, which shows the fitness of the different initial crossover lengths over time. While the different initial lengths have widely different results in the middle of the run, they appear to be converging near the end. What likely happened is that the different runs require different amounts of time to converge the crossover length to the correct size. Thus, they will take longer to converge, but will likely reach similar results.

While some of the problems' final solution quality was marginally impacted by the initial crossover length, it is important to compare this impact with what this parameter replaces. When performing the comparisons with traditional crossovers,

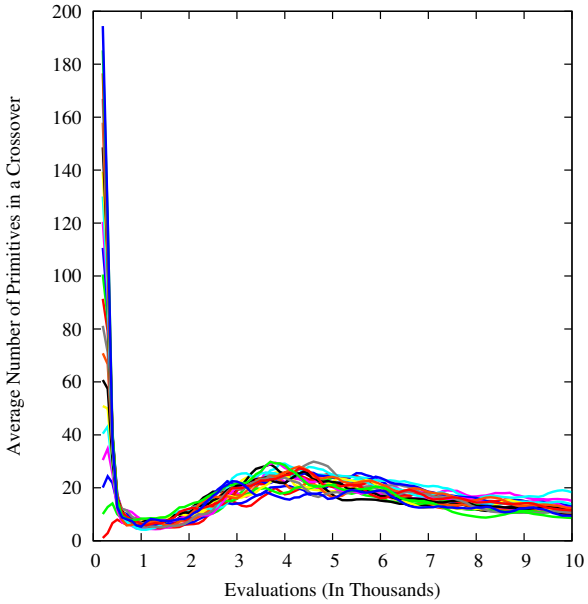

Figure 1.7: Crossover length on Rastrigin

SCX always used an initial length of five. On Rosenbrock and DTrap, this value falls between the two worst initial lengths tested. On NK, the only other problem to receive statistically different results based on initial crossover length, the optimal value found was 80. Even the worst settings tested for this parameter resulted in SCX outperforming the best of the traditional crossovers.

It is also important to note that in all problems except DTrap, SCX was using the best configuration for the traditional crossover. It is possible that some initial crossover lengths obtained statistically different results than others due to configuration. For example, Rosenbrock and NK both used drastically higher mutation rates than other problems that saw much less parameter sensitivity.

1.8. CONCLUSION

Properly configuring an EA requires a lot of testing, and poorly chosen crossover operators can greatly reduce solution quality. To minimize these issues and create more powerful crossover operators, Self-Configuring Crossover (SCX) is introduced in

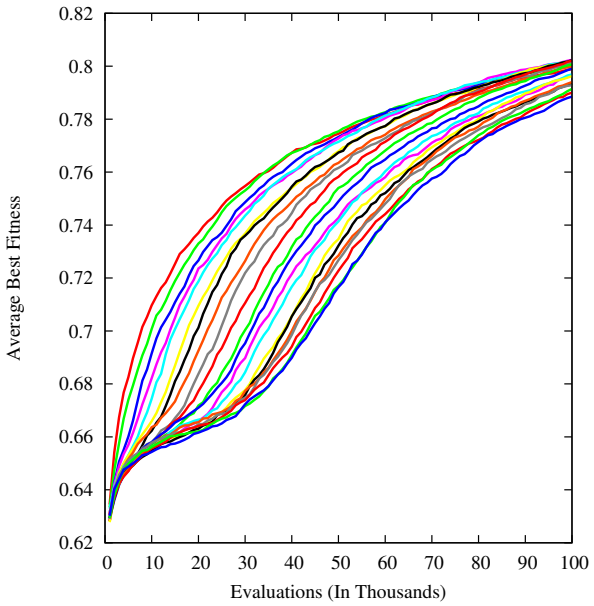

Figure 1.8: Fitness for different initial crossover lengths on NK

this paper. This method expands an individual's genome to include an LGP encoded crossover operator, used whenever that individual creates a child. This operator is defined using two primitives, *Swap* and *Merge*, chosen to represent the building blocks of existing crossovers.

To test this new method, three traditional operators (arithmetic crossover, n-point crossover, uniform crossover) were used for comparison on five benchmark problems (Rosenbrock, Rastrigin, Offset Rastrigin, NK, DTrap). Each comparison operator was tuned extensively by a Meta-EA to help ensure the best possible performance. Whichever operator obtained the best results was then statistically compared with SCX. On all problems tested, SCX was able to statistically outperform the traditional operators.

Since SCX introduces a new parameter, initial crossover length, analysis was performed to test how much impact this parameter has on final solution quality. For all problems, statistically optimal results at a confidence of α=*0.05* could be reached by a range of 80 or more values. Also, while the initial crossover length may effect SCX's performance, without optimization SCX still outperformed all traditional

crossovers on all problems tested. As a result, the effect of this parameter should be considered negligible.

1.9. FUTURE WORK

SCX needs to be compared on a wider range of problems with state-of-the-art crossover operators, such as network crossover [10] and linkage learning [11], as well as advanced EAs such as Estimation of Distribution Algorithms [22] and CMA-ES [23]. SCX also needs to be generalized to cover permutation based problems. All of the experiments performed in this paper used configurations optimized for crossovers other than SCX. To better examine the full potential of SCX, experiments need to be performed using SCX with optimal configurations. While a considerable amount of work was done to optimize how SCX encoded crossovers and performed crossover evolution, further study to improve these mechanics is expected to result in further SCX improvement.

Table 1.4: P-value comparison with optimum initial crossover length

Initial Length	**Rosenbrock**	**Rastrigin**	**Offset Rastrigin**	**NK**	**DTrap**
1	0.000	1.000	1.000	0.014	0.000
10	0.001	1.000	1.000	0.051	0.004
20	0.000	1.000	1.000	0.077	0.002
30	0.000	1.000	1.000	0.245	0.073
40	0.001	1.000	1.000	0.298	0.471
50	0.002	1.000	1.000	0.689	0.978
60	0.010	1.000	1.000	0.167	0.361
70	0.009	1.000	1.000	0.080	1.000
80	0.076	1.000	1.000	1.000	1.000
90	0.034	1.000	1.000	0.050	1.000
100	0.033	1.000	1.000	0.015	1.000
110	0.002	1.000	1.000	0.001	1.000
120	0.401	1.000	1.000	0.001	0.912
130	0.201	1.000	1.000	0.000	1.000
140	1.000	1.000	1.000	0.000	1.000
150	0.194	1.000	1.000	0.000	1.000
160	0.611	1.000	1.000	0.000	1.000
170	0.626	1.000	1.000	0.000	1.000
180	0.100	1.000	1.000	0.000	1.000
190	0.184	1.000	1.000	0.000	1.000
200	0.420	1.000	1.000	0.000	1.000

