Complex Systems 4 (1990) 415–444

Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale

David E. Goldberg
University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA

Kalyanmoy Deb
University of Alabama, Tuscaloosa, AL 35487 USA

Bradley Korb
McDonnell Douglas Space Systems Company, Huntsville, AL 35806 USA

Abstract. Genetic algorithms have been finding increasing application to difficult search and optimization problems in science and engineering. As more demands are placed on the methodology, the remaining challenges to the technique have become increasingly apparent, and that they have remained largely unanswered has become less than acceptable to a user base seeking to cash in on the method's oft-repeated promises of robustness. Foremost among these challenges is the so-called *linkage problem*. In difficult problems — so-called *deceptive* problems — if needed bits or features are not coded tightly on the artificial chromosome, simple genetic algorithms will be forced to bypass global optima and instead be misled toward less acceptable local optima. Standard suggestions to circumvent this problem through inversion or other reordering operators have not been successful, and it has been suggested that many of these operators are too slow to be of timely use. These difficulties led to the invention of so-called *messy genetic algorithms* that were introduced in a previous paper. Simply stated, messy genetic algorithms tackle the linkage problem by finding tightly coded building blocks initially and then juxtaposing them to find globally optimal structures. In the original study, a 30-bit, order-three deceptive problem was solved to global optimality, but a number of challenges were posed for the messy methodology. Specifically, nonuniform building block scale and size were highlighted as difficulties that must be overcome if mGAs were to become a technique of general applicability. These challenges have been overcome, and this paper presents theoretical and computational results using mGAs on problems of varying building block size and scale. The problem of nonuniform building block scale is tackled with a technique called *genic selective crowding*, and the problem of nonuniform

building-block size is handled with *null bits* and *tie breaking*. Together, these techniques appear to overcome these difficult hurdles, yielding convergence to global optima in problems of bounded deception. Furthermore, this desirable convergence is performed with remarkable efficiency. Theoretical computations are presented that show that mGAs converge in time that grows only as a polynomial function of the number of decision variables on a serial machine and as a logarithmic function of the number of decision variables on a parallel machine with a polynomial number of processors. The paper also suggests a number of extensions and continuations of this work, including the trial of messy floating point codes, messy permutations, and messy classifiers (rules). Although additional basic work is both needed and recommended, the compelling convergence and efficiency demonstrated by mGAs recommends them for immediate application in some of the many tough, blind combinatorial optimization problems of science and engineering that have gone unsolved for want of more tractable solution techniques.

1. Introduction

Like neural networks and connectionist systems, genetic algorithms and artificial evolutionary systems got their start during the cybernetics movement of the late 1940s and 1950s, and as neural nets faded during the 1960s and 1970s only to receive revived attention in the last decade, so too have genetic algorithms receded from view and undergone a recent renaissance. Simply stated, genetic algorithms are search procedures based on the mechanics of natural selection and natural genetics. They combine the use of string codings or artificial chromosomes and populations with the selective and juxtapositional power of reproduction and recombination to motivate a surprisingly powerful search heuristic in many problems.

Despite their empirical success, there has been a long-standing objection to the use of GAs in arbitrarily difficult problems. To assure convergence to global optima, strings in simple GAs must be coded so that *building blocks* — short, highly fit combinations of bits — can combine to form optima. If the linkage between necessary bit combinations is too weak, in certain types of problems called *deceptive* problems [7, 8], genetic algorithms will converge to suboptimal points. A number of reordering operators have been suggested to recode strings on the fly, but these have not yet proved sufficiently powerful in empirical studies, and a recent theoretical study [9] has suggested that unary reordering operators are too slow to be of much use in searching for tight linkage.

On March 23, 1989, a new approach to this problem was launched in the Genetic Algorithms Laboratory (GALab) at the University of Alabama. On that date, globally optimal results to a 30-bit, order-three deceptive problem were obtained using a new type of genetic algorithm called a *messy genetic algorithm* [10]. Messy genetic algorithms combine the use of variable-length

strings, a two-phase selection scheme, and messy genetic operators to effect a solution to the fixed-coding problem of standard simple GAs. Prior to this discovery, no provably difficult problem had ever been solved to optimality using any GA without prior knowledge of good string orderings. Thus, these results suggest that a major impediment to the use of GAs on arbitrarily difficult problems has been removed. In the original work, two challenges were outlined, challenges that needed to be overcome if messy GAs were to become a broadly applicable tool. These challenges of nonhomogeneous subproblem scale and size have been addressed successfully in this investigation.

This paper presents the results of our study of mGAs in problems with nonuniform subfunction scale and size. With these challenges largely answered, messy genetic algorithms now appear capable of solving many difficult combinatorial optimization to global optimality in polynomial time or better. In the remainder of this report, the messy GA approach is summarized, both its operation and its theory of use. Thereafter, experiments on problems of varying scale, varying building-block size, and combined varying scale and size are presented. Directions for further study and application are also considered.

2. mGAs: How are they different? What makes them tick?

The details of simple genetic algorithms are covered in standard references [1, 6, 13], and messy GAs are described more fully in Goldberg et al. [10]. Here, fundamental differences between the usual simple GA and the messy approach are highlighted, and the salient theory of messy GAs is briefly discussed.

2.1 Differences between messy GAs and simple GAs

Messy GAs are different from simple GAs in four ways:

1. mGAs use variable-length codes that may be over- or underspecified with respect to the problem being solved.
2. mGAs use simple *cut* and *splice* operators in place of fixed-length crossover operators.
3. mGAs divide the evolutionary process into two phases: a primordial phase and a juxtapositional phase.
4. mGAs use competitive templates to accentuate salient building blocks.

Messy GAs are messy because they use variable-length strings that may be under- or overspecified with respect to the problem being solved. For example, the three-bit string 111 of a simple GA might be represented in a messy GA (using LISP-like notation) as `((1 1) (2 1) (3 1))`, where each bit is identified by its name and its value. In mGAs, since variable-length strings are allowed, interpretations must be found for strings with

to mislead similarity-based methods away from global optima and toward the complement of the global optimum. Since these functions are maximally misleading, if an algorithm can solve this class of problem, it can solve anything easier. In these first studies, test functions have been constructed from sums of disjoint deceptive subfunctions. The restriction to nonoverlapping subfunctions can be partially lifted, a matter to be discussed as an extension to this work.

2.4 Competitive templates

Competitive templates are critical to the success of mGAs in problems over a fixed set of Boolean variables, because they permit the accurate evaluation of partial strings. The argument is straightforward, yet subtle. Assume that order-k building blocks are being processed, and further assume that a competitive template is available that is locally optimal to the level $k-1$. If the function is deterministic and nonstationary, then the only structures that will achieve a function value better than that of the competitive template alone are those that are building blocks at the level k. Moreover, among directly competing gene combinations, the best building block at the level k will get the best increment over the competitive template value. In this way, mGAs are able to separate the value of a bit combination from the string without prior function knowledge.

The idea of using locally optimal templates to the previous level suggests the most practical way of using mGAs. Starting at the level $k = 1$, an order-1 optimal template can be found, which in turn is used to find a level $k = 2$ template, and so on. In this way, mGAs can climb the *ladder of deception* one rung at a time, obtaining useful intermediate results at the same time the solution is being refined. It is interesting to note that this ladder-climbing analogy carries over to the computational cost of solutions with increasing k, a matter addressed in the next subsection.

