Complex Systems **3** (1989) 493–530

Messy Genetic Algorithms: Motivation, Analysis, and First Results

David E. Goldberg
Bradley Korb
Kalyanmoy Deb
Department of Engineering Mechanics, University of Alabama,
Tuscaloosa, AL 35487, USA

Abstract. This paper defines and explores a somewhat different type of genetic algorithm (GA) — a messy genetic algorithm (mGA). Messy GAs process variable-length strings that may be either under- or over-specified with respect to the problem being solved. As nature has formed its genotypes by progressing from simple to more complex life forms, messy GAs solve problems by combining relatively short, well-tested building blocks to form longer, more complex strings that increasingly cover all features of a problem. This approach stands in contrast to the usual fixed-length, fixed-coding genetic algorithm, where the existence of the requisite tight linkage is taken for granted or ignored altogether. To compare the two approaches, a 30-bit, order-three-deceptive problem is searched using a simple GA and a messy GA. Using a random but fixed ordering of the bits, the simple GA makes errors at roughly three-quarters of its positions; under a worst-case ordering, the simple GA errs at all positions. In contrast to the simple GA results, the messy GA repeatedly solves the same problem to optimality. Prior to this time, no GA had ever solved a provably difficult problem to optimality without prior knowledge of good string arrangements. The mGA presented herein repeatedly achieves globally optimal results without such knowledge, and it does so at the very first generation in which strings are long enough to cover the problem. The solution of a difficult nonlinear problem to optimality suggests that messy GAs can solve more difficult problems than has been possible to date with other genetic algorithms. The ramifications of these techniques in search and machine learning are explored, including the possibility of messy floating-point codes, messy permutations, and messy classifiers.

1. Introduction

Barring a few notable exceptions, most current genetic algorithms (GAs) use decidedly *neat* codings and operators. Whether the domain is search, optimization, or machine learning, fixed-length, fixed-locus strings processed by one- or two-cut recombination operators are the rule in current work. Even those examples that violate the usual tidiness of conventional GAs do so in a rigid way, restricting the operators to process strings with only limited abandon. By contrast, nature's climb out of the primordium has occurred with genotypes that exhibit redundancy, overspecification, underspecification, changing length, and changing structure. Juxtaposition of the tidiness of current GA practice with the less-than-orderliness of nature begs us to question whether current GAs are losing important processing power by refusing to go along with nature's somewhat messy scheme of things.

In this paper, we claim that neatness is a limiting factor to the advance of GA art. Specifically, and more positively, we assert that allowing more messy strings and operators permits genetic algorithms to form and exploit tighter, higher-performance building block than is possible with random, fixed codings and relatively slow reordering operators such as inversion.

The remainder of this paper presents this argument in greater detail. Specifically, the reasons behind the current rigid state of affairs are examined and the essential limitation in such tidy GAs is uncovered. This immediately suggests the possibility of more flexible, messy codings and operators, and a simple version of a messy GA is defined in detail; some schema analysis reinforces the potential of such methods, and results from some initial proof-of-principle experiments demonstrate the ability of messy codings and operators to search for optima in problems too hard to be solved by simple GAs. In particular, a messy GA solves an order-three-deceptive, 30-bit problem to optimality; in the same problem, a simple GA using a random ordering of its genes makes errors at 75% of the bit positions. This is the first time that any GA has been reported to solve a provably difficult problem to optimality without prior knowledge of good string orderings. Some remaining challenges are discussed, and a number of extensions to these methods are also suggested, including messy floating-point codes, messy permutations, and messy classifiers.

2. Background: The neats and scruffies of genetic search

Why do genetic algorithmists do what they currently do? Although it is difficult to trace the precise steps leading to the formation of standard practice in any field, two publications — and their subsequent interpretation by other GA researchers — are largely responsible for the neatness of the current state of affairs. There are also a number of early GA studies that foreshadow a possibly messier GA future. In this section, we examine these "neats" and "scruffies"[1] of recent genetic algorithm history.

2.1 Why are GAs so neat?

Two signposts of recent GA history mark the headwaters of neat GA implementations: the publication of De Jong's (1975) Ph.D. dissertation and Holland's (1975) book.

The current standard GA practice of using fixed-length strings and neat genetic operators traces its roots to De Jong's (1975) pivotal dissertation. What is elsewhere [13] termed a simple genetic algorithm was first abstracted and tested in De Jong's pioneering work. Although Hollstien's (1971) dissertation was an important precursor, De Jong was the first to consider simple, fixed-length bit strings and stripped-down recombination and mutation operators in a carefully controlled suite of computational experiments guided by knowledge of the schema theorem [19]. Although we are about to examine an argument against adhering to a number of the implementation details of this work, it is important to recognize its importance. The contribution of this work was in its ruthless abstraction and simplification; De Jong got somewhere not in spite of his simplification but because of it (and because of his careful experimental design and execution), and the work's clarity has led many others to follow his good example. We should point out that De Jong did not intend then for his work to be slavishly imitated; his dissertation expressed an interest in more complex problems, codings, and operators, and his subsequent writing and research have demonstrated strong interest in problems that require less structured representations.

The other event that solidified the current tradition of neatness was the 1975 publication of Holland's *Adaptation in Natural and Artificial Systems* (*ANAS*). Holland's book laid the theoretical foundation for De Jong's and all subsequent GA work by mathematically identifying the combined role in genetic search of similarity subsets (schemata), minimal operator disruption, and reproductive selection. The intent of the book was quite broad, and although the proofs were presented in terms of neat codings and simple operators, these were not intended to limit GA practice. For example, Holland's discussion of not-so-neat operators such as intrachromosomal duplication, deletion, and segregation clearly demonstrates his support for less tidy operators, even when their inclusion might prevent as complete a theoretical analysis. Nonetheless, subsequent researchers have tended to take the theoretical suggestions in *ANAS* quite literally, thereby reinforcing the implementation success of De Jong's neat codings and operators.

[1] It is said that there are two types of AI enthusiast: neats and scruffies. Neats work on theorem provers, predicate logic, and other tidy objects about which things can sometimes be proved. Scruffies fool around with search, heuristics, and anything else that occasionally works. Although genetic algorithmists are generally considered to be card-carrying scruffies, it is ironic that the community is herein accused of having its shoe laces too neatly knotted and cravats too tightly pulled.

It is interesting, if not ironic, that neither man intended for his work to be taken quite so literally. Although De Jong's implementation simplifications established usable technique in accordance with Holland's theoretical simplifications, subsequent researchers have tended to treat both accomplishments as inviolate gospel.

2.2 Has anyone been messy?

Since the pivotal publications of Holland and De Jong, many have followed in their footsteps. This is not to say that no one has ever strayed from the straight and neater. Here, we examine a number of not-so-neat GAs, including Cavicchio's pattern detector GA, Frantz's work on inversion, Smith's poker player, and the various permutation GAs.

Cavicchio's (1970) study of GAs in a search for pattern recognition detectors was an early example of the use of variable-length strings. There, pixels on a two-dimensional grid were numbered, and alternating groups of positive and negative integers determined membership in different detectors that were then used by a pattern detection algorithm of fixed structure. In this particular application, there was no harm in mentioning the same pixel twice, nor was there any difficulty in not mentioning a pixel. In many other problem domains, the twin problems of over- and underspecification must be handled.

Smith (1980) used variable-length rules in his study of machine learning in a poker-playing task. His LS-1 system adopted modified crossover operators that permitted crosses both at and within rule boundaries. Smith's work was also notable for its inclusion of an inversion operator which permitted the somewhat successful rearrangement of rules on a string in the hope that such reordering would permit the juxtaposition of closely related rules. Since Smith, a number of machine learning applications have used variable-length strings and modified recombination operators [5,10,22].

Prior to Smith, Frantz (1972) had considered shifting loci in his study of nonlinear function optimization using inversion over strings of constant length. Unfortunately, choice of a relatively easy function permitted a simple GA without inversion to converge to near-optimal points; little differential advantage was possible for the GA with inversion over that without, and none was shown. Nonetheless, the work was a milestone in that it investigated changing locus GAs, a subject we shall return to in a later section.

A number of other authors [6,16,21] have investigated reordering operators in a somewhat different manner than Frantz by considering permutation representations and various permutation crossover operators that take bits and pieces of mated permutations to create offspring. The theory involving the schemata being processed has been developed [13,16], and the operators have been demonstrated with some success; however, whether these operators may be used to search for good bit orderings remains an open question.

In the next section, we examine the possibility of relaxing some of the neat ways ingrained in GA practice in the hope of developing more autonomous and more effective GAs.

3. Motivation for messy GAs: Better building blocks

Why consider abandoning the simplicity of fixed-length codes and one- or two-cut recombination operators? Parsimony in scientific endeavors is no vice, and making matters more complex must be done to some end good. Here, we look to other operators in a quest for better schema processing, and when schemata are an issue, we must appeal to the schema theorem. In this section, we review the fundamental theorem and consider how it may be used to support a case for better building block processing.

3.1 Remember the schema theorem

Recall that the schema theorem bounds the expected growth (or decay) of the number m of schemata $\mathbf{h}$ contained within a population at generation t as follows:

$$m(\mathbf{h},t+1) \geq m(\mathbf{h},t)\frac{f(\mathbf{h})}{\bar{f}}\left[1 - p_c\frac{\delta(\mathbf{h})}{l-1} - p_m o(\mathbf{h})\right] \tag{3.1}$$

where $f(\mathbf{h})$ is the average fitness of the string representatives of $\mathbf{h}$ contained in the population, $\bar{f}$ is the average fitness of the population, l is the string length, p_c is the crossover probability, $\delta(\mathbf{h})$ is the defining length of the schema $\mathbf{h}$ (the physical distance between the outermost defining positions of the schema), $o(\mathbf{h})$ is the order of the schema (the number of fixed positions contained in the schema), and p_m is the probability of mutation. Note that the schema theorem as presented here (and as usually presented elsewhere) is talking about a fixed-length, fixed-locus coding under simple crossover and mutation. This is most readily apparent from the constancy and form of the terms containing δ, o, and l.