3. LINKAGE TREE GENETIC ALGORITHMS: VARIANTS AND ANALYSIS

ABSTRACT

Discovering and exploiting the linkage between genes during evolutionary search allows the Linkage Tree Genetic Algorithm (LTGA) to maximize crossover effectiveness, greatly reducing both population size and total number of evaluations required to reach success on decomposable problems. This paper presents a comparative analysis of the most prominent LTGA variants and a newly introduced variant. While the deceptive trap problem (Trap-k) is one of the canonical benchmarks for testing LTGA, when LTGA is combined with applying steepest ascent hill climbing to the initial population, as is done in all significant LTGA variations, Trap-k is trivially solved. This paper introduces the deceptive step trap problem (StepTrap-k,s), which shows the novel combination of smallest first subtree ordering with global mixing (LTS-GOMEA) is effective for black box optimization, while least linked first subtree ordering (LT-GOMEA) is effective on problems where partial reevaluation is possible. Finally, nearest neighbor NK landscapes show that global mixing is not effective on problems with complex overlapping linkage structure that cannot be modeled correctly by a linkage tree, emphasizing the need to extend how LTGA stores linkage to allow the power of global mixing to be applied to these types of problems.

1.1. INTRODUCTION

Linkage learning has been shown to benefit the performance of evolutionary algorithms and a variety of linkage algorithms have been developed which use non-tree structures [24]. The canonical Linkage Tree Genetic Algorithm (LTGA) variant was introduced in 2010 [25, 26] with two promising additional variants being intro-

duced in 2011 [27, 28]. The basis for all LTGA variants is the identification of linked substructures in the genome by examining the entropy in the current population and representing the linkages as a hierarchical tree. Each node in the tree represents a cluster of linked genes and the children of each node are non-overlapping subsets of their parent. To exploit the linkage information, LTGA performs a multistage crossover designed to allow lots of exploration without disrupting interdependent genes. The LTGA variants differ in the exact method for how they construct their linkage tree, apply it during crossover, and select parents to perform crossover on, with little analysis of the advantages and disadvantages of each of these options. Furthermore, most of the work done with LTGA employs Steepest Ascent Hill Climbing (SAHC) on the initial population to help the initial tree construction, the effects of which warrant further investigation.

Table 1.1 shows the principal differences between the different LTGA variants in terms of option combinations. The next few sections provide an introduction to how LTGA works and explain the rationale driving these different option combinations. The effects of each option are discussed and some experimental evidence provided to support this discussion. To isolate the effects of each, a new LTGA variation consisting of a unique option combination is introduced. Also, a new benchmark is introduced in Section 1.7 to better understand how LTGA exploits the local search on the population. Finally, the conclusions will summarize the findings and indicate which variant is best under what circumstances.

1.2. INITIAL POPULATION

In LTGA's canonical form, the initial population undergoes SAHC before starting evolution. In each SAHC iteration, all of an individual's neighbors are evaluated. If any of those neighbors are better than the current individual, then the best neighbor replaces it. SAHC continues until no improvement can be found. On

Table 1.1: Existing variations on LTGA

Dimension	**Options**
Initial Population	Random / SAHC
Clustering	Full / Pairwise
Subtree Ordering	Least Linked First / Smallest First
Crossover	Two-Parent / Global Mixing / Global Best Next
Variations	**Options Used**
Original [25, 26]	SAHC, Full, Least Linked First, Two-Parent
Original+	SAHC, Pairwise, Least Linked First, Two-Parent
Pairwise [27]	SAHC, Pairwise, Smallest First, Two-Parent
LT-GOMEA [28, 29]	SAHC, Pairwise, Least Linked First, Global Mixing
LTS-GOMEA	SAHC, Pairwise, Smallest First, Global Mixing

binary problems, the SAHC neighborhood consists of all individuals within one bit flip of the original individual. The primary reason given for why LTGA uses SAHC on the initial population is to discover linkage information [27]. The linkage tree is constructed using all of the individuals in the current population and is built before any type of selection occurs. As a result, if SAHC is not performed on the initial population and no other steps are taken, LTGA attempts to determine the linkage of randomly generated genes. Not only does this create a meaningless tree, [27] argues that applying those trees can be very harmful to the search's success. If SAHC is not used, they suggest some form of selection be performed prior to the initial tree creation, such as tournament selection.

Using SAHC on the initial population can result in up to μn^2 evaluations. On some problems it is possible to do partial reevaluation such that neighbors of an individual can be evaluated at a much lower cost than a brand new individual [29]. In such cases, the total number of SAHC iterations is counted, which has an upper bound of μn steps. As a promising offset to SAHC's cost based on population size, [29] shows how performing local search can reduce the population size required to achieve success.

1.3. CONSTRUCTING TREES

LTGA stores linkage information as a tree. Each node in the tree represents a cluster of genes that LTGA believes to be interdependent. The more linked two genes, the further from the root they will be linked. Each tree is built by first creating a subtree with one node for each locus in the genome. LTGA then iteratively joins the two subtrees that it considers the most linked into a binary tree whose root contains all of the loci of both subtrees. As a result, the number of subtrees is reduced by one each iteration, meaning the total number of nodes in the final tree is twice the number of dimensions.

To aid in explaining the LTGA variants, Table 1.2 provides an example population and the construction of a linkage tree for that population. The first three loci (a, b, and c) and the last three loci (d, e, and f) are likely linked, respectively. When constructing this tree, the first merge combines a and b. In the next iteration, LTGA can combine any of the remaining individual loci with each other, as well as with the newly created subtree ab, but it can no longer combine loci with just a or b. This represents one of the limitations of LTGA, since it is unable to fully capture a relationship such as a linked to b and b linked to c without a linked to c. The closest approximation LTGA can make to this type of linkage is if a is more tightly linked to b than b is to c, then the linkage tree can contain both ab and abc, which is one of the primary advantages of a tree representation over simple clusters. Also, since LTGA rebuilds the linkage tree each generation, changes in the population can allow it to shift priorities, for instance from ab to bc. It is important to note that the tree provided in Table 1.2 is not the only one that can be constructed for this population and that LTGA contains no bias based on where loci are positioned in the genome. LTGA also has no bias on merging larger or smaller subtrees first.