2.5 mGA complexity

In the pilot study, the complexity of messy GAs was not discussed, but subsequent analysis has shown that they are polynomial on serial machines and logarithmic on parallel machines. The remainder of this subsection examines this argument in detail.

Analysis of the complexity of the basic mGA is straightforward, if we assume that function evaluations require much greater processing time than genetic operators, that is $t_f \gg t_{ga}$. Consider a problem with ℓ decision variables being optimized to order k. We recognize immediately that there are $m = \ell/k$ building blocks to be discovered. Further analysis proceeds by considering the processing in the separate phases.

During the primordial phase, the mGA starts with a population size $n = 2^k \binom{\ell}{k}$ and a function evaluation for each population member during initialization. If the function is deterministic, the initial evaluations are not

repeated in subsequent rounds of the primordial phase. Tournament selection is performed until the population contains a proportion $O(1/m)$ of each building block. With the usual logistic growth [10], and letting $P = 1/m$, it is a straightforward matter to calculate the number of generations required until the completion of the primordial phase. Starting with the logistic growth equation,

$$P = \frac{1}{1 + (n-1)2^{-t}} \tag{2.1}$$

and solving for t, we obtain the result that

$$t = \log(n-1) - \log(m-1) \tag{2.2}$$

where logarithms here and elsewhere are taken with base two. Since $m < n$ and the population size is polynomial in ℓ, we conclude that the number of generations in the primordial phase is $O(\log \ell)$. The downsizing of the population (sometimes performed during the primordial phase through cutting the population in half every so often) does not affect this computation, because the growth is logistic whether a double round is made to maintain constant population size or a single round is made, cutting the population size in half. Overall, the primordial phase has complexity of $O(\ell^k)$ on a serial machine and $O(1)$ on a parallel machine (assuming $O(\ell^k)$ processors). In either case, the number of generations of tournament selection required to dope the population with a sufficient number of the best building blocks is $O(\log \ell)$.

To analyze the juxtapositional phase, we consider the processing in two subphases: the *lengthening* subphase and the *crossing* subphase. In both subphases, we assume a constant population size $n = O(m) = O(\ell)$, where enough duplication of building blocks is permitted to allow for probabilistic variance. During lengthening, the cut probability is small, because the building blocks are short, and with splice probability near one, the processing tends to double the string length each generation. The strings are finally long enough to cover the problem no sooner than a time governed by the equation $\ell = k2^t$. Solving for t, the duration of the lengthening phase is clearly $t = O(\log(\ell/k)) = O(\log \ell)$. With a population size of $O(m) = O(\ell)$, a serial machine requires a number of function evaluations that is of order $O(\ell \log \ell)$ during lengthening, and a parallel machine requires processing of $O(\log \ell)$.

At the beginning of the crossing subphase, we assume that the optimal building blocks make up at least half the population when compared to competing substrings. Assuming continued logistic growth from this starting condition, the proportion of optimal building blocks grows as follows:

$$P = \frac{1}{1 + 2^{-t}} \tag{2.3}$$

Phase	Duration	Serial	Parallel
Primordial	$O(\log \ell)$	$O(\ell^k)$	$O(1)$
Lengthening	$O(\log \ell)$	$O(\ell \log \ell)$	$O(\log \ell)$
Crossing	$O(\log \ell)$	$O(\ell \log \ell)$	$O(\log \ell)$
Overall mGA	$O(\log \ell)$	$O(\ell^k)$	$O(\log \ell)$

Table 1: Summary of complexity estimates for an mGA.

We expect a single instance of the k-optimal string when the population contains at least one expected copy or when

$$nP^m = 1 \tag{2.4}$$

Substituting and solving for the number of generations to convergence, we obtain

$$t = -\log(n^{1/m} - 1) \tag{2.5}$$

Letting $n = (1 + \epsilon)^m$ and expanding in a power series through quadratic terms using the binomial theorem, we conclude that $\epsilon = O(m^{-1/2})$. Checking the series with the cubic term included we see that that term dominates. Continuing this process through the jth term, we conclude that the limiting case is $\epsilon = O(m^{-(j-1)/j})$, which for large j yields $\epsilon = O(m^{-1})$. Thus, the time to convergence in the crossing phase is $t = O(\log m) = O(\log \ell)$. As with the lengthening subphase, we conclude that the crossing phase is of complexity $O(\ell \log \ell)$ on a serial machine and $O(\log \ell)$ in parallel.

These complexity estimates for each of the phases are summarized in table 1. On a serial machine, an mGA requires a number of function evaluations that grows as a polynomial function of the number of decision variables, $O(\ell^k)$. It is interesting that the computation is dominated by the initialization phase. This suggests that if prior information is available regarding the function that would permit restriction of initialization to a limited number of building blocks (something less than $O(\ell \log \ell)$), then the overall serial complexity can be reduced to a svelte $O(\ell \log \ell)$. On a parallel machine with enough processors, initialization can be done in constant time, as can the generational function evaluations during lengthening and crossing. Thus, on a large enough parallel machine, the mGA requires computations that grow only as fast as a logarithmic function of the number of decision variables. These estimates are exciting and bode well for the future of messy GAs in combinatorial function optimization, especially when considered in the light of the following conjecture.

2.6 A conjecture: mGAs find the best solution at a given level

That mGAs converge in polynomial time or better is important, but polynomial convergence is no virtue if that convergence is incorrect. Empirically,

mGAs have always found globally optimal results in problems of bounded deception, leading us to the following conjecture:

Conjecture 1. *With probability that can be made arbitrarily close to one, messy GAs converge to a solution at least as good as the truncated order-k solution, where the truncated order-k solution is defined as the optimum of the function defined by setting all Walsh coefficients of the original function at order $k+1$ or above to zero. Moreover, this convergence occurs in a time that is $O(\ell^k)$ on a serial machine and $O(\log \ell)$ on a parallel machine.*

The plausibility of this conjecture can be seen quite readily. During the primordial phase, the best building blocks grow logistically as long as the fitness signal is reliable (and as long as apples are compared to apples, but we will have more to say about this in a moment when we discuss the need for thresholding or genic selective crowding). During the lengthening portion of the juxtapositional phase, the best building blocks will hold their own on average, because reproduction will continue to increase their number at a rate near doubling, and splicing must continue to express a building block no less than half of the time (because half the time a currently expressed building block will be placed at the left end of the string, guaranteeing continued expression under the first-come-first-served rule). Thereafter during the crossing portion of the juxtapositional phase, the mGA behaves very much like a simple GA with very tight building blocks, and continued convergence proceeds according to an inequality that looks very much like the standard schema theorem.

Although the conjecture is reasonable, taking it to theoremhood is nontrivial as it is insufficient to deal with the trajectory of the population in expectation. It is also difficulty to include in detail the thresholding and tie-breaking mechanisms to be discussed in the following sections. Nonetheless, the outline gives more than a hint of the convergence mechanism underlying mGAs and provides some explanation of the remarkable empirical results observed to date.

3. Nonuniform scaling and genic selective crowding

It is clear that the test function considered in the pilot study [10] is quite difficult. Each of the 10 subfunctions has two local optima, yielding a total of $2^{10} = 1024$ optima, 1023 of them false. On the other hand, the function seems like something of a special case. After all, 10 copies of the same function were added together, thereby prohibiting any study of what happens to mGA convergence when either the *scale* of the subfunction varies or when building block *size* is different. In this section, we extend the mGA to permit the solution of problems with varying subfunction scale using a technique called *genic selective crowding* or *thresholding*. Simply stated, this technique restricts the selection procedure by requiring competitions to be held between only those individuals that have a better-than-random number of genes in

common. In this way, only apples are compared to apples, and only relevant differences in scale are used to distinguish between different building blocks.