The workings of simple GAs have been described in more detail elsewhere [13]. Briefly, a schema is expected to grow in subsequent generations if (1) it has above-average fitness, (2) it is relatively short, and (3) it is of low order. When all three conditions are met, we say that the schema in question is a *building block*. Some straightforward reasoning about building blocks unlocks the door to powerful genetic search.

3.2 Building blocks are the key

Examining the schema theorem more closely, we notice that all schemata receive increasing or decreasing numbers of trials, more or less, according to their fitness values. More specifically, a schema is expected to grow at least as fast as the growth factor γ, $\gamma = f(\mathbf{h})/\bar{f}\,[1 - p_c\,\delta(\mathbf{h})/(l-1) - p_m o(\mathbf{h})]$. Here we see the mathematical embodiment of the three conditions stated above. Whether the growth factor is greater than or less than one is a rough guide to whether a schema is or is not a building block. When $\gamma \geq 1$, future generations will contain an increased number of strings containing this template, and new strings will be created through the recombination of it and other building blocks.

This is the standard explanation of the workings of genetic algorithms, but there is an article of faith whose violation prevents simple GAs from converging to the most desirable points in certain problems. Suppose short, low-order schemata at certain locations are above average, but the combination of two such schemata (their intersection) is below average. For example, suppose the schema 00**** is highly fit and the schema ****00 is highly fit, but the schema 00***00 is much less fit than its complement, 11***11, which itself is a building block of the optimal point, 1111111. In the particular case, the GA will tend to converge to points representative of the less fit schema (perhaps points like 0011100), because with high probability, crossover will tend to disrupt the needed combination (11***11). This is the essence of what has elsewhere [11,12,14] been called *deception.* Coding-function combinations that have such misleading low-order building blocks can mislead a GA, causing it to converge to less-than-desirable points. To circumvent this difficulty, two alternatives are often suggested: prior coding of tight building blocks or the introduction of inversion.

The first of these suggestions is reasonable enough, but it assumes prior knowledge of the function being optimized. In the previous example, if we had known about the deception beforehand, we might have coded the string differently, so the four bits required to optimize the function were adjacent (e.g., 1111***). In that way, crossover would have been less likely to destroy the linkage, the schema's growth factor, γ, would have been higher, and the schema might then have constituted a building block. Although there are techniques that can identify where linkage is needed [2,12,14], in a general purpose search technique it is undesirable to require such prior knowledge (there is certainly no harm in using it if it is available, however).

If a coding for a particular problem is randomly chosen, we might be interested in knowing the probability of randomly coding a group of bits of order k with a defining length $\delta = d$ in a string of length l. Using straightforward combinatorial arguments, Frantz (1972) calculated the probability density function as

$$P\{\delta = d\} = (l-d)\frac{\binom{d-1}{k-2}}{\binom{l}{k}} \tag{3.2}$$

and the cumulative distribution as

$$P\{\delta \leq d\} = \left[\frac{k(l-d+1)+d+1}{d+1}\right]\binom{d+1}{k} / \binom{l}{k}. \tag{3.3}$$

The former may be calculated by fixing two of the k bits at the outermost positions of a $d+1$-bit template, calculating the number of possible combinations of the remaining $k-2$ bits, then sliding the template down the l-bit string one position at a time. The latter may be obtained by summing the probability density function by parts.

We have calculated the expected building block length $\langle\delta\rangle$ by taking the first moment of the probability density function and summing from $k-1$ to l. Some algebraic rearrangement and summation by parts yields the following equation:

$$\frac{\langle\delta\rangle}{l+1} = \frac{k-1}{k+1} \tag{3.4}$$

Thinking of the left-hand side as a normalized expected defining-length, it is remarkable how quickly it becomes quite long with increased k. Even for order-three building blocks the normalized expected defining length is already $2/4 = 0.5$, and matters get much worse fairly quickly. Thus, even for low-order bit combinations, the chances of being coded tightly are slim. For this reason and because prior knowledge of the function is not always available, inversion and other reordering operators have been suggested to recode strings on the fly.

Inversion strikes at the heart of the building block issue by suggesting that tight linkage can be found at the same time good bits are sought. To see how inversion might help, we first need to see how the usual string representation must be extended to permit bits to change their position (their locus) without changing the bit's meaning. Adding inversion to a simple GA requires that bits be identified with their names. For example, the string 1100011 might be identified by the string of ordered pairs

((1 1) (2 1) (3 0) (4 0) (5 0) (6 1) (7 1)).

Notice that even if the bits are randomly shuffled, because each bit is tagged by its name, we can retrieve the string's interpretation without difficulty.

With the representation so extended, simple inversion proceeds by picking two points along the string and flipping the included substring end over end. In the string above, for example, choosing inversion points between bits 2 and 3 and after bit 7 and performing inversion yields the string

((1 1) (2 1) (7 1) (6 1) (5 0) (4 0) (3 0))

Note that this fortuitous inversion brings the 1s together to form the tight building block 1111***. This example has been worked out to demonstrate the possibility of forming tight linkage, and clearly, reordering via inversion can bring together good building blocks, but whether inversion is a practical remedy to the formation of tight building blocks has been questioned. An earlier study [16] argued that inversion — a unary operator — was incapable of searching efficiently for tight building blocks because it lacked the power of juxtaposition inherent in binary operators. Put another way, inversion is to orderings what mutation is to alleles: both fight the good fight against search-stopping lack of diversity, but neither is sufficiently powerful to search for good structures, allelic or permutational, on its own when good structures require epistatic interaction of the individual parts. That same study argued that binary, crossover-like operators were necessary to obtain the binary juxtapositional processing of ordering schemata — o-schemata — as crossover obtains from allelic schemata — a-schemata.

Other arguments have been given against the ultimate usefulness of inversion. In a simple, order-two epistatic problem an idealized reordering operator modeled after inversion was shown to aid convergence [15], but it was found that the probability of reordering must be set to a relatively small value if the reordering was to permit ultimate convergence to the best. If the frequency of reordering must be kept low to guarantee convergence to good points, it is unlikely to be useful in a simultaneous search for good alleles and tight building blocks, because waiting times will be too large. Thus, our backs seem to be up against the wall. If we can't get reasonable convergence guarantees with fixed, random orderings, and if inversion is unlikely to generate the needed ordering soon enough to be of much use, how are tight building blocks and good allele combinations going to be found at the same time?

As has been hinted, our answer is to turn to nature and borrow variable-length strings and messy operators to first select and then juxtapose useful building blocks.

3.3 Two arguments from nature

Thus far, our arguments have been made on analytically theoretical grounds alone. Since GAs are loosely based on natural precedent, we might ask where nature stands on the issue of neat versus scruffy genetics. Actually, it is interesting to observe that evidence in nature can support either view, depending upon the time frame adopted.

In the short run, natural evidence supports the neats' argument. If viewed over 1,000 years or so (a blink of an eye by evolutionary standards), members of a species tend to mate with members of their own species, and recombination operators tend to be carefully tailored to maintain the full genetic complement of that particular species. Thus, viewed in the short term, intraspecies genetic interaction is crudely, but not wholly inaccurately, modeled by the usual simple GA.

Over the long haul, however, an entirely different — and messier — picture emerges. After all, nature did not start with strings of length $5.9(10^9)$ (an estimate of the number of pairs of DNA nucleotides in the human genome) or even of length two million (an estimate of the number of genes in *Homo sapiens*) and try to make man. Instead, simple life forms gave way to more complex life forms, with the building blocks learned at earlier times used and reused to good effect along the way. If we require similar building block processing, perhaps we should take a page out of nature's play book, testing simple building blocks early in a run and using these to assemble more complex structures as a simulation progresses. A number of technical problems need to be addressed in such an approach, but natural evidence supports the effort. In the next section, we investigate the details and analysis of one such messy genetic algorithm.

4. A messy GA defined

In this section, we define a straightforward mGA by describing its coding rules, decoding rules, and operators. We also perform a simple schema analysis to obtain some prior idea of expected system behavior.

4.1 Messy coding

Assume that the underlying problem has been reduced to a fitness function over the l-bit strings in the usual manner,[2] and further assume that each bit is tagged with its name (number). This is the same approach adopted in other moving-locus schemes, with two major exceptions. In the present study, we take no steps to insure that strings contain full gene complements, nor do we prevent redundant or even contradictory genes within a string. For example, the strings ((1 0) (3 1)) and ((1 0) (3 1) (2 0) (3 0)) are acceptable strings for a 3-bit problem in the messy scheme despite the underspecification in the first (no bit 2) and the overspecification in the second (two, 3-bits). As we shall soon see, this relaxation in coding permits the use of extremely simple genetic operators, but something of a price is paid in the added complexity of decoding the string. After all, if the problem has l-bits, and we have too many or too few, some additional processing is necessary to interpret the string if the objective function is to be sampled properly.

4.2 Handling overspecification

Of the twin problems introduced by the messy coding — overspecification and underspecification — overspecification is the easier to handle. Overspecification requires that we choose between conflicting genes contained within the same string. There are a number of possibilities: we could choose by using a probabilistic or deterministic voting procedure, by using positional precedence, or by using adaptive precedence.

In difficult nonlinear problems, bitwise voting rules, although egalitarian, seem unwise, because in deceptive problems wrong bits will have relatively high fitness and large numbers early in a run; their proliferation will prevent sampling of the correct building blocks.

Positional precedence is an appealing alternative, because it is simple and because it may permit the formation of a kind of intrachromosomal dominance operator. Because of these characteristics, simple positional precedence has been adopted in this study using a left-to-right scan and a first-come-first-served precedence rule.

[2] This assumption may be relaxed, and later we will examine the possibility of messy permutations, messy floating point codes, and messy classifiers (rules). Here, we construct a solution mechanism that solves the same problem as conventional GAs.

Adaptive precedence codings may also be imagined in analogy to the various adaptive diploidy and dominance schemes [18–20,24]. Simply stated, characters are added to the allelic alphabet and assigned relative priority values. The left-to-right scan would proceed as before, except that precedence would be decided first on the basis of priority, and the first-come-first-served rule would be used only to break ties between genes with equal priority values. Adaptive precedence has not proved necessary in this study; it is discussed in a later section as a possible remedy to a potential difficulty in problems with nonuniform building block order.