1.3.1. Cluster Distance. The linkage between genes is determined using the normalized variation of information, as defined by Eq. 1.5 where D is the distance

Table 1.2: Example population and tree construction

Individual 1	000000
Individual 2	111000
Individual 3	000111
Linkage Tree	abcdef
Merge 1	ab
Merge 2	abc
Merge 3	de
Merge 4	def

between cluster C_i and C_j, and H is the entropy of a cluster, as given by Eq. 1.6. $H(C_i, C_j)$ denotes the entropy of $C_i \cup C_j$, with S representing all possible strings that cluster C can be, and $p_c(S)$ being the percentage of the current population that has cluster C set to S.

$$D(C_i, C_j) = 2 - \frac{H(C_i) + H(C_j)}{H(C_i, C_j)} \tag{1.5}$$

$$H(C) = -\sum_{S} p_c(S) \log(p_c(S)) \tag{1.6}$$

As a result, $D(C_i, C_j)$ provides the ratio of the entropy of the clusters separately to the clusters joined. The minimum distance is achieved when, for each S in C_i there is exactly one S' in C_j such that all individuals who have S also have S'. This method of calculating distance measures how well the contents of one cluster can predict another, which implies they are linked.

Using the example population in Table 1.2 and starting after the first merge, lets find the distance between the cluster ab and the clusters c and d. For cluster ab, $p_{ab}(00) = 2/3$, $p_{ab}(11) = 1/3$, and all other possible strings for ab never occur in the population. Both c and d have $p(0) = 2/3$ and $p(1) = 1/3$. This means that the nominator in Eq. 1.5 will be the same for both $D(ab, c)$ and $D(ab, d)$ and also that all three clusters have the same value for H. Joined, $p_{abc}(000) = 2/3$ and $p_{abc}(111) = 1/3$ for abc and $p_{abd}(000) = 1/3$, $p_{abd}(110) = 1/3$ and $p_{abd}(111) = 1/3$ for abd. Because

the entropy for *abd* is higher than *abc*, the distance value for *ab* to *c* will be lower, and LTGA will choose to join *abc* first.

1.3.2. Pairwise Approximation. In the original LTGA variant [25, 26], hereafter referred to simply as Original, the method given in Section 1.3.1 was used to merge all clusters. However, a follow up paper showed that this method does not scale well, as calculating the entropy between all possible clusters becomes very time consuming when solving high-dimensional problems [27]. To overcome that problem, that paper proposed an estimation method, provided in Eq. 1.7.

$$D'(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{c_i \in C_i} \sum_{c_j \in C_j} D(c_i, c_j) \tag{1.7}$$

Instead of finding the entropy of an entire cluster, this measure only finds the entropy between all pairs of loci in the population. To deal with clusters of sizes greater than one, this method finds the average of the pairwise entropy for all loci in C_i with all loci in C_j. While the intent behind this metric was to reduce the time complexity used to construct trees, in which it succeeded, using the pairwise distance also led to a drop in the population size and number of evaluations required to achieve success on a number of problems [27]. The first algorithm to use this method is hereafter referred to as Pairwise, and Original using this method is called Original+.

1.4. CROSSOVER

The linkage tree is used to perform crossovers that do not disrupt gene linkage, while still exploring the search space. Each subtree in the linkage tree is used as a crossover mask, such that all of the genes in that subtree's cluster are crossed between the parents. When creating a single offspring, all of the crossover masks are used sequentially. Crossovers that create an improvement are kept, while detrimental changes are discarded. As was discussed in the context of LT-GOMEA [28, 29], this

incremental change can be seen as a form of local search. Since the leaves of the linkage tree each contain a single locus, LTGA's crossover will mimic a simple hill climber or bit flip mutation.

Table 1.3, Table 1.4, Table 1.5, and Table 1.6 all provide examples of how LTGA performs crossover, each for a different variant. The left column in each shows which subtree cluster is in use and the right column(s) show how the offspring are being modified. Each offspring is created through a series of $2N-2$ crossovers, where N is the number of problem dimensions. This includes all crossovers of length one and all other subtrees in the linkage tree. The root of the tree, which contains all of the loci, is not crossed because it will not create any new individuals.

Since the most linked genes are clustered first and are treated as a unit in all further clustering, few crossovers will occur that can disrupt their linkage. For instance, the two genes that have the minimum distance (tightest linkage) will be merged first. All other clusters will either contain both genes or none, so only two of the $2N-2$ crossovers act on one gene but not the other. Conversely, pairs of clusters with the highest distance (weakest linkage) will be merged last, allowing many crossovers to mix their gene values. In the running example, a and b were the first merged cluster, meaning that only their leaf nodes can perform a crossover that splits their values, while a and d are not linked until the root of the linkage tree, and as a result the subtrees a, d, ab, abc, de, and def can all freely mix their gene values.

As a side effect to using hierarchical clustering, LTGA is able to recover from some amount of confusion of gene linkages. Consider the reverse, a nonhierarchical crossover that always maintains the linkage between the most linked genes. There, the variation operators would be unable to increase the entropy of a cluster once it is formed, without first reducing the entropy of some other group of genes even lower. In LTGA, crossover occurs inside all clusters, with the potential of increasing the

Table 1.3: Two parent crossover with least linked first ordering as used by Original. Bold indicates which genes changed and a * indicates that changes were kept.

Parents	000000	111000
def	000**000**	111**000**
de	000**00**0	111**00**0
abc	**111**000	**000**000
ab	**11**0000	**00**1000
Single Bit Crossover Here		

entropy of that cluster, making it less likely to form that cluster in the future without impacting the diversity of the rest of the population.

1.4.1. Repeated Evaluations. If no preventative measures are taken, LTGA will create an immense number of duplicate individuals. For example, only four of the twenty individuals created in Table 1.3 are unique. As a slight prevention, all of the published LTGA variants avoid evaluating individuals that are identical to their parents, or at least make no mention of deviating from the original LTGA formulation that did this. While helpful, this does not catch all of the duplicates that are created. In the same example, all four unique individuals are created twice. For this study, a hash table is used to ensure that no individual is evaluated twice. This provides an accurate gauge of how many unique evaluations each LTGA variant uses and removes any bias against LTGA variants that are more likely to create duplicate individuals.