In the remainder of this section, the qualitative theory of genic selective crowding, a mathematical analysis of two of its important parameters, and some simulation results are presented to lend support to the use of this technique. A later section will present results using this method in combination with tie breaking on functions with mixed scale and building block size.

3.1 Theory of genic selective crowding

In the pilot study, tournament selection was used during the primordial phase by repeatedly drawing two strings chosen at random (without replacement) from the population and selecting the better string. In the problem considered in that study, that procedure worked well, because all subfunctions had identical scaling. In general, however, this procedure is flawed, because it permits substrings to be compared to one another regardless of whether they are referring to the same subfunction — regardless of whether they contain any genes in common. The pilot study recognized this challenge, and suggested a technique called *genic selective crowding* (or *thresholding*) to overcome the difficulty.

The idea of genic selective crowding is straightforward. Tournaments are held as usual, except that individuals are forced to compete with those individuals who have at least some *threshold* number of genes in common with them. In this way, a pressure is maintained for like to compete with like, helping to insure that the comparison is a meaningful one. The mechanism is not unlike that of a number of *niching* procedures in common use [1, 3, 4, 11, 13], except that allele values are not compared; only the presence of genes in common is checked.

In practice, the actual algorithm works as follows. The first candidate for selection is picked uniformly at random without replacement from a candidate permutation list that originally includes all population members in randomly generated order. The second candidate is chosen by checking the next *shuffle number*, n_{sh}, candidates in the permutation list one at a time until one is found that has at least *threshold*, θ, genes in common with the first candidate. If a candidate is found, the tournament is held in the normal manner with the better individual being selected for subsequent genetic processing. If no second candidate is found that meets the criterion in n_{sh} tries, the first candidate is chosen for subsequent processing.

The two parameters of genic selective crowding, the *threshold value* and *shuffle number*, play an important role in properly implementing the thresholding mechanism in messy GAs. It is important to choose a threshold value that discriminates between the chance occurrence of genes in common and the likelihood that such commonality is statistically significant. Intuitively, it seems reasonable to expect that the threshold will have to increase in a messy GA as the strings get longer, and an analysis of appropriate threshold values will show exactly that. It is also important to choose a shuffle number

such that there is a better than random chance of choosing at least one individual with the threshold number of bits in common. This parameter should vary as the string length, but without some further thought, it is unclear exactly how. Simplified analyses that guide reasonable choices for threshold and shuffle number are presented in the next subsection.

3.2 Reasonable values for threshold and shuffle number

Simplified analyses of threshold and shuffle number values are presented herein. Under the assumption of a randomly generated population of candidates, we require that the threshold value θ be set higher than its expected value for a given current string length and that the shuffle number n_{sh} be set so as to expect at least one occurrence of a second candidate with θ genes in common with the first candidate. Although the algebra is somewhat involved, the analysis leaves us with a straightforward procedure that appears to give good results.

Assuming an ℓ-bit problem, it is clear that an mGA may have raw strings of length λ less than or greater than ℓ. After decoding, however, the *processed length* of a string (the number of different bits mentioned in the string) must be less than or equal to the length of the problem ℓ. In the remainder of this subsection, the lengths discussed are all processed lengths. In a given threshold comparison, we consider the possibility of having different processed lengths λ_1 and λ_2. Clearly, there are $\binom{\ell}{\lambda_1} \cdot \binom{\ell}{\lambda_2}$ possible combinations of two strings of length λ_1 and λ_2. The number of combinations of two strings having x common genes between them may be estimated by fixing x positions in both the strings and calculating the number of possible combinations to place the rest $(\ell - x)$ genes in the remaining positions of both strings.

The number of combinations of two strings containing x common genes is equal to the number of combinations of choosing x positions from ℓ possible choices, times the number of combinations which allocate the $(\lambda_1 - x)$ remaining positions in the first string from the $(\ell - x)$ remaining choices, times the number of combinations which allocate the $(\lambda_2 - x)$ remaining positions in the second string from $(\ell - \lambda_1)$ remaining choices or symbolically

$$\binom{\ell}{x}\binom{\ell - x}{\lambda_1 - x}\binom{\ell - \lambda_1}{\lambda_2 - x}$$

Therefore, the probability that two strings contain exactly x common genes is given by the equation

$$p(\ell, \lambda_1, \lambda_2, x) = \frac{\binom{\ell}{x}\binom{\ell - x}{\lambda_1 - x}\binom{\ell - \lambda_1}{\lambda_2 - x}}{\binom{\ell}{\lambda_1}\binom{\ell}{\lambda_2}} \tag{3.1}$$

There are limits on the number of common genes between two strings, x that will satisfy the above equation. A little computation shows that the minimum and maximum limits on x are

$$\max(0, \lambda_1 + \lambda_2 - \ell)$$

and

$$\min(\lambda_1, \lambda_2)$$

respectively. It can be proved easily that

$$\binom{\ell}{x}\binom{\ell - x}{\lambda_1 - x} = \binom{\ell}{\lambda_1}\binom{\lambda_1}{x} \tag{3.2}$$

Substituting equation 3.2 for the first two terms in the numerator of equation 3.1 yields the point probability function for a hypergeometric distribution [5]:

$$p(\ell, \lambda_1, \lambda_2, x) = \frac{\binom{\lambda_1}{x}\binom{\ell - \lambda_1}{\lambda_2 - x}}{\binom{\ell}{\lambda_2}} \tag{3.3}$$

The limits on x mentioned above agree with that in a hypergeometric distribution. Therefore, the probability that there are a certain number of common genes between two random strings is hypergeometric. To calculate the threshold value, we compute the expected number of bits in common by summing over all possible values:

$$E[x] = \sum_{x=\max(0,\lambda_1+\lambda_2-\ell)}^{\min(\lambda_1,\lambda_2)} x \cdot p(\ell, \lambda_1, \lambda_2, x) \tag{3.4}$$

After a good bit of manipulation it is found that the expected number of bits in common is given by the simple equation, $E[x] = \frac{\lambda_1\lambda_2}{\ell}$. In the program, we simply require that a threshold be used that is at least equal to the nearest integer greater than the calculated $E[x]$ value. Therefore, the threshold value is taken as

$$\theta(\ell, \lambda_1, \lambda_2) = \lceil \tfrac{\lambda_1\lambda_2}{\ell} \rceil \tag{3.5}$$

where the operator $\lceil\ \rceil$ denotes a *ceiling operator* that calculates the nearest integer greater than the operand.