4.3 Handling underspecification

The first mGA experiments sidestep the problem of underspecification by making a simplifying assumption regarding the structure and accessibility of the fitness function. We assume that the function may be written as the sum of subfunctions f_j, where the f_j are themselves functions of nonoverlapping subsets of the Boolean variables x_i:

$$f(x_i) = \sum_{j=1}^{m} f_j(x_k, k \in K_j),\ i = 1, \ldots, l, \tag{4.1}$$

where the sets K_j form a partition of the full index set $\{1, \ldots, l\}$. When the function has this form, and when the search procedure has access to the individual subfunctions f_j, partial strings may be evaluated as follows: if a string samples one or more subfunctions fully, return the sum of the subfunction fitness values as the fitness of the string; otherwise, return a value of zero.

The assumption of such *partial string, partial evaluation* (PSPE) is a form of cheating, because it breaks the usual GA assumption of a black-box fitness function [13] and restricts the class of functions that may be handled by the method. We make the assumption for two reasons. First, many fitness functions (both natural and artificial) can evaluate a partial solution in a similar manner. In such cases, the assumption of partial string, partial evaluation is appropriate, and an investigation of the performance of an mGA under such conditions is useful. Second, where a full l-bit string must be passed to a black-box fitness function, it may be necessary to use averaging or other techniques to detect small differences in fitness. Since the most obvious schemes involve sampling error, we first consider the mGA under the assumption of PSPE to obtain a relatively noise-free trial of the messy algorithm. As it turns out, there is a relatively low-cost and noise-free way to beat the underspecification problem while still treating the function as a black box. We postpone this matter until we reconsider underspecification in a later section.

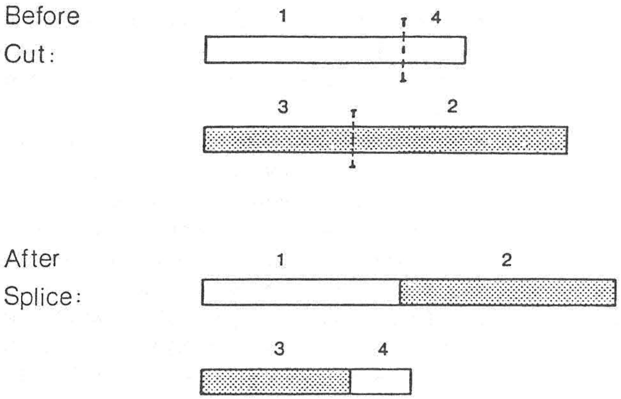

Figure 1: Schematic of cut and splice coordination.

4.4 Messy operators

Allowing variable-length, overspecified, or underspecified strings means that the usual simple crossover operator will no longer work. In this study, we replace crossover with two simpler operators: *splice* and *cut*.

The splice operator is as simple as it sounds. With specified splice probability p_s, two strings are concatenated. For example, starting with the two strings ((2 0) (5 1)) and ((3 1) (6 0) (5 1)), splice would yield the single string ((2 0) (5 1) (3 1) (6 0) (5 1)).

The cut operator is also straightforward. Using a specified bitwise cut probability p_κ, an overall cut probability is calculated as $p_c = p_\kappa(\lambda - 1)$, where λ is the current length of the string and p_c is subject to the limit $p_c \leq 1$. Thereafter, if a cut is called for, the string is cut at a position chosen uniformly at random along its length. For example, with $p_\kappa = 0.1$, the string ((2 0) (5 1) (3 1) (6 0) (5 1)) would have a probability of 0.4 being cut; supposing that a cut at location 3 was indicated, the two strings ((2 0) (5 1) (3 1)) and ((6 0) (5 1)) would be obtained.[3]

The cut and splice operators are coordinated to permit action similar to the simple crossover operator when both cuts and both splices are selected. Cut is performed first on two strings that have been mated by random choice. To see how this is accomplished, consider the schematic of mated chromosomes shown in figure 1. Assuming that a cut is called for on both chromosomes, the substrings are numbered as shown. Thereafter, the possibility of splicing is checked on successive pairs 1–2, 2–3, and 3–4. If pairing 1–2 is

[3] The cut operator can be performed cut site by cut site using independent Bernoulli trials to determine whether or not a cut is made. Doing so complicates the coordination of cut and splice unnecessarily, and only a single cut is permitted in this study.

spliced, the resulting string is placed into the new population, and the possibility of splice at between 3 and 4 is considered. If a splice is not called for between 1 and 2, 1 is inserted in the population, and a splice is considered between 2 and 3 (note the inversion-like action of this second type of splice). Similar orderings of splice trials are used when only one of the strings is cut or when neither string is cut.

In addition to cut and splice, an allelic mutation operator has been defined in the messy GA. With specified mutation probability p_m, a 0 is changed to a 1 and vice versa. For all runs presented in this paper, the mutation probability has been set to zero in order that the messy GA is given the most stringent testing possible. Without the diversity replenishing capability of mutation, the best building blocks had better be retained; otherwise, they will become extinct and will not be created again. It would also be possible to include a genic mutation operator that would change one gene to another with specified probability. No such operator has been included, although one might prove useful, if important genes were ever lost due to overzealous selection.

4.5 Reproduction revisited

For reliable selection regardless of function scaling, a form of *tournament selection* is used in this study [3,26]. Tournament selection combines selection and scaling into one step by holding tournaments between two or more combatant strings. A version without replacement is used in this study as follows:

1. Choose a tournament size $s \geq 2$.

2. Create a random permutation of the n integers.

3. Compare the fitness values of the next s population members listed in the permutation, selecting the individual with highest fitness value for subsequent genetic operation and insertion in the next generation.

4. If the n-permutation becomes exhausted, generate another.

5. Repeat starting at step 3 until no more selections are required during the current generation.

The algorithm is easily implemented and, as with other tournament selection algorithms, has desirable expected performance.

To understand the combined scaling-selection action of tournament selection, consider the expected number of copies that will be given to the best, the worst, and the median individuals within the population when the tournament size, s, is set to two. The best individual will participate in exactly two tournaments (in stochastic versions, the expected number of tournaments is two), and because the individual is best he will win both times, thus receiving exactly two copies. The worst individual will also participate in two tournaments, but will not win either. The median individual will participate in two tournaments, and because half of the individuals have higher fitness and half have lower fitness, the median should expect a single win. Thus, desirable scaling takes place without the unnecessary sorting of ranking schemes [1] or the other machinations of the usual scaling procedures [13].

4.6 Overall mGA organization

The usual simple GA proceeds by choosing an initial population at random, thereafter generating non-overlapping populations of constant size through repeated calls to reproduction, crossover, mutation, and other genetic operators. In the messy GA we use *partially enumerative initialization* combined with two phases of selection: the *primordial* phase and the *juxtapositional* phase.

In partially enumerative initialization, at least one copy of *all* possible building blocks of a specified size is provided, where the size is chosen to encompass the highest order deceptive nonlinearity suspected in the subject problem [14]. For example, with an order-2 deceptive problem we would initialize the population with all two-position combinations: on an l-bit function this would require a population of size $2^2\binom{l}{2} = 2l(l-1)$ individuals. Higher order nonlinearities would require correspondingly larger initial population sizes. In general, a population of size $n = 2^k\binom{l}{k}$ would be required for capturing deceptive nonlinearities of order k; however, these large populations are not as disadvantageous as they first appear. First, the primary factor in determining k is allied to the subtle concept of deception. A nonlinearity must be misleading if it is to cause difficulty for a GA, and oftentimes nonlinearities are simple [14]. Furthermore, weak nonlinearities may often be neglected, and many high-order nonlinearities in problems encountered in practice are weak.[4] Last, even though population sizes may be large initially, each string usually only requires a single function evaluation, and no further function evaluations are necessary before the population is reduced in size during the first phase of the mGA: *primordial selection*.

After partially enumerative initialization, the primordial phase of the mGA proceeds by selecting building blocks without invoking other genetic operators. The objective of doing so is to create an enriched population of building blocks whose combination will create optimal or very near optimal strings. Moreover, as selection proceeds during the primordial phase, the population size is reduced. Once good building blocks are chosen there is no need to maintain the population size associated with partially enumerative initialization. Reduction of the population size is accomplished by halving the population size at regular intervals.

[4] Other knowledge about the problem may also be used to reduce the number of building blocks that must be constructed during initialization. The algebraic methods suggested elsewhere [14] may be used to identify which bits are potentially epistatic, and this information may be used to restrict the initialization set. No such knowledge is assumed in this paper, although using it in this manner if it existed could eliminate many, if not most, of the function evaluations required by the mGA.

Following the primordial phase, the GA invokes cut, splice, and the other genetic operators during the *juxtapositional phase*, so named because of the juxtapositional nature of splice and cut. This phase resembles the usual processing in a simple GA except that the strings may vary in length. The population size is held constant, and in the simplest mGA, operator probabilities p_κ, p_s, and p_m are also held constant.

4.7 A simple schema analysis of cut and splice

We consider the survival — and the continued expression — of a schema under cut and splice. To do so, we must first define what we mean by a schema. Then we must calculate the probabilities of physically surviving the two primary operators. Last, we consider a schema's probability of continued expression after genetic processing.

Schemata are similarity subsets. In simple GAs, schemata may be represented by the usual similarity template notation, where a wildcard character (usually a *) is used to indicate positional indifference. In mGAs, genes are allowed to change position, and in the messy coding described earlier, the ordering of a gene does not directly affect its allele's fitness (position relative to other competing alleles will affect a gene's expression, but more on this in a moment). Thus, we define a schema in this study as a cluster of a particular set of genes containing a particular set of alleles with a defining length less than some specified value. Note that this definition is indifferent to both relative and absolute position of the genes within the cluster and within the string. Using this definition, it is a straightforward matter to calculate bounds on survival and expression probabilities.