1.4.2. Subtree Ordering. The order in which subtrees are used to perform crossover is one of the primary differences between LTGA variants. In Original and LT-GOMEA, subtrees are applied in least linked first order. In this ordering, the crossover masks are stored in a stack. All of the individual loci are pushed randomly onto the stack. During each iteration of tree construction, the newly formed cluster is also pushed onto the stack. Table 1.3 shows the ordering for the running example. Since *def* was the last cluster to be merged, it is the first to be applied. The emphasis of this choice of ordering is for crossover to perform solution refining. The initial

crossovers are likely to be large and will be based on the least amount of information in the linkage tree. These steps provide search space exploration. The more linked LTGA believes a cluster of genes to be, the more likely it believes that keeping those genes together will allow for an individual to improve by crossing them. As such, this method starts with the least predictable crossovers, followed by something very similar to local search, bringing the solution to a local optima.

An alternative subtree ordering was used in Pairwise which sorts the clusters based on size instead of linkage. This method is more akin to optimizing the parts of the problem you know the most about before using the optimized pieces to make larger changes. Table 1.4 provides an example for this ordering. Smallest first is not the direct opposite of least linked first. One major difference is that least linked first has the potential to deal with the same set of genes in a number of sequential crossovers, where as in smallest first that is very unlikely. In the example, *de* is crossed directly after *ab* and before *abc*, which means if there is a linkage between *ab* and *de* that was previously undiscovered, this ordering has the potential to try it before modifying *c*. In general, subsequent crossovers in smallest first subtree ordering are less likely to deal with the same genes.

1.4.3. Parent Selection. In Original and Pairwise, all crossovers for a single offspring are based on two parents, similar to normal Genetic Algorithms (GAs). To create each child, two parents are selected at random. For each cluster in the linkage tree, two children are created by swapping the genetic material of the parents for that cluster. If either of the children created in one step of the crossover are better than the best of the parents, both of the parents are replaced by the children before continuing on to the next subtree. This method of replacement ensures that the genetic diversity inside a single crossover is unchanged, as whatever patterns existed in the parents before crossover will still exist after crossover. Often this has a polarizing effect, as the better of the parents is likely to improve by taking the best parts of the weaker

Table 1.4: Two parent crossover with smallest first ordering as used by Pairwise. Bold indicates which genes changed and a * indicates that changes were kept.

Parents	000000	111000
Single Bit Crossover Here		
ab	**11**0000	**00**1000
de	000**00**0	111**00**0
abc	**111**000	**000**000
def	000**000**	111**000**

parent. After all of the linkage clusters have been applied, the best child created during the entire process is copied into the next generation. This means that each parent pairing creates a single child. As LTGA uses generational survivor selection, to achieve a steady population size parents must participate in creating more than one offspring each generation. Since parents are modified during the creation of their offspring, subsequent offspring created by the same parent will benefit from the learning performed during the creation of their first. Table 1.3 and Table 1.4 provide examples for this type of crossover. The subtree clusters *ab* and *abc* swap 00 with 11 and 000 with 111 respectively. While *abc* does improve one of the children, it is not better than both of the parents (it is in fact a duplicate of one of the parents) and the change is therefore not kept. If there were more genes in the individuals and the left parent was able to use *abc* as all ones better than the right, then the changes would have been kept.

The LTGA variant Linkage Tree Global Optimal Mixing Evolutionary Algorithm (LT-GOMEA) differs from Original and Pairwise by performing crossover in a method more similar to Estimation of Distribution Algorithms than GAs. Instead of selecting two parents to pull genetic material from, LT-GOMEA starts by cloning a single parent to initialize the child. For each subtree crossover mask, LT-GOMEA chooses a random parent from the population to donate genetic material to the child for that mask. If the child is improved, the changes will be kept, otherwise they are reverted. While it is not explicitly stated as part of LT-GOMEA, the donated genetic

material is assumed here to not be identical to the child's gene values, as doing so would waste a chance at improving the individual. Tables 1.5 and 1.6 provide examples of performing global mixing crossover using the population and linkage tree provided in Table 1.2. When performing crossover on *abc* and *def*, the child receives genetic material from a random parent with different genetic material than its own. In both cases this represents an improvement and the material is kept.

In [29], a variant to LT-GOMEA was proposed which tried all possible donations for each crossover before choosing to keep any. This change allows it to perform a steepest ascent local search on each subtree cluster for each individual, finding the global next best. Unfortunately, doing so incurs significant overhead and causes premature convergence due to large subtrees copying so much of an individual as to allow the current best individual to spread too rapidly.

1.5. EXPERIMENTS

While most of the permutations listed in Table 1.1 are valid ways to configure an LTGA, this paper focuses on the variants most likely to achieve interesting results. From the work with LT-GOMEA and Pairwise, it appears that subtree ordering and crossover represent the ways in which the state-of-the-art LTGA variations differ. To examine the effects of these variations, empirical test results are presented for all four combinations of least first linkage, smallest first linkage, two parent crossover, and global mixing, which are represented by Original+, Pairwise, LT-GOMEA, and the novel combination Linkage Tree Smallest First Global Mixing Evolutionary Algorithm (LTS-GOMEA). While global next best is an option for crossover, [29] tested LT-GOMEA against global next best and found it to be significantly inferior to not warrant further investigation. The full description of what options each LTGA variant employs is available in Table 1.1.

Table 1.5: Global mixing crossover with least linked first ordering as used by LT-GOMEA. Bold indicates which genes changed and an * indicates that changes were kept.

Parent	000000
def	000**111***
de	000**00**1
abc	**111**111*
ab	**00**1111
Single Bit Crossover Here	

Table 1.6: Global mixing crossover with smallest first ordering as used by LTS-GOMEA. Bold indicates which genes changed and an * indicates that changes were kept.

Parent	000000
Single Bit Crossover Here	
ab	**11**0000
de	000**11**0
abc	**111**000*
def	111**111***

In order to ensure a minimal population size, each LTGA variant was tuned to each problem using a modified version of bisection [30]. In order to remove operator bias and further generalize the method, the bisection used here starts by setting the initial population size to the minimum possible for the algorithm, which in this case is two. It then proceeds by testing to see if its guess meets the success criteria. If not, then it doubles its guess and tries again. Once the criteria are met, the current guess is used as the maximum population size and the previous guess is the minimum population size for bisection. This ensures the starting minimum is below the population size required and the starting maximum is at least as high as required. In this instance, the required success rate was 24 of 25 runs finding the global optimum. When actually gathering experimental results, all experiments used 100 runs.