With a suitable choice of threshold, we turn to calculating a reasonable value for the shuffle number. We would like to choose a shuffle number that ensures a reasonable probability of selecting a second candidate that has at least θ genes in common with the first candidate. Calculating the

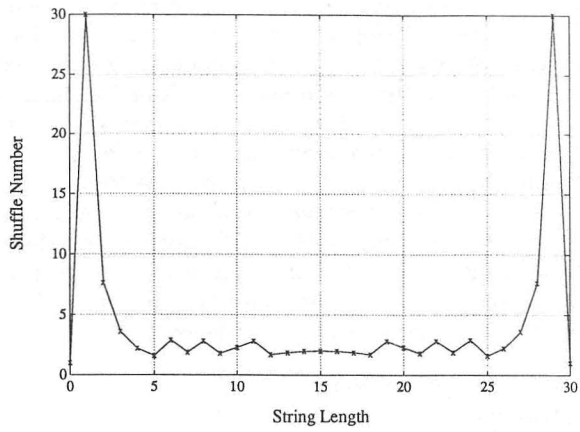

Figure 1: The shuffle number versus string length ($\lambda_1 = \lambda_2$ assumed).

cumulative probability distribution of having at least θ genes in common is a straightforward exercise:

$$P(\ell, \lambda_1, \lambda_2, \theta) = \sum_{k=\theta}^{\min(\lambda_1, \lambda_2)} p(\ell, \lambda_1, \lambda_2, k) \tag{3.6}$$

Setting the expected number of matched copies in the shuffle subpopulation to one and solving for the shuffle number yields the following:

$$n_{sh}(\ell, \lambda_1, \lambda_2, \theta) = \frac{1}{P(\ell, \lambda_1, \lambda_2, \theta)} \tag{3.7}$$

Assuming strings of equal length, $\lambda_1 = \lambda_2$, the value of shuffle number is shown as a function of string length for a 30-bit problem in figure 1. It may be shown using the usual normal approximation to the hypergeometric distribution that the lowest probabilities of occurrence of θ matches between randomly chosen strings occurs when both strings are very short or very long. Substituting appropriate length and threshold values into the probability distribution yields the equation

$$n_{sh}(\ell, \lambda_1, \lambda_2, \theta) = \ell \tag{3.8}$$

which is used regardless of string length as a reasonable bound on the necessary shuffle number.

3.3 Computational experiments with and without thresholding

The thresholding mechanism described in the previous subsections has been implemented in mGA code developed for execution on a TI Explorer. Several test functions having subfunctions with unequal scaling have been used to examine the effect of thresholding on messy GAs. Scaled versions of the original 30-bit test function are formed by multiplying each of the subfunctions by a scale factor, thereby producing an unequal selection pressure on each subfunction, making it more difficult for messy GAs to solve the problem to global optimality without a mechanism such as selective genic crowding. A performance comparison of messy GAs with and without thresholding is made by applying them on these functions. In each experiment, five simulations are performed and the average values are presented. The basic GA parameters used in all simulation runs are as follows:

number of generations	= 30;
probability of cut	= 1/60;
probability of splice	= 1.0;
probability of mutation	= 0.0.

In the following, the test functions and their corresponding simulation results are presented.

3.3.1 Test function 1: Nine up, one down

In this test function, the first subfunction is scaled down by a factor of three and the other subfunctions (two through ten) are scaled up by a factor of seven. This introduces an adverse selection pressure against the first subfunction. To make matters worse, an extra 1000 copies of each correct three-bit building block for subfunctions two through ten are added to the initial population, making a total of 9000 additional strings. This perturbation is performed in an attempt to overwhelm the poorly scaled building block. Adverse scaling together with an adverse initial proportion produces a stiff challenge for the mGA to maintain enough copies of the correct building block for the first subfunction in the population. The other GA parameters used in the simulations follow:

population size	= 41480 reduced to 2592;
string length	= 3;
number of generations	= 30.

Simulations with and without thresholding are performed, and the average of five simulations is plotted. The thresholding parameters θ and shuffle number are adopted according to the theory presented in the previous subsection, using a threshold of $\lceil \frac{\lambda_1 \lambda_2}{\ell} \rceil$ and a shuffle number of 30. Figure 2 compares the maximum number of subfunctions correct versus generation for mGAs with and without thresholding. The first subfunction and the rest of the subfunctions are plotted separately to show the convergence of

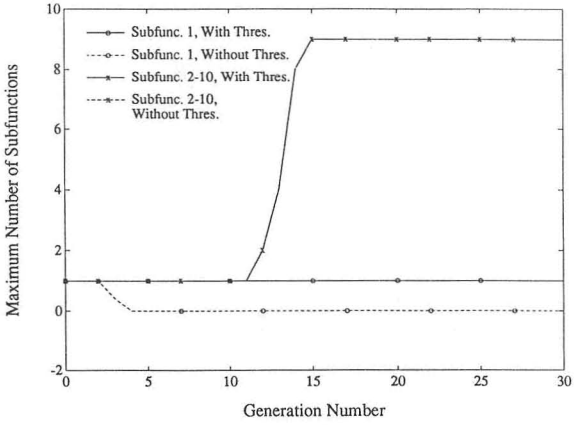

Figure 2: Maximum number of subfunctions correct in test function 1 with nine subfunctions scaled up and one scaled down. The average of five runs is shown. Without thresholding, the mGA is unable to get all ten subfunctions correctly. With thresholding, the mGA correctly finds global optima reliably.

the algorithm in each category. The mGA without thresholding is unable to maintain correct strings corresponding to the first subfunction throughout the primordial phase, whereas the mGA with thresholding maintains enough copies of the correct string corresponding to the first subfunction in successive generations to solve the problem to global optimality. Figure 3 shows the average number of subfunctions in the population versus generation number. It is clear from the figure that the mGA without thresholding loses the correct building block corresponding to the first subfunction, while the mGA with thresholding maintains all ten correct building blocks, solving the problem to global optimality quite easily.

3.3.2 Test function 2: One up, nine down

In the second test function, we investigate the reverse situation from that of the first. Here, subfunction one is scaled up by a factor of seven and the remaining nine subfunctions are scaled down by a factor three. As an added perturbation, an extra 9000 copies of the best scaled-up building block (subfunction one) are added to the initial population to try to overwhelm the poorly scaled building blocks. This test function and initial condition combination provide a stiff challenge to mGA convergence. All GA parameters are the same as those in the simulations for test function 1.

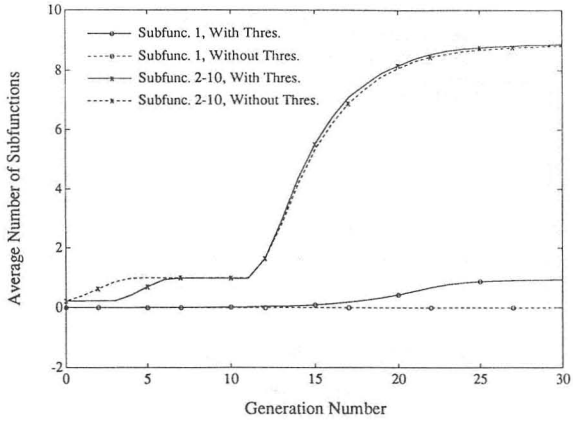

Figure 3: Average number of subfunctions correct in test function 1 with nine subfunctions scaled up and one scaled down.

Figure 4 shows the maximum number of subfunctions correct at each generation for messy GAs with and without thresholding. The graph shows that the mGA without thresholding loses the correct building blocks corresponding to the poorly scaled subfunctions and is only able to get a single subfunction answered correctly. On the other hand, the mGA with thresholding is able to maintain and recombine the correct building blocks to all subfunctions and solve the problem to global optimality. Figure 5 graphs the population average number of correct building blocks versus generation. The figure once again confirms the role of thresholding in successfully classifying the tournaments, thereby allowing only comparable building blocks to compete with one another.