The probability of a schema $\mathbf{h}$ surviving a cut, P_κ, is bounded by the following:

$$P_\kappa \geq 1 - p_c \frac{\delta(\mathbf{h})}{\lambda - 1}, \tag{4.2}$$

where δ is the schema defining length, λ is the length of the string containing the schema, and p_c is the composite cut probability defined earlier. Substituting the cutwise probability value into the expression, we obtain the bound

$$P_\kappa \geq 1 - p_\kappa \delta(\mathbf{h}). \tag{4.3}$$

The probability of a schema $\mathbf{h}$ surviving a splice, P_σ, is unity because splice is a conservative operator (actually splice is better than conservative in that it creates new combinations that did not exist prior to the operation; here, we conservatively ignore this source of new combinations).

If multiplied together, cut and splice appear to have something like the effective disruption of the simple crossover operator in a simple GA, but there is an important difference between messy GAs and simple GAs that has not been discussed. In a messy GA, the physical survival of a schema is insufficient to guarantee the optimization of difficult problems. In order to solve tough problems, the building blocks that make up the optimal solution must continue to be *expressed*. Although a schema may be physically present in a string, it may or may not be sampled, depending on its location and the presence or absence of one or more contradictory alleles. Expression is a concept usually discussed in connection with dominance and polyploidy, but the use of a redundant code and first-come-first-served precedence rules creates a type of intrachromosomal dominance operator which must be considered in this treatment.

Here, we consider the probability of continued expression of a currently expressed schema, P_ξ. So doing conservatively ignores the possible expression of a previously hidden schema. To undertake the calculation, define N as the event where a previously expressed schema is not expressed following cut and splice. Then the probability of continued expression following genetic operation may be given as

$$P_\xi = 1 - p_s P(N) \tag{4.4}$$

The event of not being expressed may be conditioned on two events: the event F that the subject string segment is placed at the front of a two-string pair, and the event B that the subject string is placed at the back of a spliced pair. This results in the computation

$$P(N) = P(N|F)P(F) + P(N|B)P(B), \tag{4.5}$$

If a string is currently expressed, and if it is placed at the front of a spliced pair, then there is no chance it won't be expressed following the splice: $P(N|F) = 0$. On the other hand, if a currently expressed schema is placed at the rear of a new string, the probability of expression, $P(N|B)$, depends upon the number and type of genes placed ahead of it. In general, the estimation of this probability is very difficult; however, we may calculate a bound on this probability using some fairly reasonable assumptions. First, we assume that the genes in the population are distributed uniformly at random. Second, we make the conservative assumption that the allele values within the uniformly distributed genes are biased against the schema at hand — the placement of even a single such gene in the front will prevent the expression of the schema in back. Under these assumptions, the probability of not being expressed given placement in the back may be calculated as

$$P(N|B) \leq 1 - \left(1 - \frac{k}{l}\right)^{\lambda^*}, \tag{4.6}$$

where λ^* is the maximum string length in the population. Applying the binomial theorem and dropping high-order terms yields the estimate

$$P(N|B) \leq \frac{k\lambda^*}{l}. \tag{4.7}$$

Recognizing that the splice operator places substrings at the front or back of the new string with equal probability, the probability of continued expression may be calculated as follows:

$$P_\xi \geq 1 - p_s \frac{k\lambda^*}{2l}. \tag{4.8}$$

Notice that the expression probability is quite large at short lengths; however, as the length of the forward segment grows, the probability of expression can become quite small.

Calculating the overall survival and expression probability of a schema, P_s, as the product of the individual probabilities, we obtain the following expression:

$$P_s \geq (1 - p_\kappa \delta(\mathbf{h})) \left(1 - p_s \frac{k\lambda^*}{2l}\right). \tag{4.9}$$

This equation may be reduced to the simpler form

$$P_s \geq 1 - p_\kappa \delta(\mathbf{h}) - p_s \frac{k\lambda^*}{2l} \tag{4.10}$$

after dropping higher order terms. The news contained in the combined equation is mixed. As long as overall string lengths remain low, the action of cut and splice is likely to be as nondisruptive as simple crossover; as the string length grows, however, disruption of expression becomes increasingly likely. This difficulty can be overcome if steps are taken to reduce the probability of the forward bits being in error; this reasoning was the primary motivation for the introduction of the primordial selection phase described earlier. By allowing the most highly fit schemata to get a significant foothold prior to juxtaposition, the probability of having large numbers of error bits at the front is greatly reduced and the probability of forming optimal or very-near-optimal strings is greatly enhanced.

4.8 Tying it all together

Thus, mGAs differ from their simpler counterparts in a number of important ways:

Use of variable-length strings that may be under- or overspecified.

Use of intrastring precedence rules (first-come-first-served or other).

Use of cut and splice instead of crossover.

Division of evolutionary processing into two phases: primordial and juxtapositional.

Use of a variable-size population with population size appropriate to the type of evolutionary phase.

Use of partially enumerative initialization of partial strings to encompass the longest misleading nonlinearity.

The proposal of these changes is not a case of being different for its own sake. Simple GAs are limited by their fixed codings, and the Band-Aid of inversion has not proved sufficiently powerful to enable them to find both good allele combinations and good orderings of those combinations at the same time. By contrast, mGAs have been designed from the beginning to find and exploit tight building blocks. In so designing, we have been careful to respect the schema theorem and other facts of GA life. In the next section, we test this new combination of operations, codings, and rules to see how it performs on a difficult, nonlinear function.

5. First results

In this section, a difficult test function is defined. Thereafter, a simple GA is used to search the function with tight, loose, and random orderings. Finally, the messy GA defined in the previous section is used to search the function, and the reasons for its success are examined.

5.1 A worthy opponent

Defining suitable test functions for artificial genetic search is not an easy task. In previous studies, researchers have chosen either some of the standard functions of the optimization literature or functions representating the class of practical problems in which the researchers were interested. These are reasonable starting points, but if one is interested in putting a GA (or other similarity-based search technique) to a stiff test, the notion of *deception* must be taken into account. Simply stated, a deceptive function is one in which low-order schema fitness averages favor a particular local optimum, but the global optimum is located at that optimum's complement.

As an example, the three-bit deceptive function of a previous study [12,14] is displayed in squashed Hamming cube format (figure 2) and as a function of the integers (figure 3). Simple calculations of schema average fitness values show that, on this function, a simple GA is likely to be led toward the second-best point, 000, instead of toward the global optimum, 111; such deception becomes more likely as the defining length of the bit combination increases (assuming that the three bits are embedded in a longer string). More intuitively, in the Hamming graph we see how the 000 optimum is surrounded by fairly fit individuals and how the 111 optimum is isolated: if the 111 optimum is to be found, it must be selected in one fell swoop; otherwise, 000 will be assembled from its more highly fit components.

Three bits does not a difficult problem make, however. To create a difficult test function, a 30-bit function is created as the sum of 10 copies of the

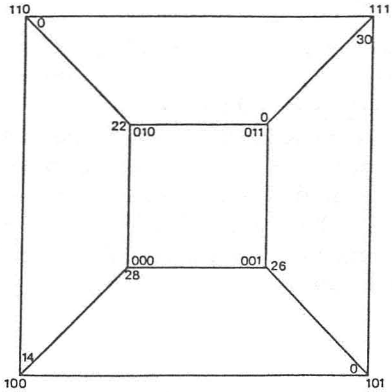

Figure 2: A fully deceptive, order-3 problem in Hamming space [12, 14].

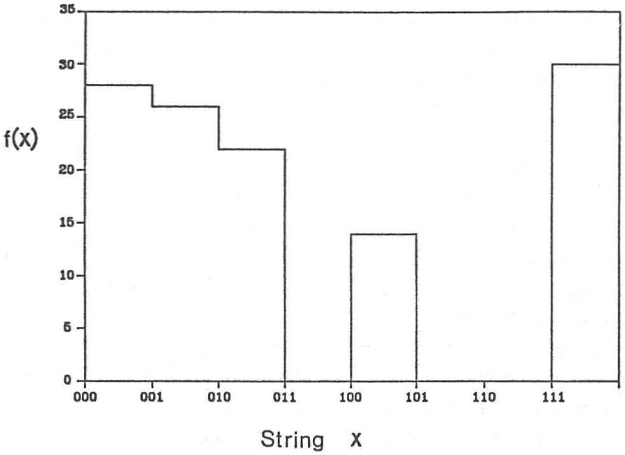

Figure 3: A fully deceptive, order-3 problem displayed as a function of the integers [12,14].

3-bit function, where each copy is associated with the next three bits. Thus, copy 1 is associated with bits 1 through 3, copy 2 is associated with bits 4 through 6, and so on. This seemingly innocuous function is difficult by most function optimization standards. The discrete space being searched is large: it contains $2^{30} = 1.074(10^9)$ or roughly a billion points. Looking at figure 2, note that each of the subfunctions has 2 optima. Thus, there must be 2^{10} local optima in the sense that no single-bit change can improve the function value. If the function were viewed as a discretized approximation to a continuous function of ten real variables (one for each copy of the function), figure 3 shows that such a representation would have 3 optima for each subfunction, meaning that there would be $3^{10} = 59,049$ optima in a ten-dimensional real space. Clearly, this is no job for gradient search or other local optimization algorithms.

In the next section, we examine how well a simple GA performs on this test function.

5.2 Three orderings and a simple GA

How well a simple GA will optimize this (or any deceptive) function is primarily determined by the ordering of the genes on the string. For example, if the genes are arranged in the order

1 2 3...28 29 30

all bits associated with a particular subfunction are as close as possible and the function is easy to optimize, because the defining length δ is small ($\delta = 2$ for all subfunctions): the schema theorem guarantees growth of the best building blocks (the 111s). On the other hand, if a poor arrangement such as

1 4 7...2 5 8...3 6 9...24 27 30

is used, the defining length of each building block is long ($\delta = 20$), the disruption is high, and the GA should converge to the fully deceptive local optimum (with 000 at the bit values of each of the 10 subfunctions). If a random ordering is used, equation (3.4) suggests that a defining length $\langle\delta\rangle = (l+1)(k-1)/(k+1) = 31(3-1)/(3+1) = 15.5$ should be expected. Checking the cumulative probability distribution, this point is roughly the 50th percentile defining length. Thus, in a randomly ordered string, we should expect approximately 50% of the strings to have defining length greater than the expected value and 50% to have a δ shorter than this.