To reduce the noise generated by the initial population, a set of individuals was created for each run of each population size on each problem. These sets of

individuals were then all optimized using SAHC. Having all versions of LTGA share initial populations for the same population size facilitates better examining of how each algorithm utilizes the information in those populations. Furthermore, any difference in population size is then more clearly indicative of a need for different initial information, and not just caused by random chance.

1.6. DECEPTIVE TRAP

The canonical benchmark for LTGA is the deceptive trap (Trap-k) problem [20], which divides the genome into separate traps of length k bits. Each trap is scored using Eq. 1.8, where t is equal to the sum of the bit values in the trap.

$$trap(t) = \begin{cases} k-1-t, & t < k \\ k, & t = k \end{cases} \tag{1.8}$$

Each trap is fully deceptive, in that all non-optimal values lead away from the global optimum of all ones toward the local optimum containing all zeros. As a result, any crossover that affects less than a complete trap will likely cause both individuals to move away from the global optima. This need to preserve linkage, coupled with trap independence, makes Trap-k an ideal proof of concept benchmark for LTGA.

LTGA needs an initial population that contains the optimized value for each trap in at least one individual. Local search on the deceptive trap function ensures that all traps in all individuals are initialized to either the optimum value (all ones) or the local optimum value (all zeros). The first benefit for LTGA is the complete removal of entropy for each trap. Independent of trap size, after local search all traps will contain less than one bit of information. As a result, when LTGA builds the gene clusters, all genes in a trap will have the minimum possible distance from each other. Furthermore, when performing a crossover using a subset of the cluster (i.e., a subtree

of size less than k), new trap values cannot be created as there is no way to change less than k bits in any individual that will improve its fitness. Thus, the only way for LTGA to lose track of a trap's grouping is if the entire population agrees on the value between two traps. In such an event it is still unlikely that LTGA will create a trap with a value other than all ones or all zeros. When performing a crossover that contains one or more complete traps, LTGA's behavior changes depending on the type of crossover in use. The final best offspring of Original and Pairwise, using two-parent crossover, will contain all of the optimized traps that started in either individual. LT-GOMEA and LTS-GOMEA will solve even faster, as after all complete traps have been crossed over, the individual produced will contain all optimized traps that exist anywhere in the population.

The second benefit of local search for LTGA is the increase in the number of optimized traps in the starting population. The probability of a trap containing an optimal value in a single individual increases from $\frac{k}{2^k}$ to $\frac{k+1}{2^k}$, since SAHC will not change any randomly generated optimized traps, and will improve all traps with a single incorrect bit to the optimum. As a minor downside, if there is any trap in which none of the individuals in the population contain the optimized version, LTGA will be unable to find the global optimum as it is incapable of creating any allele that does not exist somewhere in the current population.

Tables 1.7 and 1.8 provide the experimental results for the LTGA variations on trap-5 and trap-7, respectively. As expected, all of the algorithms required the same initial population size, due to their identical requirements on the existence of optimized traps. The rarest trap count value given for each problem is the average number of individuals in the initial population containing the optimized version of the trap with the lowest frequency of individuals containing its optimized version over all successful runs.

Table 1.7: Deceptive Trap results, k=5

N=50	**Population Size**	**Trap Rarest Count**	**Local Search Steps**	**Evals To Success**
Original+	26	1.94	513.93	1946.53 (481.90)
LT-GOMEA	26	1.94	513.93	21.31 (5.55)
Pairwise	26	1.94	513.93	2200.38 (468.01)
LTS-GOMEA	26	1.94	513.93	89.81 (0.48)
N=100				
Original+	24	1.52	922.33	6134.35 (1244.64)
LT-GOMEA	24	1.52	922.33	47.91 (10.19)
Pairwise	24	1.52	922.33	6794.50 (1307.67)
LTS-GOMEA	24	1.52	922.33	179.82 (0.48)
N=150				
Original+	32	2.26	1827.23	15430.62 (2625.32)
LT-GOMEA	32	2.26	1827.23	79.65 (14.06)
Pairwise	32	2.26	1827.23	15507.38 (2577.15)
LTS-GOMEA	32	2.26	1827.23	269.78 (0.48)

Where the versions of LTGA vary is in the number of evaluations after local search that each requires to achieve success. Most notably, both versions using global mixing (LT-GOMEA and LTS-GOMEA) require far less than either form of two-parent crossover. When performing global mixing, each crossover mask that contains exactly one trap will result in the child having the optimized version of that trap after crossover. This is because the child either has the optimized version, and therefore no chance to improve for that crossover mask, or it has the suboptimal version and will therefore randomly select a parent with the only other genetic material for that crossover mask: the optimized value. Since two-parent crossover can only create offspring containing the optimized traps from its two parents, multiple crossovers are required to build individuals such that when they are paired, at least one has the

Table 1.8: Deceptive Trap results, k=7

	Population Size	Trap Rarest Count	Local Search Steps	Evals To Success
N=49				
Original+	64	1.83	1486.62	2148.50 (571.09)
LT-GOMEA	64	1.83	1486.62	13.41 (3.43)
Pairwise	64	1.83	1486.62	2454.71 (579.23)
LTS-GOMEA	64	1.83	1486.62	90.91 (0.27)
N=98				
Original+	78	1.76	3541.38	12983.56 (2990.20)
LT-GOMEA	78	1.76	3541.38	31.89 (7.08)
Pairwise	78	1.76	3541.38	14082.40 (2884.49)
LTS-GOMEA	78	1.76	3541.38	181.92 (0.26)
N=147				
Original+	80	1.57	5410.32	27150.74 (5581.73)
LT-GOMEA	80	1.57	5410.32	53.68 (11.26)
Pairwise	80	1.57	5410.32	30189.41 (4738.40)
LTS-GOMEA	80	1.57	5410.32	272.96 (0.17)

optimized value for all traps. The difference between LT-GOMEA and LTS-GOMEA is trivial in this case, as the difference is how many crossover masks are tried before all single trap masks are applied. In LTS-GOMEA there will be $2N$ subtrees with a size less than or equal to k, as opposed to LT-GOMEA where, depending on tree construction, there can be as few as $2 \cdot (N/k) - 1$ subtrees with less than or equal linkage to the single trap subtrees. These formulas are near perfect predictors of the experimental number of evaluations found in Tables 1.7 and 1.8.