3.3.3 Test function 3: A linear scaling

Having tested the extremes of behavior, test function 3 considers a linear scaling of the 10, three-bit subfunctions, starting from a factor of 10 for subfunction one and going up to 100 for subfunction ten with an increment of 10 between each subfunction. No extra copies are added here. An initial population size equal to 32,480 is used and the population is reduced to 2030 at the end of the primordial phase. The other GA parameters including thresholding parameters used in the simulations are the same as those used in the previous test functions. Messy GAs with and without thresholding are applied to this function and the maximum and average objective function values are compared in figure 6. The plot shows that thresholding permits the mGA to maintain and recombine all ten subfunctions in the population.

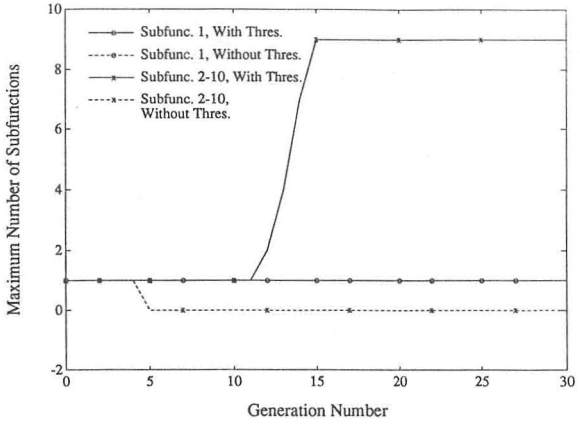

Figure 4: Maximum number of subfunctions correct in test function 2 with one subfunction scaled up and nine functions scaled down. The mGA without thresholding is only able to get the upscaled subfunction correct, while the mGA with thresholding maintains and recombines all subfunctions to obtain the global optimum.

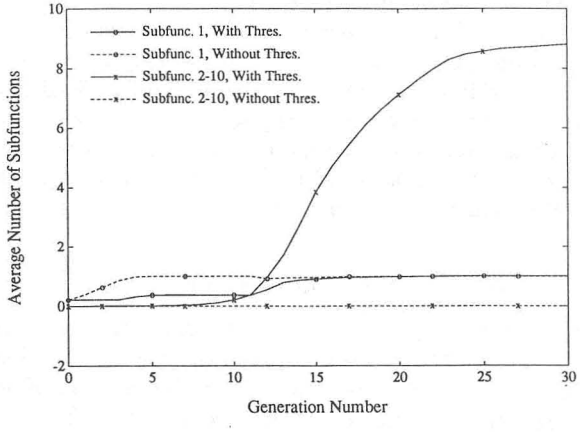

Figure 5: Average number of subfunctions correct in test function 2.

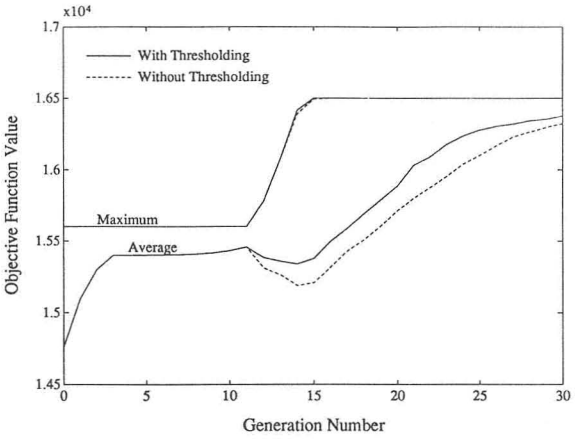

Figure 6: Maximum and average value of the objective function versus generation number on the linearly scaled test function 3. Both mGAs with and without thresholding are able to find the global optimum, but the mGA with thresholding finds the optimum more quickly and maintains higher average performance across the population.

The mGA with thresholding is also able to maintain a higher value of average fitness across the population than the mGA without thresholding. Also note that the mGA with thresholding finds globally optimal structures more quickly than the mGA without thresholding; the mGA with thresholding finds its first optimal solution at generation 15, whereas the mGA without thresholding takes one generation longer. It is actually interesting that the mGA without thresholding can solve the problem at all. Clearly, the thresholding is useful here, but the pressure applied by thresholding to maintain separate competitions appears to be more important in situations that become greatly perturbed from an ideal mix of optimal building blocks. This speaks well for the robustness of the procedure.

4. Nonuniform size, tie breaking, and null bits

In the previous section, a genic selective crowding (thresholding) mechanism successfully addressed the problem of nonuniform subfunction scale. In this section, we introduce a method that tackles the problem of nonuniform building block size. In the remainder of the section, the difficulty is further explained and the method of null bits with tie breaking is introduced. Thereafter, simulations are performed to demonstrate the efficacy of the method. Problems with combined nonuniform subfunction scale and size are also attacked using a combination of null bits, tie breaking, and thresholding.

4.1 Theory of parasitic bits, tie breaking, and null bits

It is fairly easy to understand why, if steps are not taken to enforce meaningful competitions, nonuniform scaling of different subfunctions can cause an mGA difficulty; however, it is not immediately obvious why nonuniform building block size should cause an mGA any difficulty. The problem can best be understood if we imagine a discrepancy between *function* and *algorithm* building block size. Suppose an mGA is processing at the level $k = 4$, but suppose that the longest building blocks in the function are of length 3. Such a mismatch between algorithmic and functional building blocks can cause difficulty depending on the bits that fill in the leftover positions in the shortest substrings. For example, consider the two strings `A = ((1 1) (2 1) (3 1) (8 1))` and `B = ((1 1) (2 1) (3 1) (8 0))`. Referring to the previous 30-bit problem, both contain optimal building blocks `((1 1) (2 1) (3 1))`, but they differ as to how to fill in the blank. Thinking about the desirable outcome, we would rather see string `A` selected and `B` eliminated, because the fill-in bit `(8 0)` in string `B` may prevent the expression of the optimal bit combination over positions 7-8-9, whereas the fill-in bit `(8 1)` in string `A` agrees with the correct solution and would not inhibit its expression. Yet, if nothing is done, string `B` will most often be selected in this deceptive problem because a lone zero will, on average, have higher fitness than a lone one. In some sense, such bits that ride along on a chromosome are parasites, because they agree with locally optimal solutions, but do nothing to improve the solution further. Later on, these same *parasitic* bits inhibit expression of correct bit combinations, and they must be selected against, if we are to have some hope of solving problems with differing building block size.

The pilot study suggested a mechanism to deal with this knotty problem. Specifically the inclusion of *null* or placeholder bits and the use of a tie-breaking procedure were recommended. The idea is straightforward. A number of null bits are introduced as placeholders to fill in leftover positions. Then during a tournament, if the fitness of two strings is the same, the string with the shorter *effective length* (the one with the greater number of null bits) is selected. Returning to the example given above, the addition of null bits and tie breaking fixes the problem with parasitic bits completely. Consider the modified string `A' = ((1 1) (2 1) (3 1) (8 N))` with a null bit as a placeholder for gene eight. When the objective function is sampled, this string will have the same function value as `B = ((1 1) (2 1) (3 1) (8 0))` (assuming an all-zero competitive template), but string `A'` will be preferred, because it has the shorter effective length.

In general, the number of null bits that must be added to a problem is the difference between the size of the largest and smallest subfunctions in the problem. Since this information is not usually known beforehand, a total of $k-1$ null bits should be added to the solution, thereby bounding all possible building block lengths. Another way to perform essentially the same operation is to initialize the problem with building blocks of all sizes up to k and break ties on the basis of shorter actual length. Either mechanism is reason-

able, and the biological plausibility of them both has been argued [10] on the grounds of a preference for most energy- or mass-efficient representation.