As a benchmark, we try a simple GA using fixed orderings corresponding to those suggested above. Specifically we use a tight ordering (1 2 3...), a loose ordering (1 4 7...), and a random ordering (table 1). The simple GA code is similar to the Pascal code presented elsewhere [13] except for the use

j	Left	Center	Right
1	2	17	24
2	22	21	14
3	30	9	4
4	5	10	15
5	25	26	28
6	19	7	20
7	29	8	12
8	3	1	18
9	6	23	11
10	27	16	13

Table 1: String positions of the left, center, and right bits for the jth subfunction in the random ordering.

of tournament selection. The parameters of the simple GA were fixed for all runs at the following values:

$$p_c = 1.0.$$

$$p_m = 0.0.$$

$$n = 2000.$$

For each case, three runs starting with different seed values for the pseudorandom number generator were performed. To reduce cross-case variance, the same three seed values were used for each corresponding run in different cases, thereby starting the loose, tight, and random cases with the same initial populations. The results were averaged and are presented in figures 4 and 5. The results coincide with our expectations. The tight ordering runs converged to the optimal solution. The loose ordering runs converged to a solution with errors at all bits (the solution 000...000). The random ordering runs were able to get roughly 25% of the subfunctions correct. This last result is representative of what we should expect if we were to code the problem with no prior knowledge of important bit combinations. It is not encouraging to realize that a user of a simple GA may have to put up with errors at 75% of a difficult problem's bits. This result highlights the primary weakness of simple, fixed GAs. In the next subsection, we will see whether the mGA can overcome this difficulty.

5.3 The messy GA's turn

The messy GA described in section 4 has been coded in Common Lisp for execution on a TI Explorer. The messy GA has been run on the same problem using the following parameters:

$$p_\kappa = \tfrac{1}{60}.$$

$$p_s = 1.0.$$

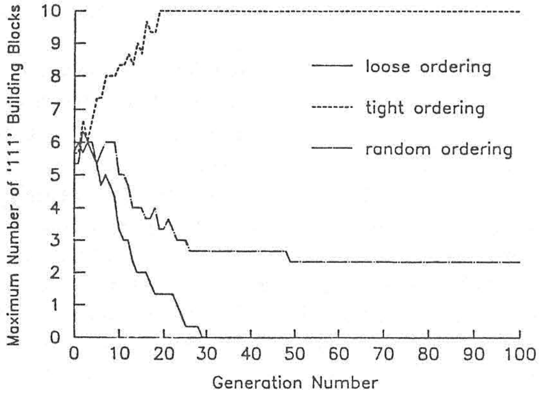

Figure 4: The generation maximum number of subfunctions optimized correctly by a simple GA graphed versus generation using tight, loose, and random orderings. The tight runs are able to optimize the function and the loose runs are not. The random runs get roughly 25% of the subfunctions correct. The population $n = 2000$.

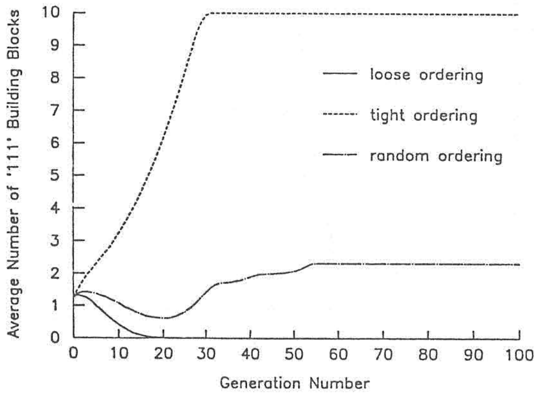

Figure 5: The generation average number of subfunctions optimized correctly by a simple GA graphed versus generation using tight, loose, and random orderings.

$p_m = 0.0$.

n =32,480 (primordial) down to 2030 (juxtapositional).

These parameters correspond roughly to those used for the simple GA benchmark runs.

The initial population is created using partially enumerative initialization as described earlier. Choosing $k = 3$ this procedure results in an initial population size of $n_0 = 2^3 \binom{30}{3}$ =32,480. The population size is cut in half every other generation thereafter until it reaches size $n = 2030$. Subsequently it is held constant through the end of the run, but the primordial phase is terminated (and the juxtapositional phase is started) after generation 11. The terminal population size ($n = 2030$) is roughly the same as the populations used in the simple GA runs (2000). Note that the simple GA has a raw material advantage over the mGA at this point, an advantage that is whittled away as the strings lengthen during the mGA's juxtapositional phase.

Results from three independent runs are shown in figures 6 and 7. In all three runs, the population contained at least one instance of an optimal solution at generation 15. This is noteworthy, because $t = 15$ is the first generation containing strings long enough to more than cover all 30 bits. Assuming no cut and 100% splice, the longest possible string length at this generation may be calculated as $\lambda_{\text{longest}} = 2^4 \cdot 3 = 48$. Thereafter, the rest

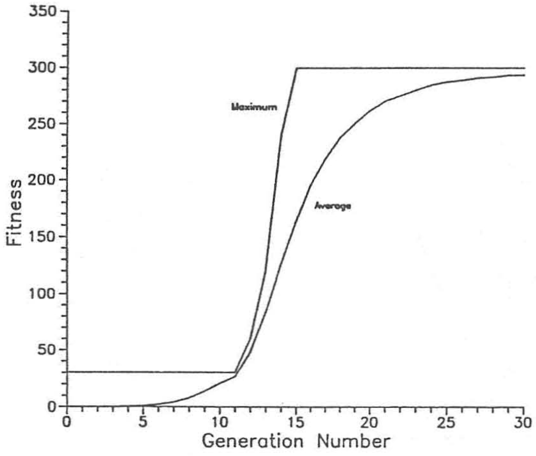

Figure 6: The messy GA converges in three independent runs to the optimal solution on the basis of generation maximum and generation average fitness (averaged over three runs). The first globally optimal solutions appear during generation 15, the first generation the strings are long enough ($\lambda \geq 30$) to contain a full solution.

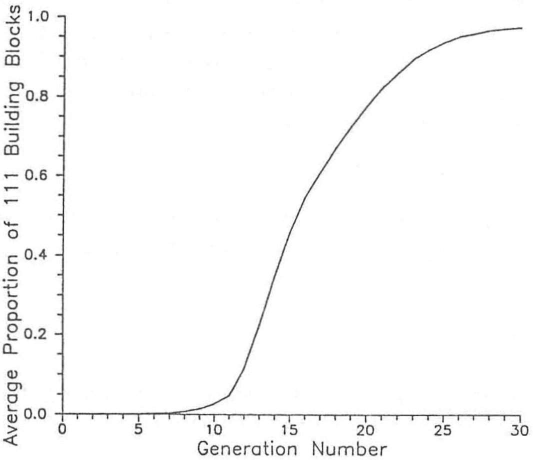

Figure 7: The average proportion of correct subfunction building building blocks (111) increases with subsequent generations.

of the population becomes dominated by optimal strings, as can be seen by examining how the average fitness approaches the maximum value (300) asymptotically.

Comparing the mGA results to the simple GA results on the basis of the number of function evaluations to optimality is enlightening. In the best simple GA, the tight ordering, optimal strings were found at generation 19, requiring a total of $(19 + 1)2000 = 40,000$ function evaluations. The mGA requires 32,480 function evaluations during the primordial phase (recall that the strings use their initial evaluation repeatedly during the primordial phase, regardless of the number of generations contained therein). Thereafter, four generations elapse before strings are found. Thus, the mGA requires a total of 32,480 $+4(2030) = 40,600$ function evaluations or roughly the same number as the simple GA. Even though the mGA had no special knowledge of good string orderings, it used roughly the same number of function evaluations as a simple GA that had access to perfect knowledge of good string orderings. The comparison is, of course, more dramatic if the mGA is compared to the random or loose cases. There, the mGA was able to solve the problem and the simple GA wasn't (and never would be). In the next subsection, we consider why the mGA is able to perform this well.

5.4 Why does the messy GA work?

To understand why the mGA works, we consider the primordial phase and the juxtapositional phase separately using crude mathematical models.

In the primordial phase, we are interested in the growth of the best building blocks. Under tournament selection two copies of the best individual are given to the best, except when they mate with one of their own, whence they receive a single copy each (or two copies, half the time). Under these conditions, the difference equations describing the proportion of best building blocks P may be written immediately:

$$P_{t+1} = 2P_t(1 - P_t) + (P_t)^2, \tag{5.1}$$

where the subscripts are time indices. Subtracting P_t from both sides and passing to the limit approximately, the following differential equation is obtained:

$$\frac{dP}{dt} = \frac{1}{P(1-P)}, \tag{5.2}$$

the logistic differential equation. Integrating between limits P_0 and P by elementary means yields the solution:

$$U = U_0 e^t, \tag{5.3}$$

where the auxiliary variable U is defined in terms of the proportion P as follows:

$$U = \frac{P}{1-P}. \tag{5.4}$$

The factor e^t may be adjusted to account (approximately) for the errors made in passing to the limit in a single step by including a factor $\beta = \ln 2$ in the exponential: $e^{\beta t} = 2^t$. Using this adjusted equation, the final proportion may be calculated as

$$P = \frac{P_0 2^t}{1 - P_0 + P_0 2^t} \tag{5.5}$$

Using this equation to predict the proportion at generation $t = 11$ we obtain $P = 0.39$, which matches well with the simulation results at the same time: from figure 7, we obtain the total number of 111 building blocks from the average number as $0.05 \cdot 10 = 0.5$.

With this analysis of the primordial phase, we must examine how well the mGA juxtaposes building blocks to form optimal or near-optimal solutions. We would hope that the mGA is at least as likely to be successful as a simple GA with a comparably high proportion of good building blocks. To evaluate whether this is true, we first consider a crude estimate of the juxtapositional power of a simple GA and compare it to a crude estimate of that of the mGA.