1.7. DECEPTIVE STEP TRAP

The artificial nature of the Trap-k problem, as exploited by SAHC, makes LTGA's performance on this benchmark an unlikely predictor of its ability on more

realistic problems. In order to create a very similar problem to Trap-k that cannot be exploited as much by SAHC, the creation of StepTrap-k,s as defined in Eq. 1.9 is proposed. Similar to Trap-k, StepTrap-k,s uses a series of non-overlapping trap functions, where each trap is scored using the sum of the bits set to one. The optimal value for each trap is all ones, with a local optimum of all zeros. All other configurations improve monotonically as they approach all zeros. The only difference between normal Trap-k and StepTrap-k,s is that StepTrap-k,s includes plateaus of length s. If s is set to one, then StepTrap-k,s is identical to Trap-k. For $s > 1$, up to $s - 1$ bits can change without a change in the fitness of non-optimal traps. By modifying Trap-k to StepTrap-k,s and setting s to two, SAHC will still create optimized traps at the same probability as in Trap-k, but it will leave the majority of traps in some partial state between all zeros and all ones. As such the loss of entropy for each trap will not be as extreme as in normal Trap-k, making LTGA more prone to clustering errors.

$$step\_trap(t) = \left\lfloor \frac{(k-s) \pmod{s} + trap(t)}{s} \right\rfloor \tag{1.9}$$

Tables 1.9 and 1.10 provide the experimental results for each variation of LTGA on the StepTrap-k,s problem using $s = 2$ with k set to 5 and 7 respectively. Unlike the uniform population sizes found on Trap-k, all LTGA variants needed a different population on StepTrap-k,s. For all N and k tested, LTS-GOMEA was the least destructive to existing optimized traps, and was therefore able to achieve the global optima with the smallest population sizes.

To understand why global mixing with smallest first ordering is the least destructive to existing optimized traps, we first need to examine how optimal mixing is able to lose an optimized trap. In order for a child to not have an optimized trap that exists in its parent, some part of the crossover step that created that mistake also has to improve another trap, raising the child's fitness more than it lost by disturbing the optimized trap. Incorrectly linked crossovers can include parts of multiple traps. If,

Table 1.9: Deceptive Step Trap results, k=5, s=2

N=50	Population Size	Rarest Trap Count	Local Search Steps	Evals To Success
Original+	73	8.66	414.4	13744 (2925.2)
LT-GOMEA	67	8.09	383.1	9643 (2606.3)
Pairwise	74	9.43	422.3	13475 (2901.1)
LTS-GOMEA	49	5.34	278.2	11192 (2163.7)
N=100				
Original+	86	9.78	893.7	46579 (7085.9)
LT-GOMEA	84	9.41	874.1	32533 (4820.6)
Pairwise	97	11.42	1011.6	50669 (7921.5)
LTS-GOMEA	62	6.32	642.7	36272 (4755.3)
N=150				
Original+	101	11.35	1520.8	91652 (13411.8)
LT-GOMEA	96	10.64	1449.6	58591 (7269.7)
Pairwise	107	12.54	1610.5	100951 (14306.7)
LTS-GOMEA	73	7.31	1103.0	68129 (10462.3)

when performing these crossovers, the amount some of the traps improve is greater than the fitness lost by breaking one of the other traps, the change is still kept.

Consider the example where a, b, c, and z are incorrectly linked together. The first three are part of one trap that is not at either optima in the parents, and the fourth is part of another trap which one of the parents has optimized. The crossover abc improves its trap by increasing its fitness two points, which offsets the crossover of z, which decreases its trap value one point. As a result, the optimized trap is lost. Smallest first ordering is sometimes able to avoid this problem by making minimal changes first. In the example, if any of a, b or c are linked before all three are combined with z and crossing that subtree can achieve even a single point of fitness

Table 1.10: Deceptive Step Trap results, k=7, s=2

N=49	Population Size	Rarest Trap Count	Local Search Steps	Evals To Success
Original+	120	4.30	535.6	36033 (4756.3)
LT-GOMEA	135	4.66	599.0	34349 (5035.3)
Pairwise	147	5.76	654.4	42689 (5746.5)
LTS-GOMEA	117	4.18	521.4	39870 (6170.5)
N=98				
Original+	233	8.71	1841.0	180075 (17685.8)
LT-GOMEA	197	7.20	1549.5	130998 (16004.0)
Pairwise	265	10.26	2090.6	222781 (25374.9)
LTS-GOMEA	143	4.37	1127.0	147621 (15173.7)
N=147				
Original+	256	9.58	2899.9	371343 (35206.8)
LT-GOMEA	317	12.17	3596.4	292347 (35440.8)
Pairwise	309	11.97	3509.9	485905 (50916.7)
LTS-GOMEA	200	6.61	2268.3	319875 (27276.5)

increase, then when crossing *abcz* there will no longer be enough improvement to offset the damage.

Conversely, because smallest first is more likely to push individual traps to their local optima before performing larger crossovers, it is less likely that larger crossovers that include partial traps will make enough of an improvement to be considered successful. This means that until better linkage information is found, most of the larger crossovers in smallest first ordering will likely be wasted. Least linked does not have this problem, and in fact the most tightly linked crossovers at the end of search are more likely to make improvements, and help prevent wasted evaluations. As an indirect added bonus, every successful crossover reduces the entropy of most if not all of the traps it effects, making the next generation's linkage tree more accurate.

This is true because as traps are pushed to either extreme, StepTrap-k,s approaches the Trap-k problem, which is very easy for LTGA to solve, as shown in Section 1.6.

Two-parent crossover is even more likely to destroy optimized traps than global mixing, as shown in Tables 1.9 and 1.10. While two-parent crossover can have all of the same flaws as global mixing for optimized trap destruction, it can also lose a trap due to parent pairing. If two parents are paired such that the worse parent has an optimized trap in a location the better parent does not, and no crossover exists that can move that trap without significantly damaging the other parent, that genetic material can be lost. It is possible that this explains why smallest first does not help Pairwise as much as LTS-GOMEA, as even though crossover is not directly destroying traps, the inability for Pairwise to move traps between parents is resulting in their loss. One final consideration about the trade off between LT-GOMEA and LTS-GOMEA is the inclusion of the initial search steps as evaluations. If either is applied to a problem where partial reevaluation is not an option, the number of evaluations required for local search will be about N times larger than the number of local search steps. LTS-GOMEA's smaller population size allows it to solve all of the StepTrap-k,s problems except $N = 49$, $k = 7$, $s = 2$ in less evaluations than LT-GOMEA if local search evaluations are included.