4.2 Monkey wrench runs

In the first size experiments, we don't actually use a problem with different building block size. Instead, we return to the original 30-bit problem of the pilot study with its ten, three-bit deceptive subfunctions. To simulate the effect of a mismatch between algorithm building block size and problem building block size, 9000 copies of four-bit parasitic strings (we call these the four-bit *monkey wrenches*, because they are added intentionally to gum up the works) are introduced on top of the normal population of 32,480, three-bit strings. The added parasitic strings contain an optimal substring (`1 1 1`) plus a spurious fourth bit having a value `0` at the first position of one of the other nine subfunctions. A simple count shows that there are a total of $9 \cdot 10 = 90$ such variations and a 100 copies of each variant are included. Five independent simulations are performed using the following GA parameters:

initial population size	= 41,480 down to 2592
probability of cut	= 1/60
probability of splice	= 1.0
probability of mutation	= 0.0

In figure 7, the maximum and average number of optimal subfunctions versus generation is shown for mGAs with and without tie breaking. The figure shows that with tie breaking, an optimal solution is found as soon as the strings are long enough to specify the solution, whereas without tie breaking the occurrence of an optimal solution is considerably delayed and unstable. The maximum and average function value of the strings versus generation are shown in figure 8.

It is interesting that the mGA without tie breaking is able to solve this problem to global optimality, albeit more slowly than the mGA with those features, even though the monkey wrenches were present to disturb the solution process. Apparently, cut and splice were sufficient to excise the bad bits that went along for the ride. In general, however, multiple parasitic bits can tag along, and to simulate this possibility a six-bit monkey wrench is devised that takes a single optimal building block and adds one zero each from three of the remaining nine subfunctions. There are $\binom{9}{3} = 84$ such six-bit monkey wrenches per function. One copy of each is included per subfunction for a total $10 \cdot 84 = 840$ substrings appended to the original initial population of 32,480, three-bit strings yielding a total of 33,320 strings in the initial population. During the primordial phase, this population is reduced to size 2082 with successive population halving; the size is held constant throughout the juxtapositional phase as per usual. Otherwise, the messy GA parameters used in these simulations are the same as those used in the previous experiment.

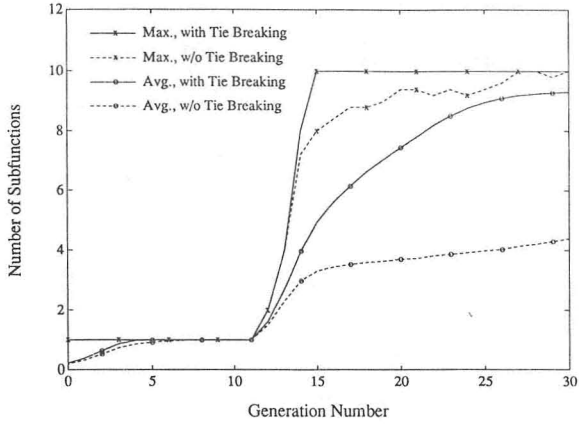

Figure 7: Average and maximum number of optimal subfunctions versus generation in the 30-bit problem with 4-bit monkey wrenches. Without null bits the discovery of the correct solution is considerably delayed and not entirely stable. With tie breaking, the mGA finds the globally optimal solution in the first generation the strings are long enough to cover the problem.

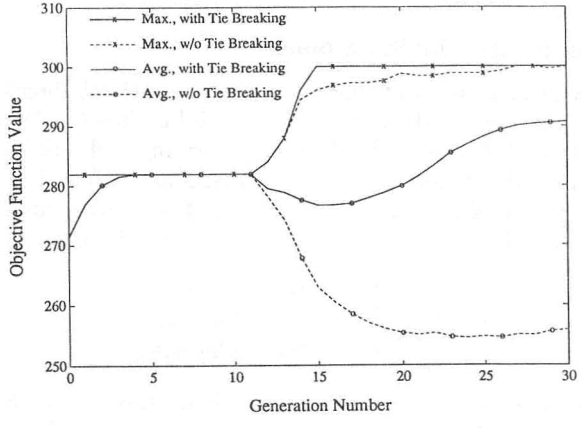

Figure 8: Average and maximum function value versus generation in the 30-bit problem with four-bit monkey wrenches.

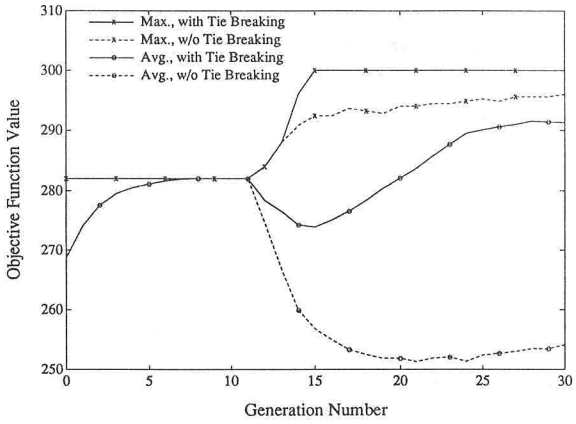

Figure 9: Average and maximum function value of the population versus generation in a problem with six-bit monkey wrenches.

The population maximum and average function values are graphed in figure 9. The tie-breaking algorithm allows messy GAs to converge to the optimal solution by maintaining all low-order building blocks in the population, whereas the mGA without tie breaking cannot solve the problem to global optimality; the presence of multiple parasitic bits is simply too disruptive.

4.3 Differently sized building blocks

The monkey wrench experiments give us confidence in the tie-breaking procedure. Here we actually construct a problem with building blocks of differing sizes and try the mGA with and without tie breaking and null bits.

Specifically, a 31-bit function with one three-bit subfunction and seven 4-bit subfunctions is designed. The three- and four-bit subfunctions use Liepins's construct [14] for a fully deceptive function of order k:

$$f(d,k) = \begin{cases} 1 - \frac{1}{2k}, & \text{if d} = 0; \\ 1, & \text{if } d = k; \\ 1 - \frac{d+1}{k}, & \text{otherwise.} \end{cases}$$

Here d is the number of ones in the substring. Thus, the function has a global optimum at $1\ldots1$, a local optimum at $0\ldots0$, and a value that declines as the function's argument gets more distant (in the sense of Hamming) from all zeroes.

Because the disparity between building block sizes is so small, only a single null bit, `(32 N)`, is required and used. GA parameters identical to

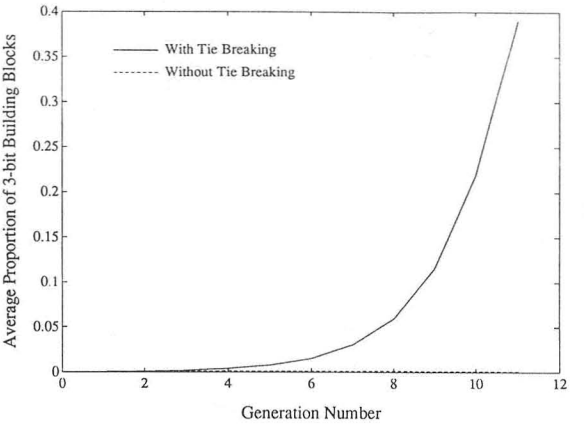

Figure 10: Average proportion of strings having a three-bit optimal building block and a null bit (only the primordial phase is shown).

the previous experiments are used, except that the population starts at size $2^4\binom{32}{4} = 575{,}360$ and is ramped down to 2247 with successive population halving. Average quantities from three independent simulations are presented. Figure 10 shows the average proportion of the three-bit optimal strings during the primordial phase. The mGA without tie breaking and null bits is unable to grow a substantial portion of the three-bit building blocks. As a result, it is also unable to optimize the function to global optimality as is seen in figure 11. By contrast, the mGA with null bits and tie breaking is able to solve the problem to global optimality quite quickly.