Assume in a simple GA with a good fixed ordering that after some number of generations, the probability of having good building blocks covering every subfunction is no less than p. Further assume that at this point, selection is indifferent to the good building blocks' competitors and no further improvement is likely (no increase in p can be assumed). We may then calculate after a sufficient number of crosses that the steady-state proportion of optimal strings after repeated crossover is at least $P^* = p^m$, where m is the number of building blocks required to optimize the function. Using this value, the waiting time (in numbers of individuals) for an optimal string may be estimated as $t \approx 1/p^m$.

This contrasts to the situation in an mGA where we note that there is no enforcement of positional diversity; nonetheless, we may argue for similar juxtapositional power if we account for the possibility of longer strings and the impact of the first-come-first-served rule. Consider the proportion of building blocks available at the end of the juxtapositional phase primordial phase, p'. After enough splices we expect to increase this proportion by a factor of m as we permit multiple building blocks per string. Ignoring increases due to further selection, the proportion of correct building blocks available for each subfunction is then $p = m \cdot p'$. In order to create optimal strings, we need to string together at least m of these building blocks without interruption by even a single bad building block. Ignoring duplications for the moment, the probability of exactly i events may be estimated using the geometric probability distribution:

$$P\{x = i\} = p^i q, \tag{5.6}$$

where $q = 1 - p$. With m subfunctions, we are interested in the probability of having m or more successes before the first failure. This may be calculated in straightforward fashion as

$$\begin{aligned} P\{x \geq m\} &= \sum_{i=m}^{\infty} p^i q \\ &= q[p^m + p^{m+1} + \ldots \\ &= qp^m \sum_{i=0}^{m} p^i \\ &= qp^m \frac{1}{q} \\ &= p^m \end{aligned} \tag{5.7}$$

Thus, the juxtapositional power of an mGA is roughly the same as that of a simple GA, and the formula has been used to size the population of the mGA runs presented earlier. With ten subfunctions and assuming a proportion of building blocks $p = 0.5$ for each subfunction, a population size of $n = 1/(0.5)^{10} = 1024$ is necessary to have a reasonable chance of seeing an optimal string as soon as the strings are long enough (four generations following the end of the primordial phase).

The foregoing analysis ignores the possibility of duplicates among the m constituents. An argument can be made that selection creates a strong pressure against duplication, leaving the simple analysis fairly useful without modification.

To see this, consider two strings associated with the 30-bit problem of a previous section, each with 111 building blocks only, one without duplication and one with duplication. Using a shorthand notation to identify the subfunctions mentioned in each string, we have the strings $A = 123$ and $B = 122$. Evaluating the two strings using the method of competitive templates (assuming a 000...000 template), we obtain fitness values $f_A = 3 \cdot 30 + 7 \cdot 28 = 286$ and $f_B = 2 \cdot 30 + 8 \cdot 28 = 284$. The duplication in string B causes it to be less fit than string A; thus, strings with fewer duplicates will be preferred to those with many, and selection will tend to drive duplicates from the system. This is true of all strings. For a given string length, strings with no duplicates will be preferred to those with duplicates, and this pressure tends to keep the number of duplicates low, thereby validating the analysis above.

With this basic understanding of the combined selective and juxtapositional power of messy GAs, we turn toward eliminating the restriction of partial string, partial evaluation.

6. Underspecification redux

In the first set of mGA experiments, we assumed that partial strings could receive a partial fitness value directly, without filling in unspecified bits. In practice, making this assumption requires special knowledge of the structure of the underlying objective function. Elsewhere, pure GAs don't require such prior knowledge [13] — pure GAs are blind — and it would be useful to devise a similarly blind formulation of an mGA. In this section, we examine a number of ways to obviate the need for the partial string, partial evaluation assumption. Specifically, we examine two approaches to underspecification through *averaging*, and show why they are unacceptable. We then develop the method of *competitive templates* and show that this approach is a useful mechanism for obtaining relatively noise-free evaluation of building blocks: using competitive templates, the mGA consistently optimizes the 30-bit problem of the previous section.

6.1 Why independent averaging won't work

The very idea of a messy GA changes our perspective of fitness evaluation. From square one, we would like to be able to evaluate the fitness of a part without possessing a whole. In many problems, this can be accomplished quite naturally, but in true black-box functions, partial string evaluation is not an option. For example, in our previous 30-bit problem, we might have partial strings such as ((1 1) (2 1) (3 1)) in the initial population, and with knowledge of the form of the particular function we know these and other such partial strings are the building blocks of optimal points, but how can they be evaluated then without resorting to the onerous assumption of partial strings, partial evaluation?

A natural reflex might be to suggest a sampling and averaging scheme. It would be easy to fill in the unspecified genes of a partial string using bits generated at random. Such random fill-in could be performed a number of times, and the average of the sample fitness values could then be used as the fitness of the partial string. This approach is attractive because of its simplicity, but the fallacy in so doing becomes apparent after some reflection.[5]

Suppose we have a building block that samples one of the 10 subfunctions comprising our 30-bit function of the previous section: for example, the partial string ((1 1) (2 1) (3 1)). By itself, this building block is worth 30; knowing that the average fitness of a single subfunction is 15, we may calculate the expected value of the building block when the other 27 positions are filled in with random bits: $f_{111} = 30 + 9 \cdot 15 = 165$. It might be useful to compare the fitness of the best building block to that of its second-best competitor, ((1 0) (2 0) (3 0)). Calculating the expected fitness in the same manner, we obtain $f_{000} = 28 + 9 \cdot 15 = 163$, All seems well and good, because in expectation, the better building block should win the competition of selection; however, consideration of the variance — consideration of the sampling error — puts this calculation in a different light. Assuming that the eight points of a single order-three function are sampled uniformly at random, we may calculate the variance of a single subfunction as $\sigma^2 = \sum f^2/8 - \bar{f}^2 = 155$. Since the three bits of each of the partial strings fix one of the ten functions, the other nine are being sampled randomly, and the variance of the sampled fitness may be calculated as the sum of the nine variance values or simply $\sigma^2_{000} = \sigma^2_{111} = 9 \cdot 155 = 1395$. Since in tournament selection, we are essentially using the sign of the difference between two such sums to choose winners, the variance of that difference is twice the value just calculated, $\sigma_{\Delta f} = 2 \cdot 1395 = 2790$. Taking the square root, we obtain a standard deviation value of 52.8, which compares unfavorably to the difference we must detect ($f_{111} - f_{000} = 2$). In other words, a single sample is unlikely to correctly identify the correct building block. More samples can help, however. Suppose that n strings are sampled for each of the 000 and 111 substrings. Setting the signal difference we wish to detect ($165 - 163 = 2$) to the standard deviation divided by the square root of the sample size and rearranging, we obtain a rough calculation of the sample size: $n \approx 2790/4 = 697.5$. Under this sampling scheme, more than 700 samples would be required per string to have even a reasonable chance at detecting the small difference between the best and second best building blocks. Even in an order-3 problem, such sampling is out of the question.

Averaging seems doomed by the concrete example of the previous paragraph, and a more general calculation shows that the naive approach to averaging is usually a problem. Generalizing the specific calculation of the previous paragraph to a problem with m subfunctions and a maximum sub-

[5] The reasoning that follows also casts doubt on the effectiveness of small-population, simple GAs in evaluating building blocks. The same variance argument applies to the usual simple GA, and unless very large populations are used, the schema-difference-to-noise ratio is likely to be too high, resulting in the high probability of selection error for many of the competing schemata in a population. This difficulty has gone largely unaddressed in the current literature of genetic algorithms.

function variance,[6] $\sigma^2_{\max}$, we may bound the variance of the difference between random samples of two different partial substrings, σ^2_d, as

$$\sigma^2_d \leq 2m\sigma^2_{\max}. \tag{6.1}$$

The factor m is used here instead of $m-1$, because an arbitrary building block does not necessarily cover even a single subfunction, leaving the possibility of nonzero variance at all subfunctions. When the difference in fitness we wish to detect, Δf, is smaller than the standard deviation of the difference divided by the square root of the sample size — when $\Delta f \ll \sigma_d/\sqrt{n}$ — averaging in this manner is unlikely to be useful.

6.2 In-common averaging doesn't work either

A somewhat more sophisticated approach to averaging might not be as noisy, and deserves our consideration. Instead of using independent random trials to fill in the missing bits, we could generate one or more random strings *in common*, and use these common strings to fill in the unspecified bits of the partial strings. The advantage of this approach becomes clear if we reexamine the example discussed above. Using common fill-in in the example, the partial strings ((1 1) (2 1) (3 1)) and ((1 0) (2 0) (3 0)) would share the same randomly selected bits at their remaining 27 genes. Assume that the fitness contribution of those 27 bits is f_{27}. For the top building block (111), the total fitness would be $f_{111} = 30 + f_{27}$, and for the second-best building block (000) the total fitness would be $f_{000} = 28 + f_{27}$. Since tournament selection chooses winners on the basis of the difference between fitness values, the better building block, in this example, would always be chosen, because $f_{111} - f_{000} = 30 - 28 + f_{27} - f_{27} = 2$. Moreover, the variance over the 27 in-common positions is zero because the template is shared by both substrings. Notice that the proper building block is selected even when only a single, common fill-in string is used.