Similar to normal Trap-k, if LTGA is able to correctly link each trap independently before all optimized versions of any trap are lost, it is very unlikely that any LTGA variant will fail to reach the global optimum. This is because any crossover that moves a single trap from one parent to another can only improve that trap's fitness, which will shift the population toward both extremes, again making it resemble Trap-k. The only way to lose an optimized trap now would be to cross multiple traps into a child, where more traps are made optimal than are made non-optimal. Smallest first ensures that any crossover that moves multiple traps will first optimize the individual traps, which again reduces the likelihood of this happening. Two-parent crossover

is also unlikely to have this issue as, independent of what order traps are crossed, optimal traps that are lost in one step can be reclaimed in subsequent crossovers with the same parent or will be optimized before they can contribute to a lost optimized trap.

1.8. NEAREST NEIGHBOR NK

NK landscapes are a set of randomly generated benchmark problems [19]. An NK landscape is defined by the number of dimensions used (N), the number of other genes each gene relies on to calculate its own fitness (k), the epistasis table defining the linkages between genes, and N fitness functions which determine the fitness of each collection of linked genes. Nearest neighbor NK landscapes are a subset of NK landscapes in which each gene is linked to the k genes that directly follow it, with the final genes wrapping around. This subset is of interest for testing LTGA as it contains overlapping blocks of genes with varying levels of linkage. Furthermore, by utilizing dynamic programming it is possible to determine the optimal value for nearest neighbor NK landscapes in polynomial time [31].

Table 1.11 shows the results of each LTGA variant tested on nearest neighbor NK using $k = 5$. Unlike the previous two benchmarks, Pairwise was able to outperform both global mixing strategies on $N = 30$ and $N = 40$ in both population size and total number of evaluations. Original+ was also able to achieve a lower number of evaluations on $N = 40$. The most likely cause for this is LTGA's inability to capture the overlapping dependencies inherent in nearest neighbor NK. When performing two-parent crossover, all of the donated information comes from the same individual. As such, any partial improvements made to a child are more likely to work well with subsequent crossovers with the same parent as both sets of genes were effective together in at least that parent.

Table 1.11: NK results, k=5

	Population Size	Local Search Steps	Evals To Success
N=30			
Original+	132	1038.06	8253.86 (3473.91)
LT-GOMEA	131	1039.16	7743.29 (4804.82)
Pairwise	110	867.97	7144.20 (3426.76)
LTS-GOMEA	151	1219.60	10634.28 (5112.47)
N=40			
Original+	323	3286.34	33141.11 (12757.48)
LT-GOMEA	501	5102.29	36300.81 (21529.69)
Pairwise	250	2549.38	28875.14 (10652.87)
LTS-GOMEA	317	3233.64	40525.74 (16589.45)
N=50			
Original+	383	4786.59	67280.11 (22403.78)
LT-GOMEA	393	4905.07	60809.10 (24655.13)
Pairwise	371	4636.94	69266.90 (21912.95)
LTS-GOMEA	276	3451.42	62564.14 (19336.66)

When performing global mixing, no similar linkage exists between donated information. As a result, global mixing will attempt to combine genetic material from many different parents, with only a limited ability to keep linked information together. Individuals created in this way are not likely to be an improvement over the existing individual, and will therefore be a wasted evaluation. Finally, the increased population size is likely a symptom of premature convergence caused by this inability to generate good solutions.

The reason why $N = 40$ switches which type of mixing is most effective is probably because as the number of problem variables increases but the amount of overlap stays constant, the effect of incorrect linkage is lessened. This may explain

why both global mixing variants were able to reduce their required population size from $N = 40$ to $N = 50$.

1.9. CONCLUSIONS

Using the Trap-k problem to test the different variations of LTGA provides little to no information, as SAHC is able to convert what is normally a very difficult problem into a trivial one for any type of LTGA. In order to provide a better benchmark for non-overlapping linked genes, this paper introduces the StepTrap-k,s problem, which modifies the traditional Trap-k to contain fitness plateaus of width s. These plateaus prevent SAHC from pushing all traps to the extremes, providing a much more challenging problem for LTGA.

This new problem reveals global mixing's ability to converge faster and use smaller population sizes than two-parent crossover. By applying the smallest crossover masks first, LTGA can better avoid being destructive to existing optimized structures in the population when incorrect linkage occurs at the expense of needing more search evaluations. As a result, on problems where local search cannot exploit partial reevaluation, the novel LTGA variant LTS-GOMEA, which uses smallest first ordering, may outperform LT-GOMEA, which uses least linked first ordering, on non-overlapping linkage problems.

Testing the LTGA variants on nearest neighbor NK landscapes provides insight to how each variant handles linkage styles less suited to tree structures. On problems with lots of overlap, the two-parent variants seem to perform better than global mixing due to the inherent linkage between subsequent crossovers between the same two parents. As global mixing is able to significantly outperform two-parent crossover in all other instances, improving how LTGA stores linkage information to allow for better representation of overlap is a very important next step for making LT-GOMEA and LTS-GOMEA more generally applicable.

SECTION

2. CONCLUSIONS

The first paper underscores the futility of attempting to manually configure an Evolutionary Algorithm (EA) optimally for a particular problem by providing experimental evidence that dynamically changing many parameters can positively effect the solution quality of an EA. The search space associated with dynamic parameter configuration is combinatorially large. Without automated controls of parameters, or redesign of the internals of an EA to no longer require parameters, both the time consuming processes available to novices and the experiential processes used by experts will not be able to cope with the exploding search space.

Achieving automated configuration of an EA is one of the great unsolved challenges in the field of Evolutionary Computation. The second paper provides a promising template for how to start tackling this challenge. Allowing the EA control of its recombination method diminishes the need to configure this aspect of the EA while improving the EA's performance and robustness. By following a similar method, it is likely possible to continue making EAs more robust, with more of the configuration controlled by evolution and less by the user.

Employing the method of redesign, the Linkage Tree Genetic Algorithm (LTGA) variants analyzed in the third paper are able to sidestep much of the difficulty of configuration. Using a single parameter, LTGA is able to solve certain classes of very difficult problems by automatically adapting to problem structure. The analysis performed provides a stepping stone to making LTGA even more flexible and powerful, while maintaining its relative independence from problem specific configuration.