4.4 Nonuniform scaling and nonuniform building block size

Thus far, we have treated the problems of nonuniform scale and size as though they always occur in separate objective functions. Of course, it is likely that a function will have both nonuniformly scaled and sized subfunctions in it. In this subsection, a test function is constructed with both difficulties, and mGAs with genic selective crowding alone, tie breaking with null bits alone, and both features together are tried and compared.

A 36-bit test function with differently scaled and sized building blocks is constructed. In the problem, three subfunctions are three bits long, three other subfunctions are four bits long, and the remaining three subfunctions are five bits long. Liepins's construct is used here for all three sizes, and each function is scaled by twice its order squared, $2k^2$. Thus, the three-bit function is multiplied by 18, the four-bit function by 32, and the five-bit function by 50.

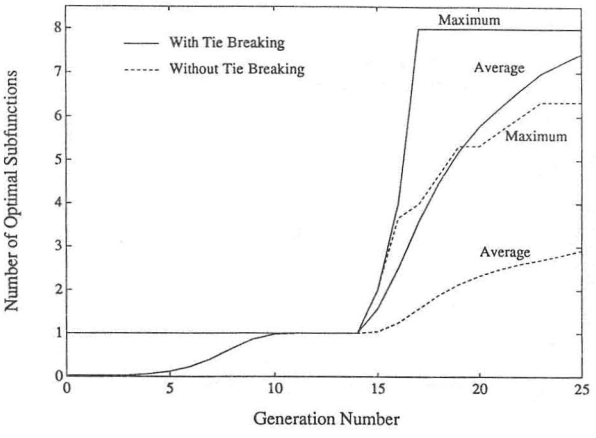

Figure 11: Average and maximum number of optimal building blocks versus generation in a problem with one three-bit subfunction and seven four-bit subfunctions.

Two null bits are required for these runs, and this dictates that an initial population required for this run be quite sizeable: $n = 2^5 \binom{38}{5} = 16{,}062{,}144$. It should be remembered, however, that this is a small portion of the search space, which is itself of size $2^{36} = 6.87(10^{10})$. Because of the large size of population required initially, the primordial selection is performed in subpopulations, and the best strings are brought forward to be considered for further selection. During primordial selection, the population size is successively reduced by half until it reaches the chosen size for juxtapositional selection, $n = 200$.

Figure 12 shows the maximum number of subfunctions correct versus generation with thresholding alone, tie breaking alone, and their combination. The results are averaged over three simulations. The figure shows that thresholding and tie breaking alone cannot solve the problem to global optimality, whereas their combination is able to maintain all subfunctions in the solution. Figure 13 shows the population maximum and average fitness in successive generations. Even though there were many copies of all order-3 building blocks in the initial population due to null-bit duplicates, neither the mGA with tie breaking alone nor the mGA with thresholding alone could maintain enough copies of them at the end of the primordial phase. On the other hand, when both thresholding and tie-breaking methods were applied, thresholding maintained a uniform selection pressure for different subfunctions, and tie breaking maintained a selection pressure for the individual building blocks with null bits. The combined action of these methods permitted the global optimum to be found repeatedly.

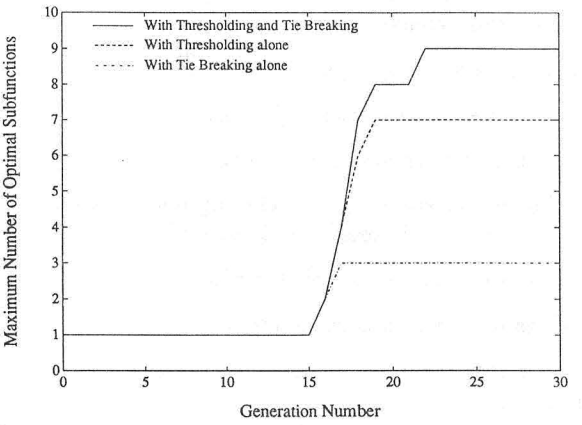

Figure 12: Maximum number of optimal subfunctions correct versus generation in a mixed size-scale problem.

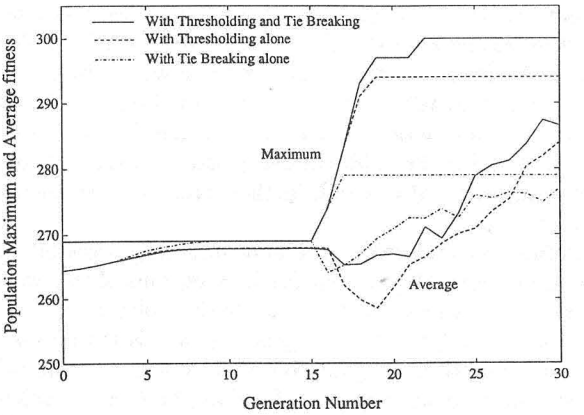

Figure 13: Population maximum and average fitness versus generation for a mixed size-scale problem.

5. Continuations and extensions

The initial investigation of mGAs and this study lead to a number of interesting continuations and extensions:

1. Analyze and solve overlapping subfunctions.
2. Prove the fundamental conjecture of mGAs.
3. Implement a parallel version of an mGA.
4. Develop other messy code types, including permutation codes, messy floating point codes, and messy classifiers (rules).
5. Extend mGAs to nondeterministic functions.
6. Extend mGAs to nonstationary functions.

In the remainder of this section, each of these possibilities is considered in somewhat more detail.

In both the pilot study and the current investigation, only nonoverlapping subfunctions were considered. This is a reasonable assumption to launch a new technique, but the question arises whether functions can be deceptive at higher levels because of a large number of low-order interactions. For example, many problems can be described with linear and quadratic interaction between Boolean decision variables, but does this imply that such functions are no more than order-2 deceptive? The answer to this and related questions lies in a careful application of the Walsh theory described elsewhere [7, 8]. Working in reverse, nondeceptive interactions can clearly be added to what otherwise would be nonoverlapping deceptive subfunctions without affecting the mGA's ability to solve the augmented problem, and it is reasonable to expect that further analytical extensions of the class of mGA-solvable problems are possible. Work in this area is important if we are to understand the full class of problems that mGAs can solve to global optimality. Regardless of these fine points, as a practical matter, mGAs may be used in the manner suggested earlier, climbing the ladder of deception one bit at a time. At some point, it becomes impractical to climb further, and the best answer so far is adopted and used.

Proving the fundamental conjecture and obtaining probabilistic bounds on the method are important, yet nontrivial, extensions of this work. On the one hand, because mGAs process strings in distinct phases, it may be easier to perform a rigorous analysis of convergence than it is with a homogeneous simple GA, where everything is going on all at once. On the other hand, these systems still have many degrees of freedom, and simple calculations in expectation (calculations like the schema theorem) are not enough to prove the conjecture. Progress here will depend on a two-pronged attack. Instead of analyzing mGA mechanics directly, simplified versions should be attacked that retain the essence of the algorithm. More sophisticated analytical horsepower will also be required, including some fairly sophisticated tools from the theory of stochastic processes.