The need for only a single sample contrasts nicely to the number required using the naive averaging approach, and it would be useful if the result generalized neatly to arbitrary building blocks, but unfortunately it does not. To see this, consider two other building blocks over different string positions: ((1 1) (2 1) (3 1)) and ((4 0) (5 0) (6 0)). If the positions are filled in common, we may write the fitness of the better building block as $f_{111} = 30 + f_{(4,5,6)} + f_{24}$, where $f_{(4,5,6)}$ is the fitness sampled at the subfunction specified over bits 4-6, and f_{24} is the fitness sampled at the 24 positions in common with the other building block. Likewise, the fitness of the other

[6] Note that this maximum variance is not bounded by the variance taken over all points. For example, fixing a 1 in the third position of the three-bit subfunction (evaluating **1) yields greater variance ($\sigma^2_{**1} = 198$) than that achieved by summing over all points ($\sigma^2_{***} = 155$). More generally, $\sigma^2_{\max}$ may be bounded by following expression: $\sigma^2_{\max} \leq 2^k(f_{\max} - f_{\min})^2$, where k is the number order (the number of bits) in the subfunction, and $f_{\max}$ and $f_{\min}$ are the highest and lowest fitness values, respectively. This result follows because the highest variance occurs when points cluster at the extremes, and the maximum variance occurs when there is an even distribution of points at each extreme.

building block may be calculated as $f_{000} = 28 + f_{(1,2,3)} + f_{24}$, where $f_{(1,2,3)}$ is the fitness sampled randomly at the subfunction specified over bits 1-3. Taking the difference, we obtain the interesting result that $f_{111} - f_{000} = 30 - 28 + f_{(4,5,6)} - f_{(1,2,3)} + f_{24} - f_{24} = 2 + f_{(4,5,6)} - f_{(1,2,3)}$. The fitness contribution of the subfunctions in common disappears, but the unspecified bits in subfunctions that are fully or partially specified by the competitor must be sampled, and if these samples are made randomly, the variance must be taken into account. In the particular case, the variance of the signal difference may be calculated as $\sigma_d^2 = 2\sigma^2 = 310$, where σ^2 is the variance of a single subfunction. Taking the indicated square root, we obtain a standard deviation of 17.6, which does not compare favorably with the difference we wish to detect ($\Delta f = 2$). Turning the argument around as before, we estimate the order of magnitude of the number of samples required as $310/4 = 77.5$. This is better than a 700 or so, but it is still impractical. To make matters worse, the bits in a partial string may not fully sample a particular subfunction. When this occurs, the schema average of that particular subfunction must be sampled in addition to the corresponding fill-in bits in the competitor.

This leads to the calculation of a bound on the variance of the signal difference. For two strings with average length $\bar{\lambda}$, the variance of the signal difference may be bounded by the following relationship:

$$\sigma_d^2 \leq 4\alpha\sigma_{\max}^2, \text{ where } \alpha = \begin{cases} 4\bar{\lambda} & \text{if } \bar{\lambda} < m/2 \\ 2m & \text{otherwise} \end{cases} \tag{6.2}$$

where $\sigma_{\max}^2$ is the maximum variance that may be achieved by sampling all or some of the bits of any subfunction. This follows, because in the worst case each of the bits of each of the strings samples a different subfunction, which means that a subfunction and its competitor must be sampled for each bit contained in a string; taking the variance of the signal difference yields the factor of four.

This result is not encouraging. Even for fairly modest building block sizes, common fill-in is too noisy to detect small signal differences reliably.[7]

6.3 An alternative: Competitive templates

The use of random templates, whether independent or in-common, is too noisy to be of much use in practical mGAs. Nonetheless the bare notion of

[7] There is another fundamental flaw in the use of averaging schemes that ultimately would prevent their use in mGAs. Even if a magic template could be devised to return a schema's average fitness, this would not by itself be useful in promoting the best bit combinations in many problems. In a messy GA, since all building blocks of a specified size are tried, and since many problems have single bits that are both deceptive and characterized by high average fitness values, the most highly fit bit combinations (as measured by schema average fitness) would take and string together a number of highly fit singletons from different subfunctions. Without other operators, these would soon dominate the population, many bits of the problem might be left uncovered, and very few multiple-bit building blocks would survive. The solution proposed shortly eliminates this problem. Other scaling problems that may arise are discussed in section 7.

using a template deserves further attention, especially if one can be devised that permits salient building blocks to stand out and be selected reliably.

The key here is the notion of salient building blocks. It may or may not be possible to preserve the relative fitness ordering of all the building blocks of the function using a fixed template; however, if the relative rankings of the best building blocks and other locally optimal building blocks are preserved using a template, and as long as other building blocks have a *template fitness* (the fitness returned by the function after filling in a partial string's unspecified positions with the template values) less than or equal to that of the lowest locally optimal point, selection should yield building blocks at least as good as the lowest locally optimal point.

This reasoning immediately suggests one solution: *use any locally optimal point as a template.* So doing will guarantee that no one-bit change can improve the function, and therefore any change that increases the fitness must be a building block. Referring back to the 30-bit problem, this would imply starting with points such as 000 . . . 000 (points with 111 building blocks also are locally optimal, but if we are interested in worst-case performance, it seems like cheating to start with partially correct solutions, so we will take steps to prevent this). In the particular problem, innocuous building blocks such as 000 will have no effect on the template's base fitness value of $10 \cdot 28 = 280$, partial sampling of a building block with a single 1 or two 1s (001 or 011), will depress the template fitness, and only the correct bit combinations will receive higher fitness than the base template value.

Using locally optimal templates seems useful, because it permits only salient building blocks to receive higher fitness than the base template value, but how can such a template be generated without prior knowledge of the function? In this study, we will adopt a greedy procedure, sweeping bit-by-bit through all l bits. The steps of this preprocessing algorithm are quite simple:

1. Generate an l-bit string at random.

2. Generate an l-permutation at random.

3. In permutation order, alternately change each bit from its current value to its complement, permanently recording those changes that improve fitness.

4. Repeat starting at step 2 a total of n_{sweeps} times.

5. Use the resulting l-bit string as the *competitive template.*

Although the suggested algorithm may require many sweeps to converge to a local optimum in the worst case, we will perform only a single sweep in this study. If a function is bitwise linear or behaves like a linear function in all neighborhoods, a single sweep guarantees convergence to a local optimum. A discussion of the mathematical criteria for a nonlinear function to be linear-like locally optimizable (LLLO) is beyond the scope of this discussion; the argument uses analysis methods described elsewhere [12,14]. We note that the three-bit function used to construct the 30-bit function is LLLO — all starting points may be optimized one bit at a time, reaching one of the two optima (000 or 111) in a single sweep.

Alternatives to this template generation method are certainly possible. For example, a simple GA can be run with a small population size, simple mutation, and shuffle crossover (where exchange of genes between chromosomes is determined by a successive coin tosses); after substantial convergence a local optimum or near local optimum is likely.

In the next subsection, the method of *competitive templates* is applied to the 30-bit problem.

6.4 Competitive templates on the 30-bit problem

The method of competitive templates is used to solve the 30-bit problem of section 5. To give the algorithm its stiffest test we use the competitive template of lowest fitness, the string $000\cdots000$, instead of generating templates at random and using the bit-scan technique described above. Since the function is LLLO, we know that all the templates generated in a single pass will be locally optimal, and we can do no worse than generate the $000\cdots000$ point.

Figure 8 shows the maximum and average fitness versus generation, averaged over three runs that used different random seeds. In all three runs,

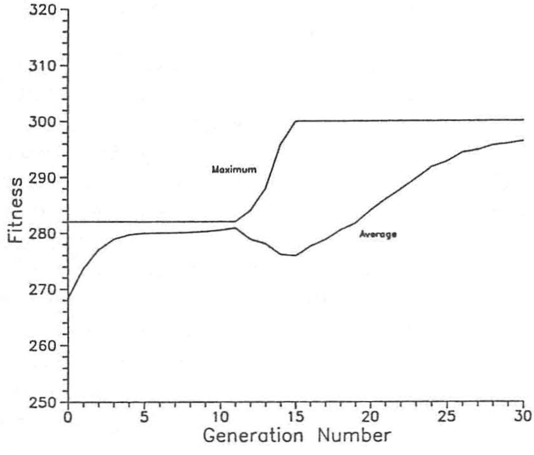

Figure 8: Generation maximum fitness and generation average fitness (both averaged over three runs) using a messy GA with the method of competitive templates.

the algorithm found an optimal point at the first generation possible (generation 15), and in terms of best performance the graph is very similar to that of figure 6. The dip in average performance may be explained by recognizing that when strings are first combined, the combining of 1s and 0s generates substandard building blocks; however, selection washes these out fairly quickly, and the operation follows the reasoning of section 4.8; the high proportion of the best building blocks guaranteed by primordial selection quickly leads to a high probability of having optimal or very near optimal points as soon as the strings are long enough.

6.5 Is the method of competitive templates biologically plausible?

At first glance, the method of competitive templates seems to be little more than a convenient hack, and to some extent the centralized implementation discussed above is just that. Certainly, there is no reason to expect that nature distributes nearly locally optimal solutions in anything similar to the manner discussed, and the implementation decisions that were made reflected the needs of computational convenience on a serial machine. On the other hand, the basic idea of reducing building sampling error by searching from a nearly locally optimal structure seems fundamental, and perhaps a biologically plausible formulation can be imagined.[8] In fact, a straightforward mechanism may be envisioned if we return to the notion of adaptive precedence mentioned briefly in section 4.2.

Imagine that the primordial selection and subsequent juxtaposition of low-order building blocks proceeds in a first phase of the evolutionary process, and further suppose that these low-order building blocks are tagged with relatively low precedence values. At the end of the first evolutionary phase, the distributed equivalent of competitive templates will exist; the strings in the population should be local optima or near local optima. Thereafter, if we can envision the random generation of building blocks of longer length that also have higher precedence values than the low-order building blocks, these will be tested on top of the locally optimal solution in much the same manner that has been built into the mGA explicitly. Of course, there is no reason to stop imagining at this point. Even longer length improved building blocks may piggyback on top of the longest solutions discovered to date as the process continues, as long as their precedence values allow the building blocks to be (at least occasionally) expressed in place of their previously discovered competitors. In this way, new building blocks are expressed in the context of nearly locally optimal solutions, thus insuring gains in the population by only those individuals that are real improvements; as occurs with the explicit formulation of competitive templates used herein, the implicit method should allow salient building blocks to be selected in relatively small populations.

[8] Our need here is similar to that of the neural network researcher when facing the biological plausibility of backpropagation, Boltzmann machines, or other weight-changing algorithms. Although we may stray far afield of nature's implementation mechanisms, we should try to imagine how the algorithm being proposed — or a reasonable approximation thereof — might be implemented in natural "hardware."

7. Some remaining challenges for mGAs

This paper has taken some important steps toward GAs that can discover good allele combinations at the same time good string orderings are sought. There are a number of challenges that remain, however, before the method is fully proven, including the problems of subfunction scaling and parasitic bits. In this section, these problems are discussed and plausible solutions are suggested.