The techniques discussed in the latter two papers represent steps towards making EAs more robust. While neither completely avoids the need for configuration, both reduce the difficultly of configuration, and as a result are more powerful and easier to apply than more traditional EAs. Future work can be done both extending and combining these techniques to be even more robust, potentially avoiding configuration all together and allowing EAs to achieve even wider application.

BIBLIOGRAPHY

[1] J. Smith. Modeling GAs with Self-Adaptive Mutation Rates. In *Proceedings of GECCO 2001*, pages 599–606, 2001.

[2] J. Cook and D. Tauritz. An Exploration into Dynamic Population Sizing. *In Proceedings of GECCO 2010 - Genetic And Evolutionary Computation Conference*, pages 807–814, 2010.

[3] A. Eiben, M. Schut, and A. deWilde. Is Self-Adaptation of Selection Pressure and Population Size Possible? In *Proceedings of PPSN IX*, pages 900–909, 2006.

[4] A. Nwamba and D. Tauritz. Futility-Based Offspring Sizing. In *Proceedings of GECCO 2009*, pages 1873–1874, 2009.

[5] J. Gomez. Self Adaptation of Operator Rates in Evolutionary Algorithms. In *Proceedings of GECCO 2010*, pages 162–173. Springer Berlin, Heidelberg, 2004.

[6] E. Smorodkina and D. Tauritz. Toward Automating EA configuration: the Parent Selection Stage. In *IEEE Congress on Evolutionary Computation*, pages 63–70, 2007.

[7] G. Papa. Parameter-less Evolutionary Search. pages 1133–1134. Atlanta, GA, USA, 2008.

[8] F. Vafaee, W. Xiao, P.C. Nelson, and C. Zhou. Adaptively Evolving Probabilities of Genetic Operators. *Machine Learning and Applications*, pages 292–299, 2008.

[9] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer, 2002.

[10] M. Hauschild and M. Pelikan. Network Crossover Performance on NK Landscapes and Deceptive Problems. *In Proceedings of GECCO 2010 - Genetic And Evolutionary Computation Conference*, pages 713–720, 2010.

[11] G. R. Harik and D. E. Goldberg. Linkage Learning Through Probabilistic Expression. *Computer Methods In Applied Mechanics and Engineering*, pages 295–310, 2000.

[12] L. Dioşan and M. Oltean. Evolving crossover operators for function optimization. pages 97–108. Springer-Verlag, 2006.

[13] I. Rechenberg. *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien des Biolischen Evolution*. Fromman-Hozlboog Verlag, Stuttgart, 1973.

[14] N. L. Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. *Proceedings of First International Conference on Genetic Algorithms*, pages 183–187, 1985.

[15] S. Meyer-Nieberg and H.G. Beyer. Self-Adaptation in Evolutionary Algorithms. *Parameter Setting in Evolutionary Algorithms*, pages 47–76, 2006.

[16] A.E. Eiben and J.E. Smith. Evolutionary strategies. In *Introduction to Evolutionary Computing*, pages 73–75. Springer-Verlag, Berlin Heidelberg, 2003.

[17] K. A. De Jong. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. *Ph.D. thesis*, 1975.

[18] H. Mühlenbein, D. Schomisch, and J. Born. The Parallel Genetic Algorithm as Function Optimizer. *Parallel Computing*, pages 619–632, 1991.

[19] S. Kauffman. The origins of order: self organization and selection in evolution. *Oxford University Press*, 1993.

[20] Kalyanmoy Deb and David Goldberg. Analyzing deception in trap functions. *In Proceedings of FOGA II: the Second Workshop on Foundations of Genetic Algorithms*, pages 93–108, 1992.

[21] S. K. Smit and A. E. Eiben. Comparing Parameter Tuning Methods for Evolutionary Algorithms. *IEEE Congress on Evolutionary Computation*, pages 399–406, 2009.

[22] Martin Pelikan, David Goldberg, and Fernando Lobo. A Survey of Optimization by Building and Using Probabilistic Models. *Computational Optimization and Applications*, 21(1):5–20, 2002.

[23] N. Hansen and A. Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. 9(2):159–195, 2001.

[24] G. R. Harik, F. G. Lobo, and K. Sastry. *Linkage learning via probabilistic modeling in the extended compact genetic algorithm.* Springer–Verlag, Berlin, 2006.

[25] D. Thierens. Linkage tree genetic algorithm: first results. *Genetic and Evol. Comp. Conf.(GECCO-2010)*, pages 1953–1957, 2010.

[26] D. Thierens. The linkage tree genetic algorithm. *Parallel Problem Solving from Nature*, pages 264–273, 2010.

[27] M. Pelikan, M. Hauschild, and D. Thierens. Pairwise and problem-specific distance metrics in the linkage tree genetic algorithm. *Genetic and Evol. Comp. Conf.(GECCO-2011)*, pages 1005–1012, 2011.

[28] D. Thierens and P.A.N Bosman. Optimal mixing evolutionary algorithms. *Genetic and Evol. Comp. Conf.(GECCO-2011)*, pages 617–624, 2011.

[29] P.A.N Bosman and D. Thierens. The roles of local search, model building and optimal mixing in evolutionary algorithms from a BBO perspective. *Genetic and Evol. Comp. Conf.(GECCO-2011)*, pages 663–670, 2011.

[30] K. Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Master's thesis, University of Illinois at Urbana-Champaign, Department of General Engineering, 2001.

[31] A. H. Wright, R. K. Thompson, and J. Zhang. The computational complexity of N-K fitness functions. *IEEE Trans. on Evolutionary Computation*, 4(4):373–379, 2000.

VITA

Brian Wesley Goldman was born in the suburbs of Saint Louis, Missouri, attending primary, middle, and high school in the Parkway school district. Brian attended Missouri University of Science and Technology (S&T) for his undergraduate degree in Computer Science, and was awarded a Bachelor of Science Summa Cum Laude in 2010. Brian graded for three courses at S&T and spent two semesters teaching the Introduction to C++ Laboratory course. He has also interned with AT&T, Dynetics, and Sandia National Laboratories. Brian has competed in ten artificial intelligence competitions, including taking first place twice in the S&T human vs computer chess tournament, second place in the University of Illinois Urbana-Champaign MechMania competition, and 25^{th} out of 4619 world wide in the Google AI competition. He received his Masters degree from S&T in May 2012.