A parallel version of an mGA should not be difficult to implement, and as it has been suggested, the logarithmic convergence guarantees of a parallel version are very attractive. A parallel version is easy to implement, because all genetic and selective processing requires only pairwise interaction. Local tournaments may be held in the neighborhood of a given processor, and mating and recombination can also be held locally.

The pilot study and this investigation have concentrated on solving problems that map from a fixed number of Boolean variables into the reals. Of course, the messy philosophy of mGAs can be extended to many different classes of problems through many different types of codes. The pilot study suggested a specific floating-point code and ongoing studies are considering it and a number of variants. Messy permutations were also suggested, and this should be a particularly fruitful avenue of research in scheduling and resource allocation problems. Another method of tackling problems over permutations is to map them to binary strings, using the Boolean satisfiability techniques suggested by De Jong and Spears [2]. This indirect approach may be fruitful in that it exploits the solid convergence of binary mGAs and the simplicity of a reasonable penalty-like method.

Messy classifiers were also suggested in the pilot study. The idea is straightforward. There is little need to carry along don't-care positions explicitly, and only information-carrying positions need to be mentioned. Moreover, messy classifiers provide a natural means of resolving the grand debate between the Michigan and Pitt approaches [6, 16]. Because mGAs can recombine strings of arbitrary length containing an arbitrary number of rules, there is no need to decide beforehand whether a single rule or a group of rules is the appropriate unit of reward. Suitable punctuation marks could be used to define corporate boundaries and rule clusters could merge or spin off subsidiaries within the normal mGA framework.

The method of competitive templates essentially removes the "deterministic noise" that exists in simple GAs because of the simultaneous variation of multiple building blocks. There are problems, however, where real noise is present, and mGAs can be extended to permit their solution. There are basically two approaches to follow. In one method, building blocks can be duplicated enough times in the initial population to ensure that their average evaluation is sufficiently accurate. In the other technique, individual copies of building blocks may be evaluated repeatedly, taking a moving average or other estimate of their function value. Either way, it should be possible to perform calculations of the duplication or repetition required to reduce the error within reasonable bounds.

It should also be possible to use mGAs on nonstationary problems. Elsewhere [6, 12, 13, 15] techniques of dominance and diploidy have been suggested and applied in nonstationary problems, and these certainly could be adapted to mGA practice. There is another possibility, however. We have already seen how the decoding of an mGA string involves a form of gene expression that is something like an intrachromosomal dominance mechanism. Why not introduce explicit dominant and recessive markers within the mGA

together with a dominance shift mechanism, thereby permitting allele combinations (building blocks) to be alternately remembered and held in abeyance as time goes on? Doing this in an mGA context has the advantage of being able to recall or store appropriately sized (and tested) building blocks, whereas it is unclear in a simple GA how to get beyond independent storage or retrieval of more than single alleles.

Although there are many fruitful avenues for continued research, it should be pointed out that messy GAs are ready for real-world application today. Their combination of polynomial or better efficiency and apparent global convergence seems difficult to beat in many blind combinatorial optimization problems.

6. Conclusions

This paper has discussed the salient features and theory of messy genetic algorithms, and it has presented the results from an investigation of techniques that permit mGAs to be applied to problems of varying subfunction scale and size.

Although more basic work is needed, mGAs are ready for real-world applications, because they work, because they are efficient, and because they are practical. The pilot study and this investigation have laid the foundation for mGAs, demonstrating that mGAs can converge to globally optimal results in the worst kind of problem, so-called deceptive functions. Because mGAs can converge in these worst-case problems, it is believed that they will find global optima in all other problems with bounded deception. Moreover, mGAs are structured to converge in computational time that grows only as a polynomial function of the number of decision variables on a serial machine and as a logarithmic function of the number of decision variables on a parallel machine. Finally, mGAs are a practical tool that can be used to climb a function's ladder of deception, providing useful and relatively inexpensive intermediate results along the way.

For these reasons and because of their potential benefit in so many areas, we recommend the immediate application of mGAs to difficult, combinatorial problems of practical import. Although several i's remain to be dotted and a number of t's are still there for the crossing, we believe that this technique will become an important weapon in the analyst or designer's arsenal to combat nontrivial blind combinatorial problems in a rational, efficient manner. Moreover, the apparent efficiency and convergence of such an inductive and speculative process gives new hope that one day a rigorous computational theory of innovation and design can be developed. While not detracting from the designer's art, such a theory would provide rigorous underpinnings in an area where jingoism has too long substituted for careful analysis.

Acknowledgments

This material is based upon work supported by Subcontract No. 045 of Research Activity AI.12 under the auspices of the Research Institute for Computing and Information Systems (RICIS) at the University of Houston, Clearlake, under NASA Cooperative Agreement NCC9-16 and by the National Science Foundation under Grant CTS-8451610. The authors acknowledge computers and software provided by the Digital Equipment Corporation and Texas Instruments Incorporated. Mr. Deb also acknowledges research support under a University of Alabama Graduate Council Fellowship.

References

[1] K.A. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, (Doctoral dissertation, University of Michigan, 1975). *Dissertation Abstracts International, 36*(10), 5140B. (University Microfilms No. 76-9381)

[2] K.A. De Jong and W.M. Spears, "Using genetic algorithms to solve NP-complete problems," *Proceedings of the Third International Conference on Genetic Algorithms* (1989) 124–132.

[3] K. Deb, *Genetic Algorithms in Multimodal Function Optimization*, Master's thesis and TCGA Report 89002 (The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa, 1989).

[4] K. Deb and D.E. Goldberg,"An investigation of niche and species formation in genetic function optimization," *Proceedings of the Third International Conference on Genetic Algorithms* (1989) 42–50.

[5] W. Feller, *An Introduction to Probability Theory and its Application* (Wiley, New York, 1968).

[6] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addison-Wesley, Reading, MA, 1989).

[7] D.E. Goldberg, "Genetic algorithms and Walsh functions: Part I, a gentle introduction," *Complex Systems*, **3** (1989) 129–152.

[8] D.E. Goldberg, "Genetic algorithms and Walsh functions: Part II, deception and its analysis," *Complex Systems*, **3** (1989) 153–171.

[9] D.E. Goldberg and C.L. Bridges, "An analysis of a reordering operator on a GA-hard problem," *Biological Cybernetics*, **62(5)** (1990) 397–405.

[10] D.E. Goldberg, B. Korb, and K. Deb, "Messy genetic algorithms: Motivation, analysis, and first results," *Complex Systems*, **3** (1989) 493–530.

[11] D.E. Goldberg and J. Richardson, J. "Genetic algorithms with sharing for multimodal function optimization," *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (1987) 41–49.

[12] D.E. Goldberg and R.E. Smith, "Nonstationary function optimization using genetic algorithms with dominance and diploidy," *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (1987) 59–68.

[13] J.H. Holland, *Adaptation in Natural and Artificial Systems* (University of Michigan Press, Ann Arbor, 1975).

[14] G.E. Liepins and M.D. Vose, *Representational Issues in Genetic Optimization*. Manuscript submitted for publication, 1989.

[15] R.E. Smith, *An Investigation of Diploid Genetic Algorithms for Adaptive Search of Non-stationary Functions*, Master's thesis and TCGA Report No. 88001 (The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa, 1988).

[16] S.W. Wilson and D.E. Goldberg, "A critical review of classifier systems," *Proceedings of the Third International Conference on Genetic Algorithms* (1989) 244–255.