7.1 Subfunction scaling

In designing the 30-bit problem used to test the mGA, care was taken to choose a problem that would mislead the algorithm if the required tight building blocks were not formed. This was the primary requirement, but in some other respects, the problem was not sufficiently representative of the kinds of problems that might be encountered in practice. In particular, the problem was homogeneous with respect to subfunction magnitude and order. The first characteristic may in other problems lead to the problem of subfunction *scaling*.

To see the difficulty, imagine two optimal building blocks associated with different non-overlapping positions: for example, $A = ((1\ 1)\ (2\ 1)\ (3\ 1))$ and $B = ((4\ 1)\ (5\ 1)\ (6\ 1))$. Suppose also that the fitness of building block A is larger than that of building block B, $f(A) > f(B)$. This creates a potential problem unless further steps are taken. In the 30-bit problem, all subfunctions were scaled identically, so there was little or no danger of permitting comparisons between building blocks that cover separate subfunctions, because the fitness value was a reflection of the building block's global rank. On the other hand, if a problem is chosen so that different subfunctions are scaled differently, then comparisons between building blocks A and B are not meaningful — tournament selection compares apples and oranges — and should be restricted.

To circumvent this difficulty in future experiments, selection will be permitted only between individuals that contain some of the same genes. The first candidate for tournament selection will be picked normally. Thereafter, the remaining candidates will be chosen as those next individuals in the permutation list that have a specified number (the threshold value) of genes in common with the first candidate. Thus, performance of this *genic, selective crowding* will help insure that tournament selection compares individuals that contain some common genes. In this way, like will be compared to like and some pressure will exist to preserve those optimal building blocks that are poorly scaled. This method is not unlike the various crowding and sharing techniques [7,8,17,19], except that alleles (the 1s and 0s) will not be checked.

7.2 Parasitic bits

Problems that contain subfunctions that are nonhomogeneous in the number of bits required to optimize the subfunctions may encourage the selection of *parasitic bits* during primordial selection. Assuming that most of the subfunctions in a particular problem are of length 4 and the problem has been initialized by all length $k = 4$ bit combinations during partially enumerative initialization, but assume that bits 1–3 are well optimized by the combination ((1 1) (2 1) (3 1)). During primordial selection, the three-bit combination will be filled in by bits that tend to agree with the competitive template. During primordial selection this does little harm, but during juxtapositional selection these parasites will tend to prevent the expression of other more useful bit combinations.

To prevent this difficulty, the genic alphabet will be extended by $k - 1$ characters, where k is the initialization building block length. These extended genes will have a single value: N (null). In this way, the extended genes may fill in a short building block in place of spurious bits, thereby permitting the subsequent expression of other bit combinations. For example, the string ((1 1) (2 1) (3 1) (31 N) would express the 111 building block as a length four string without parasites in the $l = 30$ problem.

To implement this idea, there is one further requirement. The number of null bits should be used in the selection process to break ties between strings of equal fitness. For example, the proper building block ((1 1) (2 1) (3 1) (31 N) and a string with parasitic bit ((1 1) (2 1) (3 1) (20 0)) would have the same fitness value during primordial selection. If the number of null bits is used to break ties, the proper building block would have preference when the two went head to head during tournament selection. Biologically, this tie-breaking procedure can be interpreted as a preference for most efficient representations, which may itself be argued on minimal energy grounds. Other things being equal, an organism with less hardware requires less energy and therefore has a fitness advantage, albeit a slight one, when phenotypic expression is otherwise the same.

8. Extensions

This work may be extended along a number of lines. Here, possible messy floating point codes, messy permutations, and messy classifiers are discussed.

8.1 Messy floating point codes

The ARGOT system [23] implements an adaptive floating point coding scheme, but it does so in a rigid manner with global population statistics used to guide the selection of a plethora of ad hoc operators. The variable length strings of mGAs open the door for more natural, adaptive floating point codes. In one such messy floating point code, the string ((M 3 1) (M 3 0) (E 3 0) (F 3 0)) decodes to a value of 0.05 for the third parameter, because one-half of the mantissa bits (M bits) have value 1, and because the single exponent bit dictates a unary minus (1 = plus and 0 = minus). The final F bit is used as a parameter fixing punctuation mark. Its presence prevents subsequent genes designating parameter 3 from changing the already calculated value (assuming the usual left-to-right scan). In this way, messy GAs should be able to put their precision where it is useful.

8.2 Messy permutation codes

The traveling salesman problem and other problems over permutations have received some attention with GAs. As was mentioned earlier, these studies have required specialized operators to preserve crossover operators to preserve tours. With the introduction of the messy approach, this no longer seems quite so urgent. Partial permutations may be tested during the primordial phase (using the genic selective crowding operator discussed earlier) and recombined during the juxtapositional phase. The first-come-first-served precedence scheme may work well enough, or perhaps adaptive precedence will be necessary in this class of problems.

8.3 Messy classifiers

The production rules called *classifiers* that are often used in genetics-based machine learning systems are a natural application of messy GAs. In the usual classifier system, the classifier 00###:00011 would be matched by any previously posted message that begins with an 00 and would, thus, be a candidate for posting its own message to the message list at subsequent iterations. In a messy classifier system, the rule ((M 1 0) (M 2 0) (S 4 1) (S 5 1)) would encode the same rule using match genes (M), send genes (S), and a host of other pass through characters and punctuation marks. These suggestions can take us closer toward Holland's (1975) *broadcast language* proposal in a step-wise fashion.

9. Conclusions

This paper has defined, analyzed, and partially explored a new type of genetic algorithm called a messy genetic algorithm. Messy GAs first find and emphasize the good building blocks of optimal or near-optimal solutions in what is called the primordial selection phase. Thereafter, during the juxtapositional selection phase, cut and splice operators recombine the good building blocks to form optimal points with high probability. Bounding analyses have been performed and these have helped shape the design presented herein.

A difficult test function has been designed, and in two sets of experiments the mGA is able to find its global optimum. In the first set of experiments, the assumption of partial string, partial evaluation, PSPE, was made to test the method without sampling noise. In the second set of experiments, the method of competitive templates was developed to permit noise-free partial string evaluation without the limiting assumption of PSPE. In all runs on both sets of experiments, the mGA converges to the test function global optimum. By contrast, a simple GA using a random ordering of the string is able to get only 25% of the subfunctions correct.

This is the first time that any genetic algorithm has been reported to converge to a global optimum in any provably difficult problem; however, further evaluation of the mGA technique is required before firm conclusions may be drawn, and a number of remaining challenges have been discussed in this paper. Nonetheless, the potential of these methods in very difficult optimization problems with many false optima has been demonstrated by this proof-of-principle study. As the remaining challenges are solved, these techniques may be extended to many difficult problem domains using messy floating point codes, messy permutation codes, and messy classifiers as suggested in this paper.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant CTS-8451610.

References

[1] J.E. Baker, "Adaptive selection methods for genetic algorithms," *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (1985) 100–111.

[2] A.D. Bethke, *Genetic Algorithms as Function Optimizers*, doctoral dissertation, University of Michigan, *Dissertation Abstracts International*, **41(9)** (1980) 3503B, University Microfilms No. 8106101.

[3] A. Brindle, *Genetic Algorithms for Function Optimization*, doctoral dissertation and Technical Report TR81-2, Department of Computer Science, University of Alberta, Edmonton (1981).

[4] D.J. Cavicchio, *Adaptive Search Using Simulated Evolution*, unpublished doctoral dissertation, University of Michigan, Ann Arbor (1970).

[5] N.L. Cramer, "A representation for the adaptive generation of simple sequential programs," *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (1985) 183–187.

[6] L. Davis and D. Smith, *Adaptive Design for Layout Synthesis*, Texas Instruments Internal Report, Texas Instruments, Dallas (1985).

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

[8] K. Deb, "Genetic algorithms in multimodal function optimization," master's thesis and TCGA Report No. 89002, The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa (1989).

[9] D.R. Frantz, *Non-Linearities in Genetic Adaptive Search*, doctoral dissertation, University of Michigan, *Dissertation Abstracts International* , **33(11)** (1972) 5240B–5241B, University Microfilms No. 73-11, 116.

[10] C. Fujiko and J. Dickinson, "Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma," *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (1987) 236–240.

[11] D.E. Goldberg, "Simple genetic algorithms and the minimal deceptive problem," in *Genetic Algorithms and Simulated Annealing*, L. Davis, ed. (Pitman, London, 1987) 74–88.

[12] D.E. Goldberg, "Genetic algorithms and Walsh functions: Part I, a gentle introduction," TCGA Report No. 88006, The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa (1988).

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

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

[15] D.E. Goldberg and C.L. Bridges, *An Analysis of a Reordering Operator on a GA-hard Problem*, TCGA Report No. 88005, The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa (1988).

[16] D.E. Goldberg and R. Lingle, "Alleles, loci, and the traveling salesman problem," *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (1985) 154–159.

[17] D.E. Goldberg and J. Richardson, "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.

[18] 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.

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

[20] R.B. Hollstien, *Artificial Genetic Adaptation in Computer Control Systems*, doctoral dissertation, University of Michigan, *Dissertation Abstracts International*, **32(3)** (1971) 1510B, University Microfilms No. 71-23,773.

[21] I.M. Oliver, D.J. Smith, and J.R.C. Holland, "A study of permutation crossover operators on the traveling salesman problem," *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (1987) 224–230.

[22] J.D. Schaffer, *Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms*, unpublished doctoral dissertation, Vanderbilt University, Nashville, TN (1984).

[23] C.G. Shaefer, "The ARGOT strategy: Adaptive representation genetic optimizer technique," *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (1987) 50–58.

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

[25] S.F. Smith, *A Learning System Based on Genetic Adaptive Algorithms*, unpublished doctoral dissertation, University of Pittsburgh (1980).

[26] J.Y. Suh and D. Van Gucht, *Distributed Genetic Algorithms*, Technical Report No. 225, Computer Science Department, Indiana University, Bloomington (1987).